PythonOCC 0.18.2 Documentation
Topology and Geometry Modeling

Thomas Paviot - tpaviot@gmail.com
originally written by the OpenCASCADE company

Document version 0 - October 15, 2019

Contents

1 Introduction

1.1 About this document

Modeling Data supplies data structures to represent 2D and 3D geometric models.

1.2 Credits and license

This manual was originally written by the Open CASCADE company. They're the dev team that developed OCC Technology (OCCT), the underlying c++ layer on which PythonOCC ia based.

This manual contains additions and modifications to fit with python specific syntax, and PythonOCC usage in general. However, the basic concepts are the one available from OCCT. If you need further details related to OCC Technology, be aware that they offer commercial support and trainings, just check their E-learning & Training offerings. For your information, there is not any commercial agreement between the OCC Company and PythonOCC development teams.

1.3 Online resources

If you need PythonOCC specific help, please refer to the following online resources:

- PythonOCC source code repository https://github.com/tpaviot/PythonOCC-core
- Mailing list http://groups.google.com/group/PythonOCC

Finally, email {tpaviot@gmail.com} for any other request.

1.4 Typographic conventions

python code snippets use: lower_case convention for variable names and functions, CamelCase for classes and methods. Example:

```
1 def create_box(): # this is a function
2   my_box = BRepPrimAPI_MakeBox(10, 20, 30) # a variable created from a Class
3   return my_box.Shape() # calling a .Method
4
5 a_box_shape = create_box()
```

In each code snippet, we assume that all required modules are loaded. For instance, in order to run the previous snippet, you have to import the BRepPrimAPI_Box class from the BRepPrimAPI module just before:

```
1 from OCC.BRepPrimAPI import BRepPrimAPI_MakeBox
```

1.5 Contribute improving this document

The source code for this document is available in Doxygen mardkown format. Feel free to report issues, mistakes or submit patches using the issue tracker or pull request feature at:

https://github.com/tpaviot/PythonOCC-documentation.

2 Topology

PythonOCC Topology allows accessing and manipulating data of objects without dealing with their 2D or 3D representations. Whereas PythonOCC Geometry provides a description of objects in terms of coordinates or parametric values, Topology describes data structures of objects in parametric space. These descriptions use location in and restriction of parts of this space.

Topological library allows you to build pure topological data structures. Topology defines relationships between simple geometric entities. In this way, you can model complex shapes as assemblies of simpler entities. Due to a built-in non-manifold (or mixed-dimensional) feature, you can build models mixing:

- 0D entities such as points,
- 1D entities such as curves,
- 2D entities such as surfaces,
- 3D entities such as volumes.

You can, for example, represent a single object made of several distinct bodies containing embedded curves and surfaces connected or non-connected to an outer boundary.

Abstract topological data structure describes a basic entity - a shape, which can be divided into the following component topologies:

- vertex - a zero-dimensional shape corresponding to a point in geometry,
- edge - a shape corresponding to a curve, and bound by a vertex at each extremity,
- wire - a sequence of edges connected by their vertices,
- face - part of a plane (in 2D geometry) or a surface (in 3D geometry) bounded by a closed wire,
- shell - a collection of faces connected by some edges of their wire boundaries,
- solid - a part of 3D space bound by a shell,
- compound solid - a collection of solids.

The wire and the solid can be either infinite or closed.

A face with 3D underlying geometry may also refer to a collection of connected triangles that approximate the underlying surface. The surfaces can be undefined leaving the faces represented by triangles only. If so, the model is purely polyhedral.

Topology defines the relationship between simple geometric entities, which can thus be linked together to represent complex shapes.

Abstract Topology is provided by six modules. The first three modules describe the topological data structure used in PythonOCC:

- *TopAbs* module provides general resources for topology-driven applications. It contains enumerations that are used to describe basic topological notions: topological shape, orientation and state. It also provides methods to manage these enumerations,
- *TopLoc* module provides resources to handle 3D local coordinate systems: *Datum3D*∗*and* ∗*Location*. *Datum3D* describes an elementary coordinate system, while *Location* comprises a series of elementary coordinate systems,
- *TopoDS* module describes classes to model and build data structures that are purely topological.

Three additional modules provide tools to access and manipulate this abstract topology:

- *TopTools* module provides basic tools to use on topological data structures,

- *TopExp* module provides classes to explore and manipulate the topological data structures described in the TopoDS module,
- *BRepTools* module provides classes to explore, manipulate, read and write BRep data structures. These more complex data structures combine topological descriptions with additional geometric information, and include rules for evaluating equivalence of different possible representations of the same object, for example, a point.

2.1 Shape Location

A local coordinate system can be viewed as either of the following:

- a right-handed trihedron with an origin and three orthonormal vectors. The *gp_Ax2* module corresponds to this definition,
- a transformation of a +1 determinant, allowing the transformation of coordinates between local and global references frames. This corresponds to the *gp_Trsf*.

TopLoc module distinguishes two notions:

- *TopLoc_Datum3D* class provides the elementary reference coordinate, represented by a right-handed orthonormal system of axes or by a right-handed unitary transformation,
- *TopLoc_Location* class provides the composite reference coordinate made from elementary ones. It is a marker composed of a chain of references to elementary markers. The resulting cumulative transformation is stored in order to avoid recalculating the sum of the transformations for the whole list.

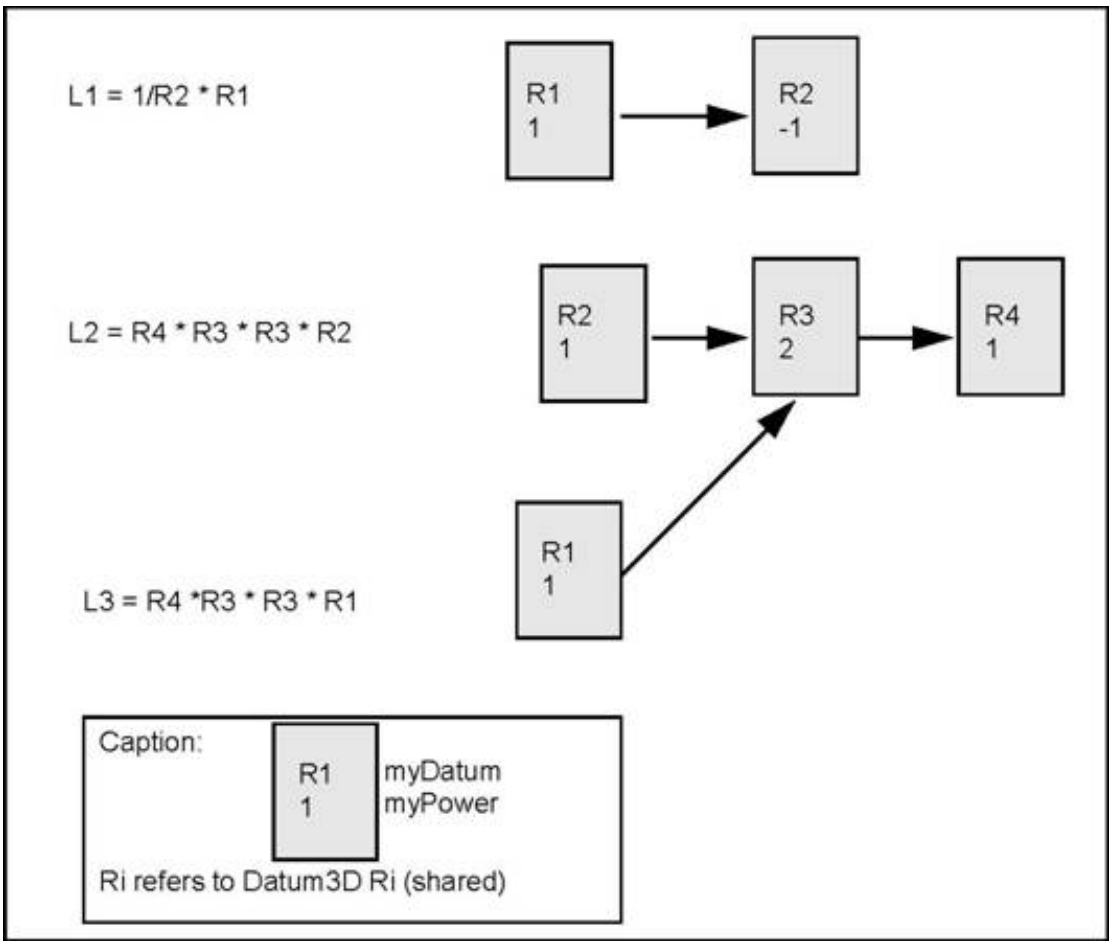

Figure 1: Structure of TopLoc_Location

Two reference coordinates are equal if they are made up of the same elementary coordinates in the same order. There is no numerical comparison. Two coordinates can thus correspond to the same transformation without being equal if they were not built from the same elementary coordinates.

For example, consider three elementary coordinates $R1, R2, R3$. The composite coordinates are:

$C1 = R1 \times R2$

$C2 = R2 \times R3$

$C3 = C1 \times R3$

$C4 = R1 \times C2$

NOTE $C3$ and $C4$ are equal because they are both $R1 \times R2 \times R3$.

The *TopLoc* module is chiefly targeted at the topological data structure, but it can be used for other purposes.

Change of coordinates

TopLoc_Datum3D class represents a change of elementary coordinates. Such changes must be shared so this class inherits from *MMgt_TShared*. The coordinate is represented by a transformation *gp_Trsfmodule*. This transformation has no scaling factor.

2.2 Naming shapes, sub-shapes, their orientation and state

The *TopAbs* module provides general enumerations describing the basic concepts of topology and methods to handle these enumerations. It contains no classes. This module has been separated from the rest of the topology because the notions it contains are sufficiently general to be used by all topological tools. This avoids redefinition of enumerations by remaining independent of modeling resources. The TopAbs module defines three notions:

- the shape type: *TopAbs_ShapeEnum*,
- the shape orientation: *TopAbs_Orientation*,
- the shape state: *StateTopAbs_State* .

2.2.1 Topological types

TopAbs contains the *TopAbs_ShapeEnum* enumeration,which lists the different topological types:

- TopAbs_COMPOUND: a group of any type of topological objects,
- TopAbs_COMPSOLID: a composite solid is a set of solids connected by their faces. It expands the notions of WIRE and SHELL to solids,
- TopAbs_SOLID: a part of space limited by shells. It is three dimensional,
- TopAbs_SHELL: a set of faces connected by their edges. A shell can be open or closed,
- TopAbs_FACE: in 2D it is a part of a plane; in 3D it is a part of a surface. Its geometry is constrained (trimmed) by contours. It is two dimensional,
- TopAbs_WIRE: a set of edges connected by their vertices. It can be an open or closed contour depending on whether the edges are linked or not,
- TopAbs_EDGE: a topological element corresponding to a restrained curve. An edge is generally limited by vertices. It has one dimension,
- TopAbs_VERTEX: a topological element corresponding to a point. It has zero dimension,
- TopAbs_SHAPE: a generic term covering all of the above.

A topological model can be considered as a graph of objects with adjacency relationships. When modeling a part in 2D or 3D space it must belong to one of the categories listed in the ShapeEnum enumeration. The TopAbsmodule lists all the objects, which can be found in any model. It cannot be extended but a subset can be used. For example, the notion of solid is useless in 2D.

The terms of the enumeration appear in order from the most complex to the most simple, because objects can contain simpler objects in their description. For example, a face references its wires, edges, and vertices.

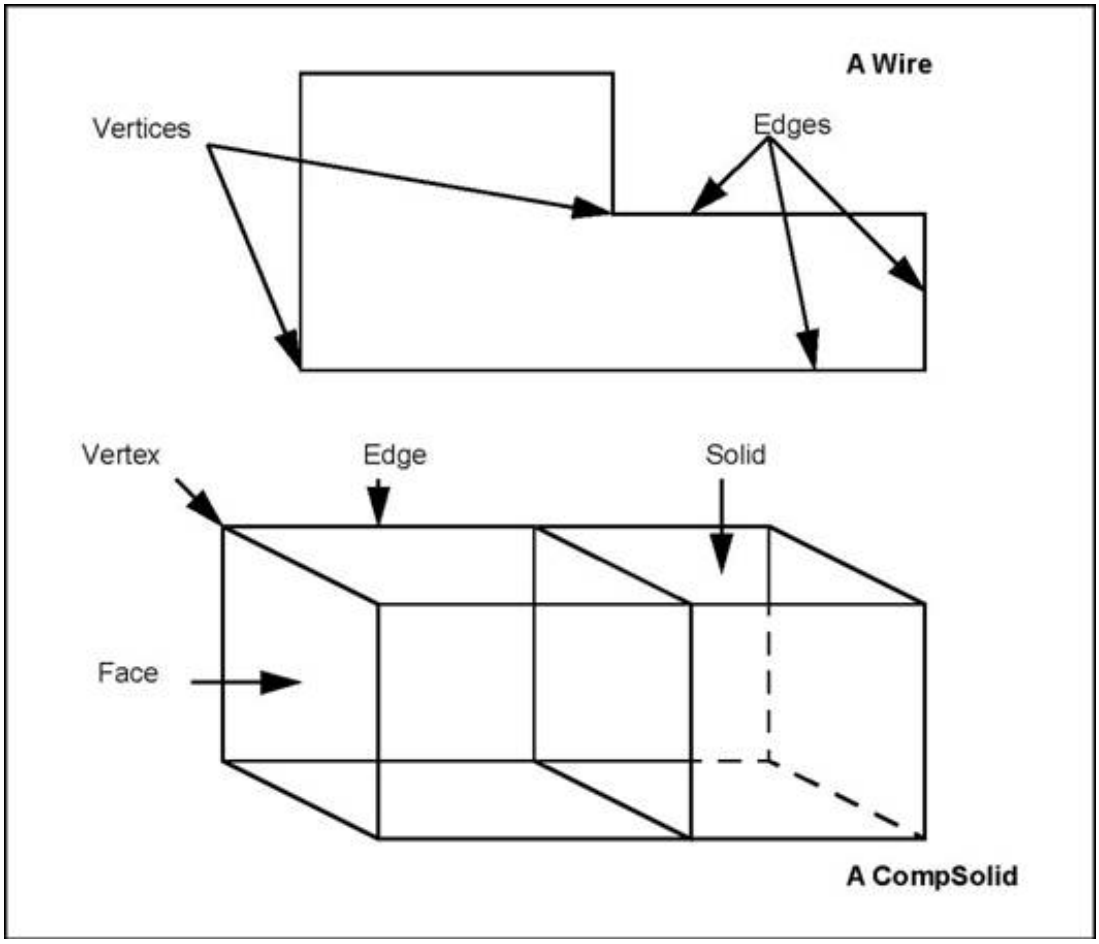

Figure 2: ShapeEnum

2.2.2 Orientation

The notion of orientation is represented by the *TopAbs_Orientation* enumeration. Orientation is a generalized notion of the sense of direction found in various modelers. This is used when a shape limits a geometric domain; and is closely linked to the notion of boundary. The three cases are the following:

- curve limited by a vertex,
- surface limited by an edge,
- space limited by a face.

In each case the topological form used as the boundary of a geometric domain of a higher dimension defines two local regions of which one is arbitrarily considered as the *default region*.

For a curve limited by a vertex the default region is the set of points with parameters greater than the vertex. That is to say it is the part of the curve after the vertex following the natural direction along the curve.

For a surface limited by an edge the default region is on the left of the edge following its natural direction. More precisely it is the region pointed to by the vector product of the normal vector to the surface and the vector tangent to the curve.

For a space limited by a face the default region is found on the negative side of the normal to the surface.

Based on this default region the orientation allows definition of the region to be kept, which is called the *interior* or *material*. There are four orientations defining the interior.

Orientation	Description
TopAbs_FORWARD	The interior is the default region.
TopAbs_REVERSED	The interior is the region complementary to the default.
TopAbs_INTERNAL	The interior includes both regions. The boundary lies inside the material. For example a surface inside a solid.
TopAbs_EXTERNAL	The interior includes neither region. The boundary lies outside the material. For example an edge in a wire-frame model.

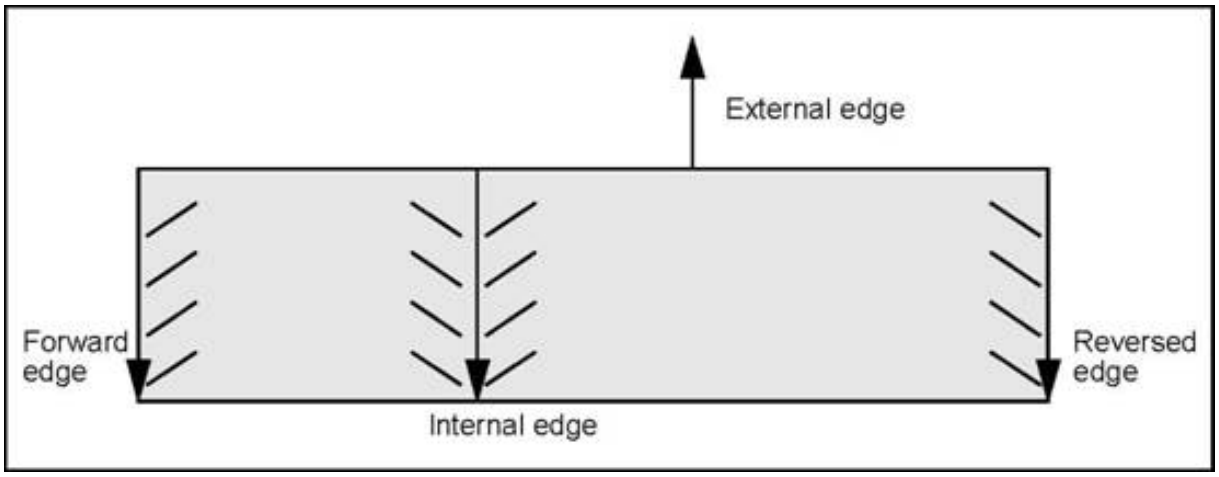

Figure 3: Four Orientations

The notion of orientation is a very general one, and it can be used in any context where regions or boundaries appear. Thus, for example, when describing the intersection of an edge and a contour it is possible to describe not only the vertex of intersection but also how the edge crosses the contour considering it as a boundary. The edge would therefore be divided into two regions - exterior and interior - with the intersection vertex as the boundary. Thus an orientation can be associated with an intersection vertex as in the following figure:

Orientation	Association
TopAbs_FORWARD	Entering
TopAbs_REVERSED	Exiting
TopAbs_INTERNAL	Touching from inside
TopAbs_EXTERNAL	Touching from outside

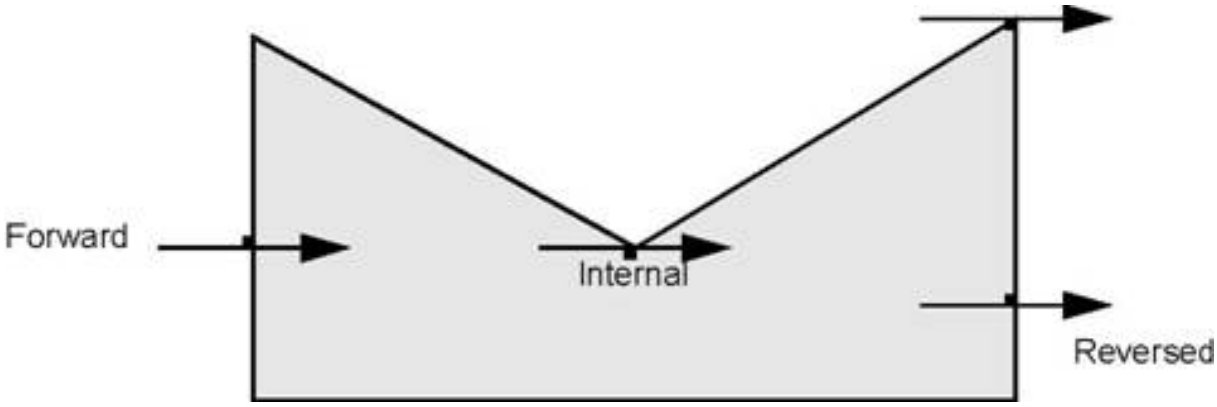

Figure 4: Four orientations of intersection vertices

Along with the Orientation enumeration the *TopAbs* module defines four methods:

- *topabs_Complement*,
- *topabs_Compose*,
- *topabs_Print*,
- *topabs_Reverse*.

2.2.3 State

The *TopAbs_State* enumeration described the position of a vertex or a set of vertices with respect to a region. There are four terms:

Position	Description
TopAbs_IN	The point is interior.
TopAbs_OUT	The point is exterior.
TopAbs_ON	The point is on the boundary(within tolerance).
TopAbs_UNKNOWN	The state of the point is indeterminate.

The UNKNOWN term has been introduced because this enumeration is often used to express the result of a calculation, which can fail. This term can be used when it is impossible to know if a point is inside or outside, which is the case with an open wire or face.

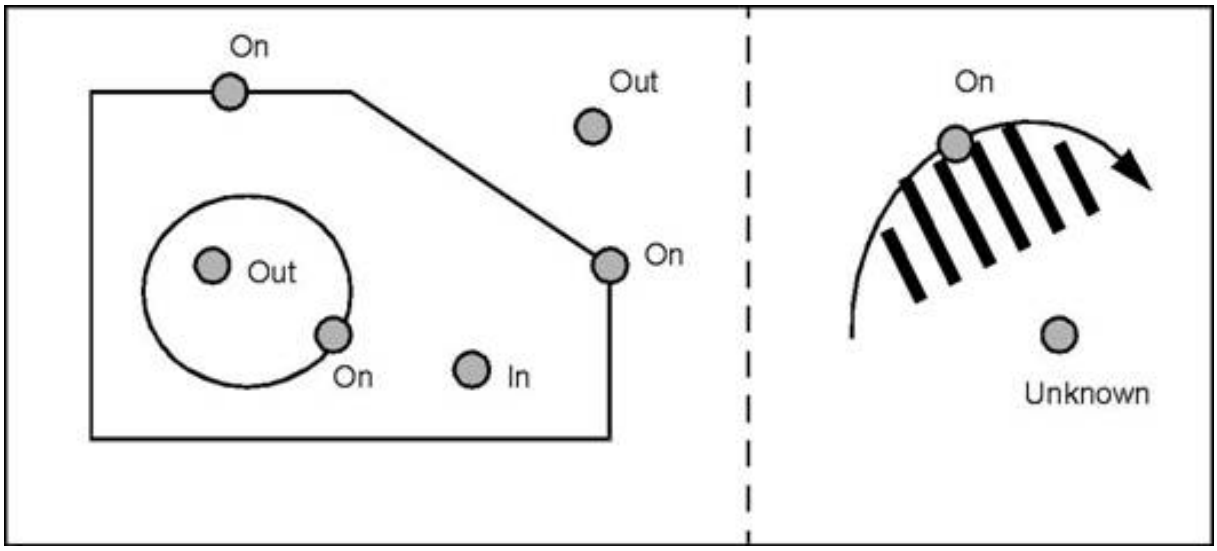

Figure 5: The four states

The State enumeration can also be used to specify various parts of an object. The following figure shows the parts of an edge intersecting a face.

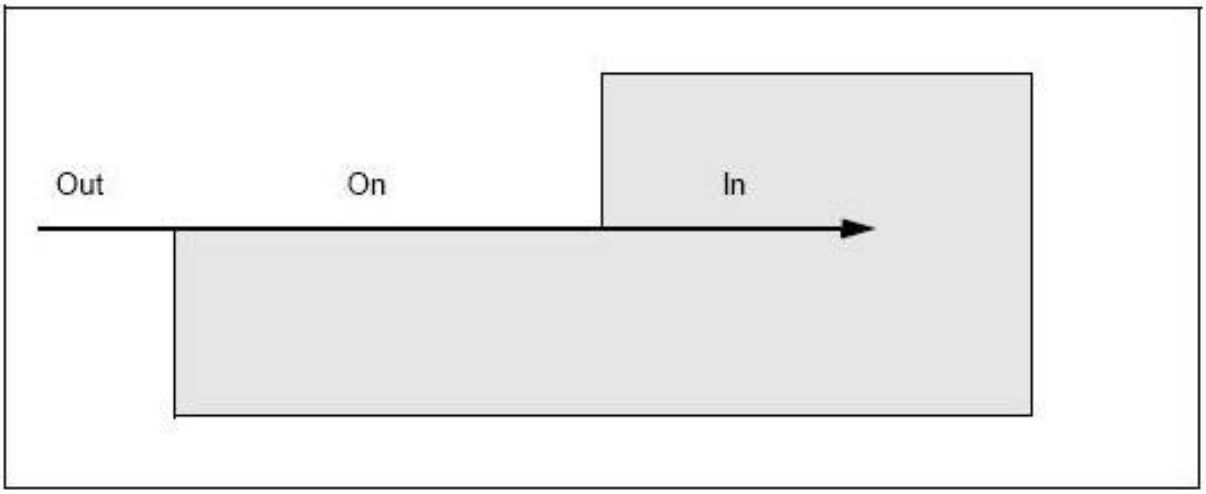

Figure 6: State specifies the parts of an edge intersecting a face

2.3 Manipulating shapes and sub-shapes

The *TopoDS* module describes the topological data structure with the following characteristics:

- reference to an abstract shape with neither orientation nor location,
- access to the data structure through the tool classes.

As stated above, PythonOCC Topology describes data structures of objects in parametric space. These descriptions use localization in and restriction of parts of this space. The types of shapes, which can be described in these terms, are the vertex, the face and the shape. The vertex is defined in terms of localization in parametric space, and the face and shape, in terms of restriction of this space.

PythonOCC topological descriptions also allow the simple shapes defined in these terms to be combined into sets. For example, a set of edges forms a wire; a set of faces forms a shell, and a set of solids forms a composite solid

(CompSolid in PythonOCC). You can also combine shapes of either sort into compounds. Finally, you can give a shape an orientation and a location.

Listing shapes in order of complexity from vertex to composite solid leads us to the notion of the data structure as knowledge of how to break a shape down into a set of simpler shapes. This is in fact, the purpose of the *TopoDS* module.

The model of a shape is a shareable data structure because it can be used by other shapes. (An edge can be used by more than one face of a solid). A shareable data structure is handled by reference. When a simple reference is insufficient, two pieces of information are added - an orientation and a local coordinate reference.

- an orientation tells how the referenced shape is used in a boundary (*TopAbs_Orientation*),
- a local reference coordinate (*TopLoc_Location*) allows referencing a shape at a position different from that of its definition.

The *TopoDS_TShape* class is the root of all shape descriptions. It contains a list of shapes. Classes inheriting *TopoDS_TShape* can carry the description of a geometric domain if necessary (for example, a geometric point associated with a TVertex). A *TopoDS_TShape* is a description of a shape in its definition frame of reference. This class is manipulated by reference.

The *TopoDS_Shape* class describes a reference to a shape. It contains a reference to an underlying abstract shape, an orientation,and a local reference coordinate. This class is manipulated by value and thus cannot be shared.

The class representing the underlying abstract shape is never referenced directly. The *TopoDS_Shape* class is always used to refer to it.

The information specific to each shape (the geometric support) is always added by inheritance to classes deriving from *TopoDS_TShape*. The following figures show the example of a shell formed from two faces connected by an edge.

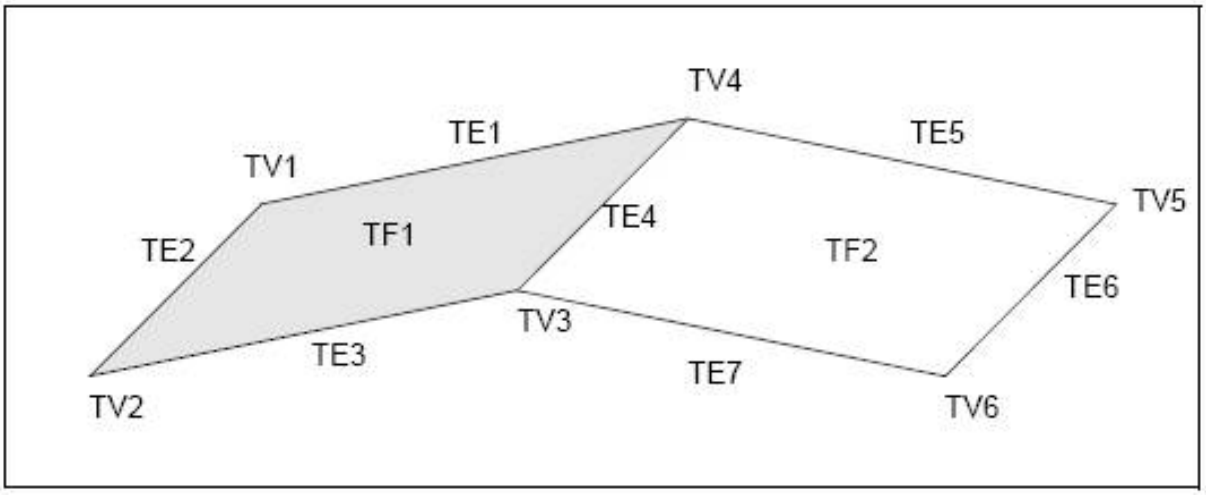

Figure 7: Structure of a shell formed from two faces

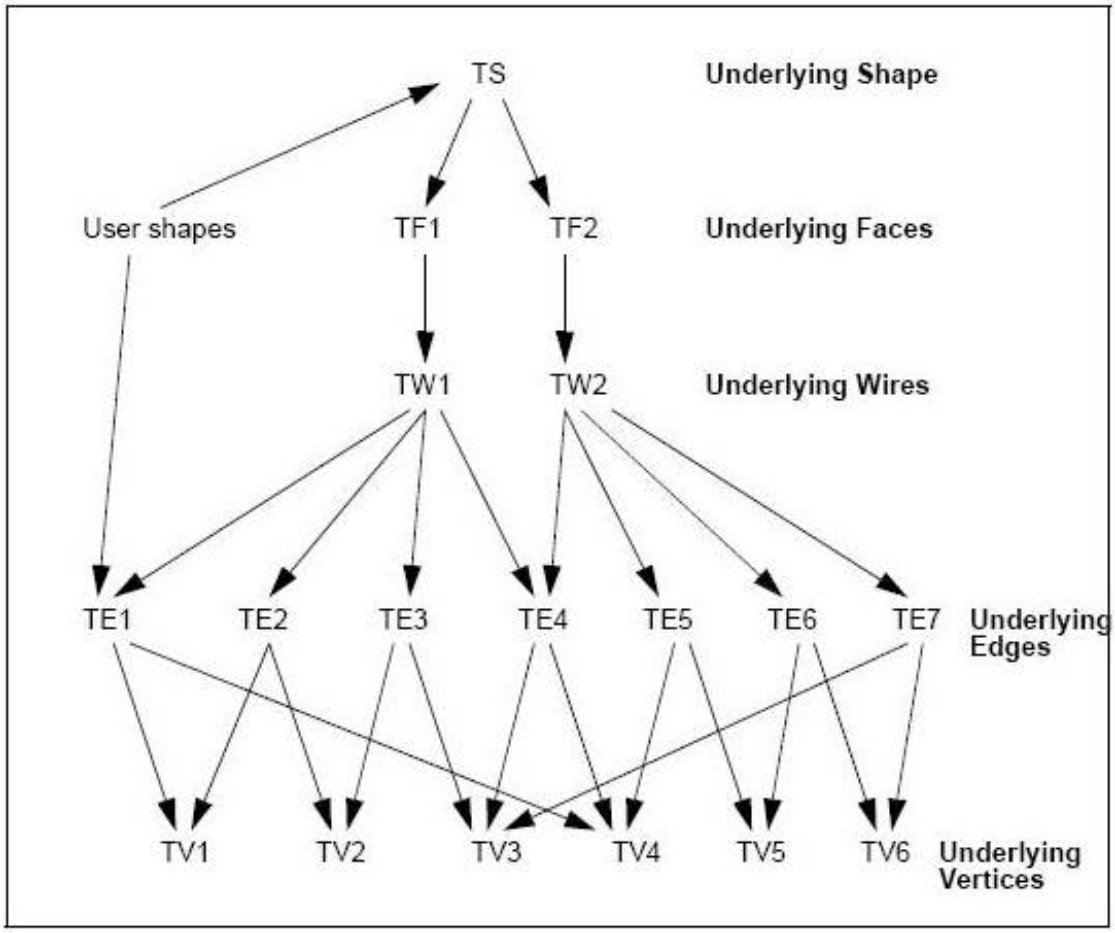

Figure 8: Data structure of the above shell

In the previous diagram, the shell is described by the underlying shape TS, and the faces by TF1 and TF2. There are seven edges from TE1 to TE7 and six vertices from TV1 to TV6.

The wire TW1 references the edges from TE1 to TE4; TW2 references from TE4 to TE7.

The vertices are referenced by the edges as follows:TE1(TV1,TV4), TE2(TV1,TV2), TE3(TV2,TV3), TE4(TV3,TV4), TE5(TV4,TV5), TE6(T5,TV6),TE7(TV3,TV6).

Note: this data structure does not contain any *back references*. All references go from more complex underlying shapes to less complex ones. The techniques used to access the information are described later. The data structure is as compact as possible. Sub-objects can be shared among different objects.

Two very similar objects, perhaps two versions of the same object, might share identical sub-objects. The usage of local coordinates in the data structure allows the description of a repetitive sub-structure to be shared.

The compact data structure avoids the loss of information associated with copy operations which are usually used in creating a new version of an object or when applying a coordinate change.

The following figure shows a data structure containing two versions of a solid. The second version presents a series of identical holes bored at different positions. The data structure is compact and yet keeps all information on the sub-elements.

The three references from *TSh2* to the underlying face *TFcyl* have associated local coordinate systems, which correspond to the successive positions of the hole.

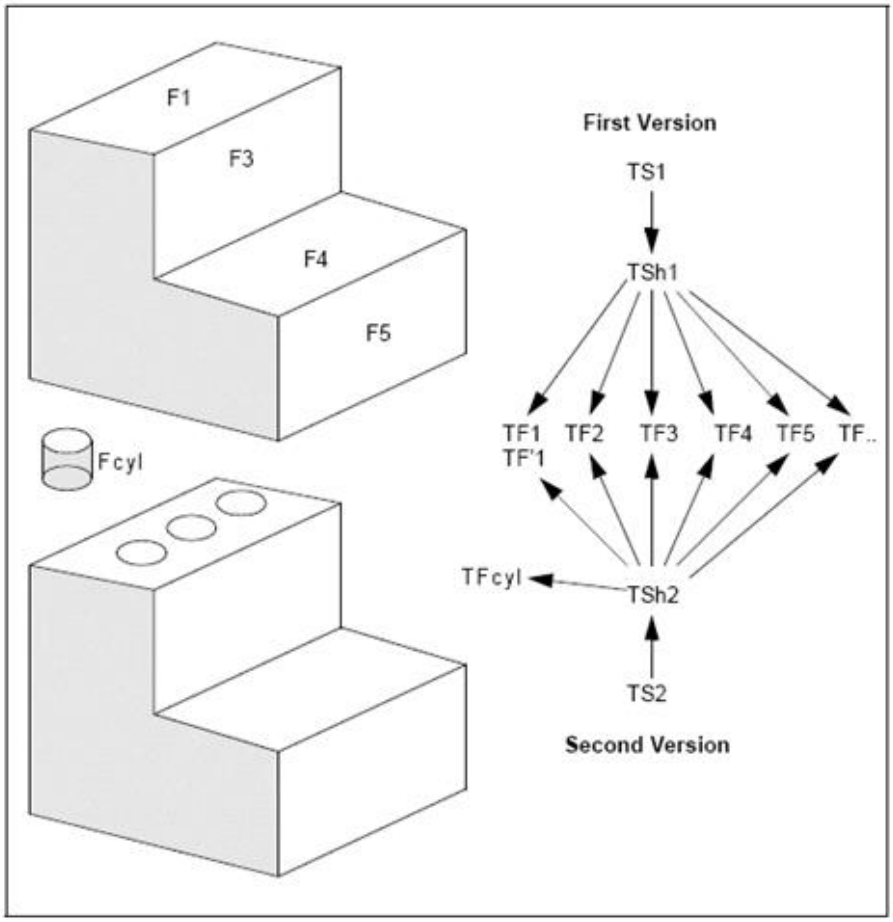

Figure 9: Data structure containing two versions of a solid

Classes inheriting TopoDS_Shape

TopoDS is based on class *TopoDS_Shape* and the class defining its underlying shape. This has certain advantages, but the major drawback is that these classes are too general. Different shapes they could represent do not type them (Vertex, Edge, etc.) hence it is impossible to introduce checks to avoid incoherences such as inserting a face in an edge.

TopoDS module offers two sets of classes, one set inheriting the underlying shape with neither orientation nor location and the other inheriting *TopoDS_Shape*, which represent the standard topological shapes enumerated in *TopAbs* module.

The following classes inherit from *TopoDS_Shape* : *TopoDS_Vertex, TopoDS_Edge, TopoDS_Wire, TopoDS_Face, TopoDS_Shell, TopoDS_Solid,TopoDS_CompSolid,* and *TopoDS_Compound*. In spite of the similarity of names with those inheriting from *TopoDS_TShape* there is a profound difference in the way they are used.

TopoDS_Shape class and the classes, which inherit from it, are the natural means to manipulate topological objects. *TopoDS_TShape* classes are hidden. *TopoDS_TShape* describes a class in its original local coordinate system without orientation. *TopoDS_Shape* is a reference to *TopoDS_TShape* with an orientation and a local reference.

TopoDS_TShape class is deferred; *TopoDS_Shape* class is not. Using *TopoDS_Shape* class allows manipulation of topological objects without knowing their type. It is a generic form. Purely topological algorithms often use the *TopoDS_Shape* class.

TopoDS_TShape class is manipulated by reference; TopoDS_Shape class by value. A TopoDS_Shape is nothing more than a reference enhanced with an orientation and a local coordinate. The sharing of *TopoDS_Shapes* is meaningless. What is important is the sharing of the underlying *TopoDS_TShapes*. Assignment or passage in argument does not copy the data structure: this only creates new *TopoDS_Shapes* which refer to the same

TopoDS_TShape.

Although classes inheriting *TopoDS_TShape* are used for adding extra information, extra fields should not be added in a class inheriting from TopoDS_Shape. Classes inheriting from TopoDS_Shape serve only to specialize a reference in order to benefit from static type control (carried out by the compiler). For example, a routine that receives a *TopoDS_Face* in argument is more precise for the compiler than the one, which receives a *TopoDS_Shape*. It is pointless to derive other classes than those found inTopoDS. All references to a topological data structure are made with the Shape class and its inheritors defined in *TopoDS*.

There are no constructors for the classes inheriting from the *TopoDS_Shape* class, otherwise the type control would disappear through **implicit casting** (a characteristic of C++). The TopoDS module provides module methods for **casting** an object of the TopoDS_Shape class in one of these sub-classes, with type verification.

The following example shows a routine receiving an argument of the *TopoDS_Shape* type, then putting it into a variable V if it is a vertex or calling the method ProcessEdge if it is an edge.

```
1 from OCC.Core.TopAbs import TopAbs_VERTEX, TopAbs_Edge
2 from OCC.Core.TopoDS import topods_Vertex, topods_Edge
3
4 def process_edge(an_edge):
5   """ an_edge : a TopoDS_Edge
6   """
7   ... do something
8
9 def process(a_shape):
10   """ a_shape : TopoDS_Shape
11   """
12   if aShape.Shapetype() == TopAbs_VERTEX:
13     v = topods_Vertex(aShape)
14   elif aShape.ShapeType() == TopAbs_EDGE:
15     e topods_Edge(a_shape)
16     process_edge(e)
17   else:
18     print("Neither a vertex nor an edge.")
```

2.4 Exploration of Topological Data Structures

The *TopExp* module provides tools for exploring the data structure described with the *TopoDS* module. Exploring a topological structure means finding all sub-objects of a given type, for example, finding all the faces of a solid.

The TopExp module provides the class *TopExp_Explorer* to find all sub-objects of a given type. An explorer is built with:

- the shape to be explored,
- the type of shapes to be found e.g. *TopAbs_VERTEX*, *TopAbs_EDGE* with the exception of *TopAbs_SHAPE*, which is not allowed,
- the type of Shapes to avoid. e.g. *TopAbs_SHELL*, *TopAbs_EDGE*. By default, this type is *TopAbs_SHAPE*. This default value means that there is no restriction on the exploration.

The *TopExp_Explorer* class visits the whole structure in order to find the shapes of the requested type not contained in the type to avoid. The example below shows how to find all faces in the shape *S*:

```
1 from OCC.Core.TopExp import TopExp_Explorer
2
3 def process_face(a_topods_shape):
4   ... ## anything that takes and processes a TopoDS_Face
5
6 def test():
```

```
7   shp = ... # any TopoDS_Shape
8   exp = TopExp_Explorer()
9   exp.Init(shp, TopAbs_FACE)
10   while exp.More():
11     process_face(exp.Current())
12     exp.Next()
```

Find all the vertices which are not in an edge

```
1 exp.Init(shp, TopAbs_VERTEX, TopAbs_EDGE)
```

Find all the faces in a SHELL, then all the faces not in a SHELL:

```
1 def test():
2   shp = ... ## any TopoDS_Shape
3   exp1 = TopExp_Explorer()
4   exp2 = TopExp_Explorer()
5   exp1.Init(shp, TopAbs_SHELL)
6   while exp1.More():
7     exp1.Next()
8     # visit all shells
9     exp2.Init(exp1.Current()
10     while exp2.More():
11       # visit all the faces of the current shell
12       ProcessFaceinAshell(exp2.Current())
13       exp2.Next()
14     exp1.Next()
15   exp1.Init(shp, TopAbs_FACE, TopAbs_SHELL)
16   while exp1.More():
17     # visit all faces not ina shell.
18     process_face(exp.Current())
19     exp1.Next()
```

The Explorer presumes that objects contain only objects of an equal or inferior type. For example, if searching for faces it does not look at wires, edges, or vertices to see if they contain faces.

The *MapShapes* method from the *topexp* class allows filling a Map. An exploration using the *TopExp_Explorer* class can visit an object more than once if it is referenced more than once. For example, an edge of a solid is generally referenced by two faces. To process objects only once, they have to be placed in a Map.

2.5 Lists and Maps of Shapes

TopTools module contains tools for exploiting the *TopoDS* data structure. It is an instantiation of the tools from *TCollection* module with the Shape classes of *TopoDS*.

- *TopTools_Array1OfShape, HArray1OfShape* - Instantiation of the *TCollection_Array1* and *TCollection_HArray1* with *TopoDS_Shape*,
- *TopTools_SequenceOfShape* - Instantiation of the *TCollection_Sequence* with *TopoDS_Shape*,
- *TopTools_MapOfShape* - Instantiation of the *TCollection_Map*. Allows the construction of sets of shapes,
- *TopTools_IndexedMapOfShape* - Instantiation of the *TCollection_IndexedMap*. Allows the construction of tables of shapes and other data structures.

With a *TopTools_Map*, a set of references to Shapes can be kept without duplication. The following example counts the size of a data structure as a number of *TShapes*.

```
def get_shape_size(a_shp):
  """ returns the size of a shape.
  This is a recursive method.
  The size of a shape is1 + the sizes of the subshapes.

  a_shp : a TopoDS_Shape
  """
  it = TopoDS_Iterator()
  it.Initialize(a_shp)
   size = 1
   while it.More():
    size += get_shape_size(it.Value())
    it.Next()
  return size
```

This program is incorrect if there is sharing in the data structure.

Thus for a contour of four edges it should count 1 wire + 4 edges +4 vertices with the result 9, but as the vertices are each shared by two edges this program will return 13. One solution is to put all the Shapes in a Map so as to avoid counting them twice, as in the following example:

```
def map_shapes(a_shape, a_map):
 """
 a_shape: a TopoDS_Shape
 a_map: a TopTools_MapOfShape
 """
 # This is a recursive auxiliary method. It stores all subShapes of aShape in a Map.
 if a_map.Add(a_shape):
   ## Add returns True if aShape was not already in the Map.
   it = TopoDS_Iterator()
    it.Initiliaze(a_shape)
    while it.More():
      map_shapes(it.Value(), a_map)
      it.Next()


def size(a_shape):
  """
  a_shape: a TopoDS_Shape
  """
  # Store Shapes in a Mapand return the size.
  m = TopTools_MapOfShape()
  map_shapes(a_shape, m)
  return m.Extent()


sh = ... # any TopoDS_Shape
print(size(sh))
```

The following example is more ambitious and writes a program which copies a data structure using an *TopTools_IndexedMap*. The copy is an identical structure but it shares nothing with the original. The principal algorithm is as follows:

- all TopoDS_Shape instances in the structure are put into an *TopTools_IndexedMap*,
- a table of TopoDS_Shape instances is created in parallel with the map to receive the copies,
- the structure is copied using the auxiliary recursive function, which copies from the map to the array.

```
def copy(a_shape, a_builder):
 """ aShape: a TopoDS_Shape instance
 a_builder: a TopoDS_Builder instance
 Returns a TopoDS_Shape
 """
 # Copies the wholestructure of a_shape using a_builder.
 # Stores all the sub-Shapes in an IndexedMap.
 theMap = TopTools_IndexedMapOfShape()
 it = TopoDS_Iterator()
  theMap.Add(a_shape)
  for i in range(1, theMap.Extent()):
   it.Initialize(theMap(i))
   while it.More():
     s=It.Value()
     s.Location(Identity)
     s.Orientation(TopAbs_FORWARD)
     theMap.Add(s)
     it.Next()
```

2.5.1 Wire Explorer

BRepTools_WireExplorer class can access edges of a wire in their order of connection.

For example, in the wire in the image we want to recuperate the edges in the order {e1, e2, e3,e4, e5} :

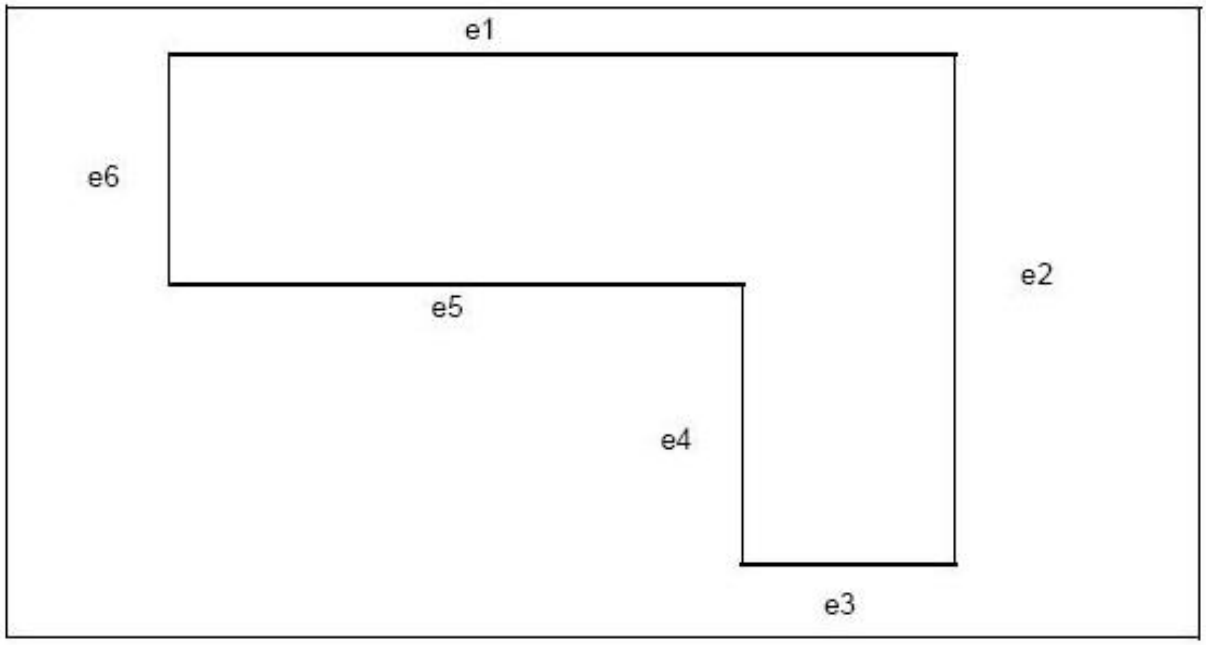

Figure 10: A wire composed of 6 edges.

TopExp_Explorer, however, recuperates the lines in any order.

```
w = ... # any TopoDS_Wire
ex = BRepTools_WireExplorer()
ex.Init(w)
while ex.More():
 process_current_endge(ex.Current())
 # ProcessTheVertexConnectingTheCurrentEdgeToThePrevious
```

```
7  one(ex.CurrentVertex())
8  ex.Next()
```

3 Geometry Utilities

Geometry Utilities provide the following services:

- creation of shapes by interpolation and approximation,
- direct construction of shapes,
- conversion of curves and surfaces to BSpline curves and surfaces,
- computation of the coordinates of points on 2D and 3D curves,
- calculation of extrema between shapes.

3.1 Interpolations and Approximations

In modeling, it is often required to approximate or interpolate points into curves and surfaces. In interpolation, the process is complete when the curve or surface passes through all the points; in approximation, when it is as close to these points as possible.

Approximation of Curves and Surfaces groups together a variety of functions used in 2D and 3D geometry for:

- the interpolation of a set of 2D points using a 2D BSpline or Bezier curve,
- the approximation of a set of 2D points using a 2D BSpline or Bezier curve,
- the interpolation of a set of 3D points using a 3D BSpline or Bezier curve, or a BSpline surface,
- the approximation of a set of 3D points using a 3D BSpline or Bezier curve, or a BSpline surface.

You can program approximations in two ways:

- using high-level functions, designed to provide a simple method for obtaining approximations with minimal programming,
- using low-level functions, designed for users requiring more control over the approximations.

3.1.1 Analysis of a set of points

The class *PEquation* from *GProp* module allows analyzing a collection or cloud of points and verifying if they are coincident, collinear or coplanar within a given precision. If they are, the algorithm computes the mean point, the mean line or the mean plane of the points. If they are not, the algorithm computes the minimal box, which includes all the points.

3.1.2 Basic Interpolation and Approximation

Modules *Geom2dAPI* and *GeomAPI* provide simple methods for approximation and interpolation with minimal programming

2D Interpolation

The class *Interpolate* from *Geom2dAPI* module allows building a constrained 2D BSpline curve, defined by a table of points through which the curve passes. If required, the parameter values and vectors of the tangents can be given for each point in the table.

3D Interpolation

The class *Interpolate* from *GeomAPI* module allows building a constrained 3D BSpline curve, defined by a table of points through which the curve passes. If required, the parameter values and vectors of the tangents can be given for each point in the table.

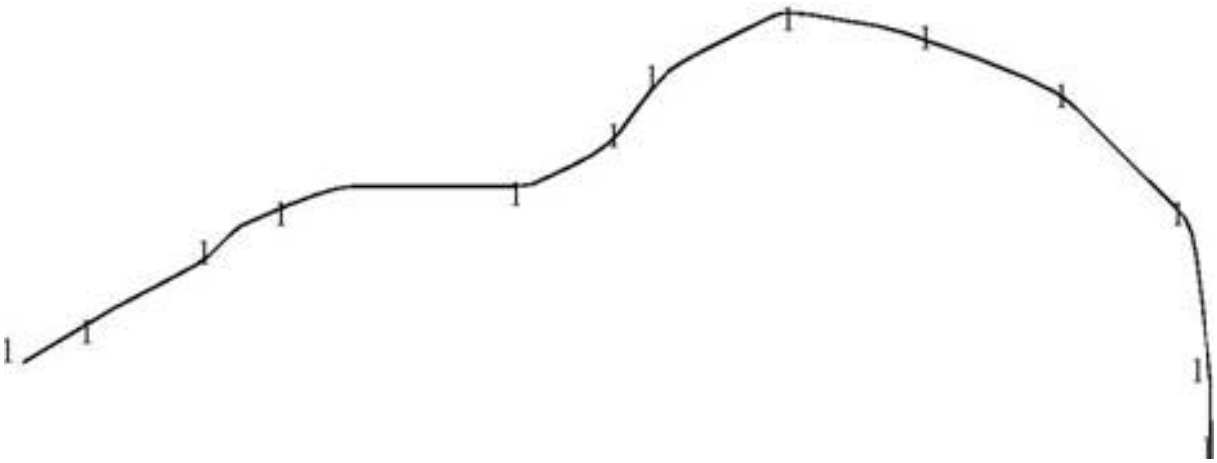

Figure 11: Approximation of a BSpline from scattered points

This class may be instantiated as follows:

```
1 interp = GeomAPI_Interpolate(points)
```

From this object, the BSpline curve may be requested as follows:

```
1 curve = interp.Curve()
```

2D Approximation

The class *Geom2dAPI_PointsToBSpline* allows building a 2DBSpline curve, which approximates a set of points. You have to define the lowest and highest degree of the curve, its continuity and a tolerance value for it.The tolerance value is used to check that points are not too close to each other, or tangential vectors not too small. The resulting BSpline curve will beC2 or second degree continuous, except where a tangency constraint is defined on a point through which the curve passes. In this case, it will be only C1continuous.

3D Approximation

The class *GeomAPI_PointsToBSpline* allows building a 3D BSplinecurve, which approximates a set of points. It is necessary to define the lowest and highest degree of the curve, its continuity and tolerance. The tolerance value is used to check that points are not too close to each other,or that tangential vectors are not too small.

The resulting BSpline curve will be C2 or second degree continuous, except where a tangency constraint is defined on a point, through which the curve passes. In this case, it will be only C1 continuous. This class is instantiated as follows:

```
1 approx = GeomAPI_PointsToBSpline(points, deg_min, deg_max, continuity, tol)
```

From this object, the BSpline curve may be requested as follows:

```
1 curve = approx.Curve()
```

Surface Approximation

The class *GeomAPI_PointsToBSplineSurface* allows building a BSpline surface, which approximates or interpolates a set of points.

3.1.3 Advanced Approximation

Modules *AppDef* and *AppParCurves* provide low-level functions, allowing more control over the approximations.

The low-level functions provide a second API with functions to:

- define compulsory tangents for an approximation. These tangents have origins and extremities,
- approximate a set of curves in parallel to respect identical parameterization,
- smooth approximations. This is to produce a faired curve.

You can also find functions to compute:

- the minimal box which includes a set of points,
- the mean plane, line or point of a set of coplanar, collinear or coincident points.

Approximation by multiple point constraints

AppDef module provides low-level tools to allow parallel approximation of groups of points into Bezier or B-Spline curves using multiple point constraints.

The following low level services are provided:

- Definition of an array of point constraints:

  The class *AppDef_MultiLine* allows defining a given number of multi-point constraints in order to build the multi-line, multiple lines passing through ordered multiple point constraints.

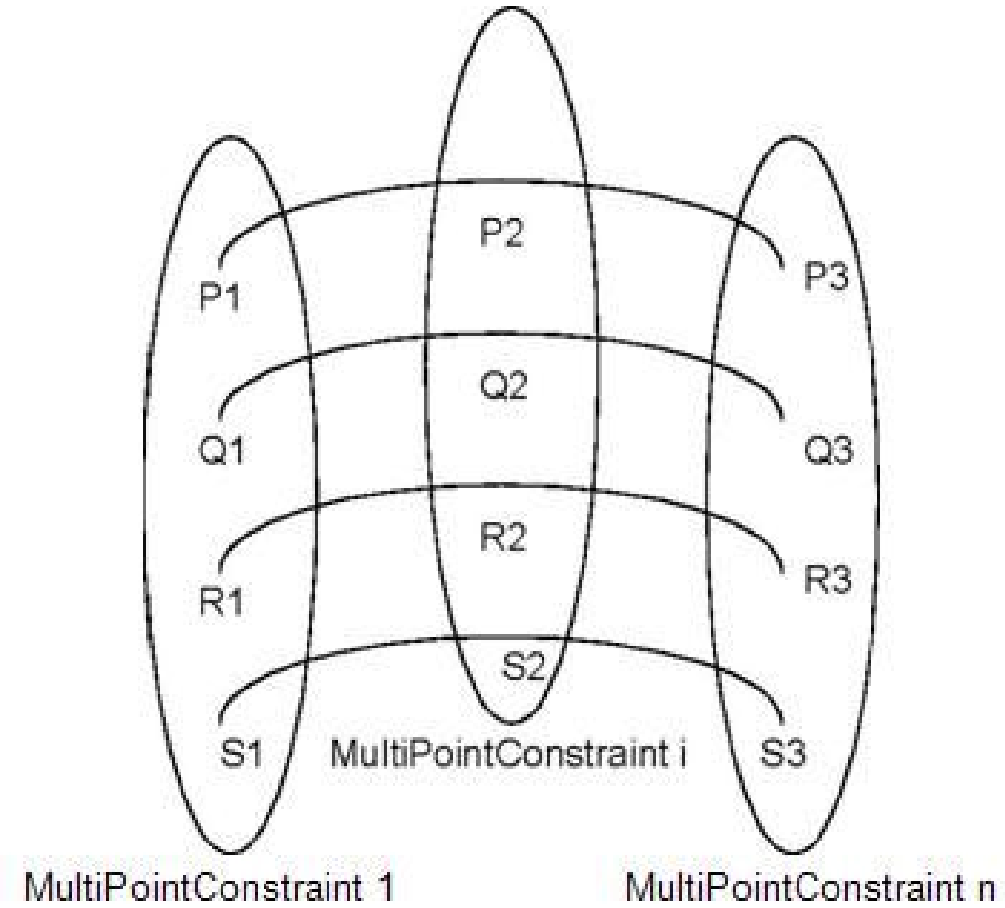

Figure 12: Definition of a MultiLine using Multiple Point Constraints

In this image:

- Pi, Qi, Ri ... Si can be 2D or 3D points,
- defined as a group: Pn, Qn, Rn, ... Sn form a MultipointConstraint. They possess the same passage, tangency and curvature constraints,
- $P1$, $P2$, ... Pn, or the Q, R, ... or S series represent the lines to be approximated.

- Definition of a set of point constraints:

  The class *AppDef_MultiPointConstraint* allows defining a multiple point constraint and computing the approximation of sets of points to several curves.

- Computation of an approximation of a Bezier curve from a set of points:

  The class *AppDef_Compute* allows making an approximation of a set of points to a Bezier curve

- Computation of an approximation of a BSpline curve from a set of points:

  The class *AppDef_BSplineCompute* allows making an approximation of a set of points to a BSpline curve.

- Definition of Variational Criteria:

The class *AppDef_Variational* allows fairing the approximation curve to a given number of points using a least squares method in conjunction with a variational criterion, usually the weights at each constraint point.

Approximation by parametric or geometric constraints

AppParCurves module provides low-level tools to allow parallel approximation of groups of points into Bezier or B-Spline curve with parametric or geometric constraints, such as a requirement for the curve to pass through given points, or to have a given tangency or curvature at a particular point.

The algorithms used include:

- the least squares method,
- a search for the best approximation within a given tolerance value.

The following low-level services are provided:

- Association of an index to an object:

The class *AppParCurves_ConstraintCouple* allows you associating an index to an object to compute faired curves using *AppDef_TheVariational*.

- Definition of a set of approximations of Bezier curves:

The class *AppParCurves_MultiCurve* allows defining the approximation of a multi-line made up of multiple Bezier curves.

- Definition of a set of approximations of BSpline curves:

The class *AppParCurves_MultiBSpCurve* allows defining the approximation of a multi-line made up of multiple BSpline curves.

- Definition of points making up a set of point constraints

The class *AppParCurves_MultiPoint* allows defining groups of 2D or 3D points making up a multi-line.

Example: How to approximate a curve with respect to tangency

To approximate a curve with respect to tangency, follow these steps:

1. Create an object of type *AppDef_MultiPointConstraints* from the set of points to approximate and use the method *SetTang∗to set the tangency vectors.*
2. *Create an object of type ∗AppDef_MultiLine∗from the ∗AppDef_MultiPointConstraint*.
3. Use *AppDef_BSplineCompute*, which instantiates *Approx_BSplineComputeLine* to perform the approximation.

3.2 Direct Construction

Direct Construction methods from *gce*, *GC* and *GCE2d* modules provide simplified algorithms to build elementary geometric entities such as lines, circles and curves. They complement the reference definitions provided by the *gp*, *Geom* and *Geom2d* modules.

The algorithms implemented by *gce*, *GCE2d* and *GC* modules are simple: there is no creation of objects defined by advanced positional constraints (for more information on this subject, see *Geom2dGcc* and *GccAna*, which describe geometry by constraints).

For example, to construct a circle from a point and a radius using the *gp* module, it is necessary to construct axis *Ax2d* before creating the circle. If *gce* module is used, and *Ox* is taken for the axis, it is possible to create a circle directly from a point and a radius.

Another example is the class *gce_MakeCirc* providing a framework for defining eight problems encountered in the geometric construction of circles and implementing the eight related construction algorithms.

The object created (or implemented) is an algorithm which can be consulted to find out, in particular:

- its result, which is a *gp_Circ*,
- its status. Here, the status indicates whether or not the construction was successful.

If it was unsuccessful, the status gives the reason for the failure.

```
1 p1 = gp_Pnt(0., 0., 0.)
2 p2 = gp_Pnt(0., 10., 0.)
3 p3 = gp_Pnt(10., 0., 0.)
4 mc = gce_MakeCirc(p1, p2, p3)
5 if mc.IsDone():
6   c = mc.Value()
```

In addition, *gce*, *GCE2d* and *GC* each have a *Root* class, i.e. *gce_Root*, *GCE_Root* and *GC_Root*. This class is the root of all classes in the module, which return a status. The returned status (successful construction or construction error) is described by the enumeration *gce_ErrorType*. Note, that classes, which construct geometric transformations do not return a status, and therefore do not inherit from *Root*.

3.2.1 Non-persistent entities

The following algorithms used to build entities from non-persistent *gp* entities are provided by *gce* module.

- 2D line parallel to another at a distance,
- 2D line parallel to another passing through a point,
- 2D circle passing through two points,
- 2D circle parallel to another at a distance,
- 2D circle parallel to another passing through a point,
- 2D circle passing through three points,
- 2D circle from a center and a radius,
- 2D hyperbola from five points,
- 2D hyperbola from a center and two apexes,
- 2D ellipse from five points,
- 2D ellipse from a center and two apexes,
- 2D parabola from three points,
- 2D parabola from a center and an apex,
- line parallel to another passing through a point,
- line passing through two points,
- circle coaxial to another passing through a point,
- circle coaxial to another at a given distance,
- circle passing through three points,
- circle with its center, radius, and normal to the plane,

- circle with its axis (center + normal),
- hyperbola with its center and two apexes,
- ellipse with its center and two apexes,
- plane passing through three points,
- plane from its normal,
- plane parallel to another plane at a given distance,
- plane parallel to another passing through a point,
- plane from an array of points,
- cylinder from a given axis and a given radius,
- cylinder from a circular base,
- cylinder from three points,
- cylinder parallel to another cylinder at a given distance,
- cylinder parallel to another cylinder passing through a point,
- cone from four points,
- cone from a given axis and two passing points,
- cone from two points (an axis) and two radii,
- cone parallel to another at a given distance,
- cone parallel to another passing through a point,
- all transformations (rotations, translations, mirrors,scaling transformations, etc.).

Each class from *gp* module, such as *gp_Circ, gp_Circ2d, gp_Mirror, gp_Mirror2d*, etc., has the corresponding *gce_MakeCirc, gce_MakeCirc2d, gce_MakeMirror, gce_MakeMirror2d*, etc. class from *gce* module.

It is possible to create a point using a *gce* module class, then question it to recover the corresponding *gp* object.

```
1 point1 = ... # any gp_Pnt2d
2 point2 = ... # any gp_Pnt2d
3
4 l = gce_MakeLin2d(point1, point2)
5 if l.Status() == gce_Done:
6  l2d = l.Value() # returns a gp_Lin2d
```

3.2.2 Persistent entities

GC and *GCE2d* modules provides an implementation of algorithms used to build entities from *Geom* and *Geom2D* modules. They implement the same algorithms as the *gce* module but create *persistent* entities, and also contain algorithms for trimmed surfaces and curves. The following algorithms are available:

- arc of a circle trimmed by two points,
- arc of a circle trimmed by two parameters,
- arc of a circle trimmed by one point and one parameter,
- arc of an ellipse from an ellipse trimmed by two points,
- arc of an ellipse from an ellipse trimmed by two parameters,

- arc of an ellipse from an ellipse trimmed by one point and one parameter,
- arc of a parabola from a parabola trimmed by two points,
- arc of a parabola from a parabola trimmed by two parameters,
- arc of a parabola from a parabola trimmed by one point and one parameter,
- arc of a hyperbola from a hyperbola trimmed by two points,
- arc of a hyperbola from a hyperbola trimmed by two parameters,
- arc of a hyperbola from a hyperbola trimmed by one point and one parameter,
- segment of a line from two points,
- segment of a line from two parameters,
- segment of a line from one point and one parameter,
- trimmed cylinder from a circular base and a height,
- trimmed cylinder from three points,
- trimmed cylinder from an axis, a radius, and a height,
- trimmed cone from four points,
- trimmed cone from two points (an axis) and a radius,
- trimmed cone from two coaxial circles.

Each class from *GCE2d* module, such as *GCE2d_Circle, GCE2d_Ellipse, GCE2d_Mirror*, etc., has the corresponding *Geom2d_MakeCircle, Geom2d_MakeEllipse, Geom2d_MakeMirror*, etc. class from *Geom2d* module. Besides, the class *GCE2d_MakeArcOfCircle* returns an object of type *Geom2d_TrimmedCurve*.

Each class from *GC* module, such as *GC_Circle, GC_Ellipse, GC_Mirror*, etc., has the corresponding *Geom_MakeCircle, Geom_MakeEllipse, Geom_MakeMirror*, etc. class from *Geom* module. The *Value* method for following classes return objects of type *Geom_TrimmedCurve*:

- *GC_MakeArcOfCircle*
- *GC_MakeArcOfEllipse*
- *GC_MakeArcOfHyperbola*
- *GC_MakeArcOfParabola*
- *GC_MakeSegment*

3.3 Conversion to and from BSplines

The conversion to and from BSplines component has two distinct purposes:

- firstly, it provides a homogeneous formulation which can be used to describe any curve or surface. This is useful for writing algorithms for a single data structure model. The BSpline formulation can be used to represent most basic geometric objects provided by the components which describe geometric data structures ("Fundamental Geometry Types", "2D Geometry Types" and "3D Geometry Types" components),
- secondly, it can be used to divide a BSpline curve or surface into a series of curves or surfaces, thereby providing a higher degree of continuity. This is useful for writing algorithms which require a specific degree of continuity in the objects to which they are applied. Discontinuities are situated on the boundaries of objects only.

The "Conversion to and from BSplines" component is composed of three modules.

The *Convert* module provides algorithms to convert the following into a BSpline curve or surface:

- a bounded curve based on an elementary 2D curve (line, circle or conic) from the *gp* module,
- a bounded surface based on an elementary surface (cylinder, cone, sphere or torus) from the *gp* module,
- a series of adjacent 2D or 3D Bezier curves defined by their poles.

These algorithms compute the data needed to define the resulting BSpline curve or surface. This elementary data (degrees, periodic characteristics, poles and weights, knots and multiplicities) may then be used directly in an algorithm, or can be used to construct the curve or the surface by calling the appropriate constructor provided by the classes *Geom2d_BSplineCurve, Geom_BSplineCurve* or *Geom_BSplineSurface*.

The *Geom2dConvert* module provides the following:

- a global function which is used to construct a BSpline curve from a bounded curve based on a 2D curve from the Geom2d module,
- a splitting algorithm which computes the points at which a 2D BSpline curve should be cut in order to obtain arcs with the same degree of continuity,
- global functions used to construct the BSpline curves created by this splitting algorithm, or by other types of segmentation of the BSpline curve,
- an algorithm which converts a 2D BSpline curve into a series of adjacent Bezier curves.

The *GeomConvert* module also provides the following:

- a global function used to construct a BSpline curve from a bounded curve based on a curve from the Geom module,
- a splitting algorithm, which computes the points at which a BSpline curve should be cut in order to obtain arcs with the same degree of continuity,
- global functions to construct BSpline curves created by this splitting algorithm, or by other types of BSpline curve segmentation,
- an algorithm, which converts a BSpline curve into a series of adjacent Bezier curves,
- a global function to construct a BSpline surface from a bounded surface based on a surface from the Geom module,
- a splitting algorithm, which determines the curves along which a BSpline surface should be cut in order to obtain patches with the same degree of continuity,
- global functions to construct BSpline surfaces created by this splitting algorithm, or by other types of BSpline surface segmentation,
- an algorithm, which converts a BSpline surface into a series of adjacent Bezier surfaces,
- an algorithm, which converts a grid of adjacent Bezier surfaces into a BSpline surface.

3.4 Points on Curves

The Points on Curves component comprises high level functions providing an API for complex algorithms that compute points on a 2D or 3D curve.

The following characteristic points exist on parameterized curves in 3d space:

- points equally spaced on a curve,

- points distributed along a curve with equal chords,
- a point at a given distance from another point on a curve.

GCPnts module provides algorithms to calculate such points:

- *GCPnts_AbscissaPoint* calculates a point on a curve at a given distance from another point on the curve.
- *GCPnts_UniformAbscissa* calculates a set of points at a given abscissa on a curve.
- *GCPnts_UniformDeflection* calculates a set of points at maximum constant deflection between the curve and the polygon that results from the computed points.

Example: Visualizing a curve.

Let us take an adapted curve **C**, i.e. an object which is an interface between the services provided by either a 2D curve from the module Geom2d (in case of an Adaptor_Curve2d curve) or a 3D curve from the module Geom (in case of an Adaptor_Curve curve), and the services required on the curve by the computation algorithm. The adapted curve is created in the following way:

2D case :

```
1 mycurve = some Handle_Geom2d_Curve
2 c = Geom2dAdaptor_Curve(mycurve)
```

3D case :

```
1 mycurve = some Handle_Geom_Curve
2 c = GeomAdaptor_Curve(mycurve)
```

The algorithm is then constructed with this object:

```
1 my_algo = GCPnts_UniformDeflection
2 deflection = ... # any floar number
3 my_algo.Initialize(c, deflection)
4 if myAlgo.IsDone():
5   nbr = myAlgo.NbPoints()
6   for i in range(1, param):
7     param = myAlgo.Parameter(i)
```

3.5 Extrema

The classes to calculate the minimum distance between points, curves, and surfaces in 2d and 3d are provided by *GeomAPI* and *Geom2dAPI* modules.

These modules calculate the extrema of distance between:

- point and a curve,
- point and a surface,
- two curves,
- a curve and a surface,
- two surfaces.

Extrema between Curves

The *Geom2dAPI_ExtremaCurveCurve* class allows calculation of all extrema between two 2D geometric curves. Extrema are the lengths of the segments orthogonal to two curves.

The *GeomAPI_ExtremaCurveCurve* class allows calculation of all extrema between two 3D geometric curves. Extrema are the lengths of the segments orthogonal to two curves.

Extrema between Curve and Surface

The *GeomAPI_ExtremaCurveSurface* class allows calculation of all extrema between a 3D curve and a surface. Extrema are the lengths of the segments orthogonal to the curve and the surface.

Extrema between Surfaces

The *GeomAPI_ExtremaSurfaceSurface* class allows calculation of all extrema between two surfaces. Extrema are the lengths of the segments orthogonal to two surfaces.

4 2D Geometry

Geom2d module defines geometric objects in 2dspace. The objects are non-persistent and are handled by reference. In particular, *Geom2d* module provides classes for:

- description of points, vectors and curves,
- their positioning in the plane using coordinate systems,
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following objects are available:

- point,
- cartesian point,
- vector,
- direction,
- vector with magnitude,
- axis,
- curve,
- line,
- conic: circle, ellipse, hyperbola, parabola,
- rounded curve: trimmed curve, NURBS curve, Bezier curve,
- offset curve.

Before creating a geometric object, it is necessary to decide how the object is handled. The objects provided by *Geom2d* module are handled by reference rather than by value. Copying an instance copies the handle, not the object, so that a change to one instance is reflected in each occurrence of it. If a set of object instances is needed rather than a single object instance, *TColGeom2d* module can be used. This module provides standard and frequently used instantiations of one-dimensional arrays and sequences for curves from *Geom2d* module. All objects are available in two versions:

- handled by reference,
- handled by value.

The key characteristic of *Geom2d* curves is that they are parameterized. Each class provides functions to work with the parametric equation of the curve, and, in particular, to compute the point of parameter u on a curve and the derivative vectors of order $1, 2.., N$ at this point.

As a consequence of the parameterization, a *Geom2d* curve is naturally oriented.

Parameterization and orientation differentiate elementary *Geom2d*$*$*curves from their equivalent as provided by* $*$*gp* module. *Geom2d* module provides conversion functions to transform a *Geom2d* object into a *gp* object, and vice-versa, when this is possible.

Moreover, *Geom2d* module provides more complex curves, including Bezier curves, BSpline curves, trimmed curves and offset curves.

Geom2d objects are organized according to an inheritance structure over several levels.

Thus, an ellipse (specific class *Geom2d_Ellipse*) is also a conical curve and inherits from the abstract class *Geom2d_Conic*, while a Bezier curve (concrete class *Geom2d_BezierCurve*) is also a bounded curve and inherits from the abstract class *Geom2d_BoundedCurve*; both these examples are also curves (abstract class

Geom2d_Curve). Curves, points and vectors inherit from the abstract class∗Geom2d_Geometry,∗ which describes the properties common to any geometric object from the *Geom2d* module.

This inheritance structure is open and it is possible to describe new objects, which inherit from those provided in the *Geom2d* module, provided that they respect the behavior of the classes from which they are to inherit.

Finally, *Geom2d* objects can be shared within more complex data structures. This is why they are used within topological data structures, for example.

Geom2d module uses the services of the *gp* module to:

- implement elementary algebraic calculus and basic analytic geometry,
- describe geometric transformations which can be applied to *Geom2d* objects,
- describe the elementary data structures of *Geom2d* objects.

However, the *Geom2d* module essentially provides data structures and not algorithms. You can refer to the *GCE2d* module to find more evolved construction algorithms for *Geom2d* objects.

5 3D Geometry

The *Geom* module defines geometric objects in 3d space and contains all basic geometric transformations, such as identity, rotation, translation, mirroring, scale transformations, combinations of transformations, etc. as well as special functions depending on the reference definition of the geometric object (e.g. addition of a control point on a B-Spline curve,modification of a curve, etc.).

In particular, it provides classes for:

- description of points, vectors, curves and surfaces,
- their positioning in 3D space using axis or coordinate systems,
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following non-persistent and reference-handled objects are available:

- point,
- cartesian point,
- vector,
- direction,
- vector with magnitude,
- axis,
- curve,
- line,
- conic: circle, ellipse, hyperbola, parabola,
- offset curve,
- elementary surface: plane, cylinder, cone, sphere, torus,
- bounded curve: trimmed curve, NURBS curve, Bezier curve,
- bounded surface: rectangular trimmed surface, NURBS surface, Bezier surface,
- swept surface: surface of linear extrusion, surface of revolution,
- offset surface.

The key characteristic of *Geom* curves and surfaces is that they are parameterized. Each class provides functions to work with the parametric equation of the curve or surface, and, in particular, to compute:

- the point of parameter u on a curve, or
- the point of parameters (u, v) on a surface. together with the derivative vectors of order $1, 2, ...N$ at this point.

As a consequence of this parameterization, a Geom curve or surface is naturally oriented.

Parameterization and orientation differentiate elementary Geom curves and surfaces from the classes of the same (or similar) names found in *gp* module. *Geom* module also provides conversion functions to transform a Geom object into a *gp* object, and vice-versa, when such transformation is possible.

Moreover, $*$Geom$*$module provides more complex curves and surfaces, including:

- Bezier and BSpline curves and surfaces,

- swept surfaces, for example surfaces of revolution and surfaces of linear extrusion,
- trimmed curves and surfaces, and
- offset curves and surfaces.

Geom objects are organized according to an inheritance structure over several levels. Thus, a sphere (concrete class *Geom_SphericalSurface*) is also an elementary surface and inherits from the abstract class *Geom_ElementarySurface*, while a Bezier surface (concrete class *Geom_BezierSurface*) is also a bounded surface and inherits from the abstract class *Geom_BoundedSurface*; both these examples are also surfaces (abstract class *Geom_Surface*). Curves, points and vectors inherit from the abstract class *Geom_Geometry,* which describes the properties common to any geometric object from the *Geom* module.

This inheritance structure is open and it is possible to describe new objects, which inherit from those provided in the Geom module, on the condition that they respect the behavior of the classes from which they are to inherit.

Finally, Geom objects can be shared within more complex data structures. This is why they are used within topological data structures, for example.

If a set of object instances is needed rather than a single object instance, *TColGeom* module can be used. This module provides instantiations of one- and two-dimensional arrays and sequences for curves from *Geom* module. All objects are available in two versions:

- handled by reference and
- handled by value.

The *Geom* module uses the services of the *gp* module to:

- implement elementary algebraic calculus and basic analytic geometry,
- describe geometric transformations which can be applied to Geom objects,
- describe the elementary data structures of Geom objects.

However, the Geom module essentially provides data structures, not algorithms.

You can refer to the *GC* module to find more evolved construction algorithms for Geom objects.

6 Properties of Shapes

6.1 Local Properties of Shapes

BRepLProp module provides the Local Properties of Shapes component, which contains algorithms computing various local properties on edges and faces in a BRep model.

The local properties which may be queried are:

- for a point of parameter u on a curve which supports an edge :
  - the point,
  - the derivative vectors, up to the third degree,
  - the tangent vector,
  - the normal,
  - the curvature, and the center of curvature;
- for a point of parameter (u, v) on a surface which supports a face :
  - the point,
  - the derivative vectors, up to the second degree,
  - the tangent vectors to the u and v isoparametric curves,
  - the normal vector,
  - the minimum or maximum curvature, and the corresponding directions of curvature;
- the degree of continuity of a curve which supports an edge, built by the concatenation of two other edges, at their junction point.

Analyzed edges and faces are described as *BRepAdaptor* curves and surfaces, which provide shapes with an interface for the description of their geometric support. The base point for local properties is defined by its u parameter value on a curve, or its (u, v) parameter values on a surface.

6.2 Local Properties of Curves and Surfaces

The "Local Properties of Curves and Surfaces" component provides algorithms for computing various local properties on a Geom curve (in 2D or 3D space) or a surface. It is composed of:

- *Geom2dLProp* module, which allows computing Derivative and Tangent vectors (normal and curvature) of a parametric point on a 2D curve;
- *GeomLProp* module, which provides local properties on 3D curves and surfaces
- *LProp* module, which provides an enumeration used to characterize a particular point on a 2D curve.

Curves are either *Geom_Curve* curves (in 3D space) or *Geom2d_Curve* curves (in the plane). Surfaces are *Geom_Surface* surfaces. The point on which local properties are calculated is defined by its u parameter value on a curve, and its (u, v) parameter values on a surface.

It is possible to query the same local properties for points as mentioned above, and additionally for 2D curves:

- the points corresponding to a minimum or a maximum of curvature,
- the inflection points.

Example: How to check the surface concavity

To check the concavity of a surface, proceed as follows:

1. sample the surface and compute at each point the Gaussian curvature.
2. if the value of the curvature changes of sign, the surface is concave or convex depending on the point of view.
3. to compute a Gaussian curvature, use the class *GeomLProp_SLprops*, which instantiates the generic class *LProp_SLProps* and use the method *GaussianCurvature*.

6.3 Global Properties of Shapes

The Global Properties of Shapes component provides algorithms for computing the global properties of a composite geometric system in 3D space, and frameworks to query the computed results.

The global properties computed for a system are :

- mass,
- mass center,
- matrix of inertia,
- moment about an axis,
- radius of gyration about an axis,
- principal properties of inertia such as principal axis, principal moments, and principal radius of gyration.

Geometric systems are generally defined as shapes. Depending on the way they are analyzed, these shapes will give properties of:

- lines induced from the edges of the shape,
- surfaces induced from the faces of the shape,
- volumes induced from the solid bounded by the shape.

The global properties of several systems may be brought together to give the global properties of the system composed of the sum of all individual systems.

The Global Properties of Shapes component is composed of:

- seven functions for computing global properties of a shape: one function for lines, two functions for surfaces and four functions for volumes. The choice of functions depends on input parameters and algorithms used for computation (*brepgprop_∗∗ global functions),*
- *a framework for computing global properties for a set of points (∗GProp_PGProps*),
- a general framework to bring together the global properties retained by several more elementary frameworks, and provide a general programming interface to consult computed global properties.

Modules *GeomLProp* and *Geom2dLProp* provide algorithms calculating the local properties of curves and surfaces

A curve (for one parameter) has the following local properties:

- point,
- derivative,
- tangent,

- normal,
- curvature,
- center of curvature.

A surface (for two parameters u and v) has the following local properties:

- point,
- derivative (for u and v),
- tangent line (for u and v),
- normal,
- max curvature,
- min curvature,
- main directions of curvature,
- mean curvature,
- gaussian curvature.

Among others, the following classes/methods are available:

- *GeomLProp_CLProps* - calculates the local properties of a curve (tangency, curvature,normal),
- *Geom2dLProp_CurAndInf2d* - calculates the maximum and minimum curvatures and the inflection points of 2d curves,
- *GeomLProp_SLProps* - calculates the local properties of a surface (tangency, the normal and curvature),
- *Geom2dLProp_Curve2dTool.Continuity* - calculates regularity at the junction of two curves.

Note that the B-spline curve and surface are accepted but they are not cut into pieces of the desired continuity. It is the global continuity, which is seen.

6.4 Adaptors for Curves and Surfaces

Some PythonOCC general algorithms may work theoretically on numerous types of curves or surfaces.

To do this, they simply get the services required of the analyzed curve or surface through an interface so as to a single API, whatever the type of curve or surface. These interfaces are called adaptors.

For example, *Adaptor3d_Curve* is the abstract class which provides the required services by an algorithm which uses any 3d curve.

GeomAdaptor module provides interfaces:

- on a Geom curve,
- on a curve lying on a Geom surface,
- on a Geom surface.

Geom2dAdaptor module provides interfaces :

- on a *Geom2d* curve.

BRepAdaptor module provides interfaces:

- on a face,
- on an edge.

When you write an algorithm which operates on geometric objects, use *Adaptor3d* (or *Adaptor2d*) objects. As a result, you can use the algorithm with any kind of object, if you provide for this object an interface derived from *Adaptor3d* or *Adaptor2d*. These interfaces are easy to use: simply create an adapted curve or surface from a *Geom2d* curve, and then use this adapted curve as an argument for the algorithm which requires it.

7 History of this document:

january 2018: first public release - tagged v0