Object Libraries

Modeling Algorithms User's Guide

Version 6.5.4 / October 2012

Copyright © 2012, by OPEN CASCADE S.A.S.

PROPRIETARY RIGHTS NOTICE: All rights reserved. Verbatim copying and distribution of this entire document are permitted worldwide, without royalty, in any medium, provided the copyright notice and this permission notice are preserved.

The information in this document is subject to change without notice and should not be construed as a commitment by OPEN CASCADE S.A.S.

OPEN CASCADE S.A.S. assures no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such a license.

CAS.CADE, **Open CASCADE** and **Open CASCADE Technology** are registered trademarks of OPEN CASCADE S.A.S. Other brand or product names are trademarks or registered trademarks of their respective holders.

NOTICE FOR USERS:

This User Guide is a general instruction for Open CASCADE Technology study. It may be incomplete and even contain occasional mistakes, particularly in examples, samples, etc.

OPEN CASCADE S.A.S. bears no responsibility for such mistakes. If you find any mistakes or imperfections in this document, or if you have suggestions for improving this document, please, contact us and contribute your share to the development of Open CASCADE Technology: bugmaster@opencascade.com

http://www.opencascade.com/contact/

Table of Contents

TABLE OF CONTENTS **2**

1. INTRODUCTION **5**

1. 1 THE MODELING ALGORITHMS MODULE 5
1. 2 THE TOPOLOGY API 5
1. 2. 1 Error Handling in the Topology API *8*

2. GEOMETRIC TOOLS **10**

2. 1 OVERVIEW 10
2. 2 INTERSECTIONS 10
2. 2. 1 Geom2dAPI_InterCurveCurve *11*
2. 2. 2 Intersection of Curves and Surfaces *12*
2. 2. 3 Intersection of two Surfaces *13*
2. 3 INTERPOLATIONS 13
2. 3. 1 Geom2dAPI_Interpolate *13*
2. 3. 2 GeomAPI_Interpolate *14*
2. 4 LINES AND CIRCLES FROM CONSTRAINTS 15
2. 5 SERVICES PROVIDED 16
2. 6 TYPES OF ALGORITHMS 17
2. 7 PERFORMANCE FACTORS 18
2. 8 CONVENTIONS 19
2. 8. 1 Exterior/Interior *19*
2. 8. 2 Orientation of a Line *20*
2. 9 EXAMPLES 20
2. 9. 1 Line tangent to two circles *20*
2. 9. 2 Circle of given radius tangent to two circles *24*
2. 10 THE ALGORITHMS 27
2. 10. 1 The Qualifiers *27*
2. 10. 2 General Remarks about the Algorithms *27*
2. 10. 3 The Analytic Algorithms *28*
2. 10. 4 The Geometric Algorithms *28*
2. 10. 5 The Iterative Algorithms *29*
2. 11 CURVES AND SURFACES FROM CONSTRAINTS 29
2. 11. 1 FairCurve *30*

2. 11. 2 GeomFill *31*
2. 11. 3 GeomPlate *32*
2. 12 PROJECTIONS 36
2. 12. 1 Projection of a Point onto a Curve *36*
2. 12. 2 Geom2dAPI_ProjectPointOnCurve *37*
2. 12. 3 Redefined operators *38*
2. 12. 4 Access to lower-level functionalities *39*
2. 12. 5 GeomAPI_ProjectPointOnCurve *39*
2. 12. 6 Projection of a Point on a Surface *41*
2. 12. 7 Access to lower-level functionalities *44*
2. 12. 8 Switching from 2d and 3d Curves *44*
3. TOPOLOGICAL TOOLS 46
3. 1 OVERVIEW 46
3. 2 STANDARD TOPOLOGICAL OBJECTS 47
3. 2. 1 BRepBuilderAPI_MakeShape *47*
3. 2. 2 BRepBuilderAPI_ModifyShape *47*
3. 2. 3 Making Vertices, Edges and Faces *47*
3. 2. 4 Making Wires and Shells *61*
3. 2. 5 Modification Operators *63*
4. CONSTRUCTION OF PRIMITIVES 65
4. 1 MAKING PRIMITIVES 65
4. 1. 1 BRepPrimAPI_MakeBox *65*
4. 1. 2 BRepPrimAPI_MakeWedge *66*
4. 1. 3 BRepPrimAPI_MakeOneAxis *67*
4. 1. 4 BRepPrimAPI_MakeCylinder *68*
4. 1. 5 BRepPrimAPI_MakeCone *69*
4. 1. 6 BRepPrimAPI_MakeSphere *70*
4. 1. 7 BRepPrimAPI_MakeTorus *72*
4. 1. 8 BRepPrimAPI_MakeRevolution *74*
4. 2 SWEEPING: PRISM, REVOLUTION AND PIPE 74
4. 2. 2 BRepPrimAPI_MakeSweep *75*
4. 2. 3 BRepPrimAPI_MakePrism *75*
4. 2. 4 BRepPrimAPI_MakeRevol *77*
5. BOOLEAN OPERATIONS 79
5. 1 BOOLEAN OPERATORS 79
5. 1. 1 BRepAlgoAPI_BooleanOperation *80*

5. 1. 2 BRepAlgoAPI_Fuse *80*
5. 1. 3 BRepAlgoAPI_Common *81*
5. 1. 4 BRepAlgoAPI_Cut *81*
5. 1. 5 BRepAlgoAPI_Section *81*
6. FILLETS AND CHAMFERS 83
6. 1 FILLET CONSTRUCTOR 83
6. 1. 1 BRepFilletAPI_MakeFillet *83*
6. 2 BRepFilletAPI_MakeFillet2d *86*
6. 1 CHAMFER CONSTRUCTOR 88
6 . 3 BRepFilletAPI_MakeChamfer *88*
7. OFFSETS, DRAFTS, PIPES AND EVOLVED SHAPES 89
7. 1 SHELLING OPERATOR 89
7. 2 MODIFICATION OPERATORS 90
7. 3 PIPE CONSTRUCTOR 92
7. 4 CONSTRUCTOR OF EVOLVED SOLID 93
8. SEWING OPERATORS 95
8. 1 BRepBuilderAPI_Sewing *95*
8. 2 BRepOffsetAPI_FindContiguousEdges *96*
9. FEATURES 97
9. 1 THE BREPFEAT CLASSES AND THEIR USE 97
9. 1. 1 Form classes *97*
9. 1. 2 The Gluer class *109*
9. 1. 3 The SplitShape Class *111*
10. HIDDEN LINE REMOVAL 112
10. 1 OVERVIEW 112
10. 2 THE SERVICES PROVIDED 115
10. 2. 1 HLRBRep *115*
9. 2. 2 Restrictions in use *117*
10. 3 EXAMPLES OF USE 117
10. 3. 1 HLRBRep_Algo *117*
10. 3. 2 HLRBRep_PolyAlgo *118*
10. MESHING OF SHAPES 120

1. Introduction

1. 1 The Modeling Algorithms Module

This manual explains how to use the Modeling Algorithms. It provides basic documentation on modeling algorithms. For advanced information on Modeling Algorithms, see our offerings on our web site at www.opencascade.org/support/training/

The Modeling Algorithms module brings together a wide range of topological algorithms used in modeling. Along with these tools, you will find the geometric algorithms, which they call.

The algorithms available are divided into:

- Geometric tools
- Topological tools
- The Topology API

1. 2 The Topology API

The Topology API of Open CASCADE Technology (**OCCT**) includes the following six packages:

- BRepAlgoAPI
- BRepBuilderAPI
- BRepFilletAPI
- BRepFeat
- BRepOffsetAPI
- BRepPrimAPI

The classes in these six packages provide the user with a simple and powerful interface.

- A simple interface: a function call works ideally,
- A powerful interface: including error handling and access to extra information provided by the algorithms.

As an example, the class `BRepBuilderAPI_MakeEdge` can be used to create a linear edge from two points.

```
gp_Pnt P1(10,0,0), P2(20,0,0);
TopoDS_Edge E = BRepBuilderAPI_MakeEdge(P1,P2);
```

This is the simplest way to create edge E from two points P1, P2, but the developer can test for errors when he is not as confident of the data as in the previous example.

Example

```
#include <gp_Pnt.hxx>
#include <TopoDS_Edge.hxx>
#include <BRepBuilderAPI_MakeEdge.hxx>
void EdgeTest()
{
  gp_Pnt P1;
  gp_Pnt P2;
  BRepBuilderAPI_MakeEdge ME(P1,P2);
  if (!ME.IsDone())
  {
  // doing ME.Edge() or E = ME here
  // would raise StdFail_NotDone
  Standard_DomainError::Raise
  (“ProcessPoints::Failed to createan edge”);
  }
  TopoDS_Edge E = ME;
}
```

In this example an intermediary object ME has been introduced. This can be tested for the completion of the function before accessing the result. More information on **error handling** in the topology programming interface can be found in the next section.

`BRepBuilderAPI_MakeEdge` provides valuable information. For example, when creating an edge from two points, two vertices have to be created from the points. Sometimes you may be interested in getting these vertices quickly without exploring the new edge. Such information can be provided when using a class. The following example shows a function creating an edge and two vertices from two points.

Example

```
void MakeEdgeAndVertices(const gp_Pnt& P1,
   const gp_Pnt& P2,
   TopoDS_Edge& E,
   TopoDS_Vertex& V1,
   TopoDS_Vertex& V2)
{
   BRepBuilderAPI_MakeEdge ME(P1,P2);
   if (!ME.IsDone()) {
    Standard_DomainError::Raise
    (“MakeEdgeAndVerices::Failed to create an edge”);
    }
   E = ME;
   V1 = ME.Vextex1();
   V2 = ME.Vertex2();
}
```

The `BRepBuilderAPI_MakeEdge` class provides the two methods Vertex1 and Vertex2, which return the two vertices used to create the edge.

How can `BRepBuilderAPI_MakeEdge` be both a function and a class? It can do this because it uses the casting capabilities of C++. The `BRepBuilderAPI_MakeEdge` class has a method called Edge; in the previous example the line E = ME could have been written.

Example

```
E = ME.Edge();
```

This instruction tells the C++ compiler that there is an **implicit casting** of a `BRepBuilderAPI_MakeEdge` into a `TopoDS_Edge` using the Edge method. It means this method is automatically called when a `BRepBuilderAPI_MakeEdge` is found where a `TopoDS_Edge` is required.

This feature allows you to provide classes, which have the simplicity of function calls when required and the power of classes when advanced processing is necessary. All the benefits of this approach are explained when describing the topology programming interface classes.

1. 2. 1 Error Handling in the Topology API

A method can report an error in the two following situations:

- The data or arguments of the method are incorrect, i.e. they do not respect the restrictions specified by the methods in its specifications. Typical example: creating a linear edge from two identical points is likely to lead to a zero divide when computing the direction of the line.
- Something unexpected happened. This situation covers every error not included in the first category. Including: interruption, programming errors in the method or in another method called by the first method, bad specifications of the arguments (i.e. a set of arguments that was not expected to fail).

The second situation is supposed to become increasingly exceptional as a system is debugged and it is handled by the **exception mechanism**. Using exceptions avoids handling error statuses in the call to a method: a very cumbersome style of programming.

In the first situation, an exception is also supposed to be raised because the calling method should have verified the arguments and if it did not do so, there is a bug. For example if before calling MakeEdge you are not sure that the two points are non-identical, this situation must be tested.

Making those validity checks on the arguments can be tedious to program and frustrating as you have probably correctly surmised that the method will perform the test twice. It does not trust you.

As the test involves a great deal of computation, performing it twice is also time-consuming.

Consequently, you might be tempted to adopt the *highly inadvisable* style of programming illustrated in the following example:

Example

```
#include <Standard_ErrorHandler.hxx>
try {
  TopoDS_Edge E = BRepBuilderAPI_MakeEdge(P1,P2);
  // go on with the edge
  }
catch {
  // process the error.
  }
```

To help the user, the Topology API classes only raise the exception **StdFail_NotDone**. Any other exception means that something happened which was unforeseen in the design of this API.

The **NotDone** exception is only raised when the user tries to access the result of the computation and the original data is corrupted. At the construction of the class instance, if the algorithm cannot be completed, the internal flag NotDone is set. This flag can be tested and in some situations a more complete description of the error can be queried. If the user ignores the NotDone status and tries to access the result, an exception is raised.

For example, with the BRepBuilderAPI_MakeEdge class:

Example

```
BRepBuilderAPI_MakeEdge ME(P1,P2);
if (!ME.IsDone()) {
// doing ME.Edge() or E = ME here
// would raise StdFail_NotDone
Standard_DomainError::Raise
(“ProcessPoints::Failed to create an edge”);
}
TopoDS_Edge E = ME;
```

---

2. Geometric Tools

2. 1 Overview

Open CASCADE Technology geometric tools include:

- Computation of intersections
- Interpolation laws
- Computation of curves and surfaces from constraints
- Computation of lines and circles from constraints
- Projections

2. 2 Intersections

The *Geom2dAPI_InterCurveCurve* class allows the evaluation of the intersection points (*gp_Pnt2d*) between two geometric curves (*Geom2d_Curve*) and the evaluation of the points of self-intersection of a curve.

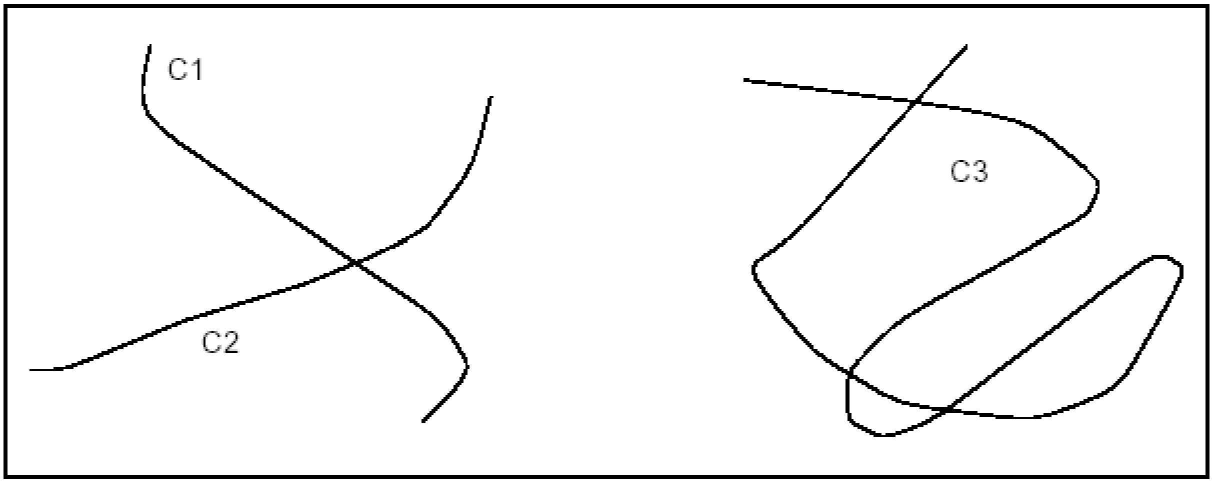


Figure 1. Intersection and self-intersection of curves

In both cases, the algorithm requires a value for the tolerance (Standard_Real) for the confusion between two points. The default tolerance value used in all constructors is *1.0e-6.*

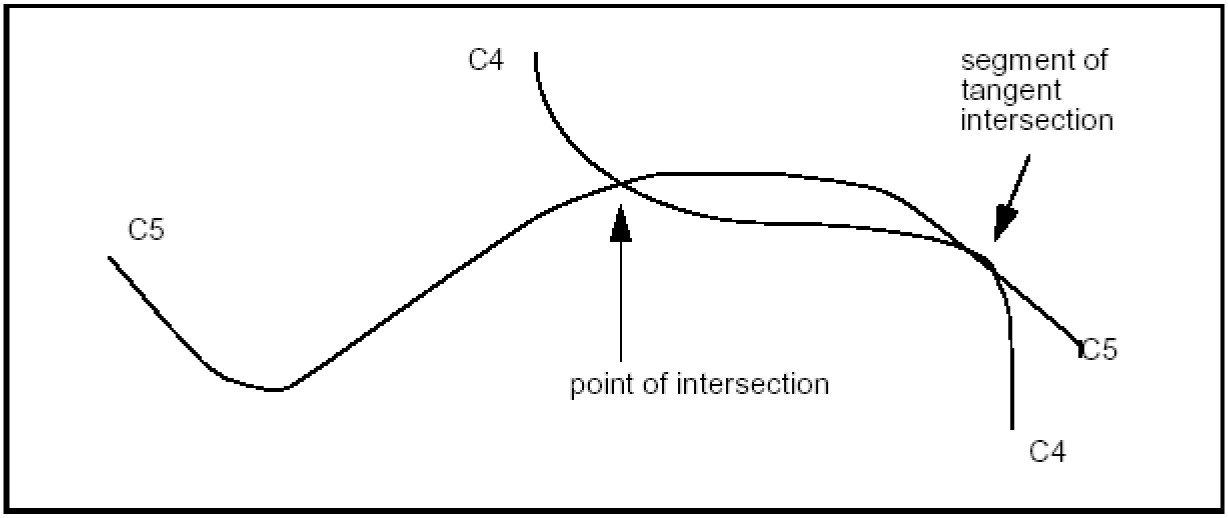


Figure 2. Intersection and tangent intersection

The algorithm returns a point in the case of an intersection and a segment in the case of tangent intersection.

2. 2. 1 Geom2dAPI_InterCurveCurve

This class may be instantiated in either of the following two ways:

Intersection of curves C1 and C2.

```
Geom2dAPI_InterCurveCurve Intersector(C1,C2,tolerance);
```

Self-intersection of curve C3.

```
Geom2dAPI_InterCurveCurve Intersector(C3,tolerance);
```

Calling the number of intersection points

```
Standard_Integer N = Intersector.NbPoints();
```

Calling an intersection point

To select the desired point, pass an integer index value in argument.

```
gp_Pnt2d P = Intersector.Point(Index);
```

Calling the number of intersection segments

```
Standard_Integer M = Intersector.NbSegments();
```

Calling an intersection segment

To select the desired segment pass integer index values in argument.

```
Handle(Geom2d_Curve) Seg1, Seg2;
Intersector.Segment(Index,Seg1,Seg2);
// if intersection of 2 curves
Intersector.Segment(Index,Seg1);
// if self-intersection of a curve
```

Access to lower-level functionalities

If you need access to a wider range of functionalities the following method will return the algorithmic object for the calculation of intersections:

```
Geom2dInt_GInter& TheIntersector = Intersector.Intersector();
```

2. 2. 2 Intersection of Curves and Surfaces

The *GeomAPI_IntCS* class is used to compute the intersection points between a curve and a surface.

This class is instantiated as follows:

```
GeomAPI_IntCS Intersector(C, S);
```

Calling the number of intersection points

```
Standard_Integer nb = Intersector.NbPoints();
```

Calling the intersection points

```
gp_Pnt& P = Intersector.Point(Index);
```

where *Index* is an integer between *1* and *nb*.

2. 2. 3 Intersection of two Surfaces

The *GeomAPI_IntSS* class is used to compute the intersection of two surfaces from *Geom_Surface* with respect to a given tolerance.

This class is instantiated as follows:

```
GeomAPI_IntSS Intersector(S1, S2, Tolerance);
```

Once the *GeomAPI_IntSS* object has been created, it can be interpreted.

Calling the number of intersection curves

```
Standard_Integer nb = Intersector. NbLines();
```

Calling the intersection curves

```
Handle(Geom_Curve) C = Intersector.Line(Index)
```

where *Index* is an integer between *1* and *nb*.

2. 3 Interpolations

Interpolation provides functionalities for interpolating BSpline curves, whether in 2D, using *Geom2dAPI_Interpolate*, or 3D using *GeomAPI_Interpolate*.

2. 3. 1 Geom2dAPI_Interpolate

This class is used to interpolate a BSplineCurve passing through an array of points. If tangency is not requested at the point of interpolation, continuity will be *C2* . If tangency is requested at the point, continuity will be *C1*. If Periodicity is requested, the curve will be closed and the junction will be the first point given. The curve will then have a continuity of *C1* only.

This class may be instantiated as follows:

```
Geom2dAPI_Interpolate
   (const Handle_TColgp_HArray1OfPnt2d& Points,
   const Standard_Boolean PeriodicFlag,
   const Standard_Real Tolerance);

Geom2dAPI_Interpolate Interp(Points, Standard_False,
                                     Precision::Confusion());
```

Calling the BSpline curve

From the object defined above the BSpline curve may be requested.

```
Handle(Geom2d_BSplineCurve) C = Interp.Curve();
```

Note that the *Handle(Geom2d_BSplineCurve)* operator has been redefined by the method Curve(). Consequently, it is unnecessary to pass via the construction of an intermediate object of the *Geom2dAPI_Interpolate* type and the following syntax is correct.

```
Handle(Geom2d_BSplineCurve) C =
      Geom2dAPI_Interpolate(Points,
                            Standard_False,
                            Precision::Confusion());
```

2. 3. 2 GeomAPI_Interpolate

This class may be instantiated as follows:

```
GeomAPI_Interpolate
      (const Handle_TColgp_HArray1OfPnt& Points,
      const Standard_Boolean PeriodicFlag,
      const Standard_Real Tolerance);

GeomAPI_Interpolate Interp(Points, Standard_False,
                                     Precision::Confusion());
```

Calling the BSpline curve

From the object defined above the BSpline curve may be requested.

```
Handle(Geom_BSplineCurve) C = Interp.Curve();
```

Note that the *Handle(Geom_BSplineCurve)* operator has been redefined by the method Curve(). Thus, it is unnecessary to pass via the construction of an intermediate object of the GeomAPI_Interpolate type and the following syntax is correct.

```
Handle(Geom_BSplineCurve)  C  =  GeomAPI_Interpolate(Points,
Standard_False, 1.0e-7);
```

Boundary conditions may be imposed with the method Load.

```
GeomAPI_Interpolate AnInterpolator
(Points, Standard_False, 1.0e-5);
AnInterpolator.Load (StartingTangent, EndingTangent);
```

2. 4 Lines and Circles from Constraints

There are two packages of importance for the end-user - *Geom2dGcc* and *GccAna*. *Geom2dGcc* deals with reference-handled geometric objects from the *Geom2d* package while *GccAna* deals with value-handled geometric objects from the *gp* package.

The *Geom2dGcc* package solves geometric constructions of lines and circles expressed by constraints such as tangency or parallelism, that is, a constraint expressed in geometric terms. As a simple example the following figure shows a line which is constrained to pass through a point and be tangent to a circle.

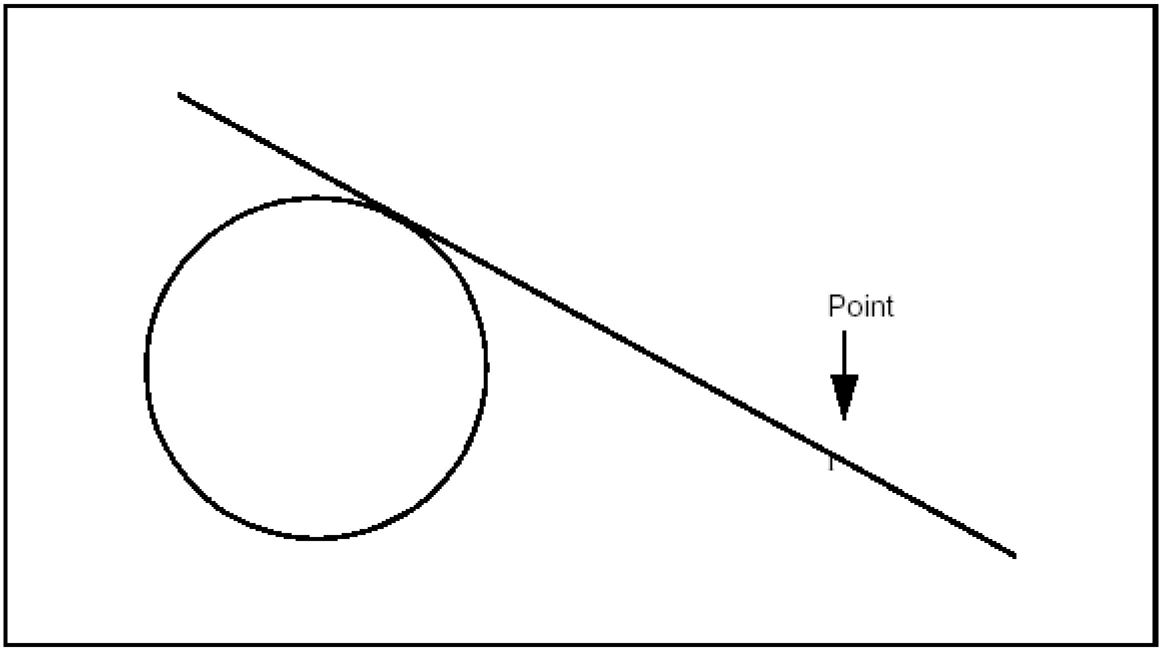


Figure 3. A constrained line

The *Geom2dGcc* package focuses on algorithms; it is useful for finding results, but it does not offer any management or modification functions, which could be applied to the constraints or their arguments. This package is designed to offer optimum performance, both in rapidity and precision. Trivial cases (for example, a circle centered on one point and passing through another) are not treated.

The *Geom2dGcc* package deals only with 2d objects from the *Geom2d* package. These objects are the points, lines and circles available.

All other lines such as Bezier curves and conic sections - with the exception of circles -are considered general curves and must be differentiable twice.

The *GccAna* package deals with points, lines, and circles from the *gp* package. Apart from constructors for lines and circles, it also allows the creation of conics from the bisection of other geometric objects.

2. 5 Services provided

Provides an implementation of analytic algorithms using value-handled entities only which are used to create 2D lines or circles with geometric constraints. The algorithms available are:

- circle tangent to three elements (lines, circles, curves, points),

- circle tangent to two elements and having a radius,
- circle tangent to two elements and centered on a third element,
- circle tangent to two elements and centered on a point,
- circle tangent to one element and centered on a second,
- bisector of two points,
- bisector of two lines,
- bisector of two circles,
- bisector of a line and a point,
- bisector of a circle and a point,
- bisector of a line and a circle,
- line tangent to two elements (points, circles, curves),
- line tangent to one element and parallel to a line,
- line tangent to one element and perpendicular to a line,
- line tangent to one element and forming angle with a line.

2. 6 Types of algorithms

There are three categories of available algorithms, which complement each other:

- analytic,
- geometric,
- iterative.

An analytic algorithm will solve a system of equations, whereas a geometric algorithm works with notions of parallelism, tangency, intersection and so on.

Both methods can provide solutions. An iterative algorithm, however, seeks to refine an approximate solution.

2. 7 Performance factors

The appropriate algorithm is the one, which reaches a solution of the required accuracy in the least time. Only the solutions actually requested by the user should be calculated. A simple means to reduce the number of solutions is the notion of "qualifier". There are four qualifiers, which are:

- Unqualified: the position of the solution is undefined with respect to this argument.
- Enclosing: the solution encompasses this argument.
- Enclosed: the solution is encompassed by this argument.
- Outside: the solution and argument are outside each other.

2. 8 Conventions

2. 8. 1 Exterior/Interior

It is not hard to define the interior and exterior of a circle. As is shown in the following diagram, the exterior is indicated by the sense of the binormal, that is to say the right side according to the sense of traversing the circle. The left side is therefore the interior (or "material").

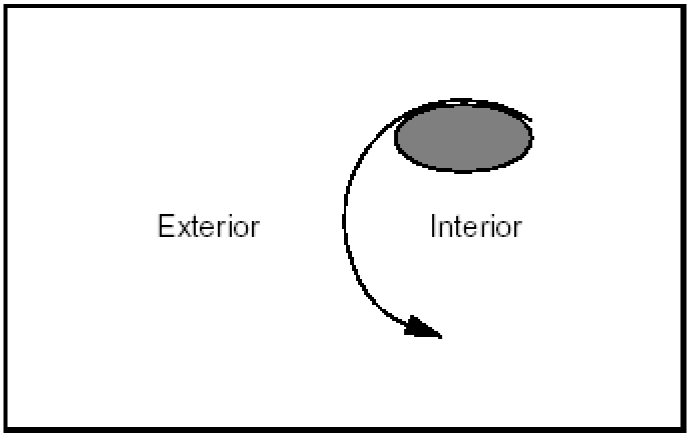


Figure 4. Exterior/Interior of a Circle

By extension, the interior of a line or any open curve is defined as the left side according to the passing direction, as shown in the following diagram:

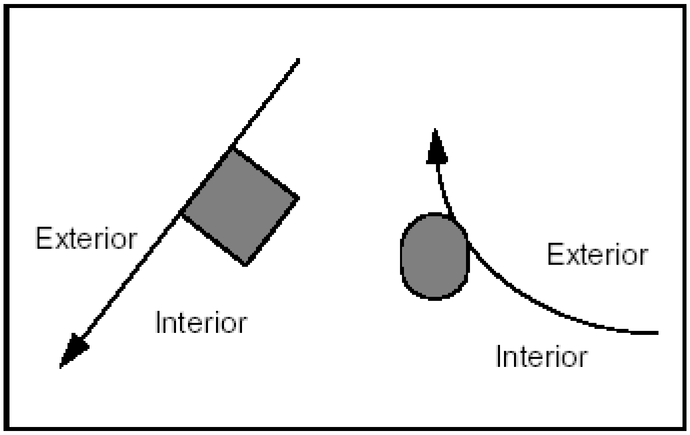


Figure 5. Exterior/Interior of a Line and a Curve

2. 8. 2 Orientation of a Line

It is sometimes necessary to define in advance the sense of travel along a line to be created. This sense will be from first to second argument.

The following figure shows a line, which is first tangent to circle C1 which is interior to the line, and then passes through point P1.

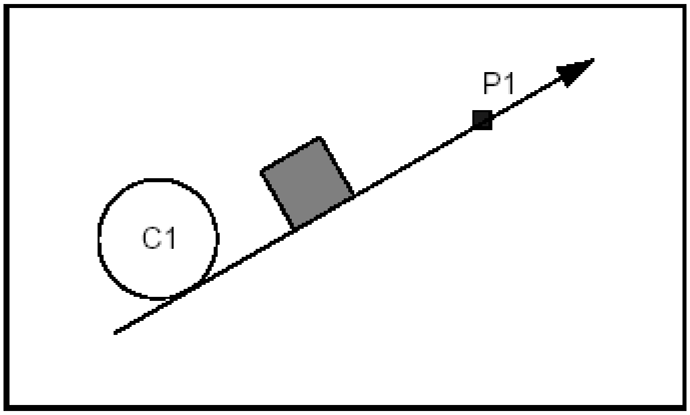


Figure 6. An Oriented Line

2. 9 Examples

2. 9. 1 Line tangent to two circles

The following four diagrams illustrate four cases of using qualifiers in the creation of a line. The fifth shows the solution if no qualifiers are given.

Note that the qualifier "Outside" is used to mean "Mutually exterior".

Example 1 Case 1

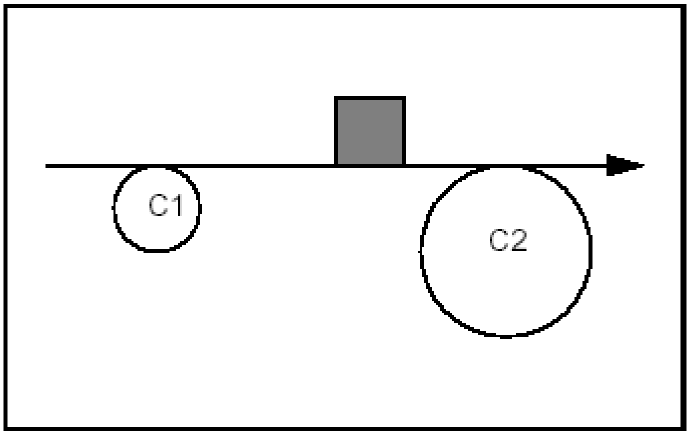


Figure 7. Both circles outside

Constraints:

Tangent and Exterior to C1.

Tangent and Exterior to C2.

Syntax:

```
GccAna_Lin2d2Tan
      Solver(GccEnt::Outside(C1),
            GccEnt::Outside(C2),
            Tolerance);
```

Example 1 Case 2

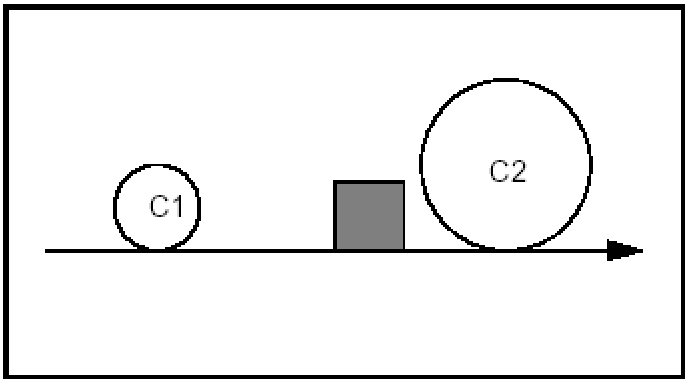


Figure 8. Both circles enclosed

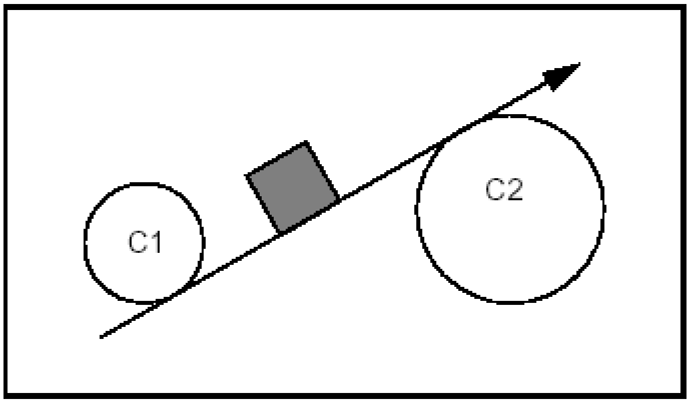
Constraints:

Tangent and Including C1.

Tangent and Including C2.

Syntax:

```
GccAna_Lin2d2Tan
      Solver(GccEnt::Enclosing(C1),
            GccEnt::Enclosing(C2),
            Tolerance);
```



Example 1 Case 3

Figure 9. C1 enclosed, C2 outside

Constraints:

Tangent and Including C1.

Tangent and Exterior to C2.

Syntax:

```
GccAna_Lin2d2Tan
      Solver(GccEnt::Enclosing(C1),
            GccEnt::Outside(C2),
            Tolerance);
```

Example 1 Case 4

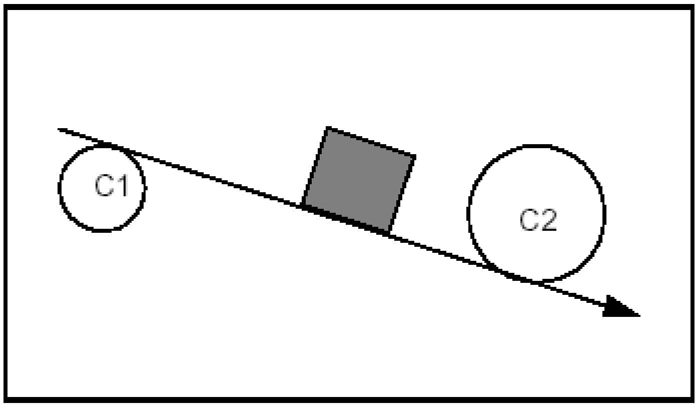


Figure 10. C1 outside, C2 enclosed

Constraints:

Tangent and Exterior to C1.

Tangent and Including C2.

Syntax:

```
GccAna_Lin2d2Tan
      Solver(GccEnt::Outside(C1),
            GccEnt::Enclosing(C2),
            Tolerance);
```

Example 1 Case 5

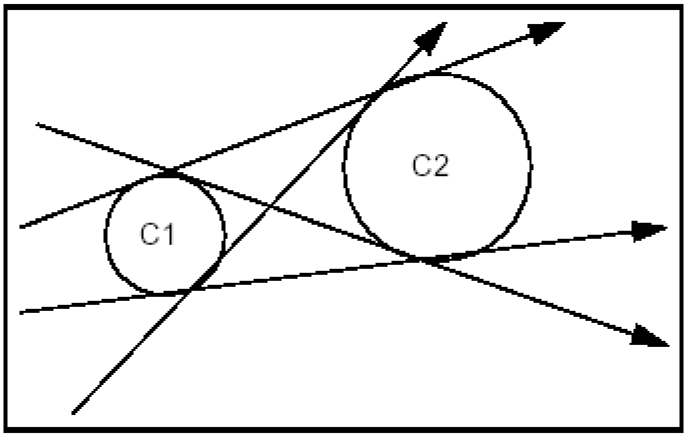


Figure 11. With no qualifiers specified

Constraints:

Tangent and Undefined with respect to C1.

Tangent and Undefined with respect to C2.

Syntax:

```
GccAna_Lin2d2Tan
      Solver(GccEnt::Unqualified(C1),
            GccEnt::Unqualified(C2),
            Tolerance);
```

2. 9. 2 Circle of given radius tangent to two circles

The following four diagrams show the four cases in using qualifiers in the creation of a circle.

Example 2 Case 1

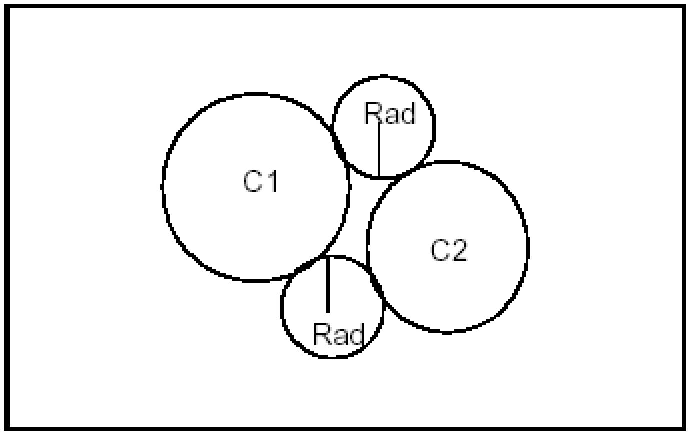


Figure 12. Both solutions outside

Constraints:

Tangent and Exterior to C1.

Tangent and Exterior to C2.

Syntax:

```
GccAna_Circ2d2TanRad
      Solver(GccEnt::Outside(C1),
            GccEnt::Outside(C2), Rad, Tolerance);
```

Example 2 Case 2

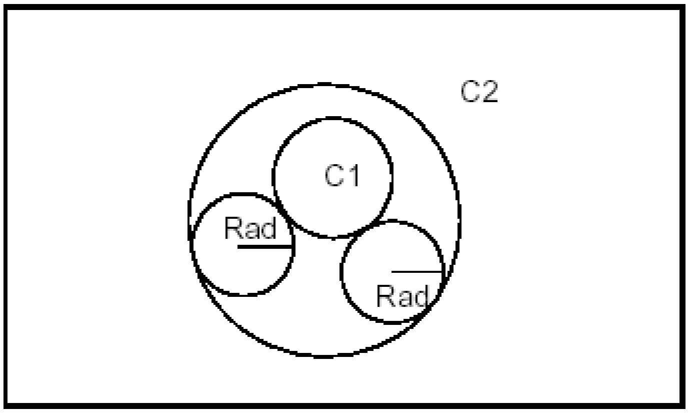


Figure 13. C2 encompasses C1.

Constraints:

Tangent and Exterior to C1.

Tangent and Included by C2.

Syntax:

```
GccAna_Circ2d2TanRad
      Solver(GccEnt::Outside(C1),
            GccEnt::Enclosed(C2), Rad, Tolerance);
```

Example 2 Case 3

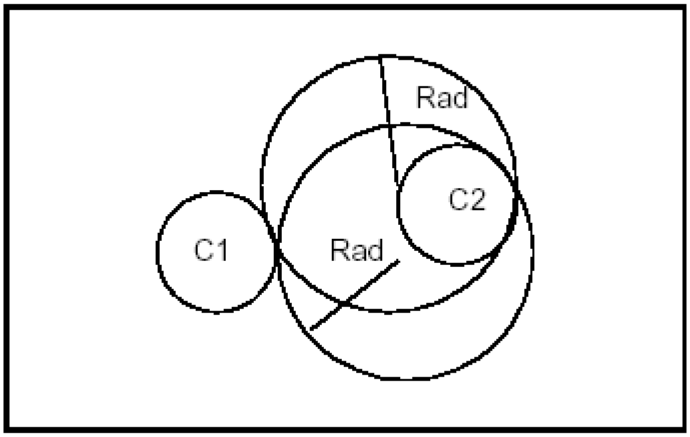


Figure 14. Solutions enclose C2

Constraints:

Tangent and Exterior to C1.

Tangent and Including C2.

Syntax:

```
GccAna_Circ2d2TanRad
       Solver(GccEnt::Outside(C1),
              GccEnt::Enclosing(C2), Rad, Tolerance);
```

Example 2 Case 4

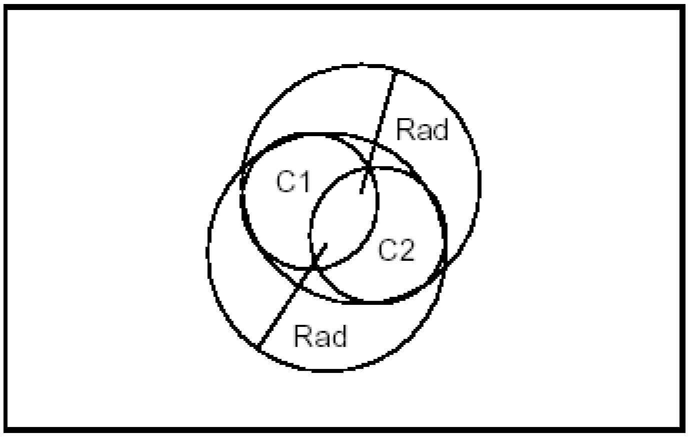


Figure 15. Solutions enclose C1 & C2

Constraints:

Tangent and Enclosing C1.

Tangent and Enclosing C2.

Syntax:

```
GccAna_Circ2d2TanRad
       Solver(GccEnt::Enclosing(C1),
              GccEnt::Enclosing(C2), Rad, Tolerance);
```

Example 2 Case5

The following syntax will give all the circles of radius *Rad,* which are tangent to *C1* and *C2* without discrimination of relative position:

```
GccAna_Circ2d2TanRad Solver(GccEnt::Unqualified(C1),
                            GccEnt::Unqualified(C2),
                            Rad,Tolerance);
```

2. 10 The Algorithms

The objects created by this toolkit are non-persistent.

2. 10. 1 The Qualifiers

The *GccEnt* package contains the following package methods:

- Unqualified,
- Enclosing,
- Enclosed,
- Outside.

This enables the creation of expressions as for example in Figure 6_12.

```
GccAna_Circ2d2TanRad
      Solver(GccEnt::Outside(C1),
            GccEnt::Enclosing(C2), Rad, Tolerance);
```

This can be expressed as "Find all the circles of radius Rad, which are tangent to both circle C1 and C2, C1 being outside and C2 being inside."

2. 10. 2 General Remarks about the Algorithms

We consider the following to be the case:

- If a circle passes through a point then the circle is tangential to it.
- A distinction is made between the trivial case of being centered on a point and the complex case of being centered on a line.

2. 10. 3 The Analytic Algorithms

The GccAna package implements the analytic algorithms. It deals only with points, lines, and circles from the gp package. Here is a list of the services offered:

Creation of a Line:

Tangent (point **|** circle) & Parallel (line)

Tangent (point **|** circle) & Perpendicular (line **|** circle)

Tangent (point **|** circle) & Oblique (line)

Tangent (2 { point **|** circle })

Bisector(line **|** line)

Creation of Conics:

Bisector (point **|** point)

Bisector (line **|** point)

Bisector (circle **|** point)

Bisector (line **|** line)

Bisector (circle **|** line)

Bisector (circle **|** circle)

Creation of a Circle:

Tangent (point **|** line **|** circle) & Center (point)

Tangent (3 { point **|** line **|** circle })

Tangent (2 { point **|** line **|** circle }) & Radius (real)

Tangent (2 { point **|** line **|** circle }) & Center (line **|** circle)

Tangent (point **|** line **|** circle) & Center (line **|** circle) & Radius (real)

For each algorithm, the desired tolerance (and angular tolerance if appropriate) is given as an argument. Calculation is done to the highest precision available from the hardware.

2. 10. 4 The Geometric Algorithms

The *Geom2dGcc* package offers algorithms, which produce 2d lines or circles with geometric constraints. For arguments, it takes curves for which an approximate solution is not requested. A tolerance value on the result is given as a starting parameter. Here is a list of the services offered:

Creation of a Circle:

Tangent (curve) & Center (point)

Tangent (curve , point **|** line **|** circle **|** curve) & Radius (real)

Tangent (2 {point **|** line **|** circle}) & Center (curve)

Tangent (curve) & Center (line **|** circle **|** curve) & Radius (real)

Tangent (point **|** line **|** circle) & Center (curve) & Radius (real)

All calculations will be done to the highest precision available from the hardware.

2. 10. 5 The Iterative Algorithms

The Geom2dGcc package offers iterative algorithms find a solution by refining an approximate solution. It produces 2d lines or circles with geometric constraints. For all geometric arguments except points, an approximate solution may be given as a starting parameter. The tolerance or angular tolerance value is given as an argument. The following is a list of the services offered:

Creation of a Line:

Tangent (curve) & Oblique (line)

Tangent (curve , { point **|** circle **|** curve })

Creation of a Circle:

Tangent (curve , 2 { point **|** circle **|** curve })

Tangent (curve , { point **|** circle **|** curve })

& Center (line **|** circle **|** curve)

2. 11 Curves and Surfaces from Constraints

The *GeomFill* and *GeomPlate* packages provide tools for creating surfaces either from boundary curves or respecting curve and point constraints.

The *FairCurve* package provides a set of classes to create faired 2D curves or 2D curves with minimal variation in curvature.

2. 11. 1 FairCurve

The FairCurve package provides the following services:

Creation of Batten Curves

The class Batten allows you to produce faired curves defined on the basis of one or more constraints on each of the two reference points. These include point, angle of tangency and curvature settings.

The following constraint orders are available:

- 0 the curve must pass through a point
- 1 the curve must pass through a point and have a given tangent
- 2 the curve must pass through a point, have a given tangent and a given curvature.

Only constraint orders of 0 and 1are used.

The function Curve returns the result as a 2D BSpline curve.

Creation of Minimal Variation Curves

The class MinimalVariation allows you to produce curves with minimal variation in curvature at each reference point.

The following constraint orders are available:

- 0 the curve must pass through a point
- 1 the curve must pass through a point and have a given tangent
- 2 the curve must pass through a point, have a given tangent and a given curvature.

Constraint orders of 0, 1 and 2 can be used. The algorithm minimizes tension, sagging and jerk energy.

The function *Curve* returns the result as a 2D BSpline curve.

Specifying the length of the curve

If you want to give a specific length to a batten curve, use:

```
b.SetSlidingFactor(L / b.SlidingOfReference())
```

where b is the name of the batten curve object

Aesthetic Considerations

Free sliding is generally more aesthetically pleasing than constrained sliding.

However, the computation can fail with values such as angles greater than p/2, because in this case, the length is theoretically infinite.

Warning

In other cases, when sliding is imposed and the sliding factor is too large, the batten can collapse.

Controlling Computation Time

The constructor parameters, Tolerance and NbIterations, control how precise the computation is, and how long it will take.

2. 11. 2 GeomFill

The *GeomFill* package provides the following services for creating surfaces from boundary curves:

Creation of Bezier surfaces

The class *BezierCurves* allows you to produce a Bezier surface from contiguous Bezier curves. Note that problems may occur with rational Bezier Curves.

Creation of BSpline surfaces

The class *BSplineCurves* allows you to produce a BSpline surface from contiguous BSpline curves. Note that problems may occur with rational BSplines.

Creation of a Pipe

The class *Pipe* allows you to produce a pipe by sweeping a curve (the section) along another curve (the path). The result is a BSpline surface.

Filling a contour

The class *GeomFill_ConstrainedFilling* allows you to fill a contour defined by two, three or four curves as well as by tangency constraints. The resulting surface is a BSpline.

Creation of a Boundary

The class *GeomFill_SimpleBound* allows you to define a boundary for the surface, which you want to construct.

Creation of a Boundary with an adjoining surface

The class *GeomFill_BoundWithSurf* allows you to define a boundary for the surface, which you want to construct. This boundary will already be joined to another surface.

Filling styles

The enumerations *FillingStyle* specify the styles used to build the surface. These include:

- *Stretch* - the style with the flattest patches
- *Coons* - a rounded style with less depth than *Curved*
- *Curved* - the style with the most rounded patches.

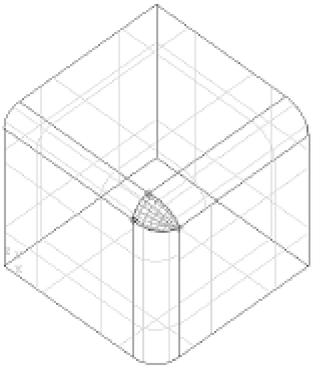


Figure 16. Intersecting filleted edges with differing radii, presenting a gap which has been filled by a surface.

2. 11. 3 GeomPlate

The *GeomPlate* package provides the following services for creating surfaces respecting curve and point constraints:

Definition of a Framework

The class *BuildPlateSurface* allows you to create a framework to build surfaces according to curve and point constraints as well as tolerance settings. The result is returned with the function *Surface*.

Note that you do not have to specify an initial surface at the time of construction.

You can add one later or, if none is loaded, one will automatically be computed.

Definition of a Curve Constraint

The class *CurveConstraint* allows you to define curves as constraints to the surface, which you want to build.

Definition of a Point Constraint

The class *PointConstraint* allows you to define points as constraints to the surface, which you want to build.

Applying Geom_Surface to Plate Surfaces

The class *Surface* allows you to describe the characteristics of plate surface objects returned by **BuildPlateSurface::Surface** using the methods of *Geom_Surface*

Approximating a Plate surface to a BSpline

The class *MakeApprox* allows you to convert a *GeomPlate* surface into a *Geom_BSplineSurface*.

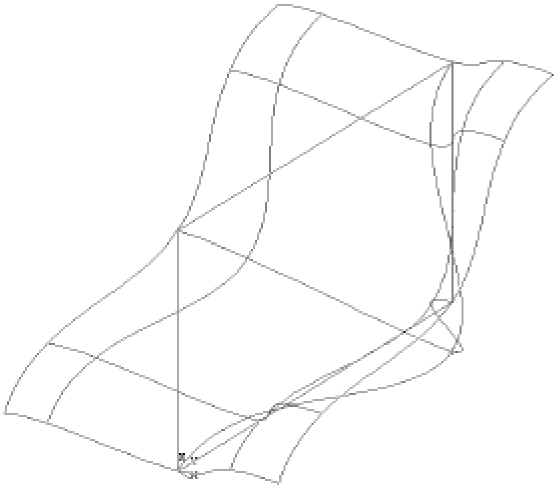


Figure 17. A surface generated from four curves and a point.

Example

Create a Plate surface and approximate it from a polyline as a curve constraint and a point constraint

```
Standard_Integer NbCurFront=4,
NbPointConstraint=1;
gp_Pnt P1(0.,0.,0.);
gp_Pnt P2(0.,10.,0.);
gp_Pnt P3(0.,10.,10.);
gp_Pnt P4(0.,0.,10.);
gp_Pnt P5(5.,5.,5.);
BRepBuilderAPI_MakePolygon W;
W.Add(P1);
W.Add(P2);
W.Add(P3);
W.Add(P4);
W.Add(P1);
// Initialize a BuildPlateSurface
GeomPlate_BuildPlateSurface BPSurf(3,15,2);
```

```
// Create the curve constraints
BRepTools_WireExplorer anExp;
for(anExp.Init(W); anExp.More(); anExp.Next())
{
TopoDS_Edge E = anExp.Current();
Handle(BRepAdaptor_HCurve) C = new
BRepAdaptor_HCurve();
C->ChangeCurve().Initialize(E);
Handle(BRepFill_CurveConstraint) Cont= new
BRepFill_CurveConstraint(C,0);
BPSurf.Add(Cont);
}
// Point constraint
Handle(GeomPlate_PointConstraint) PCont= new
GeomPlate_PointConstraint(P5,0);
BPSurf.Add(PCont);
// Compute the Plate surface
BPSurf.Perform();
// Approximation of the Plate surface
Standard_Integer MaxSeg=9;
Standard_Integer MaxDegree=8;
Standard_Integer CritOrder=0;
Standard_Real dmax,Tol;
Handle(GeomPlate_Surface) PSurf = BPSurf.Surface();
dmax = Max(0.0001,10*BPSurf.G0Error());
Tol=0.0001;
GeomPlate_MakeApprox
Mapp(PSurf,Tol,MaxSeg,MaxDegree,dmax,CritOrder);
Handle (Geom_Surface) Surf (Mapp.Surface());
// create a face corresponding to the approximated Plate
Surface
Standard_Real Umin, Umax, Vmin, Vmax;
PSurf->Bounds( Umin, Umax, Vmin, Vmax);
BRepBuilderAPI_MakeFace MF(Surf,Umin, Umax, Vmin, Vmax);
```

2. 12 Projections

This package provides functionality for projecting points onto 2D and 3D curves and surfaces.

2. 12. 1 Projection of a Point onto a Curve

The *Geom2dAPI_ProjectPointOnCurve* class allows calculation of all the normals projected from a point (*gp_Pnt2d*) onto a geometric curve (*Geom2d_Curve*). The calculation may be restricted to a given domain.

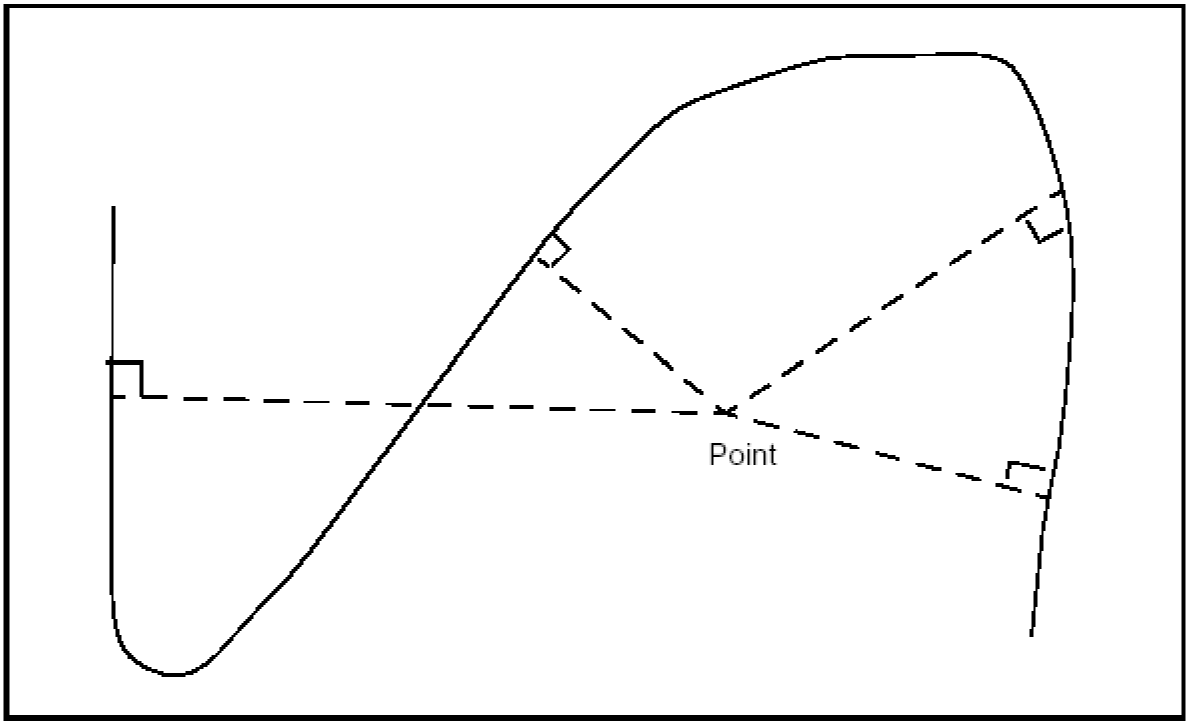


Figure 18. Normals from a point to a curve

NOTE

Note that the curve does not have to be a Geom2d_TrimmedCurve. The algorithm will function with any class inheriting Geom2d_Curve.

2. 12. 2 Geom2dAPI_ProjectPointOnCurve

This class may be instantiated as in the following example:

```
gp_Pnt2d P;
Handle(Geom2d_BezierCurve) C =
      new Geom2d_BezierCurve(args);
Geom2dAPI_ProjectPointOnCurve Projector (P, C);
```

To restrict the search for normals to a given domain *[U1,U2]*, use the following constructor:

```
Geom2dAPI_ProjectPointOnCurve Projector (P, C, U1, U2);
```

Having thus created the *Geom2dAPI_ProjectPointOnCurve* object, we can now interrogate it.

Calling the number of solution points

```
Standard_Integer NumSolutions = Projector.NbPoints();
```

Calling the location of a solution point

The solutions are indexed in a range from *1* to *Projector.NbPoints()*. The point, which corresponds to a given index Index may be found:

```
gp_Pnt2d Pn = Projector.Point(Index);
```

Calling the parameter of a solution point

For a given point corresponding to a given index Index:

```
Standard_Real U = Projector.Parameter(Index);
```

This can also be programmed as:

```
Standard_Real U;
Projector.Parameter(Index,U);
```

Calling the distance between the starting point and another

We can find the distance between the initial point and a point, which corresponds to the given index, *Index*:

```
Standard_Real D = Projector.Distance(Index);
```

Calling the nearest solution point

This class offers a method to return the closest solution point to the starting point. This solution is accessed as follows:

```
gp_Pnt2d P1 = Projector.NearestPoint();
```

Calling the parameter of the nearest solution point

```
Standard_Real U = Projector.LowerDistanceParameter();
```

Calling the minimum distance from the point to the curve

```
Standard_Real D = Projector.LowerDistance();
```

2. 12. 3 Redefined operators

Some operators have been redefined to help you find the closest solution.

Standard_Real() Returns the minimum distance from the point to the curve.

```
Standard_Real D = Geom2dAPI_ProjectPointOnCurve (P,C);
```

Standard_Integer() Returns the number of solutions.

```
Standard_Integer N =
Geom2dAPI_ProjectPointOnCurve (P,C);
```

gp_Pnt2d() Returns the nearest solution point.

```
gp_Pnt2d P1 = Geom2dAPI_ProjectPointOnCurve (P,C);
```

Using these operators makes coding easier when you only need the nearest point. Thus:

```
Geom2dAPI_ProjectPointOnCurve Projector (P, C);
gp_Pnt2d P1 = Projector.NearestPoint();
```

can be written more concisely as:

```
gp_Pnt2d P1 = Geom2dAPI_ProjectPointOnCurve (P,C);
```

However, note that in this second case no intermediate

Geom2dAPI_ProjectPointOnCurve object is created, and thus it is impossible to have access to other solution points.

2. 12. 4 Access to lower-level functionalities

If you want to use the wider range of functionalities available from the Extrema package, a call to the Extrema() method will return the algorithmic object for calculating extrema. For example:

```
Extrema_ExtPC2d& TheExtrema = Projector.Extrema();
```

2. 12. 5 GeomAPI_ProjectPointOnCurve

This class is instantiated as in the following example:

```
gp_Pnt P;
Handle(Geom_BezierCurve) C =
   new Geom_BezierCurve(args);
GeomAPI_ProjectPointOnCurve Projector (P, C);
```

If you wish to restrict the search for normals to the given domain [U1,U2], use the following constructor:

```
GeomAPI_ProjectPointOnCurve Projector (P, C, U1, U2);
```

Having thus created the GeomAPI_ProjectPointOnCurve object, you can now interrogate it.

Calling the number of solution points

```
Standard_Integer NumSolutions = Projector.NbPoints();
```

Calling the location of a solution point

The solutions are indexed in a range from 1 to Projector.NbPoints(). The point, which corresponds to a given index may be found:

```
gp_Pnt Pn = Projector.Point(Index);
```

Calling the parameter of a solution point

For a given point corresponding to a given index:

```
Standard_Real U = Projector.Parameter(Index);
```

This can also be programmed as:

```
Standard_Real U;
Projector.Parameter(Index,U);
```

Calling the distance between the starting point and another

The distance between the initial point and a point, which corresponds to a given index, may be found:

```
Standard_Real D = Projector.Distance(Index);
```

Calling the nearest solution point

This class offers a method to return the closest solution point to the starting point. This solution is accessed as follows:

```
gp_Pnt P1 = Projector.NearestPoint();
```

Calling the parameter of the nearest solution point

```
Standard_Real U = Projector.LowerDistanceParameter();
```

Calling the minimum distance from the point to the curve

Standard_Real D = Projector.LowerDistance();

Redefined operators

Some operators have been redefined to help you find the nearest solution.

Standard_Real() Returns the minimum distance from the point to the curve.

```
Standard_Real D = GeomAPI_ProjectPointOnCurve (P,C);
```

Standard_Integer() Returns the number of solutions.

Standard_Integer N = GeomAPI_ProjectPointOnCurve (P,C);

gp_Pnt2d() Returns the nearest solution point.

```
gp_Pnt P1 = GeomAPI_ProjectPointOnCurve (P,C);
```

Using these operators makes coding easier when you only need the nearest point. In this way,

```
GeomAPI_ProjectPointOnCurve Projector (P, C);
gp_Pnt P1 = Projector.NearestPoint();
```

can be written more concisely as:

```
gp_Pnt P1 = GeomAPI_ProjectPointOnCurve (P,C);
```

In the second case, however, no intermediate GeomAPI_ProjectPointOnCurve object is created, and it is impossible to access other solutions points.

Access to lower-level functionalities

If you want to use the wider range of functionalities available from the Extrema package, a call to the Extrema() method will return the algorithmic object for calculating the extrema. For example:

```
Extrema_ExtPC& TheExtrema = Projector.Extrema();
```

2. 12. 6 Projection of a Point on a Surface

The GeomAPI_ProjectPointOnSurf class allows calculation of all the normals projected from a point from gp_Pnt onto a geometric surface from Geom_Surface.

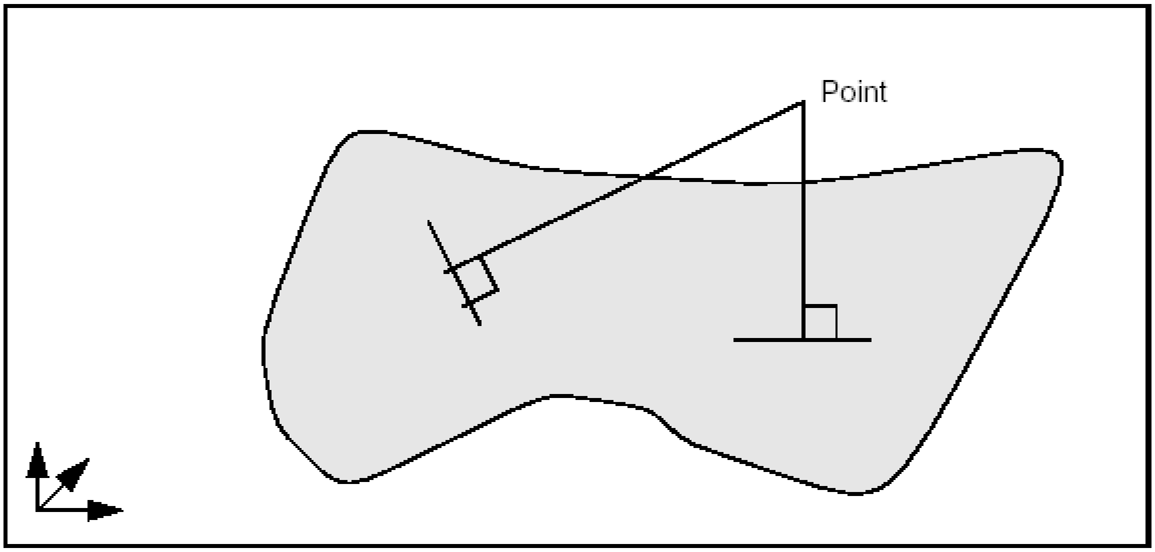


Figure 19. Normals from a point to a surface

NOTE

Note that the surface does not have to be of the Geom_RectangularTrimmedSurface type.

The algorithm will function with any class inheriting Geom_Surface.

GeomAPI_ProjectPointOnSurf

This class is instantiated as in the following example:

```
gp_Pnt P;
Handle (Geom_Surface) S = new Geom_BezierSurface(args);
GeomAPI_ProjectPointOnSurf Proj (P, S);
```

To restrict the search for normals within the given rectangular domain [U1, U2, V1, V2], use the following constructor:

```
GeomAPI_ProjectPointOnSurf Proj (P, S, U1, U2, V1, V2);
```

The values of U1, U2, V1 and V2 lie at or within their maximum and minimum limits, i.e.:

Umin<= U1<U2<=Umax

Vmin<= V1<V2<=Vmax

Having thus created the GeomAPI_ProjectPointOnSurf object, you can interrogate it.

Calling the number of solution points

```
Standard_Integer NumSolutions = Proj.NbPoints();
```

Calling the location of a solution point

The solutions are indexed in a range from 1 to Proj.NbPoints(). The point corresponding to the given index may be found:

```
gp_Pnt Pn = Proj.Point(Index);
```

Calling the parameters of a solution point

For a given point corresponding to the given index:

```
Standard_Real U,V;
Proj.Parameters(Index, U, V);
```

Calling the distance between the starting point and another

The distance between the initial point and a point corresponding to the given index may be found:

```
Standard_Real D = Projector.Distance(Index);
```

Calling the nearest solution point

This class offers a method, which returns the closest solution point to the starting point. This solution is accessed as follows:

```
gp_Pnt P1 = Proj.NearestPoint();
```

Calling the parameters of the nearest solution point

```
Standard_Real U,V;
Proj.LowerDistanceParameters (U, V);
```

Calling the minimum distance from the point to the surface

```
Standard_Real D = Proj.LowerDistance();
```

Redefined operators

Some operators have been redefined to help you find the nearest solution.

Standard_Real() Returns the minimum distance from the point to the surface.

```
Standard_Real D = GeomAPI_ProjectPointOnSurf (P,S);
```

Standard_Integer() Returns the number of solutions.

```
Standard_Integer N = GeomAPI_ProjectPointOnSurf (P,S);
```

gp_Pnt2d() Returns the nearest solution point.

```
gp_Pnt P1 = GeomAPI_ProjectPointOnSurf (P,S);
```

Using these operators makes coding easier when you only need the nearest point. In this way,

```
GeomAPI_ProjectPointOnSurface Proj (P, S);
gp_Pnt P1 = Proj.NearestPoint();
```

can be written more concisely as:

```
gp_Pnt P1 = GeomAPI_ProjectPointOnSurface (P,S);
```

In the second case, however, no intermediate *GeomAPI_ProjectPointOnSurf* object is created, and it is impossible to access other solution points.

2. 12. 7 Access to lower-level functionalities

If you want to use the wider range of functionalities available from the Extrema package, a call to the Extrema() method will return the algorithmic object for calculating the extrema as follows:

```
Extrema_ExtPS& TheExtrema = Proj.Extrema();
```

2. 12. 8 Switching from 2d and 3d Curves

The To2d and To3d package methods are used to;

- build a 2d curve from a 3d Geom_Curve lying on a gp_Pln plane
- build a 3d curve from a Geom2d_Curve and a gp_Pln plane.

These methods are called as follows:

```
Handle(Geom2d_Curve) C2d = GeomAPI::To2d(C3d, Pln);
Handle(Geom_Curve) C3d = GeomAPI::To3d(C2d, Pln);
```

3. Topological Tools

3. 1 Overview

Open CASCADE Technology topological tools include:

- Standard topological objects combining topological data structure and boundary representation
- Geometric Transformations
- Conversion to NURBS geometry
- Finding Planes
- Duplicating Shapes
- Checking Validity

The standard topological objects include

- Vertices
- Edges
- Wires
- Faces
- Shells
- Solids.

3. 2 Standard Topological Objects

3. 2. 1 BRepBuilderAPI_MakeShape

The deferred class BRepBuilderAPI_MakeShape is the root of all the classes of BRepBuilderAPI, which build shapes. It inherits from the class BRepBuilderAPI_Command. It provides a field to store the constructed shape.

3. 2. 2 BRepBuilderAPI_ModifyShape

Class BRepBuilderAPI_ModifyShape is a deferred class used as a root for the shape modifications. It inherits BRepBuilderAPI_MakeShape and implements the methods used to trace the history of all sub-shapes.

3. 2. 3 Making Vertices, Edges and Faces

The following classes are used to create topology from geometry. They all have the default precision as tolerance.

BRepBuilderAPI_MakeVertex

Use this class to create a new vertex from a 3D point from gp.

Example

```
gp_Pnt P(0,0,10);
TopoDS_Vertex V = BRepBuilderAPI_MakeVertex(P);
```

NOTE

Note that this always creates a new vertex. This class has no other methods.

BRepBuilderAPI_MakeEdge

Use this class to create edges. An edge is created from a curve and vertices. The basic method is to construct an edge from a curve, two vertices, and two parameters. All other constructions are derived from this one. The basic method and its arguments are described first, followed by the other methods. The BRepBuilderAPI_MakeEdge class can provide extra information and return an error status.

Basic Edge construction

Example

```
Handle(Geom_Curve) C = ...; // a curve
TopoDS_Vertex V1 = ...,V2 = ...;// two Vertices
Standard_Real p1 = ..., p2 = ..;// two parameters
TopoDS_Edge E = BRepBuilderAPI_MakeEdge(C,V1,V2,p1,p2);
```

C is the domain of the edge. V1 is the first vertex, it is oriented FORWARD, V2 is the second vertex, it is oriented REVERSED. p1 and p2 are the parameters for the vertices V1 and V2 on the curve. The default tolerance is associated with this edge. The following figure illustrates this construction:

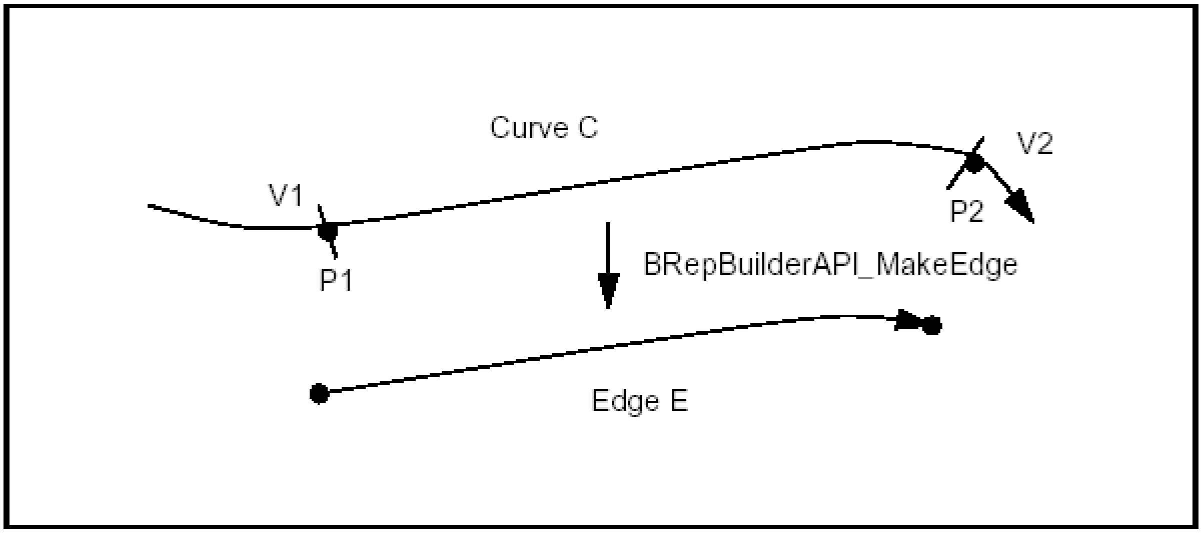


Figure 20. Basic Edge Construction

The following rules apply to the arguments:

The curve

- Must not be a Null Handle.
- If the curve is a trimmed curve, the basis curve is used.

The vertices

- Can be null shapes. When V1 or V2 is Null the edge is open in the corresponding direction and the corresponding parameter p1 or p2 must be infinite (i.e p1 is RealFirst(), p2 is RealLast()).
- Must be different vertices if they have different 3d locations and identical vertices if they have the same 3d location (identical vertices are used when the curve is closed).

The parameters

- Must be increasing and in the range of the curve.

  C->FirstParameter() <= p1 < p2 <= C->LastParameter()
- If the parameters are decreasing, the Vertices are switched, i.e. V2 becomes V1 and V1 becomes V2.
- On a periodic curve the parameters p1 and p2 are adjusted by adding or subtracting the period to obtain p1 in the range of the curve and p2 in the range p1 < p2 <= p1+ Period. So on a parametric curve p2 can be greater than the curve’s second parameter, see the figure below.
- Can be infinite but the corresponding vertex must be Null (see above).
- The distance between the Vertex 3d location and the point evaluated on the curve with the parameter must be lower than the default precision.

The figure below illustrates two special cases, a semi-infinite edge and an edge on a periodic curve.

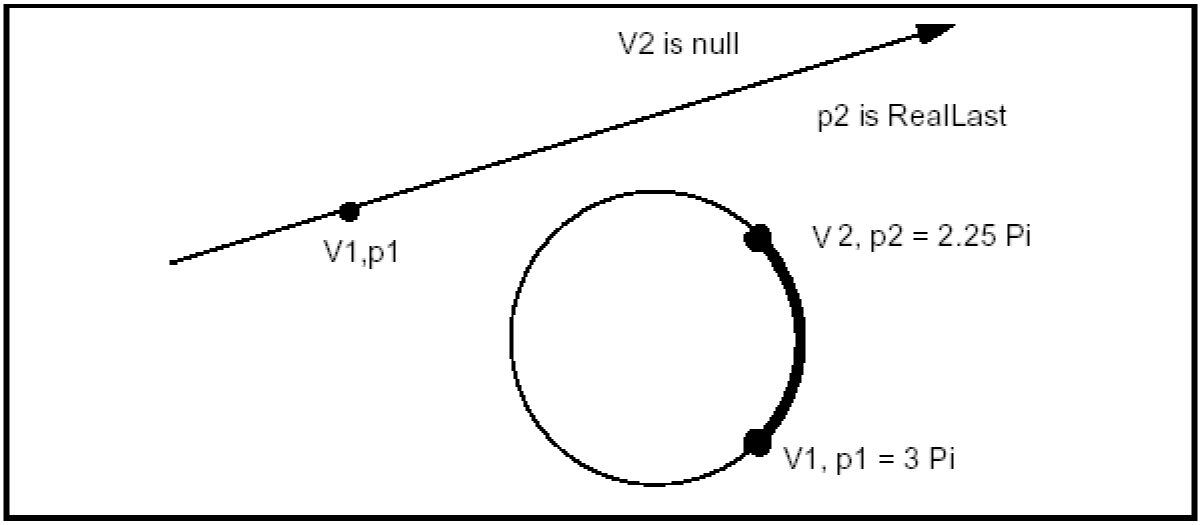


Figure 21. Infinite and Periodic Edges

Other Edge constructions

The BRepBuilderAPI_MakeEdge class provides methods, which are all simplified calls of the previous one:

- The parameters can be omitted. They are computed by projecting the vertices on the curve.
- 3d points (Pnt from gp) can be given in place of vertices. Vertices are created from the points. Giving vertices is useful when creating connected vertices.
- The vertices or points can be omitted if the parameters are given. The points are computed by evaluating the parameters on the curve.
- The vertices or points and the parameters can be omitted. The first and last parameters of the curve are used.

The five following methods are thus derived from the basic construction:

Example

```
Handle(Geom_Curve) C = ...; // a curve
TopoDS_Vertex V1 = ...,V2 = ...;// two Vertices
Standard_Real p1 = ..., p2 = ..;// two parameters
```

```
gp_Pnt P1 = ..., P2 = ...;// two points
TopoDS_Edge E;
// project the vertices on the curve
E = BRepBuilderAPI_MakeEdge(C,V1,V2);
// Make vertices from points
E = BRepBuilderAPI_MakeEdge(C,P1,P2,p1,p2);
// Make vertices from points and project them
E = BRepBuilderAPI_MakeEdge(C,P1,P2);
// Computes the points from the parameters
E = BRepBuilderAPI_MakeEdge(C,p1,p2);
// Make an edge from the whole curve
E = BRepBuilderAPI_MakeEdge(C);
```

Six methods (the five above and the basic method) are also provided for curves from the gp package in place of Curve from Geom. The methods create the corresponding Curve from Geom and are implemented for the following classes:

gp_Lin	creates a Geom_Line
gp_Circ	creates a Geom_Circle
gp_Elips	creates a Geom_Ellipse
gp_Hypr	creates a Geom_Hyperbola
gp_Parab	creates a Geom_Parabola

There are also two methods to construct edges from two vertices or two points. These methods assume that the curve is a line; the vertices or points must have different locations.

Example

```
TopoDS_Vertex V1 = ...,V2 = ...;// two Vertices
gp_Pnt P1 = ..., P2 = ...;// two points
TopoDS_Edge E;

// linear edge from two vertices
E = BRepBuilderAPI_MakeEdge(V1,V2);

// linear edge from two points
E = BRepBuilderAPI_MakeEdge(P1,P2);
```

Other information and error status

The BRepBuilderAPI MakeEdge when used as a class can provide the two vertices. This is useful when the vertices were not provided as arguments, for example when the edge was constructed from a curve and parameters. The two methods Vertex1 and Vertex2 return the vertices. Note that the returned vertices can be null if the edge is open in the corresponding direction.

The **Error** method returns a term of the BRepBuilderAPI_EdgeError enumeration. It can be used to analyze the error when the IsDone method returns False. The terms are:

- **EdgeDone**

No error occurred, IsDone returns True.

- **PointProjectionFailed**

No parameters were given but the projection of the 3D points on the curve failed. This happens when the point distance to the curve is greater than the precision.

- **ParameterOutOfRange**

The given parameters are not in the range C->FirstParameter(), C->LastParameter()

- **DifferentPointsOnClosedCurve**

The two vertices or points have different locations but they are the extremities of a closed curve.

- **PointWithInfiniteParameter**

A finite coordinate point was associated with an infinite parameter (see the Precision package for a definition of infinite values).

- **DifferentsPointAndParameter**

The distance of the 3D point and the point evaluated on the curve with the parameter is greater than the precision.

- **LineThroughIdenticPoints**

Two identical points were given to define a line (construction of an edge without curve), gp::Resolution is used for the confusion test.

The following example creates a wire from a set of parameters as described in the following figure.

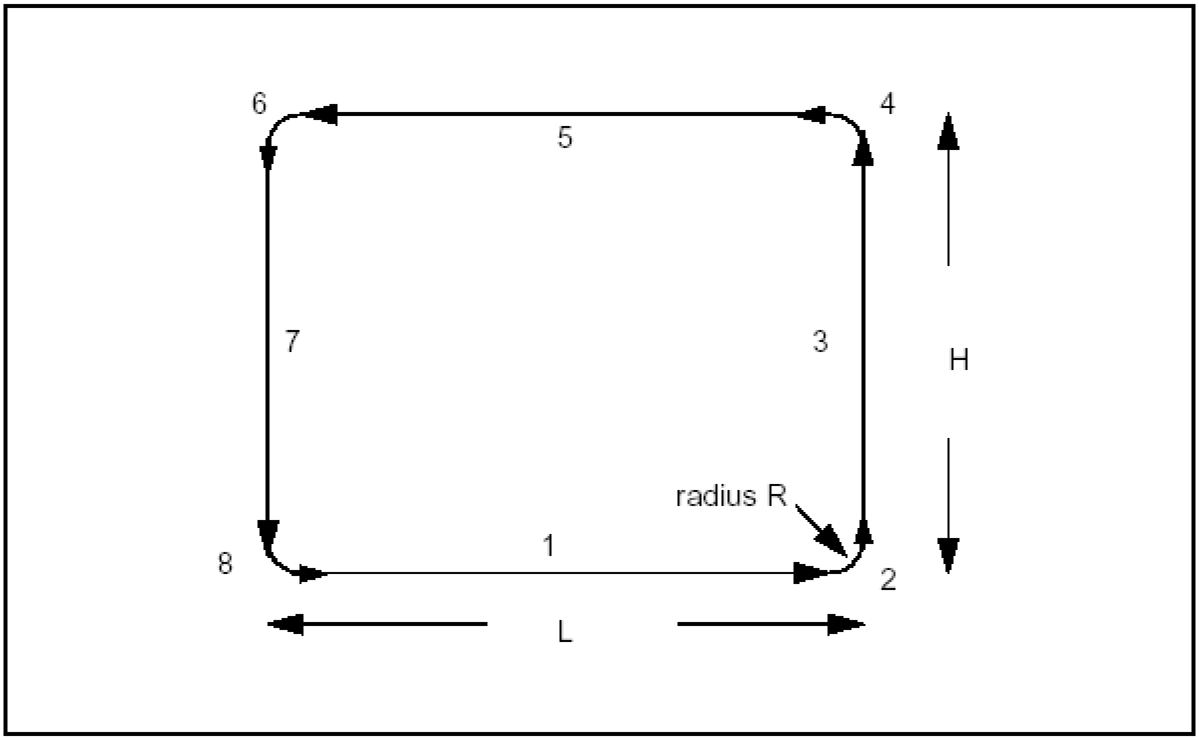


Figure 22. Creating a Wire

Example

```
// Make a rectangle centered on the origin
// of dimensions H, L with fillets of radius R.
// The edges and the vertices are stored in the arrays
// theEdges and theVertices
// We use the class Array1OfShape
// (i.e. not arrays of edges or vertices)
#include <BRepBuilderAPI_MakeEdge.hxx>
#include <TopoDS_Shape.hxx>
```

```
#include <gp_Circ.hxx>
#include <gp.hxx>
#include <TopoDS_Wire.hxx>
#include <TopTools_Array1OfShape.hxx>
#include <BRepBuilderAPI_MakeWire.hxx>

// The MakeArc method to make an edge and two vertices
void MakeArc(Standard_Real x,Standard_Real y,
        Standard_Real R,
        Standard_Real ang,
        TopoDS_Shape& E,
        TopoDS_Shape& V1,
        TopoDS_Shape& V2)
{
gp_Ax2 Origin = gp::XOY();
gp_Vec Offset(x, y, 0.);
Origin.Translate(Offset);
BRepBuilderAPI_MakeEdge
      ME(gp_Circ(Origin,R), ang, ang+PI/2);
E = ME;
V1 = ME.Vertex1();
V2 = ME.Vertex2();
}

TopoDS_Wire MakeFilletedRectangle(const Standard_Real H,
                                const Standard_Real L,
                                const Standard_Real R)
{
TopTools_Array1OfShape theEdges(1,8);
TopTools_Array1OfShape theVertices(1,8);

// First create the circular edges and the vertices
// using the MakeArc function described above.
void MakeArc(Standard_Real, Standard_Real,
Standard_Real, Standard_Real,
TopoDS_Shape&, TopoDS_Shape&, TopoDS_Shape&);

Standard_Real x = L/2 - R, y = H/2 - R;
MakeArc(x,-y,R,3.*PI/2.,theEdges(2),theVertices(2),
      theVertices(3));
MakeArc(x,y,R,0.,theEdges(4),theVertices(4),
      theVertices(5));
MakeArc(-x,y,R,PI/2.,theEdges(6),theVertices(6),
      theVertices(7));
MakeArc(-x,-y,R,PI,theEdges(8),theVertices(8),
      theVertices(1));
// Create the linear edges
for (Standard_Integer i = 1; i <= 7; i += 2)
{
theEdges(i) = BRepBuilderAPI_MakeEdge
(TopoDS::Vertex(theVertices(i)),TopoDS::Vertex
(theVertices(i+1)));
}
// Create the wire using the BRepBuilderAPI_MakeWire
BRepBuilderAPI_MakeWire MW;
for (i = 1; i <= 8; i++)
{
MW.Add(TopoDS::Edge(theEdges(i)));
}
return MW.Wire();
}
```

BRepBuilderAPI_MakeEdge2d

Use this class to make edges on a working plane from 2d curves. The working plane is a default value of the BRepBuilderAPI package (see the *Plane* methods).

The BRepBuilderAPI_MakeEdge2d class is strictly similar to the BRepBuilderAPI_MakeEdge class using 2D geometry from gp and Geom2d instead of 3D geometry.

BRepBuilderAPI_MakePolygon

Construction of polygons

The BRepBuilderAPI_MakePolygon class is used to build polygonal wires from vertices or points. Points are automatically changed to vertices as in BRepBuilderAPI_MakeEdge.

The basic usage of BRepBuilderAPI_MakePolygon is to create a wire by adding vertices or points using the Add method. At any moment, the current wire can be extracted. The close method can be used to close the current wire. In the following example, a closed wire is created from an array of points.

Example

```
#include <TopoDS_Wire.hxx>
#include <BRepBuilderAPI_MakePolygon.hxx>
#include <TColgp_Array1OfPnt.hxx>

TopoDS_Wire ClosedPolygon(const TColgp_Array1OfPnt& Points)
{
BRepBuilderAPI_MakePolygon MP;
for(Standard_Integer
i=Points.Lower();i<=Points.Upper();i++)
{
MP.Add(Points(i));
}
MP.Close();
return MP;
}
```

Short-cuts are provided for 2, 3, or 4 points or vertices. Those methods have a Boolean last argument to tell if the polygon is closed. The default value is False.

Two examples:

Example of a closed triangle from three vertices:

```
TopoDS_Wire                       W                          =
BRepBuilderAPI_MakePolygon(V1,V2,V3,Standard_True);
```

Example of an open polygon from four points:

```
TopoDS_Wire W = BRepBuilderAPI_MakePolygon(P1,P2,P3,P4);
```

Other information

The BRepBuilderAPI_MakePolygon class maintains a current wire. The current wire can be extracted at any moment and the construction can proceed to a longer wire.

After each point insertion, the class maintains the last created edge and vertex, which are returned by the methods Edge, FirstVertex and LastVertex.

When the added point or vertex has the same location as the previous one it is not added to the current wire but the most recently created edge becomes Null. The **Added** method can be used to test this condition. The MakePolygon class never raises an error. If no vertex has been added, the Wire is Null. If two vertices are at the same location, no edge is created.

BRepBuilderAPI_MakeFace

Use this class to create faces. A face is created from a surface and wires. An underlying surface is constructed from a surface and optional parametric values. Wires can be added to the surface. A planar surface can be constructed from a wire. An error status can be returned after face construction.

Basic Face construction

A face can be constructed from a surface and four parameters to determine a limitation of the UV space. The parameters are optional, if they are omitted the natural bounds of the surface are used. Up to four edges and vertices are created with a wire. No edge is created when the parameter is infinite.

Example

```
Handle(Geom_Surface) S = ...; // a surface
Standard_Real umin,umax,vmin,vmax; // parameters
TopoDS_Face                     F                     =
BRepBuilderAPI_MakeFace(S,umin,umax,vmin,vmax);
```

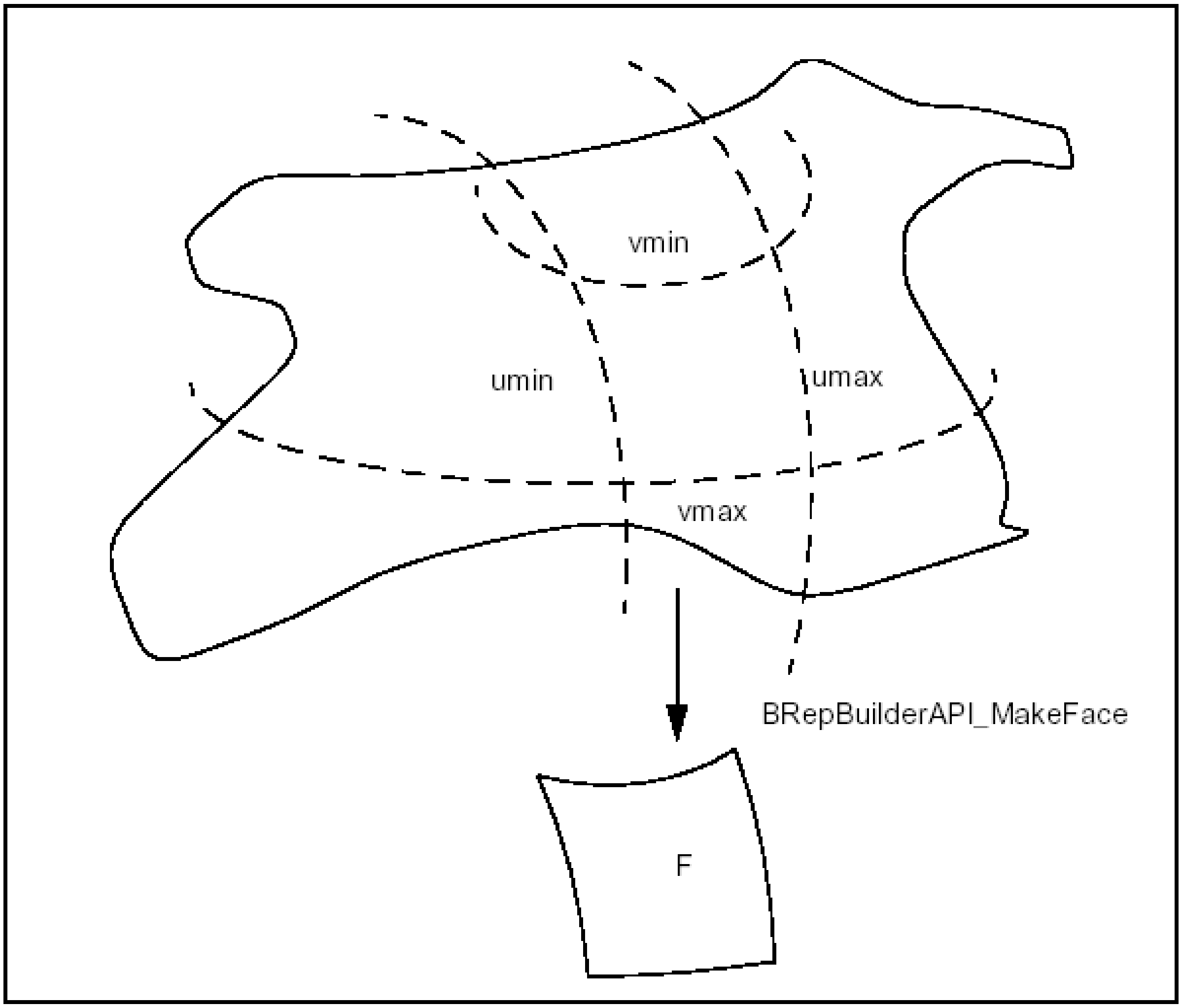

vmin
umin
umax
vmax
BRepBuilderAPI_MakeFace
F

Figure 23. Basic Face Construction

To make a face from the natural boundary of a surface, the parameters are not required:

Example

```
Handle(Geom_Surface) S = ...; // a surface
TopoDS_Face F = BRepBuilderAPI_MakeFace(S);
```

The constraint on the parameters as similar to the constraints in BRepBuilderAPI_MakeEdge.

- umin,umax (vmin,vmax) must be in the range of the surface and must be increasing.
- On a U (V) periodic surface umin and umax (vmin,vmax) are adjusted.
- umin, umax, vmin, vmax can be infinite. There will be no edge in the corresponding direction.

Other face constructions

The two basic constructions (from a surface and from a surface and parameters) are implemented for all the gp package surfaces, which are transformed in the corresponding Surface from Geom.

gp_Pln	creates a Geom_Plane
gp_Cylinder	creates a Geom_CylindricalSurface
gp_Cone	creates a Geom_ConicalSurface
gp_Sphere	creates a Geom_SphericalSurface
gp_Torus	creates a Geom_ToroidalSurface

Once a face has been created, a wire can be added using the Add method. For example, the following code creates a cylindrical surface and adds a wire.

Example

```
gp_Cylinder C = ..; // a cylinder
TopoDS_Wire W = ...;// a wire
BRepBuilderAPI_MakeFace MF(C);
MF.Add(W);
TopoDS_Face F = MF;
```

More than one wire can be added to a face, provided that they do not cross each other and they define only one area on the surface. (Note that this is not checked). The edges on a Face must have a parametric curve description.

If there is no parametric curve for an edge of the wire on the Face it is computed by projection.

For one wire, a simple syntax is provided to construct the face from the surface and the wire. The above lines could be written:

Example

```
TopoDS_Face F = BRepBuilderAPI_MakeFace(C,W);
```

A planar face can be created from only a wire, provided this wire defines a plane. For example, to create a planar face from a set of points you can use BRepBuilderAPI_MakePolygon and BRepBuilderAPI_MakeFace.

Example

```
#include <TopoDS_Face.hxx>
#include <TColgp_Array1OfPnt.hxx>
#include <BRepBuilderAPI_MakePolygon.hxx>
#include <BRepBuilderAPI_MakeFace.hxx>

TopoDS_Face     PolygonalFace(const     TColgp_Array1OfPnt&
thePnts)
{
BRepBuilderAPI_MakePolygon MP;
for(Standard_Integer i=thePnts.Lower();
i<=thePnts.Upper(); i++)
{
MP.Add(thePnts(i));
}
MP.Close();
TopoDS_Face F = BRepBuilderAPI_MakeFace(MP.Wire());
return F;
}

The last use of MakeFace is to copy an existing face to add
new wires. For example
the following code adds a new wire to a face.

TopoDS_Face F = ...; // a face
TopoDS_Wire W = ...; // a wire
F = BRepBuilderAPI_MakeFace(F,W);
```

To add more than one wire an instance of the BRepBuilderAPI_MakeFace class can be created with the face and the first wire and the new wires inserted with the Add method.

Error status

The Error method returns an error status, which is a term from the BRepBuilderAPI_FaceError enumeration.

FaceDone	No error occurred.
NoFace	No initialization of the algorithm; empty constructor was used.
NotPlanar	No surface was given and the wire was not planar.
CurveProjectionFailed	No curve was found in the parametric space of the surface for an edge.
ParametersOutOfRange	The parameters umin,umax,vmin,vmax are out of the surface.

3. 2. 4 Making Wires and Shells

Composite shapes are built not from geometry, but by the assembly of other shapes. Composite shapes are:

- The Wire made from edges.
- The Shell made from faces.
- The Solid made from shells.

BRepBuilderAPI_MakeWire

The BRepBuilderAPI_MakeWire class can build a wire from one or more edges or connect new edges to an existing wire.

Basic wire constructions

Up to four edges the class can be used directly, for example:

Example

```
TopoDS_Wire W = BRepBuilderAPI_MakeWire(E1,E2,E3,E4);
```

For a higher or unknown number of edges the Add method must be used; for example, to build a wire from an array of shapes (to be edges).

Example

```
TopTools_Array1OfShapes theEdges;
BRepBuilderAPI_MakeWire MW;
for (Standard_Integer i = theEdge.Lower();
i <= theEdges.Upper(); i++)
      MW.Add(TopoDS::Edge(theEdges(i));
TopoDS_Wire W = MW;
```

The class can be constructed with a wire. A wire can also be added. In this case, all the edges of the wires are added. For example to merge two wires:

Example

```
#include <TopoDS_Wire.hxx>
#include <BRepBuilderAPI_MakeWire.hxx>

TopoDS_Wire MergeWires (const TopoDS_Wire& W1,
            const TopoDS_Wire& W2)
{
BRepBuilderAPI_MakeWire MW(W1);
MW.Add(W2);
return MW;
}
```

Other information

The BRepBuilderAPI_MakeWire class connects the edges to the wire. When a new edge is added if one of its vertices is shared with the wire it is considered as connected to the wire. If there is no shared vertex, the algorithm searches for a vertex of the edge and a vertex of the wire, which are at the same location (the tolerances of the vertices are used to test if they have the same location). If such a pair of vertices is found, the edge is copied with the vertex of the wire in place of the original vertex. All the vertices of the edge can be exchanged for vertices from the wire. If no connection is found the wire is considered to be disconnected. This is an error.

The BRepBuilderAPI_MakeWire class can return the last edge added to the wire (Edge method). This edge can be different from the original edge if it was copied.

Error Status

The Error method returns a term of the BRepBuilderAPI_WireError enumeration:

WireDone	No error occurred.
EmptyWire	No initialization of the algorithm, empty constructor was used.
DisconnectedWire	The last added edge was not connected to the wire.
NonManifoldWire	The wire with some singularity.

BRepBuilderAPI_MakeShell

Use the MakeShell class to build a Shell from a set of Faces. What may be important is that each face should have the required continuity. That is why an initial surface is broken up into faces.

BRepBuilderAPI_MakeSolid

Use the MakeSolid class to build a Solid from a set of Shells. Its use is similar to the use of the MakeWire class: shells are added to the solid in the same way that edges are added to the wire in MakeWire.

3. 2. 5 Modification Operators

BRepBuilderAPI_Transform

The Transform from BRepBuilderAPI class is used to apply a transformation to a shape (see class Trsf from gp). The methods have a boolean argument to copy or share the original shape, as long as the transformation allows (it is only possible for direct isometric transformations). By default, the original shape is shared.

The following example deals with the rotation of shapes.

Example

```
TopoDS_Shape myShape1 = ...;
```

```
// The original shape 1
TopoDS_Shape myShape2 = ...;
// The original shape2
gp_Trsf T;
T.SetRotation(gp_Ax1(gp_Pnt(0.,0.,0.),gp_Vec(0.,0.,1.)),
            2.*PI/5.);
BRepBuilderAPI_Transformation theTrsf(T);
theTrsf.Perform(myShape1);
TopoDS_Shape myNewShape1 = theTrsf.Shape()
theTrsf.Perform(myShape2,Standard_True);
// Here duplication is forced
TopoDS_Shape myNewShape2 = theTrsf.Shape()
...
```

BRepBuilderAPI_Copy

Use the BRepBuilderAPI_Copy class to duplicate a shape. A new shape is thus created.

In the following code, a solid is copied:

Example

```
TopoDS Solid MySolid;
....// Creates a solid

TopoDS_Solid myCopy = BRepBuilderAPI_Copy(mySolid);
```

4. Construction of Primitives

4. 1 Making Primitives

The following classes are used to build primitive objects. They include boxes, wedges and rotational objects. They can be used to build solids or shells. These classes provide Shell and Solid methods to return the corresponding results.

The methods are overloaded to be cast automatically to TopoDS_Shell or TopoDS_Solid.

4. 1. 1 BRepPrimAPI_MakeBox

Use the MakeBox class to build a parallelepiped box. The result is either a Shell or a Solid. There are four ways to build a box:

From three dimensions dx,dy,dz. The box is parallel to the axes and extends for [0,dx] [0,dy] [0,dz]

From a point and three dimensions. The same as above but the point is the new origin.

From two points, the box is parallel to the axes and extends on the intervals defined by the coordinates of the two points.

From a system of axes (gp_Ax2) and three dimensions. Same as the first way but the box is parallel to the given system of axes.

An error is raised if the box is flat in any dimension using the default precision. The following code shows how to create a box:

Example

```
TopoDS_Solid theBox = BRepPrimAPI_MakeBox(10.,20.,30.);
```

The following figure illustrates the four methods to build a box.

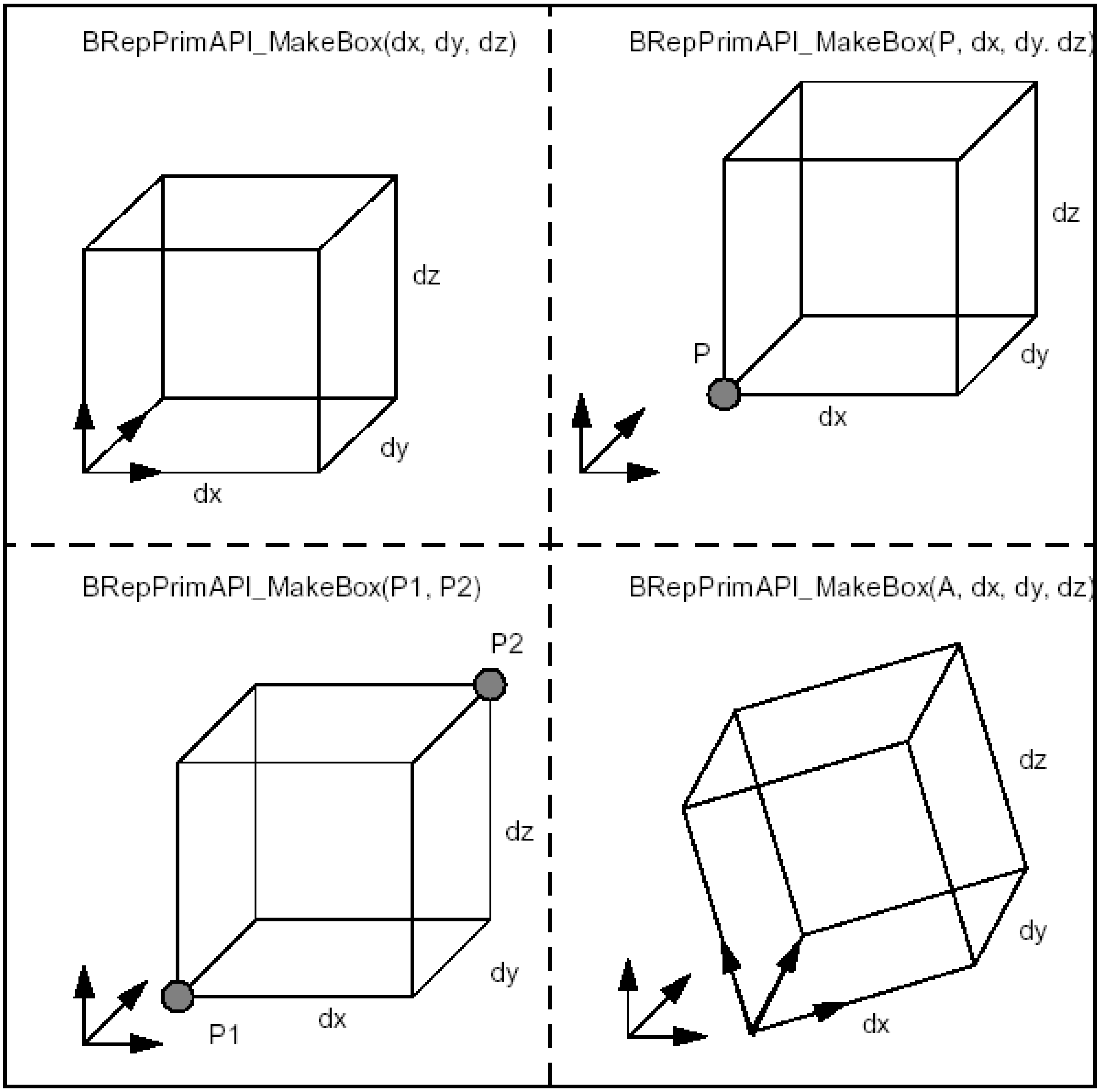

BRepPrimAPI_MakeBox(dx, dy, dz)
dz
dy
dx
BRepPrimAPI_MakeBox(P, dx, dy. dz)
dz
P
dy
dx
BRepPrimAPI_MakeBox(P1, P2)
P2
dz
dy
P1
dx
BRepPrimAPI_MakeBox(A, dx, dy, dz)
dz
dy
dx

Figure 24. Making Boxes

4. 1. 2 BRepPrimAPI_MakeWedge

Use the BRepPrimAPI_MakeWedge class to build a wedge. A wedge is a slanted box, i.e. a box with angles. The wedge is constructed in much the same way as a box i.e. from three dimensions dx,dy,dz plus arguments or from an axis system, three dimensions, and arguments.

The following figure shows two ways to build wedges. One is to add an ltx dimension, which is the length in x of the face at dy. The second is to add xmin, xmax, zmin, zmax to describe the face at dy.

The first method is a particular case of the second with:

xmin = 0, xmax = ltx, zmin = 0, zmax = dz

To make a centered pyramid you can use:

xmin = xmax = dx / 2, zmin = zmax = dz / 2

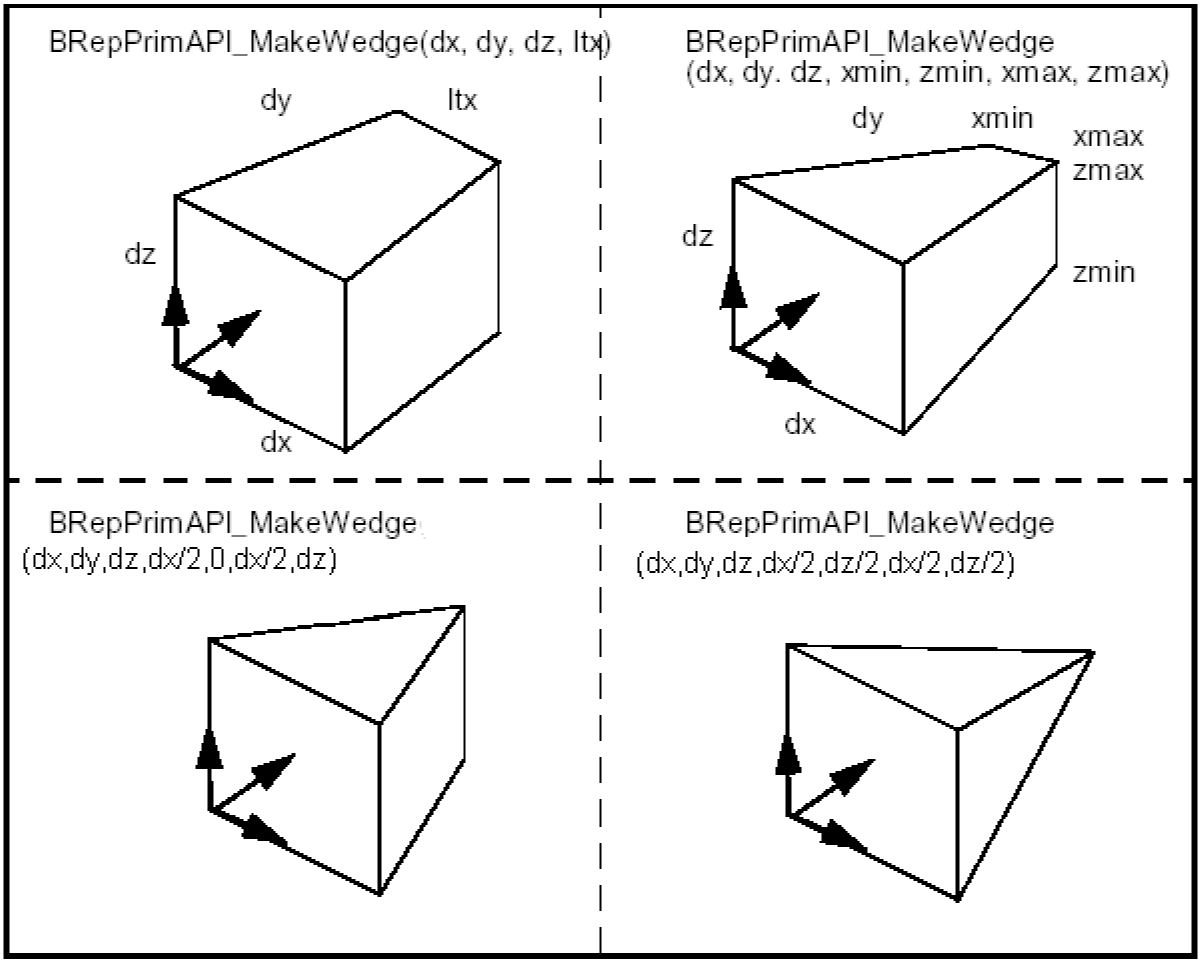

BRepPrimAPI_MakeWedge(dx, dy, dz, ltx)
dy
ltx
dz
dx
BRepPrimAPI_MakeWedge
(dx, dy. dz, xmin, zmin, xmax, zmax)
dy
xmin
xmax
zmax
dz
zmin
dx
BRepPrimAPI_MakeWedge
(dx,dy,dz,dx/2,0,dx/2,dz)
BRepPrimAPI_MakeWedge
(dx,dy,dz,dx/2,dz/2,dx/2,dz/2)

Figure 25. Making Wedges

4. 1. 3 BRepPrimAPI_MakeOneAxis

The BRepPrimAPI_MakeOneAxis class is a deferred class used as a root class for all the classes constructing rotational primitives. Rotational primitives are created by rotating a curve around an axis. They cover the cylinder, the cone, the sphere, the torus, and the revolution, which provides all the other curves.

The particular constructions of these primitives are described, but they all have some common arguments, which are:

- A system of coordinates, where the Z axis is the rotation axis..
- An angle in the range [0,2*PI].
- A vmin, vmax parameter range on the curve.

The result of the OneAxis construction is a Solid, a Shell, or a Face. The face is the face covering the rotational surface. Remember that you will not use the OneAxis directly but one of the derived classes, which provide improved constructions. The following figure illustrates the OneAxis arguments.

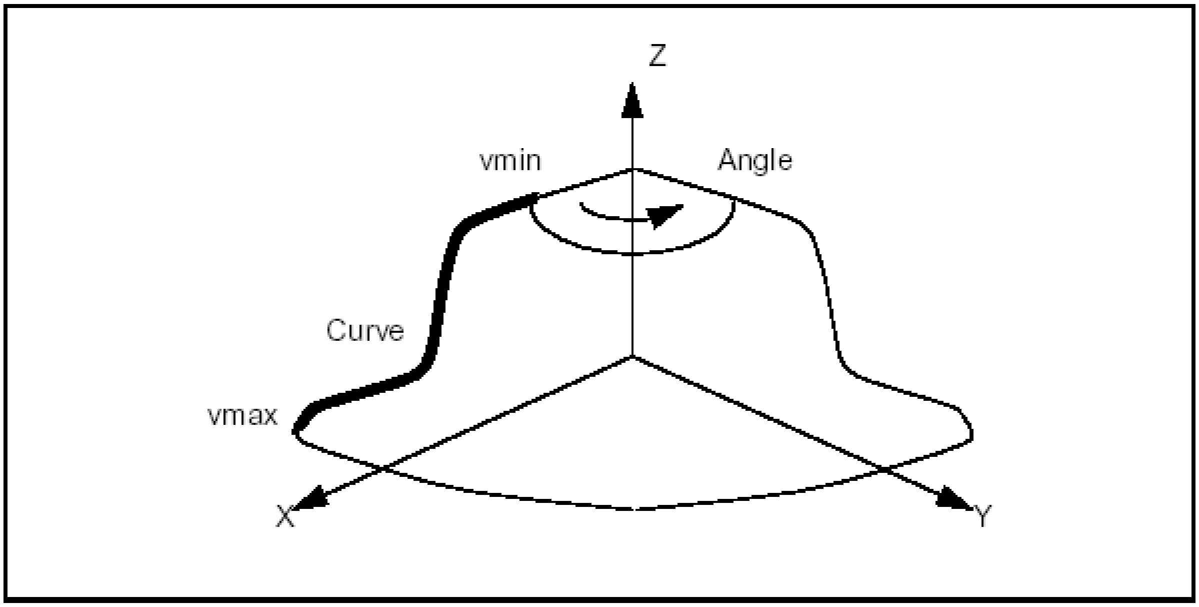


Figure 26. MakeOneAxis arguments

4. 1. 4 BRepPrimAPI_MakeCylinder

Use the MakeCylinder class to make cylindrical primitives. A cylinder is created either in the default coordinate system or in a given coordinate system (gp_Ax2). There are two constructions:

- Radius and height, to build a full cylinder.
- Radius, height and angle to build a portion of a cylinder.

The following code builds the cylindrical face of the figure, which is a quarter of cylinder along the Y axis with the origin at X,Y,Z, a length of DY, and a radius R.

Example

```
Standard_Real X = 20, Y = 10, Z = 15, R = 10, DY = 30;
// Make the system of coordinates
gp_Ax2 axes = gp::ZOX();
axes.Translate(gp_Vec(X,Y,Z));
TopoDS_Face F =
   BRepPrimAPI_MakeCylinder(axes,R,DY,PI/2.);
```

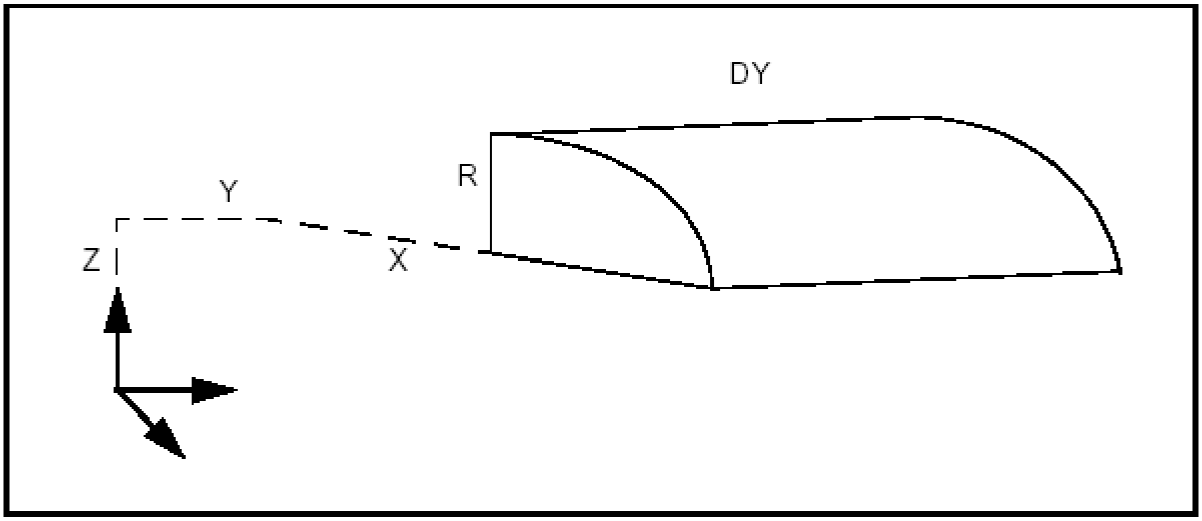


Figure 27. Example of a Cylinder

4. 1. 5 BRepPrimAPI_MakeCone

Use the BRepPrimAPI_MakeCone class to make conical primitives. Like a cylinder, a cone is created either in the default coordinate system or in a given coordinate system (gp_Ax2). There are two constructions:

- Two radii and height, to build a full cone. One of the radii can be null to make a sharp cone.
- Radii, height and angle to build a truncated cone.

The following code builds the solid cone of the figure, which is located in the default system with radii R1 and R2 and height H.

Example

```
Standard_Real R1 = 30, R2 = 10, H = 15;
TopoDS_Solid S = BRepPrimAPI_MakeCone(R1,R2,H);
```

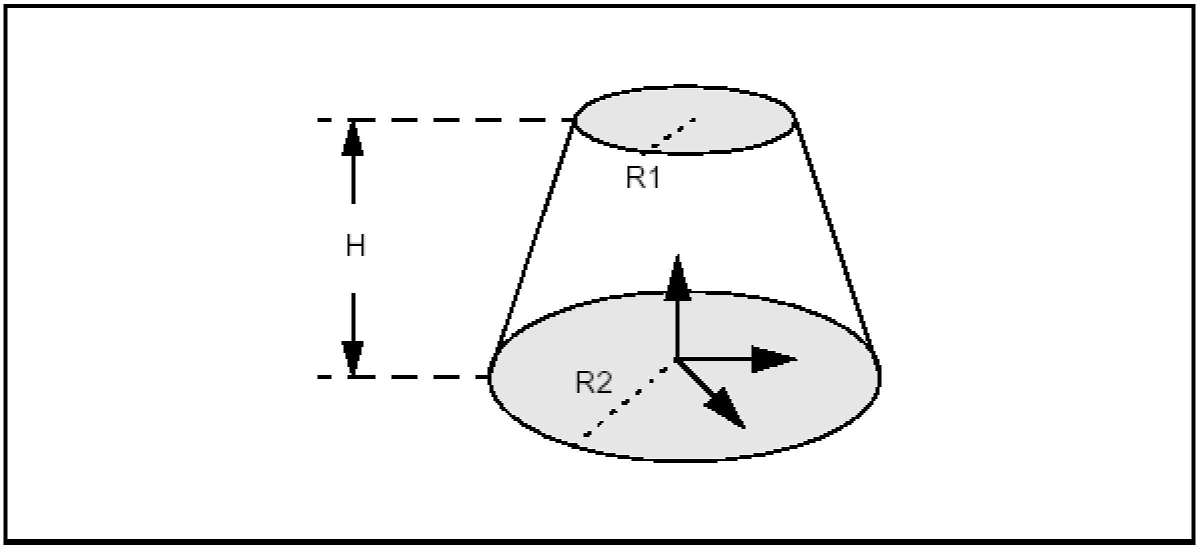


Figure 28. Example of a Cone

4. 1. 6 BRepPrimAPI_MakeSphere

Use the BRepPrimAPI_MakeSphere class to make spherical primitives. Like a cylinder, a sphere is created either in the default coordinate system or in a given coordinate system (gp_Ax2). There are four constructions:

- A radius; builds a full sphere. (see Figure 29.)

- A radius and an angle; build a portion of a sphere.
- A radius and two angles; build a segment of a sphere between two latitudes. The angles a1, a2 must verify the relation:

  PI/2 <= a1 < a2 <= PI/2. (see Figure 29.)
- A radius and three angles; build a portion of a strip of sphere. (see Figure 29.)

The following code builds four spheres from a radius and three angles.

Example

```
Standard_Real R = 30, ang =
      PI/2, a1 = -PI/2.3, a2 = PI/4;
TopoDS_Solid S1 = BRepPrimAPI_MakeSphere(R);
TopoDS_Solid S2 = BRepPrimAPI_MakeSphere(R,ang);
TopoDS_Solid S3 = BRepPrimAPI_MakeSphere(R,a1,a2);
TopoDS_Solid S4 = BRepPrimAPI_MakeSphere(R,a1,a2,ang);
```

Note that we could equally well choose to create Shells instead of Solids.

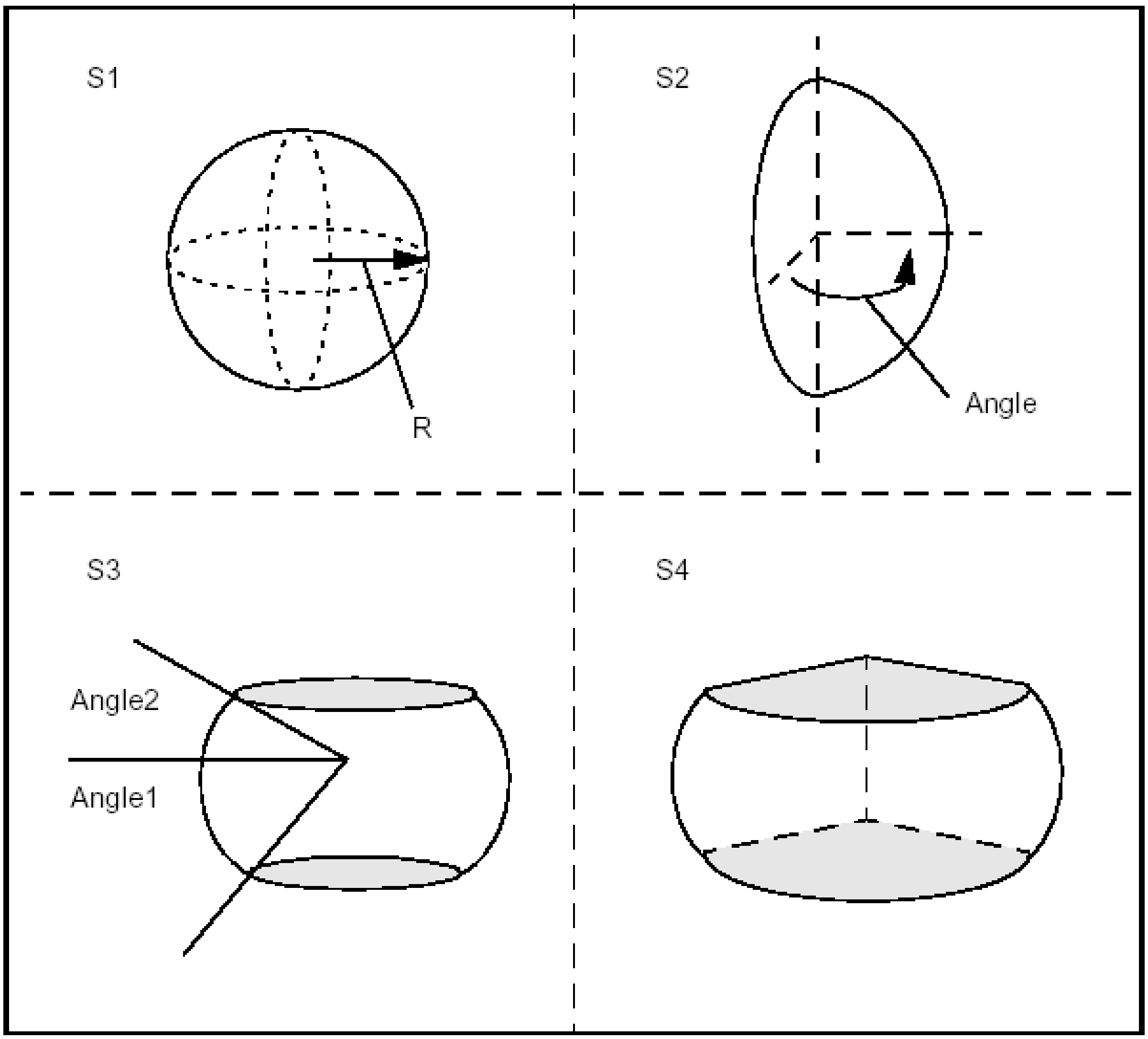


Figure 29. Examples of Spheres

4. 1. 7 BRepPrimAPI_MakeTorus

Use the BRepPrimAPI_MakeTorus class to make toroidal primitives. Like the other primitives, a torus is created either in the default coordinate system or in a given coordinate system (gp_Ax2). There are four constructions similar to the sphere constructions:

- Two radii; build a full torus.
- Two radii and an angle; build a portion of a torus.

- Two radii and two angles; build a segment of a torus between two latitudes. The angles a1, a2 must verify the relation:

  0 < a2 - a1 < 2*PI

- Two radii and three angles; build a portion of a segment of torus.

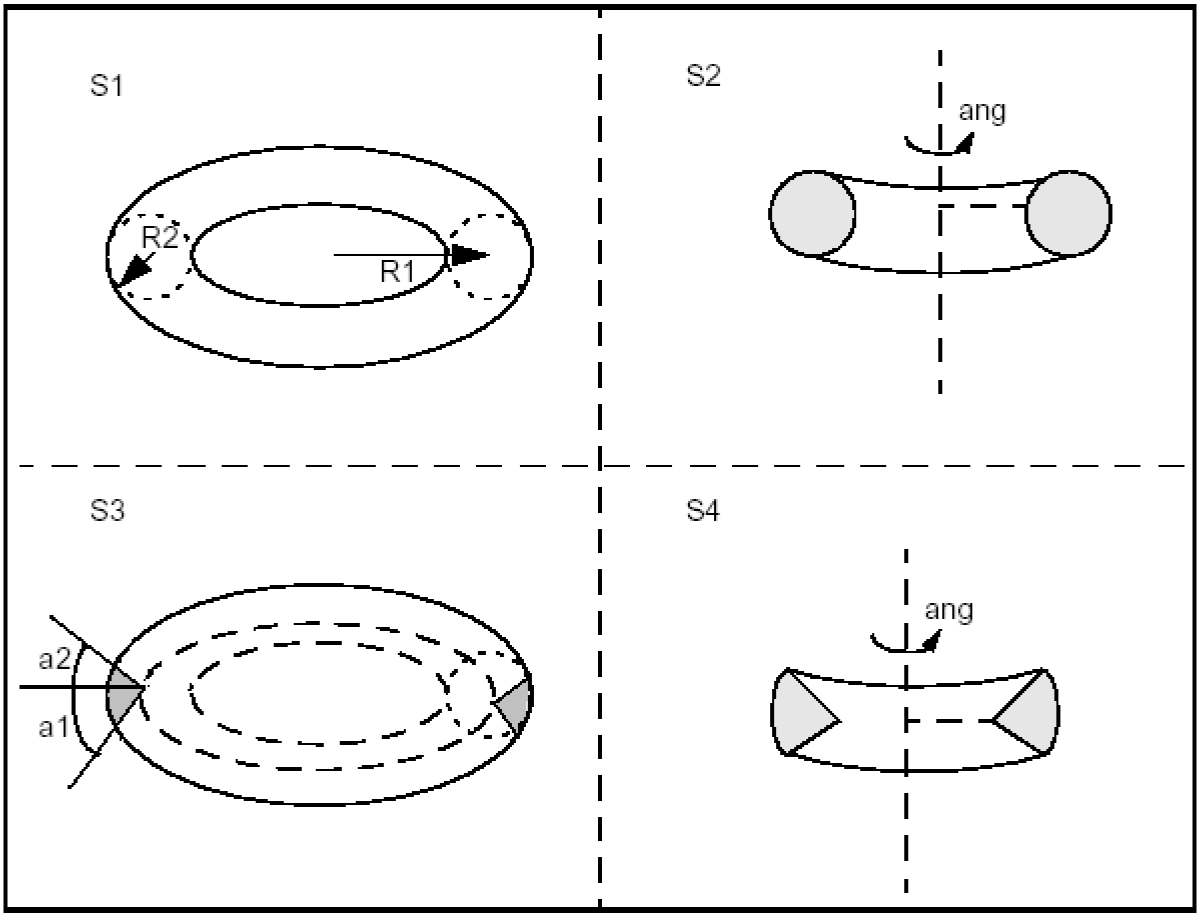


Figure 30. Examples of Tori

The following code builds four toroidal shells from two radii and three angles.

Example

```
Standard_Real R1 = 30, R2 = 10, ang = PI, a1 = 0,
      a2 = PI/2;
TopoDS_Shell S1 = BRepPrimAPI_MakeTorus(R1,R2);
TopoDS_Shell S2 = BRepPrimAPI_MakeTorus(R1,R2,ang);
TopoDS_Shell S3 = BRepPrimAPI_MakeTorus(R1,R2,a1,a2);
TopoDS_Shell S4 =
       BRepPrimAPI_MakeTorus(R1,R2,a1,a2,ang);
```

Note that we could equally well choose to create Solids instead of Shells.

4. 1. 8 BRepPrimAPI_MakeRevolution

Use the BRepPrimAPI_MakeRevolution class to build a uniaxe primitive from a curve. As with other uniaxe primitives it can be created in the default coordinate system or in a given coordinate system.

The curve can be any Geom_Curve, provided it is planar and lies in the same plane as the Z-axis of local coordinate system. There are four modes of construction:

- From a curve, use the full curve and make a full rotation.
- From a curve and an angle of rotation.
- From a curve and two parameters to trim the curve. The two parameters must be growing and within the curve range.
- From a curve, two parameters, and an angle. The two parameters must be growing and within the curve range.

4. 2 Sweeping: Prism, Revolution and Pipe

Sweeps are the objects you obtain by sweeping a **profile** along a **path.** The profile can be any topology. The path is usually a curve or a wire. The profile generates objects according to the following rules:

- Vertices generate Edges
- Edges generate Faces.
- Wires generate Shells.
- Faces generate Solids.
- Shells generate Composite Solids

It is forbidden to sweep Solids and Composite Solids. A Compound generates a

Compound with the sweep of all its elements.

Three kinds of sweeps are implemented in BRepPrimAPI, the linear sweep called **Prism**, the rotational sweep called **Revol** and the general sweep called **Pipe**.

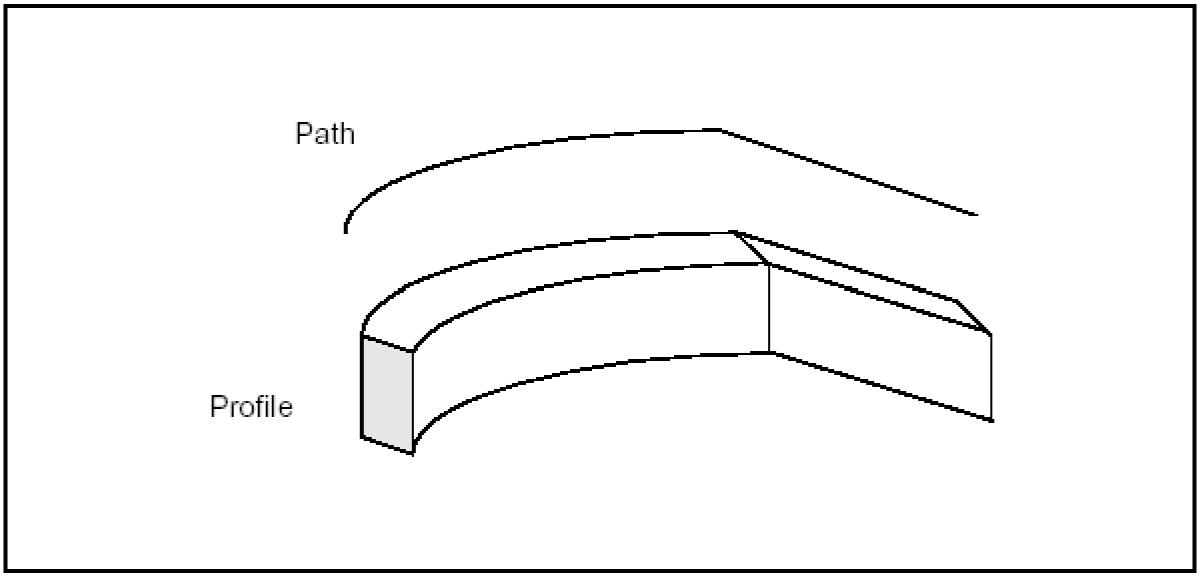


Figure 31. Generating a sweep

4. 2. 2 BRepPrimAPI_MakeSweep

The BRepPrimAPI_MakeSweep class is a deferred class used as a root of the sweep classes BRepPrimAPI_MakePrism and BRepPrimAPI_MakeRevol. It has currently no special services for the end user.

4. 2. 3 BRepPrimAPI_MakePrism

Use the BRepPrimAPI_MakePrism class to make a linear **prism** from a shape. A prism is created from a shape and a vector or a direction.

From a vector, a finite prism is created. From a direction, an infinite or semiinfinite prism is created. A Boolean argument is used to toggle the semi-infinite or infinite prism. All constructors have a boolean argument to copy or share the original shape. The default is to share it. The following code, using a face, a direction and a length, creates a finite, an infinite, and a semi-infinite solid.

Example

```
TopoDS_Face F = ..; // The swept face
```

```
gp_Dir direc(0,0,1);
Standard_Real l = 10;
// create a vector from the direction and the length
gp_Vec v = direc;
v *= l;
TopoDS_Solid P1 = BRepPrimAPI_MakePrism(F,v);
// finite
TopoDS_Solid P2 = BRepPrimAPI_MakePrism(F,direc);
// infinite
TopoDS_Solid                      P3                      =
BRepPrimAPI_MakePrism(F,direc,Standard_False);
// semi-infinite
```

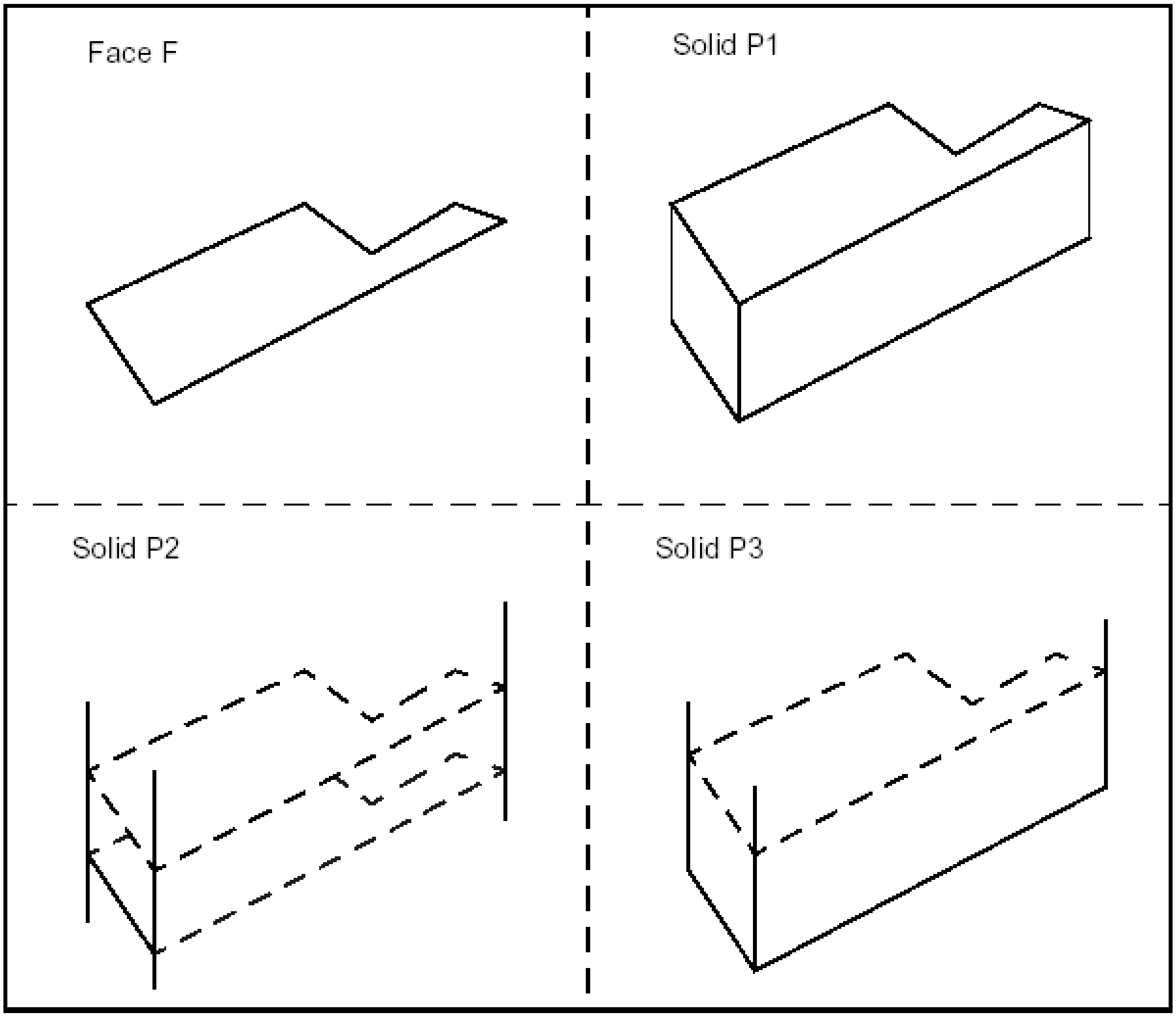


Figure 32. Finite, infinite, and semi-infinite prisms

4. 2. 4 BRepPrimAPI_MakeRevol

Use the BRepPrimAPI_MakeRevol class to make a revolved sweep. A **revol** is created from a shape, an axis (gp_Ax1), and an angle. The angle has a default value of 2*PI which means a closed revol.

BRepPrimAPI_MakeRevol constructors have a last argument to copy or share the original shape. The following code, using a face, an axis and an angle makes a full and a partial revol.

Example

```
TopoDS_Face F = ...; // the profile
gp_Ax1 axis(gp_Pnt(0,0,0),gp_Dir(0,0,1));
Standard_Real ang = PI/3;
TopoDS_Solid R1 = BRepPrimAPI_MakeRevol(F,axis);
// Full revol
TopoDS_Solid R2 = BRepPrimAPI_MakeRevol(F,axis,ang);
// Partial revol
```

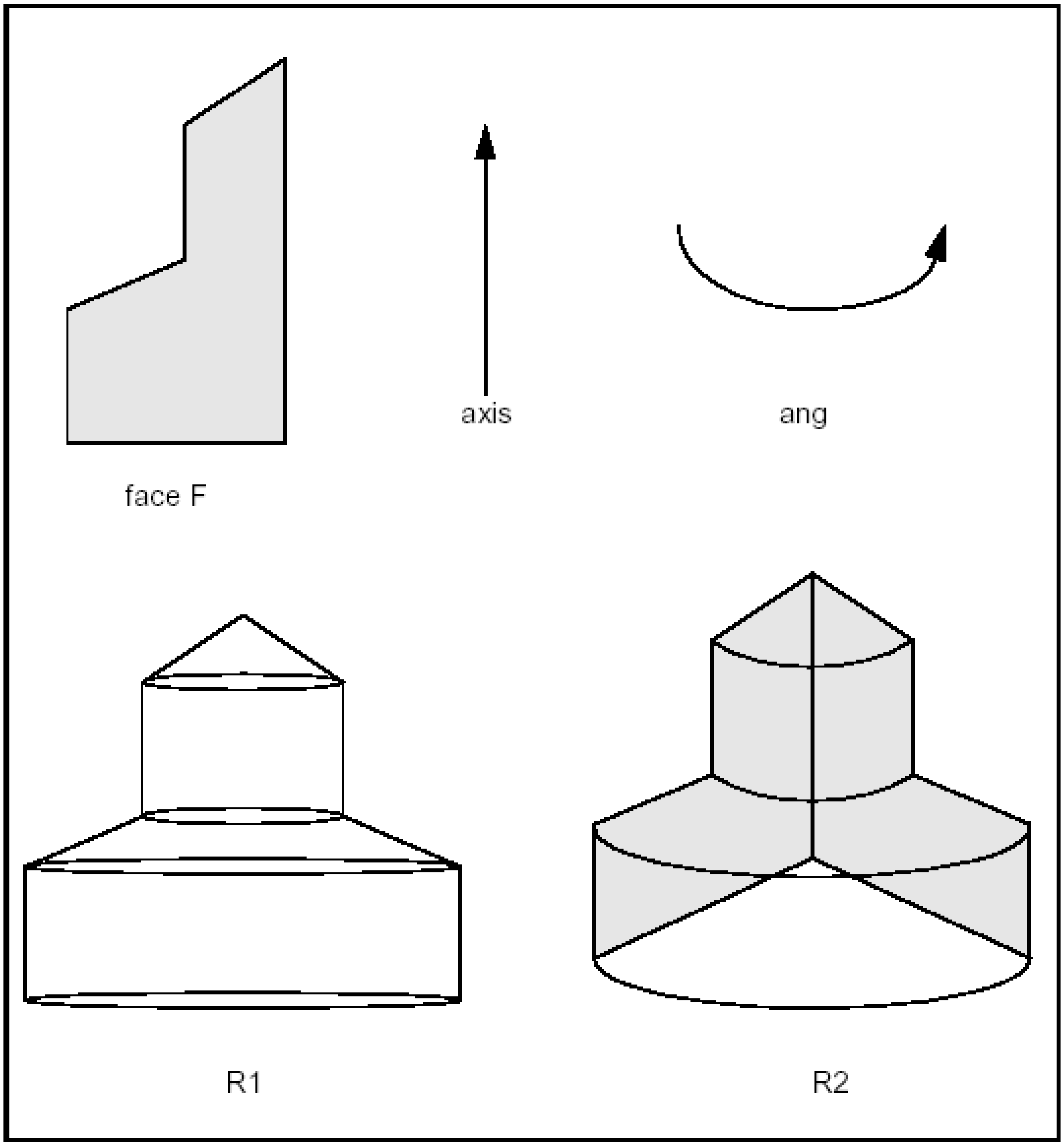

face F
axis
ang
R1
R2

Figure 33. Full and partial revols

5. Boolean Operations

5. 1 Boolean Operators

Boolean operations are used to create new shapes from the combinations of two shapes S1, S2.

Fusion:	gets all the points in S1 or S2.
Common:	gets all the points in S1 and S2.
Cut S1 by S2:	gets all the points in S1 and not in S2.

The following figure illustrates the boolean operations.

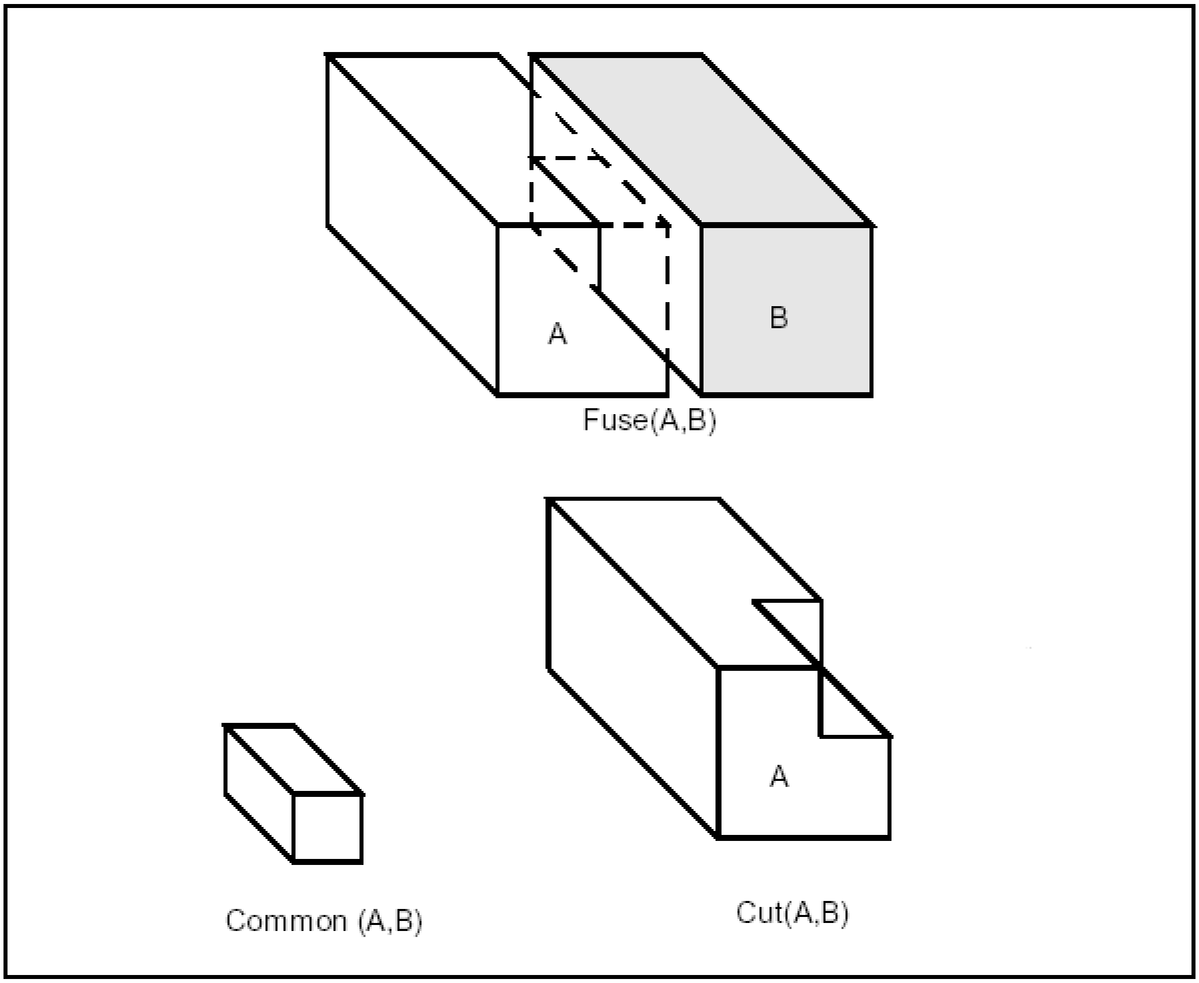


Figure 34. Boolean Operations

5. 1. 1 BRepAlgoAPI_BooleanOperation

The BRepAlgoAPI_BooleanOperation class is the deferred root class for Boolean operations.

5. 1. 2 BRepAlgoAPI_Fuse

The BRepAlgoAPI_Fuse class performs the fuse operations.

Example

```
TopoDS_Shape A = ..., B = ...;
TopoDS_Shape S = BRepAlgoAPI_Fuse(A,B);
```

5. 1. 3 BRepAlgoAPI_Common

The BRepAlgoAPI_Common class performs the common operations.

Example

```
TopoDS_Shape A = ..., B = ...;
TopoDS_Shape S = BRepAlgoAPI_Common(A,B);
```

5. 1. 4 BRepAlgoAPI_Cut

The BRepAlgoAPI_Cut class performs the cut operations.

Example

```
TopoDS_Shape A = ..., B = ...;
TopoDS_Shape S = BRepAlgoAPI_Cut(A,B);
```

5. 1. 5 BRepAlgoAPI_Section

The BRepAlgoAPI_Section class performs the section, described as a TopoDS_Compound made of TopoDS_Edge.

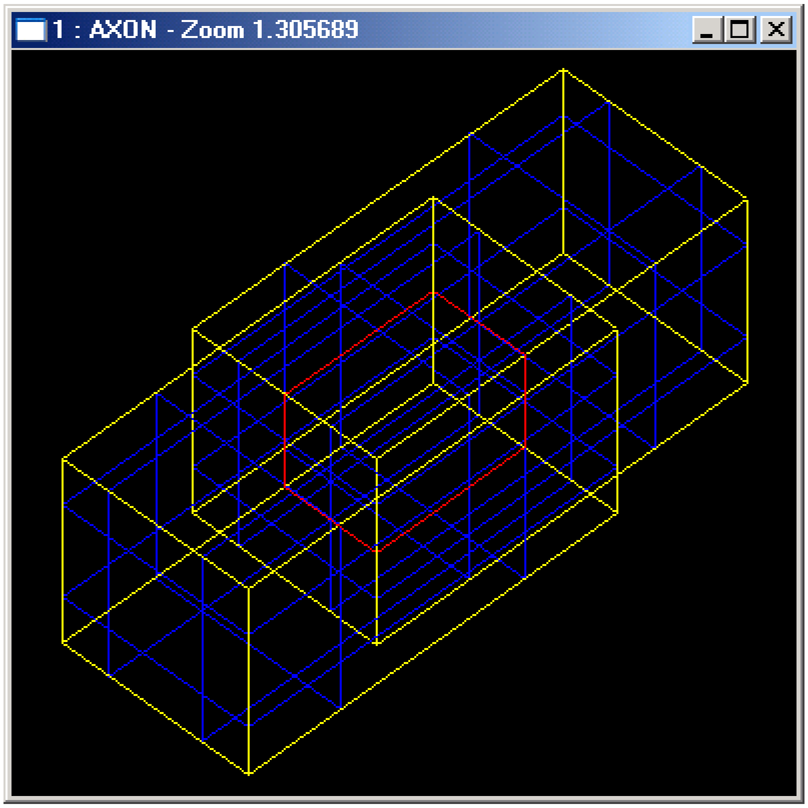

1 : AXON - Zoom 1.305689

Figure 35. Section operation

Example

```
TopoDS_Shape A = ..., TopoDS_ShapeB = ...;
TopoDS_Shape S = BRepAlgoAPI_Section(A,B);
```

---

6. Fillets and Chamfers

6. 1 Fillet Constructor

6. 1. 1 BRepFilletAPI_MakeFillet

Use the BRepFilletAPI_MakeFillet class to add fillets on a shape. A fillet is a smooth face replacing a sharp edge.

First, give a shape, which will be filleted. This is done at the construction of the class.

Then add fillet descriptions using the Add method. A fillet description contains an edge and a radius. Of course the edge must be shared by two faces. The fillet is automatically extended to all edges in a smooth continuity with the original edge.

Finally, perform the operation by asking for the result as for any class inherited from MakeShape.

It is not an error to Add a fillet twice, the last description holds.

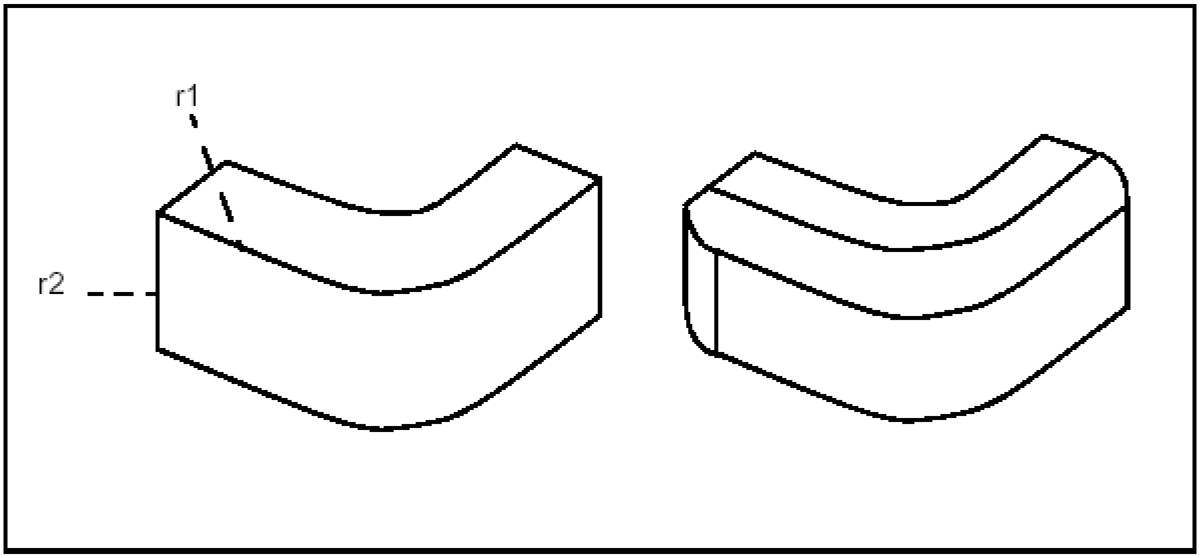


Figure 36. Filleting two edges using radii r1 and r2.

In the following example a filleted box with dimensions a,b,c and radius r is created.

Example 1 Constant radius.

```
#include <TopoDS_Shape.hxx>
#include <TopoDS.hxx>
#include <BRepPrimAPI_MakeBox.hxx>
#include <TopoDS_Solid.hxx>
#include <BRepFilletAPI_MakeFillet.hxx>
#include <TopExp_Explorer.hxx>


TopoDS_Shape FilletedBox(const Standard_Real a,
                         const Standard_Real b,
                         const Standard_Real c,
                         const Standard_Real r)
{
      TopoDS_Solid Box = BRepPrimAPI_MakeBox(a,b,c);
      BRepFilletAPI_MakeFillet MF(Box);

      // add all the edges to fillet
      TopExp_Explorer ex(Box,TopAbs_EDGE);
      while (ex.More())
      {
      MF.Add(r,TopoDS::Edge(ex.Current()));
      ex.Next();
      }
      return MF.Shape();
}
```

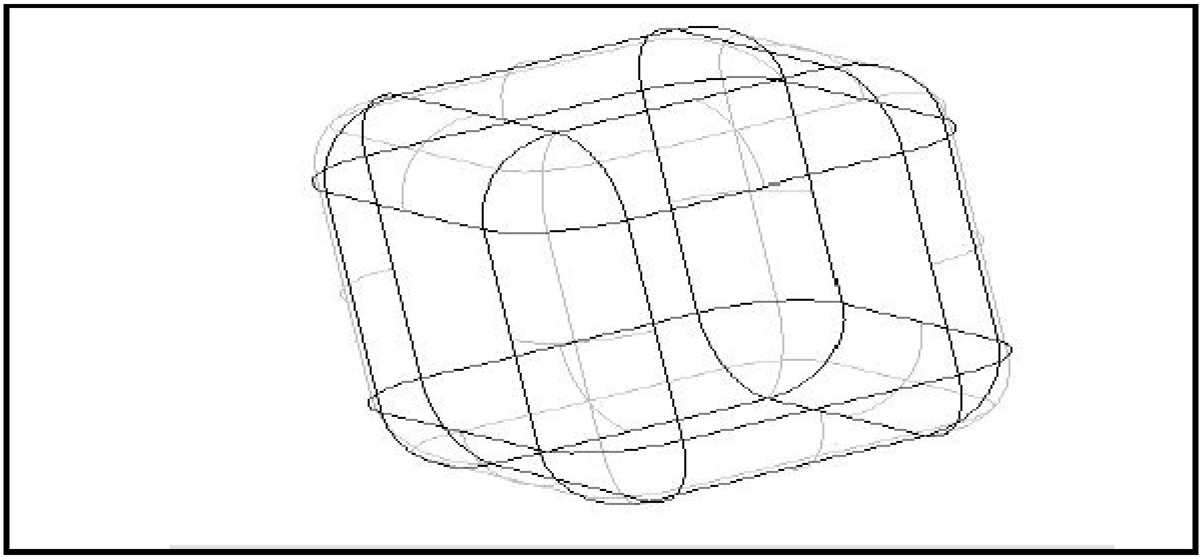


Figure 37. Filleting a box

Example 2 Evolutive radius

```
void CSampleTopologicalOperationsDoc::OnEvolvedblend1()
{
    TopoDS_Shape theBox = BRepPrimAPI_MakeBox(200,200,200);

    BRepFilletAPI_MakeFillet Rake(theBox);
    ChFi3d_FilletShape FSh = ChFi3d_Rational;
    Rake.SetFilletShape(FSh);

    TColgp_Array1OfPnt2d ParAndRad(1, 6);
    ParAndRad(1).SetCoord(0., 10.);
    ParAndRad(1).SetCoord(50., 20.);
    ParAndRad(1).SetCoord(70., 20.);
    ParAndRad(1).SetCoord(130., 60.);
    ParAndRad(1).SetCoord(160., 30.);
    ParAndRad(1).SetCoord(200., 20.);

    TopExp_Explorer ex(theBox,TopAbs_EDGE);
    Rake.Add(ParAndRad, TopoDS::Edge(ex.Current()));
    TopoDS_Shape evolvedBox = Rake.Shape();
}
```

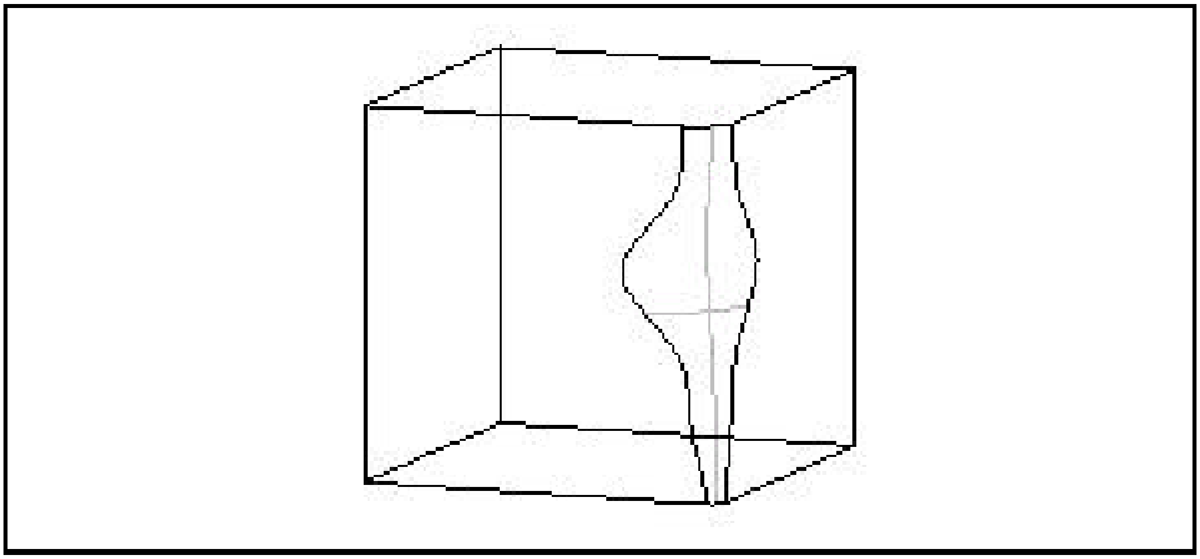


Figure 38. Evolutive radius fillet

6. 2 BRepFilletAPI_MakeFillet2d

BRepFilletAPI_MakeFillet2d is used to construct fillets and chamfers on planar faces.

A fillet is defined as a smooth edge on the face whereas a chamfer is defined as a rectilinear edge replacing a sharp vertex of the face.

1. Give the face on which the fillets (or the chamfers) are to be built.
2. Indicate which vertex is to be deleted and give the fillet radius with the AddFillet method in order to add a fillet. Or add a chamfer with the AddChamfer method. A chamfer can be described by
    - two edges and two distances
    - one edge, one vertex, one distance and one angle.

Fillets and chamfers are calculated when addition is complete.

3. Modification of a chamfer or fillet2d is possible.

A new face is created, the original face remaining unchanged.

You can ask the builder to obtain information on the modifications, which have been performed:

- new edge, modified or unchanged edge
- edge E1 has been changed into edge E1’

- edge E1’ is built from edge E1

If face F2 is created by the 2d fillet and chamfer builder from face F1, the builder can be rebuilt (the builder recovers the status it had before deletion). To do so, use the following syntax:

```
BRepFilletAPI_MakeFillet2d builder;
builder.Init(F1,F2);
```

Example

```
#include “BRepPrimAPI_MakeBox.hxx”
#include “TopoDS_Shape.hxx”
#include “TopExp_Explorer.hxx”
#include “BRepFilletAPI_MakeFillet2d.hxx”
#include “TopoDS.hxx”
#include “TopoDS_Solid.hxx”

TopoDS_Shape FilletFace(const Standard_Real a,
                        const Standard_Real b,
                        const Standard_Real c,
                        const Standard_Real r)

{
      TopoDS_Solid Box = BRepPrimAPI_MakeBox (a,b,c);
      TopExp_Explorer ex1(Box,TopAbs_FACE);

      const TopoDS_Face& F = TopoDS::Face(ex1.Current());
      BRepFilletAPI_MakeFillet2d MF(F);
      TopExp_Explorer ex2(F, TopAbs_VERTEX);
      while (ex2.More())
      {
      MF.AddFillet(TopoDS::Vertex(ex2.Current()),r);
      ex2.Next();
      }
      // while...
      return MF.Shape();
}
```

6. 1 Chamfer Constructor

6 . 3 BRepFilletAPI_MakeChamfer

The use of the BRepFilletAPI_MakeChamfer class is similar to the use of BRepFilletAPI_MakeFillet, except for the following:

1. The surfaces created are ruled and not smooth.
2. The Add syntax for selecting edges requires one or two distances, one edge and one face (contiguous to the edge):

`Add(dist, E, F)`

`Add(d1, d2, E, F) with d1 on the face F.`

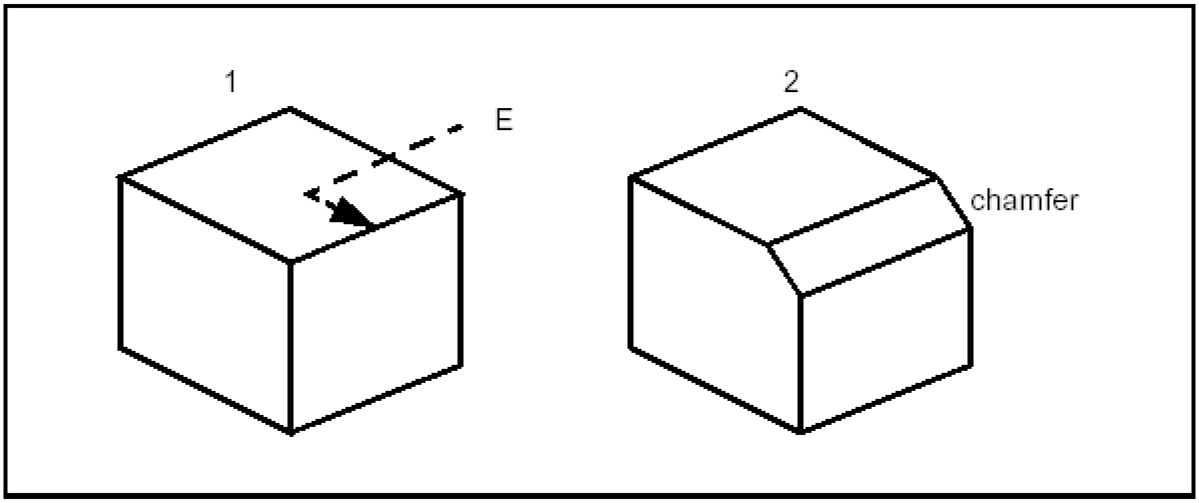


Figure 39. Creating a chamfer

7. Offsets, Drafts, Pipes and Evolved shapes

7. 1 Shelling operator

BRepOffsetAPI_MakeThickSolid

Shelling is used to offset given faces of a solid by a specific value. It rounds or intersects adjacent faces along its edges depending on the convexity of the edge. The constructor takes the solid, the list of faces to remove and an offset value as input.

Example

```
TopoDS_Solid SolidInitial = ...;

Standard_Real                Of          = ...;
TopTools_ListOfShape         LCF;
TopoDS_Shape                 Result;
Standard_Real                Tol = Precision::Confusion();

for (Standard_Integer i = 1 ;i <= n; i++) {
      TopoDS_Face SF = ...; // a face from SolidInitial
      LCF.Append(SF);
}

Result = BRepOffsetAPI_MakeThickSolid     (SolidInitial,
                                          LCF,
                                          Of,
                                          Tol);
```

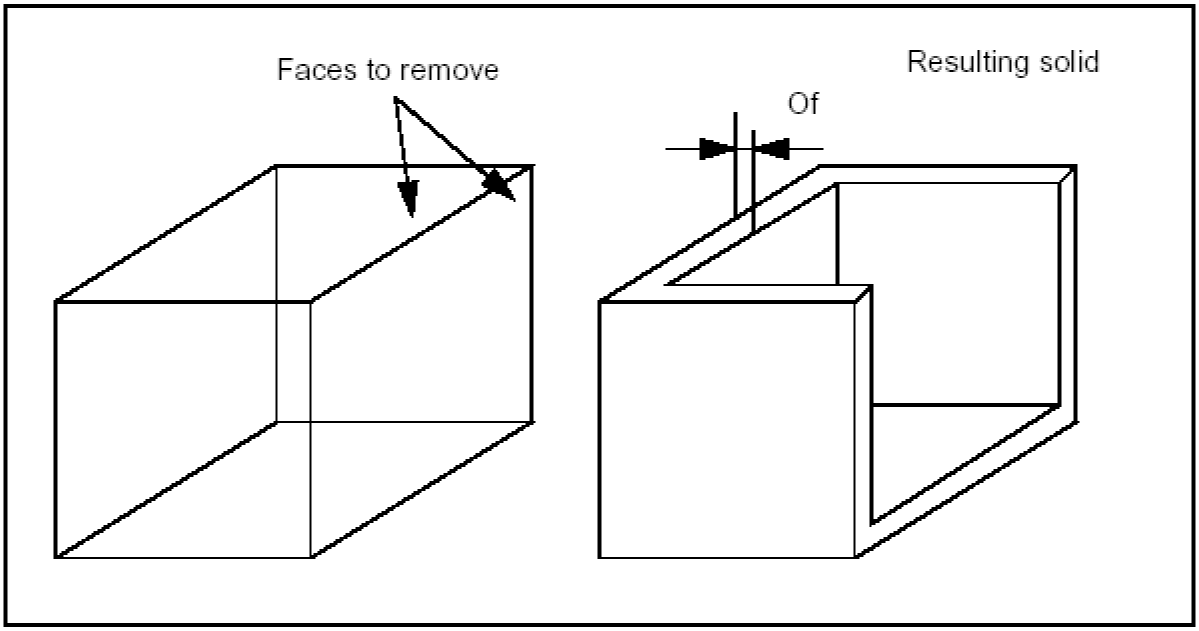


Figure 40. Shelling

7. 2 Modification operators

BRepOffsetAPI_DraftAngle

Use the BRepOffsetAPI_DraftAngle class to modify a shape by applying draft angles to planar, cylindrical and conical faces of the shape.

The class is created or initialized from a shape, then faces to be modified are added; for each face, three arguments are used:

- Direction: the direction with which the draft angle is measured
- Angle: value of the angle
- Neutral plane: intersection between the face and the neutral plane is invariant.

The following code places a draft angle on several faces of a shape; the same direction, angle and neutral plane are used for each face:

Example

```
TopoDS_Shape myShape = ...
// The original shape
```

```
TopTools_ListOfShape ListOfFace;
// Creation of the list of faces to be modified
...

gp_Dir Direc(0.,0.,1.);
// Z direction
Standard_Real Angle = 5.*PI/180.;
// 5 degree angle
gp_Pln Neutral(gp_Pnt(0.,0.,5.), Direc);
// Neutral plane Z=5
BRepOffsetAPI_DraftAngle theDraft(myShape);
TopTools_ListIteratorOfListOfShape itl;
for (itl.Initialize(ListOfFace); itl.More(); itl.Next()) {
   theDraft.Add(TopoDS::Face(itl.Value()),Direc,Angle,Neutral);
   if (!theDraft.AddDone()) {
      // An error has occurred. The faulty face is given by
      // ProblematicShape
      break;
   }
}
if (!theDraft.AddDone()) {
   // An error has occurred
   TopoDS_Face guilty = theDraft.ProblematicShape();
   ...
}
theDraft.Build();
if (!theDraft.IsDone()) {
   // Problem encountered during reconstruction
   ...
}
else {
   TopoDS_Shape myResult = theDraft.Shape();
   ...
}
```

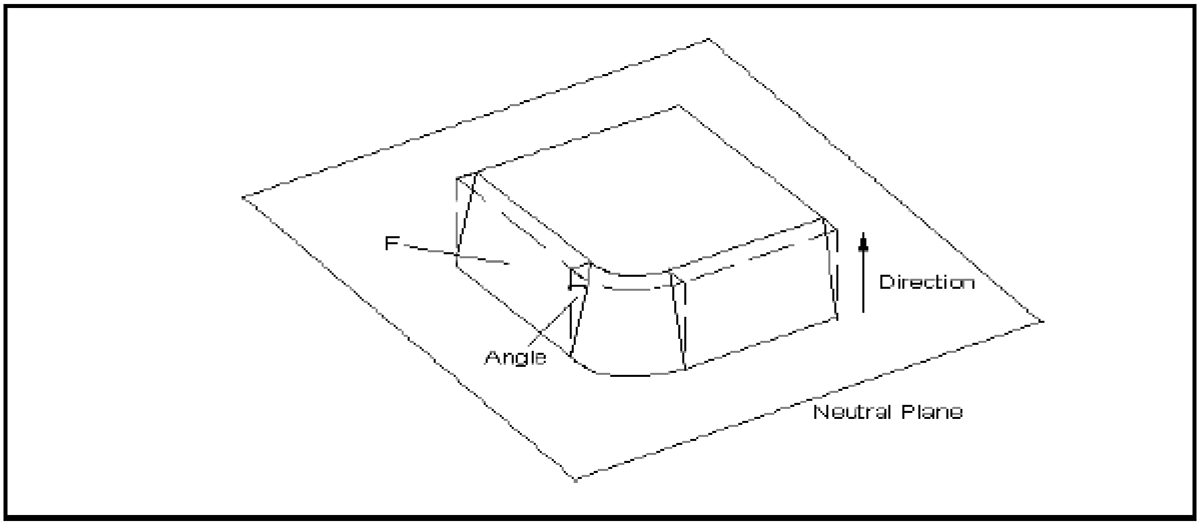


Figure 41. Example of DraftAngle

7. 3 Pipe Constructor

BRepOffsetAPI_MakePipe

Use the BRepOffsetAPI_MakePipe class to make a pipe. A pipe is created from a Spine, which is a Wire and a Profile which is a Shape. This implementation is limited to spines with smooth transitions, sharp transitions are precessed by BRepOffsetAPI_MakePipeShell. To be more precise the continuity must be G1, which is to say at neighboring edges the tangent must have the same direction, though not necessarily the same magnitude.

Whatever angle the spine makes with the profile is preserved throughout the pipe.

Example

```
TopoDS_Wire Spine = ...;
TopoDS_Shape Profile = ...;
TopoDS_Shape Pipe = BRepOffsetAPI_MakePipe(Spine,Profile);
```

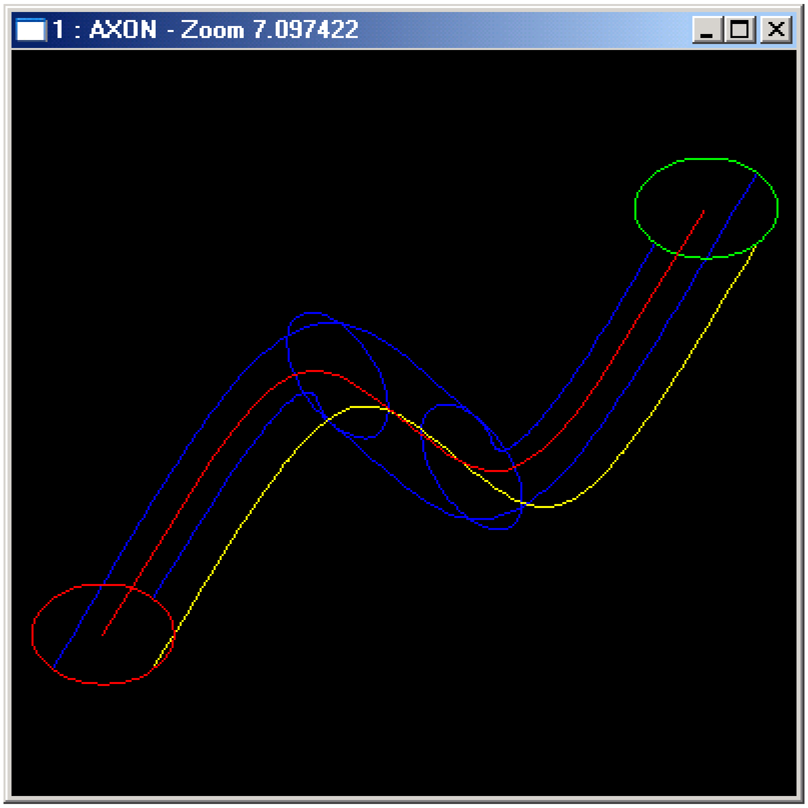


Figure 42. Example of MakePipe

7. 4 Constructor of Evolved Solid

BRepOffsetAPI_MakeEvolved

Use the BRepOffsetAPI_MakeEvolved class to create an evolved solid. The evolved shell or solid is created from a Spine (planar face or wire) and a profile (wire).

The evolved solid is the unlooped sweep generated by the spine and the profile.

The evolved solid is created by sweeping the profile's reference axes on the spine. The origin of the axes moves to the spine, the X axis and the local tangent coincide and the Z axis is normal to the face.

The reference axes of the profile can be defined following two distinct modes:

1. The reference axes of the profile are the origin axes.
2. The references axes of the profile are calculated as follows:

- the origin is given by the point on the spine which is the closest to the profile

- the X axis is given by the tangent to the spine at the point defined above
- the Z axis is the normal to the plane which contains the spine.

Example

```
TopoDS_Face Spine = ...;
TopoDS_Wire Profile = ...;
TopoDS_Shape Evol =
BRepOffsetAPI_MakeEvolved(Spine,Profile);
```

---

8. Sewing operators

8. 1 BRepBuilderAPI_Sewing

The BRepOffsetAPI_Sewing class is used to sew TopoDS Shapes together along their common edges. The edges can be partially shared as in the following example.

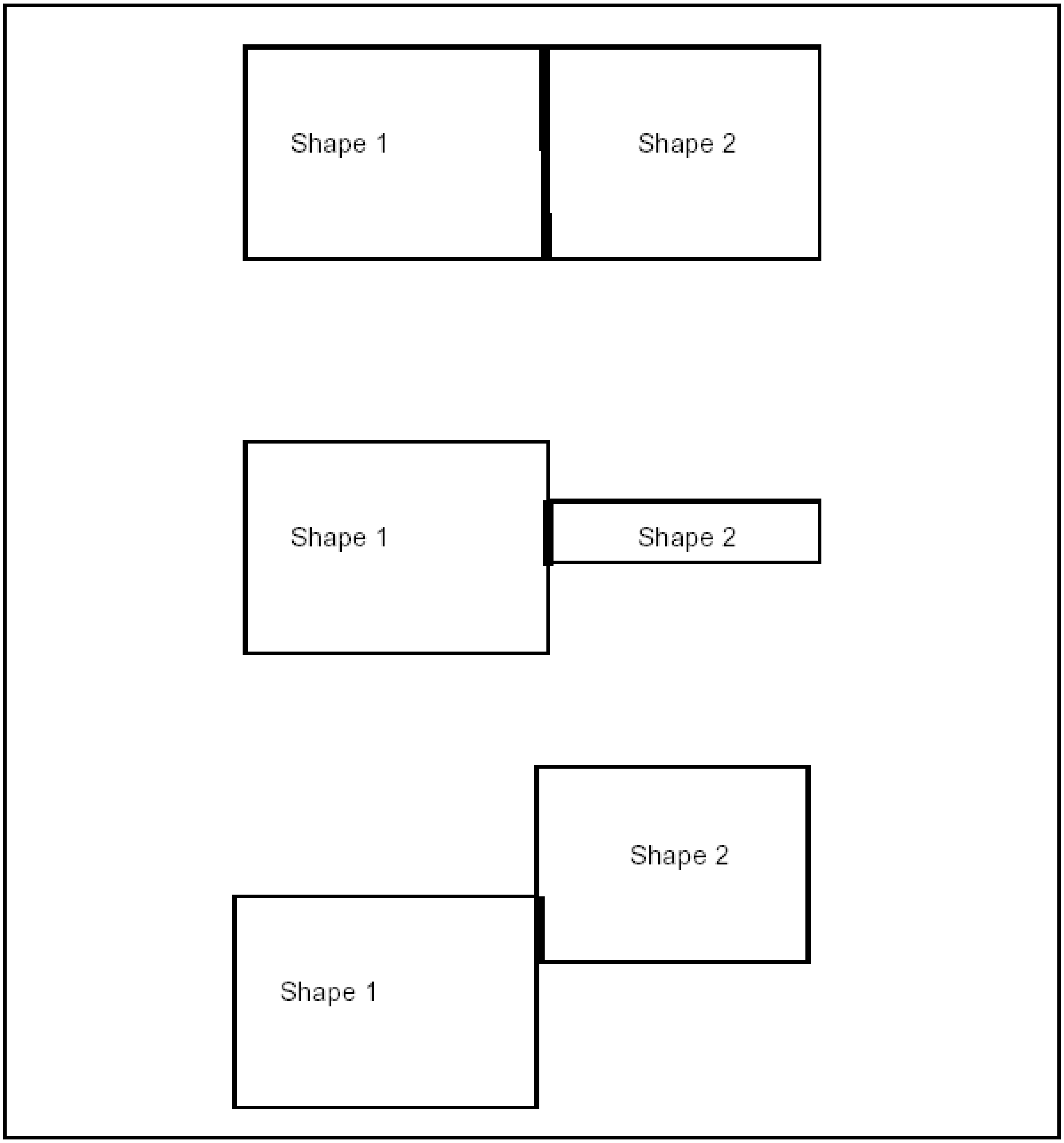

Shape 1
Shape 2
Shape 1
Shape 2
Shape 2
Shape 1

Figure 43. Shapes with partially shared edges

The constructor takes as arguments the tolerance (default value is 10^{-6}) and a flag, which is used to mark the degenerate shapes.

The *Add* method is used to add shapes, as it is needed.

The *Perform* method forces calculation of the sewed shape.

The *SewedShape* method returns the result.

Additional methods can be used to give additional information on the number of free boundaries, of multiple edges and of degenerate shapes.

8. 2 BRepOffsetAPI_FindContiguousEdges

The BRepOffsetAPI_FindContiguousEdges class is used to find edges, which coincide among a set of shapes within the given tolerance; these edges can be analyzed on tangency, continuity (C1, G2, etc.)...

The constructor takes as arguments the tolerance defining the edge proximity (10^{-6} by default) and a flag used to mark degenerated shapes.

The *Add* method is used to add shapes, which are to be analyzed.

The *NbEdges* method returns the total number of edges.

The *NbContiguousEdges* returns the number of contiguous edges within the given tolerance as defined above.

The *ContiguousEdge* method takes an edge number as an argument and returns the TopoDS edge contiguous to another edge.

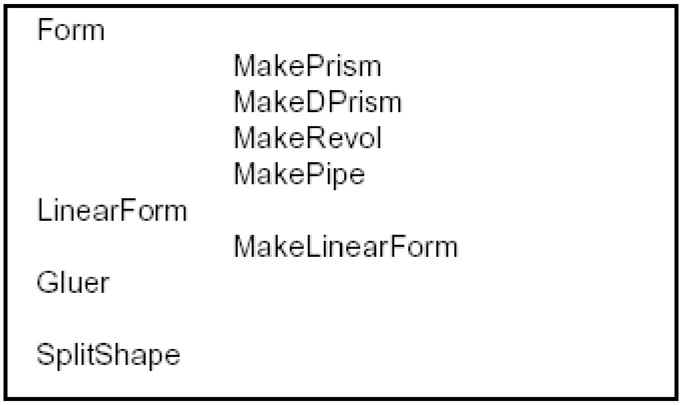
The *ContiguousEdgeCouple* gives all the edges or portions of edges (sections), which are common to the edge with the number given above.

The *SectionToBoundary* method is used to find the original edge on the original shape from the section.

9. Features

9. 1 The BRepFeat Classes and their use

The BRepFeat package is used to manipulate extensions of the classical boundary representation of shapes closer to features. In that sense, BRepFeat is an extension of the BRepBuilderAPI package.



- Form
  - MakePrism
  - MakeDPrism
  - MakeRevol
  - MakePipe
- LinearForm
  - MakeLinearForm
- Gluer

- SplitShape

Figure 44. The BRepFeat classes

9. 1. 1 Form classes

The Form from BRepFeat class is a deferred class used as a root for form features. It inherits MakeShape from BRepBuilderAPI and provides implementation of methods using to keep track of all sub-shapes.

MakePrism

The MakePrism from BRepFeat class is used to build a prism interacting with a shape. It is created or initialized from

- a shape (the basic shape),
- the base of the prism,
- a face (the face of sketch on which the base has been defined and used

to determine whether the base has been defined on the basic shape or not),

- a direction,
- a Boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another Boolean indicating if the self-intersections have to be found (not used in every case).

There are six Perform methods:

Perform(Height)	The resulting prism is of the given length.
Perform(Until)	The prism is defined between the position of the base and the given face.
Perform(From, Until)	The prism is defined between the two faces From and Until.
PerformUntilEnd()	The prism is semi-infinite, limited by the actual position of the base.
PerformFromEnd(Until)	The prism is semi-infinite, limited by the face Until.
PerformThruAll()	The prism is infinite. In the case of a depression, the result is similar to a cut with an infinite prism. In the case of a protrusion, infinite parts are not kept in the result.

NOTE

The Add method can be used prior to using the Perform methods to indicate that a face generated by an edge "slides" onto a face of the basic shape.

In the following sequence, a protrusion is performed, i.e. a face of the shape is changed into a prism.

Example

```
TopoDS_Shape Sbase = ...;  // an initial shape
TopoDS_Face Fbase = ....; // a base of prism

gp_Dir Extrusion (.,.,.);

// An empty face is given as the sketch face

BRepFeat_MakePrism  thePrism(Sbase,  Fbase,  TopoDS_Face(),
Extrusion, Standard_True, Standard_True);

thePrism, Perform(100.);
if (thePrism.IsDone()) {
      TopoDS_Shape theResult = thePrism;
      ...
}
```

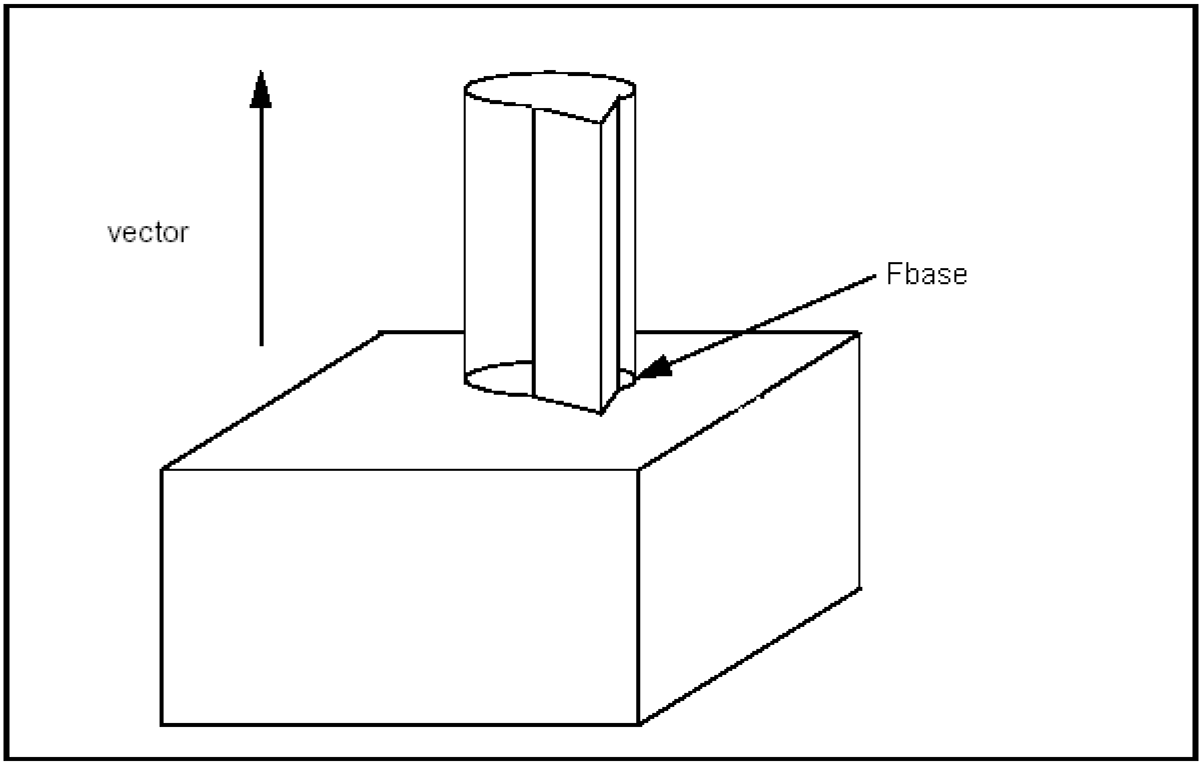


Figure 45. Fusion with MakePrism

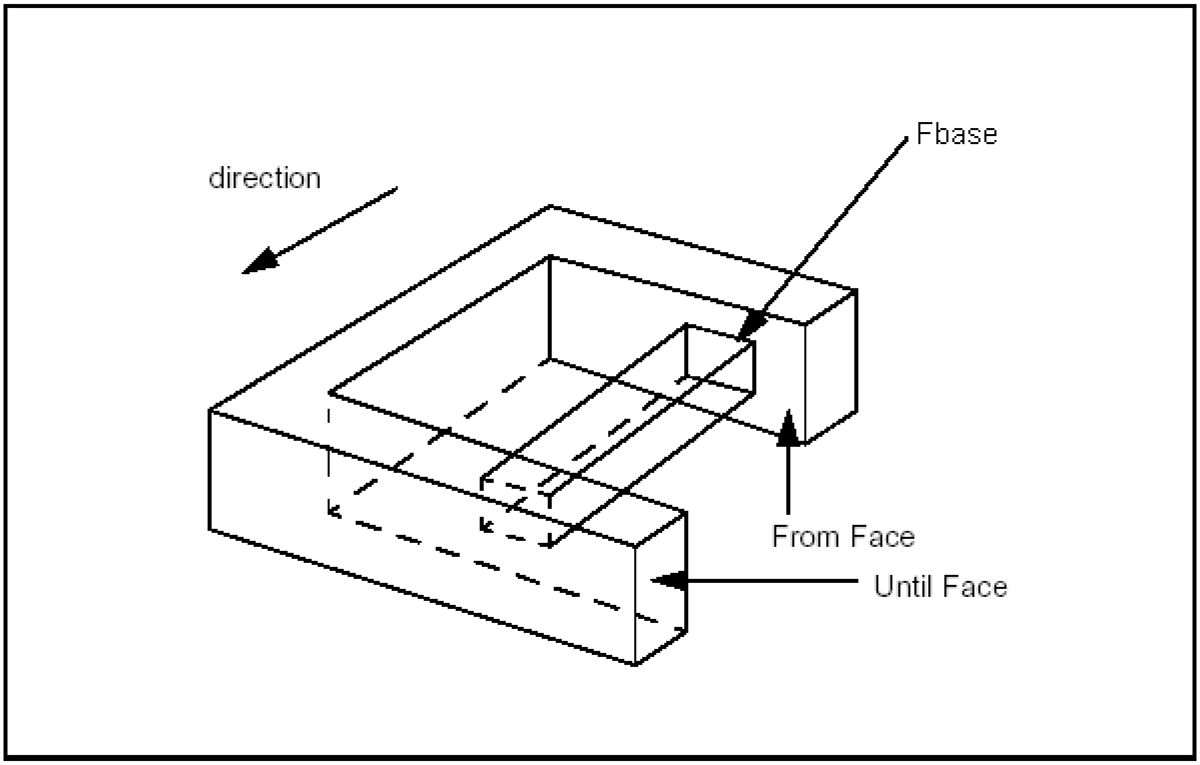


Figure 46. Creating a prism between two faces with Perform(From, Until)

MakeDPrism

The MakeDPrism from BRepFeat class is used to build draft prism topologies interacting with a basis shape . These can be depressions or protrusions. A class object is created or initialized from

- a shape (basic shape),
- the base of the prism,
- a face (face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
- an angle,
- a Boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another Boolean indicating if self-intersections have to be found (not used in every case).

Evidently the input data for MakeDPrism are the same as for MakePrism except for a new parameter Angle and a missing parameter Direction: the direction of the prism generation is determined automatically as the normal to the base of the prism.

The semantics of draft prism feature creation is based on the construction of shapes:

- along a length
- up to a limiting face
- from a limiting face to a height.

The shape defining construction of the draft prism feature can be either the supporting edge or the concerned area of a face.

In the case of the supporting edge, this contour can be attached to a face of the basis shape by binding. When the contour is bound to this face, the information that the contour will slide on the face becomes available to the relevant class methods.

In the case of the concerned area of a face, you could, for example, cut it out and move it to a different height, which will define the limiting face of a protrusion direction or depression.

There are six Perform methods:

Perform(Height)	The resulting prism is of the given length.
Perform(Until)	The prism is defined between the position of the base and the given face.
Perform(From, Until)	The prism is defined between the two faces From and Until.
PerformUntilEnd()	The prism is semi-infinite, limited by the actual position of the base.
PerformFromEnd(Until)	The prism is semi-infinite, limited by the face Until.
PerformThruAll()	The prism is infinite. In the case of a depression, the result is similar to a cut with an infinite prism. In the case of a protrusion, infinite parts are not kept in the result.

NOTE

The Add method can be used prior to using the Perform methods to indicate that a face generated by an edge "slides" onto a face of the basic shape.

Example

```
MakeDPrism

TopoDS_Shape S = BRepPrimAPI_MakeBox(400.,250.,300.);
TopExp_Explorer Ex;
Ex.Init(S,TopAbs_FACE);
Ex.Next();
Ex.Next();
Ex.Next();
Ex.Next();
Ex.Next();
TopoDS_Face F = TopoDS::Face(Ex.Current());
Handle(Geom_Surface) surf = BRep_Tool::Surface(F);
gp_Circ2d
c(gp_Ax2d(gp_Pnt2d(200.,130.),gp_Dir2d(1.,0.)),50.);
BRepBuilderAPI_MakeWire MW;
Handle(Geom2d_Curve) aline = new Geom2d_Circle(c);
MW.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,PI));
MW.Add(BRepBuilderAPI_MakeEdge(aline,surf,PI,2.*PI));
BRepBuilderAPI_MakeFace MKF;
MKF.Init(surf,Standard_False);
MKF.Add(MW.Wire());
TopoDS_Face FP = MKF.Face();
BRepLib::BuildCurves3d(FP);
BRepFeat_MakeDPrism MKDP (S,FP,F,10*PI180,Standard_True,
                        Standard_True);
MKDP.Perform(200);
TopoDS_Shape res1 = MKDP.Shape();
```

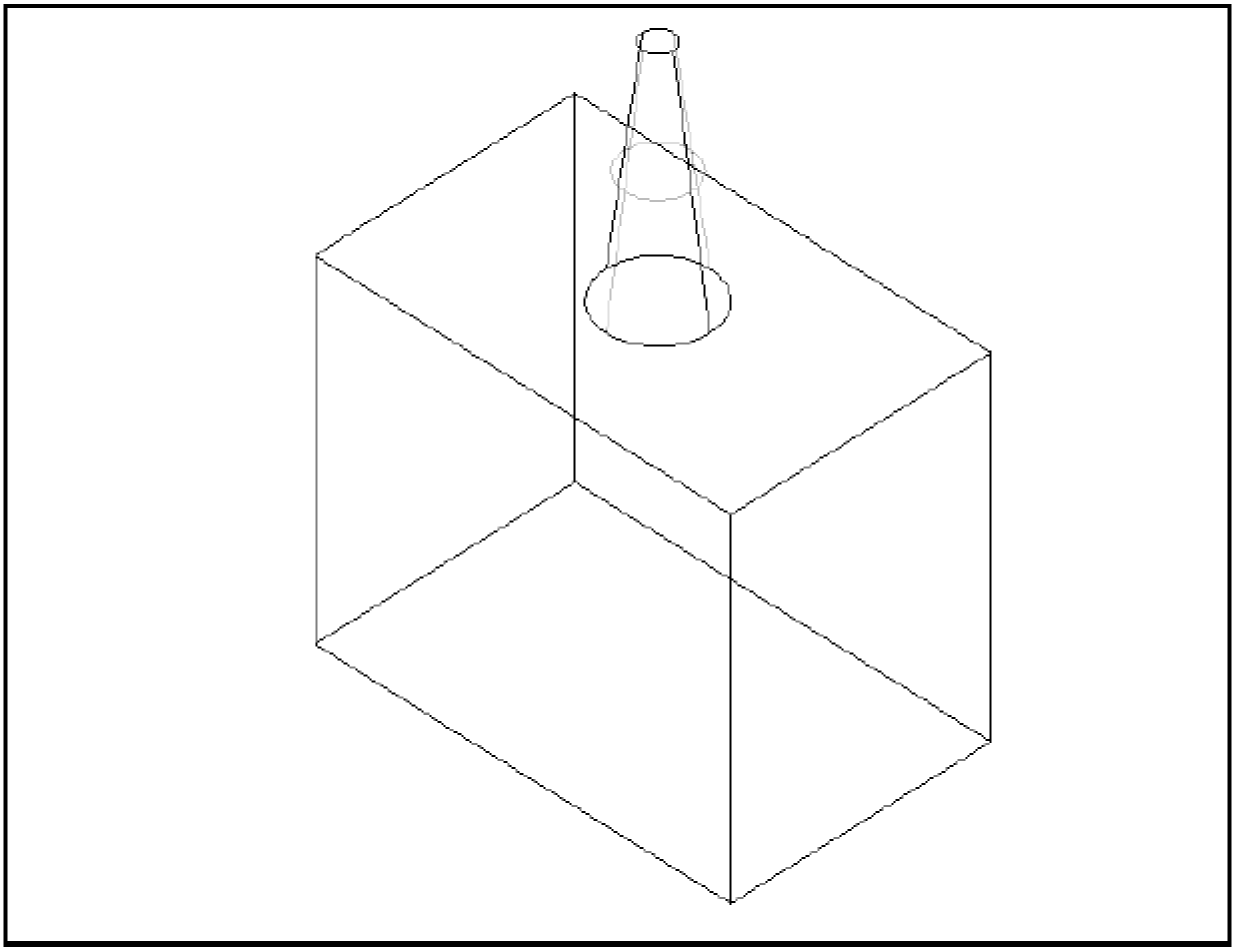


Figure 47. Creating a tapered prism

MakeRevol

The MakeRevol from BRepFeat class is used to build a revol interacting with a shape. It is created or initialized from

- a shape (the basic shape,)
- the base of the revol,
- a face (the face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
- an axis of revolution,
- a boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another boolean indicating whether the self-intersections have to be found

(not used in every case).

There are four Perform methods:

Perform(Angle)	The resulting revol is of the given magnitude.
Perform(Until)	The revol is defined between the actual position of the base and the given face.
Perform(From, Until)	The revol is defined between the two faces, From and Until.
PerformThruAll()	The result is similar to Perform(2*PI).

NOTE

The Add method can be used prior to using the Perform methods in order to indicate that a face generated by an edge "slides" onto a face of the basic shape.

In the following sequence, a face is revolved and the revol is limited by some face of the basis shape.

Example

```
TopoDS_Shape Sbase = ...;  // an initial shape
TopoDS_Face Frevol = ....; // a base of prism
TopoDS_Face FUntil = ....; // face limiting the revol

gp_Dir RevolDir (.,.,.);
gp_Ax1 RevolAx(gp_Pnt(.,.,.), RevolDir);

// An empty face is given as the sketch face

BRepFeat_MakeRevol theRevol(Sbase, Frevol, TopoDS_Face(),
            RevolAx, Standard_True, Standard_True);

theRevol.Perform(FUntil);
if (theRevol.IsDone()) {
      TopoDS_Shape theResult = theRevol;
      ...
}
```

MakePipe

Constructs compound shapes with pipe features. These can be depressions or protrusions. A class object is created or initialized from

- a shape (basic shape),
- a base face (profile of the pipe)
- a face (face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
- a spine wire
- a Boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another Boolean indicating if self-intersections have to be found (not used in every case).

There are three perform methods:

Perform()	The pipe is defined along the entire path (spine wire)
Perform(Until)	The pipe is defined along the path until a given face
Perform(From, Until)	The pipe is defined between the two faces From and Until.

Example

```
//MakePipe

TopoDS_Shape S = BRepPrimAPI_MakeBox(400.,250.,300.);
TopExp_Explorer Ex;
Ex.Init(S,TopAbs_FACE);
Ex.Next();
Ex.Next();
TopoDS_Face F1 = TopoDS::Face(Ex.Current());
Handle(Geom_Surface) surf = BRep_Tool::Surface(F1);
BRepBuilderAPI_MakeWire MW1;
```

```
gp_Pnt2d p1,p2;
p1 = gp_Pnt2d(100.,100.);
p2 = gp_Pnt2d(200.,100.);
Handle(Geom2d_Line) aline = GCE2d_MakeLine(p1,p2).Value();

MW1.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,p1.Distance(p
2)));
p1 = p2;
p2 = gp_Pnt2d(150.,200.);
aline = GCE2d_MakeLine(p1,p2).Value();

MW1.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,p1.Distance(p
2)));
p1 = p2;
p2 = gp_Pnt2d(100.,100.);
aline = GCE2d_MakeLine(p1,p2).Value();

MW1.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,p1.Distance(p
2)));
BRepBuilderAPI_MakeFace MKF1;
MKF1.Init(surf,Standard_False);
MKF1.Add(MW1.Wire());
TopoDS_Face FP = MKF1.Face();
BRepLib::BuildCurves3d(FP);
TColgp_Array1OfPnt CurvePoles(1,3);
gp_Pnt pt = gp_Pnt(150.,0.,150.);
CurvePoles(1) = pt;
pt = gp_Pnt(200.,100.,150.);
CurvePoles(2) = pt;
pt = gp_Pnt(150.,200.,150.);
CurvePoles(3) = pt;
Handle(Geom_BezierCurve) curve = new Geom_BezierCurve
      (CurvePoles);
TopoDS_Edge E = BRepBuilderAPI_MakeEdge(curve);
TopoDS_Wire W = BRepBuilderAPI_MakeWire(E);
BRepFeat_MakePipe MKPipe (S,FP,F1,W,Standard_False,
      Standard_True);
MKPipe.Perform();
TopoDS_Shape res1 = MKPipe.Shape();
```

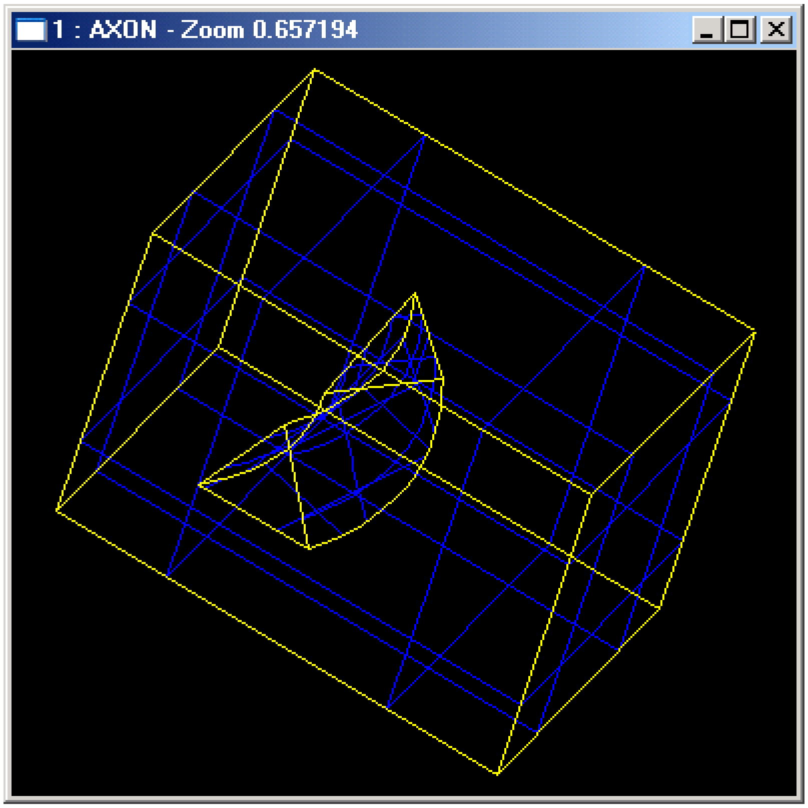


Figure 48. Creating a pipe depression

MakeLinearForm

Builds a rib or a groove along a developable, planar surface.

The semantics of mechanical features is built around giving thickness to a contour. This thickness can either be symmetrical - on one side of the contour - or dissymmetrical - on both sides. As in the semantics of form features, the thickness is defined by construction of shapes in specific contexts.

The development contexts differ, however, in the case of mechanical features.

Here they include extrusion:

- to a limiting face of the basis shape
- to or from a limiting plane
- to a height.

A class object is created or initialized from

- a shape (basic shape),

- a wire (base of rib or groove)
- a plane (plane of the wire)
- direction1 (a vector along which thickness will be built up)
- direction2 (vector opposite to the previous one along which thickness will be built up, may be null)
- a Boolean indicating the type of operation (fusion=rib or cut=groove) on the basic shape,
- another Boolean indicating if self-intersections have to be found (not used in every case).

There is one Perform method:

Perform()	Performs a prism from the wire along the direction1 and direction2 interacting basis shape Sbase. The height of the prism is Magnitude(Direction1)+Magnitude(direction2). Reconstructs the feature topologically

Example-

```
BRepBuilderAPI_MakeWire mkw;
gp_Pnt p1 = gp_Pnt(0.,0.,0.);
gp_Pnt p2 = gp_Pnt(200.,0.,0.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(200.,0.,50.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(50.,0.,50.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(50.,0.,200.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(0.,0.,200.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
mkw.Add(BRepBuilderAPI_MakeEdge(p2,gp_Pnt(0.,0.,0.)));
TopoDS_Shape S =
BRepBuilderAPI_MakePrism(BRepBuilderAPI_MakeFace
      (mkw.Wire()),gp_Vec(gp_Pnt(0.,0.,0.),gp_P
       nt(0.,100.,0.)));
TopoDS_Wire                        W                        =
BRepBuilderAPI_MakeWire(BRepBuilderAPI_MakeEdge(gp_Pnt
      (50.,45.,100.),
gp_Pnt(100.,45.,50.)));
Handle(Geom_Plane) aplane =
      new Geom_Plane(gp_Pnt(0.,45.,0.), gp_Vec(0.,1.,0.));
BRepFeat_MakeLinearForm aform(S, W, aplane, gp_Dir
      (0.,5.,0.), gp_Dir(0.,-3.,0.), 1, Standard_True);
aform.Perform();
TopoDS_Shape res = aform.Shape();
```

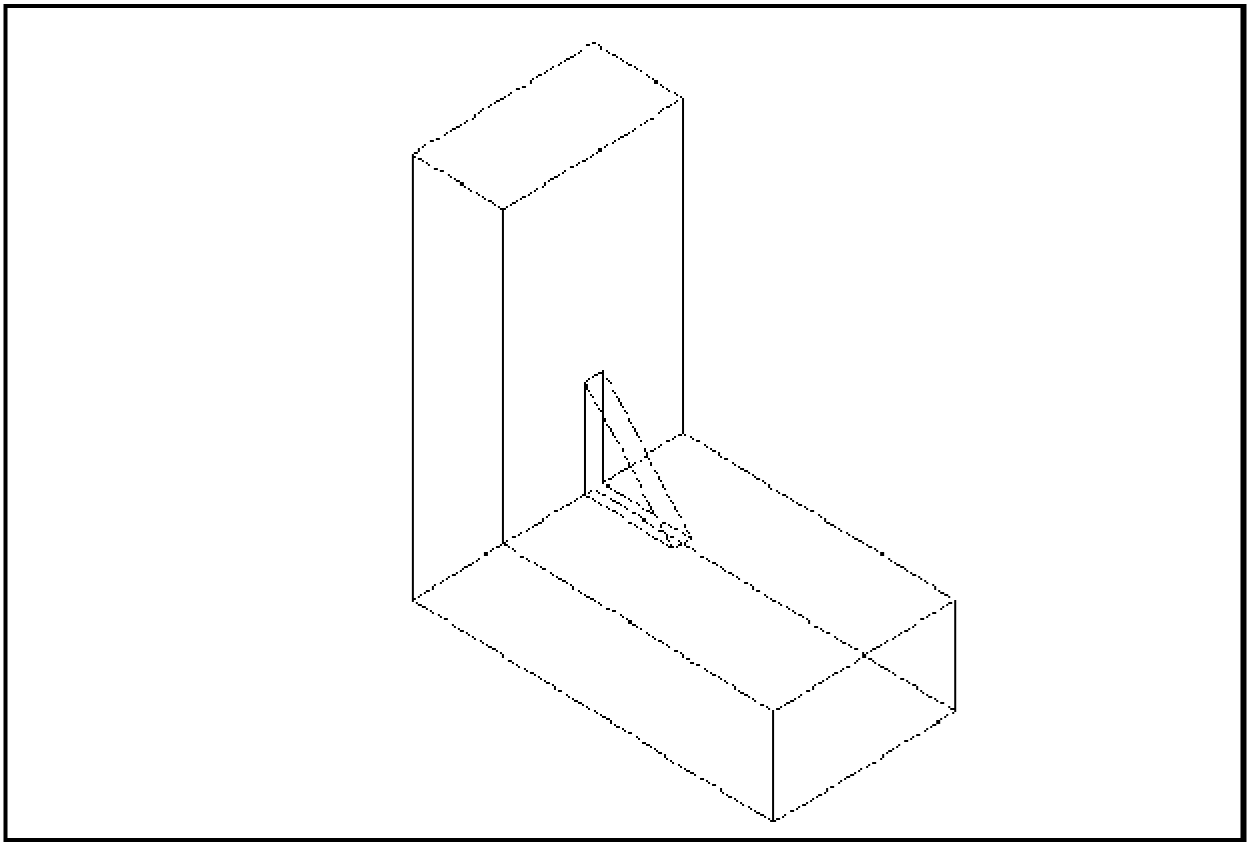


Figure 49. Creating a rib

9. 1. 2 The Gluer class

The Gluer from BRepFeat class is used to glue two solids along faces. The contact faces of the glued shape must not have parts outside the contact faces of the basic shape.

The class is created or initialized from two shapes: the “glued” shape and the basic shape (on which the other shape is glued).

Two Bind methods are used to bind a face of the glued shape to a face of the basic shape and an edge of the glued shape to an edge of the basic shape.

NOTE

Every face and edge has to be bounded, especially if two edges of two glued faces are coincident they must be explicitly bounded.

Example

```
TopoDS_Shape Sbase = ...; // the basic shape
TopoDS_Shape Sglued = ...; // the glued shape

TopTools_ListOfShape Lfbase;
TopTools_ListOfShape Lfglued;
// Determination of the glued faces
...

BRepFeat_Gluer theGlue(Sglue, Sbase);
TopTools_ListIteratorOfListOfShape itlb(Lfbase);
TopTools_ListIteratorOfListOfShape itlg(Lfglued);
for (; itlb.More(); itlb.Next(), itlg(Next()) {
      const TopoDS_Face& f1 = TopoDS::Face(itlg.Value());
      const TopoDS_Face& f2 = TopoDS::Face(itlb.Value());
      theGlue.Bind(f1,f2);
      // for example, use the class FindEdges from LocOpe
      to
      // determine coincident edges
      LocOpe_FindEdge fined(f1,f2);
      for       (fined.InitIterator();       fined.More();
      fined.Next()) {
      theGlue.Bind(fined.EdgeFrom(),fined.EdgeTo());
      }
}
theGlue.Build();
if (theGlue.IsDone() {
      TopoDS_Shape theResult = theGlue;
...
}
```

9. 1. 3 The SplitShape Class

The SplitShape from BRepFeat class is used to split faces of a shape with wires or edges. The shape containing the new entities is rebuilt, sharing the unmodified ones.

The class is created or initialized from a shape (the basic shape).

Three Add methods are available:

Add(Wire, Face)	Adds a new wire on a face of the basic shape.
Add(Edge, Face)	Adds a new edge on a face of the basic shape.
Add(EdgeNew, EdgeOld)	Adds a new edge on an existing one (the old edge must contain the new edge).

NOTE

The added wires and edges must define closed wires on faces or wires located between two existing edges. Existing edges must not be intersected.

Example

```
TopoDS_Shape Sbase = ...; // basic shape
TopoDS_Face Fsplit = ...; // face of Sbase
TopoDS_Wire Wsplit = ...; // new wire contained in Fsplit
BRepFeat_SplitShape Spls(Sbase);
Spls.Add(Wsplit, Fsplit);
TopoDS_Shape theResult = Spls;
...
```

10. Hidden Line Removal

10. 1 Overview

In order to have the precision required in industrial design, drawings need to offer the possibility of removing lines, which are hidden in a given projection.

To do this, the Hidden Line Removal component provides two algorithms:

`HLRBRep_Algo and HLRBRep_PolyAlgo.`

These two algorithms are based on the principle of comparing each edge of the shape to be visualized with each of its faces, and calculating the visible and the hidden parts of each edge. Note that these are not the sort of algorithms used in generating shading, which calculate the visible and hidden parts of each face in a shape to be visualized by comparing each face in the shape with every other face in the same shape.

These algorithms operate on a shape and remove or indicate edges hidden by faces. For a given projection, they calculate a set of lines characteristic of the object being represented. They are also used in conjunction with extraction utilities, which reconstruct a new, simplified shape from a selection of the results of the calculation. This new shape is made up of edges, which represent the shape visualized in the projection.

`HLRBRep_Algo` takes the shape itself into account whereas `HLRBRep_PolyAlgo` works with a polyhedral simplification of the shape. When you use `HLRBRep_Algo`, you obtain an exact result, whereas, when you use `HLRBRep_PolyAlgo`, you reduce computation time.

No smoothing algorithm is provided. Consequently, a polyhedron will be treated as such and the algorithms will give the results in the form of line segments conforming to the mathematical definition of the polyhedron. This is always the case with `HLRBRep_PolyAlgo`.

`HLRBRep_Algo` and `HLRBRep_PolyAlgo` can deal with any kind of object - assemblies of volumes, surfaces, and lines, for example - as long there are no unfinished objects or points within it.

You can choose to display the following types of line if they are present in the projection:

- sharp edges
- smooth edges - transition edges between two surfaces
- sewn edges - a double edge where a face in a periodic surface is sewn together.

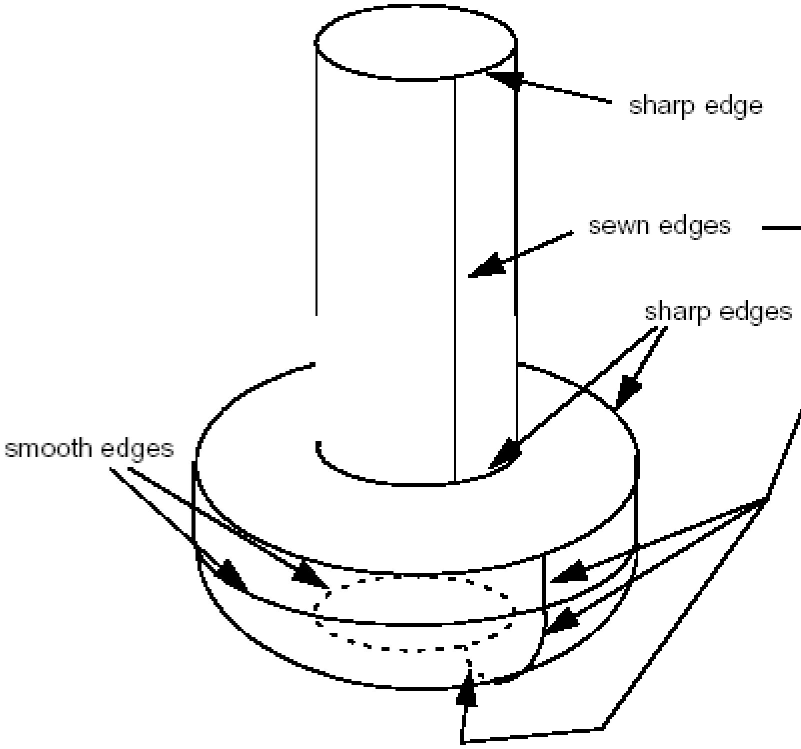


Figure 50. sharp, smooth and sewn edges in a simple screw shape

- isoparameters
- outlines (edges added to the topology in order to represent the contours visible in a particular projection)

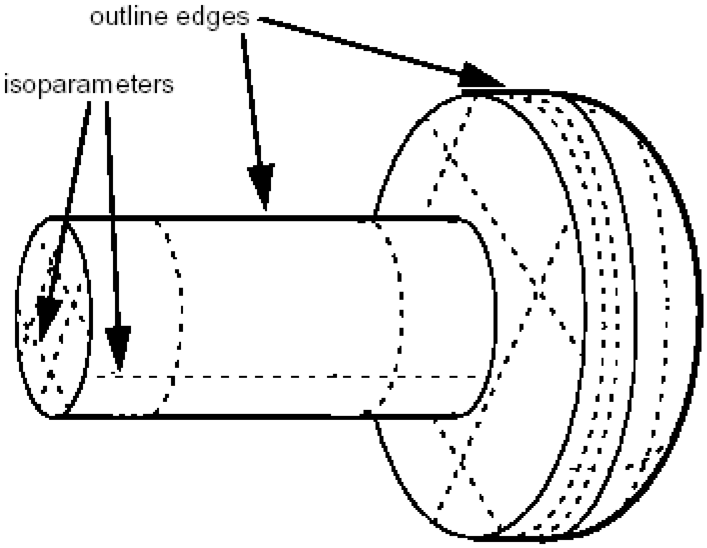

outline edges
isoparameters

Figure 51. outline edges and isoparameters in the same shape

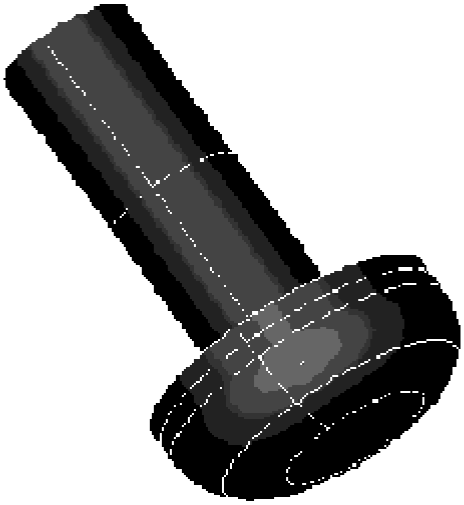


Figure 52. A simple screw shape seen with shading

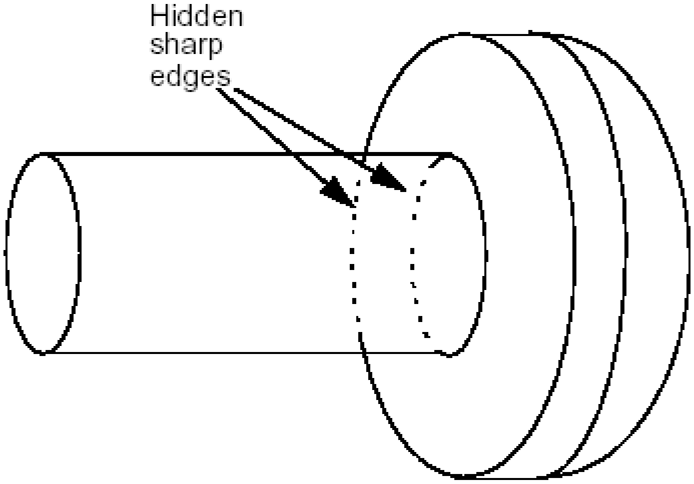


Figure 53. An extraction showing hidden sharp edges

10. 2 The Services Provided

10. 2. 1 HLRBRep

The `HLRBRep` package provides the following services:

Loading shapes to be treated

`HLRBRep_Algo` and `HLRBRep_PolyAlgo` provide algorithms for removal of hidden lines. `HLRBRep_Algo` operates on a shape with isoparameters; it takes the shape itself into account whereas `HLRBRep_PolyAlgo` works with a polyhedral simplification of the shape.

To do pass a `TopoDS_Shape` to an `HLRBRep_Algo` object, use `HLRBRep_Algo::Add`. With an `HLRBRep_PolyAlgo` object, use `HLRBRep_PolyAlgo::Load`. If you wish to add several shapes, use `Add` or `Load` as often as necessary.

Setting the view parameters

`HLRBRep_Algo::Projector` and `HLRBRep_PolyAlgo::Projector` set a projector object which defines the parameters of the view. This object is an `HLRAlgo_Projector`.

Computing the projections

`HLRBRep_PolyAlgo::Update` launches the calculation of outlines of the shape visualized by the `HLRBRep_PolyAlgo` framework.

In the case of `HLRBRep_Algo`, use `HLRBRep_Algo::Update`. With this algorithm, you must also call the method `HLRBRep_Algo::Hide` to calculate visible and hidden lines of the shape to be visualized. With an `HLRBRep_PolyAlgo` object, visible and hidden lines are computed by `HLRBRep_PolyHLRToShape` (see below).

Extracting edges.

The classes `HLRBRep_HLRToShape` and `HLRBRep_PolyHLRToShape` present a range of extraction filters for an `HLRBRep_Algo` object and an `HLRBRep_PolyAlgo` object, respectively. They highlight the type of edge you want from the results calculated by the algorithm on a shape. With both extraction classes, you can highlight the following types of output:

- visible sharp edges
- hidden sharp edges
- visible smooth edges
- hidden smooth edges
- visible sewn edges
- hidden sewn edges
- visible outline edges
- hidden outline edges.

To perform the extraction on an `HLRBRep_PolyHLRToShape` object, use the `HLRBRep_PolyHLRToShape::Update` function.

In an `HLRBRep_HLRToShape` object, built from an `HLRBRepAlgo` object, you can also highlight:

- visible isoparameters and
- hidden isoparameters.

9. 2. 2 Restrictions in use

- Points are not treated
- Z-clipping planes are not used
- Infinite faces or lines are not treated.

10. 3 Examples of Use

10. 3. 1 HLRBRep_Algo

Example

```
// Build The algorithm object
myAlgo = new HLRBRep_Algo();

// Add Shapes into the algorithm
TopTools_ListIteratorOfListOfShape anIterator(myListOfShape);
for (;anIterator.More();anIterator.Next())
myAlgo->Add(anIterator.Value(),myNbIsos);

// Set The Projector (myProjector is a
HLRAlgo_Projector)
myAlgo->Projector(myProjector);

// Build HLR
myAlgo->Update();

// Set The Edge Status
myAlgo->Hide();

// Build the extraction object :
HLRBRep_HLRToShape aHLRToShape(myAlgo);
```

```
// extract the results :
TopoDS_Shape VCompound          = aHLRToShape.VCompound();
TopoDS_Shape Rg1LineVCompound                          =
aHLRToShape.Rg1LineVCompound();
TopoDS_Shape RgNLineVCompound                          =
aHLRToShape.RgNLineVCompound();
TopoDS_Shape OutLineVCompound                          =
aHLRToShape.OutLineVCompound();
TopoDS_Shape IsoLineVCompound                          =
aHLRToShape.IsoLineVCompound();
TopoDS_Shape HCompound          = aHLRToShape.HCompound();
TopoDS_Shape Rg1LineHCompound                          =
aHLRToShape.Rg1LineHCompound();
TopoDS_Shape RgNLineHCompound                          =
aHLRToShape.RgNLineHCompound();
TopoDS_Shape OutLineHCompound                          =
aHLRToShape.OutLineHCompound();
TopoDS_Shape IsoLineHCompound                          =
aHLRToShape.IsoLineHCompound();
```

10. 3. 2 HLRBRep_PolyAlgo

Example

```
  // Build The algorithm object
  myPolyAlgo = new HLRBRep_PolyAlgo();

  // Add Shapes into the algorithm
  TopTools_ListIteratorOfListOfShape
  anIterator(myListOfShape);
  for (;anIterator.More();anIterator.Next())
  myPolyAlgo->Load(anIterator.Value());

  // Set The Projector (myProjector is a
  HLRAlgo_Projector)
```

```
myPolyAlgo->Projector(myProjector);

// Build HLR
myPolyAlgo->Update();

// Build the extraction object :
HLRBRep_PolyHLRToShape aPolyHLRToShape;
aPolyHLRToShape.Update(myPolyAlgo);

// extract the results :
TopoDS_Shape VCompound =
aPolyHLRToShape.VCompound();
TopoDS_Shape Rg1LineVCompound =
aPolyHLRToShape.Rg1LineVCompound();
TopoDS_Shape RgNLineVCompound =
aPolyHLRToShape.RgNLineVCompound();
TopoDS_Shape OutLineVCompound =
aPolyHLRToShape.OutLineVCompound();
TopoDS_Shape HCompound =
aPolyHLRToShape.HCompound();
TopoDS_Shape Rg1LineHCompound =
aPolyHLRToShape.Rg1LineHCompound();
TopoDS_Shape RgNLineHCompound =
aPolyHLRToShape.RgNLineHCompound();
TopoDS_Shape OutLineHCompound =
aPolyHLRToShape.OutLineHCompound();
```

10. Meshing of Shapes

The HLRBRep_PolyAlgo algorithm works with triangulation of shapes. This is provided by the function BRepMesh::Mesh, which adds a triangulation of the shape to its topological data structure. This triangulation is computed with a given deflection.

Example

```
Standard_Real radius=10. ;
Standard_Real height=25. ;
BRepBuilderAPI_MakeCylinder myCyl (radius, height) ;
TopoDS_Shape myShape = myCyl.Shape() ;
Standard_Real Deflection = 0.01 ;
BRepMesh::Mesh (myShape, Deflection);
```

Meshing covers a shape with a triangular mesh. Other than hidden line removal, you can use meshing to transfer the shape to another tool: a manufacturing tool, a shading algorithm, a finite element algorithm, or a collision algorithm, for example.

You can obtain information on the shape by first exploring it. To then access triangulation of a face in the shape, use BRepTool::Triangulation. To access a polygon which is the approximation of an edge of the face, use BRepTool::PolygonOnTriangulation.