C3D Labs

C3D Toolkit

incl. C3D Modeler, C3D Solver, C3D Converter

Developer Manual

2016

Content

INTRODUCTION

General Information

C3D geometric kernel can be used as a software component in Computer-Aided Design systems.

C3D geometric kernel is a software implementation of mathematical methods for building numerical models of the geometry of real and imaginary objects, as well as mathematical methods used to operate these models. Numerical models are used in systems for Computer Aided Design, Computer Aided Engineering and Computer Aided Manufacturing of the modeled objects. Numerical models of the geometry for real and imaginary objects are called geometric models.

A geometric model describes the shape of the modeled object and relations between model elements. In addition, a geometric model contains the history (methods and sequence) of its construction. Model elements have attributes that provide information about physical, technological and other properties.

Functionality

C3D geometric kernel contains the following elements of geometric model: description of the shape of the modeled object; description of relations between geometric model elements; model construction history and attributes of geometric model elements.

C3D geometric kernel uses Boundary Representation to describe the shape of the modeled object. C3D geometric kernel also supports Polygonal Representation. Solid Modeling and Surface Modeling methods are used to build a geometric model.

C3D geometric kernel builds polygonal model based on its boundary representation. A polygonal model is built by triangulating geometric model elements; it is used for visualization and calculations. In addition, C3D geometric kernel maps flat projections of geometric model (Mapping) calculates its Inertial Properties and detects collision of model elements (Collision Detection).

Relations between model elements provide geometric constraints for three- and two-dimensional objects of the geometric model. Geometric constraints are the conditions imposed on model elements that are expressed as equations. Geometric constraints permit to edit the model, to create assemblies and similar models, as well as to simulate mechanisms.

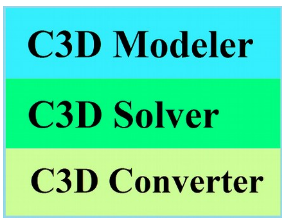
Models used to construct methods, their sequences and all the necessary input data are recorded in a construction log. The construction log permits to edit the geometric model and to rebuild it with new parameters.

Attributes can be used to store additional information on elements of the geometric model. Objects of geometric model as well as individual elements have attributes.

C3D geometric kernel uses the following formats to exchange geometric model data with other systems: STEP, IGES, ACIS (SAT), Parasolid(X_T, X_B), STL and VRML.

Structure and Distinctive Features

C3D geometric kernel consists of three modules shown in Figure.



C3D Modeler constructs a geometric model, edits the geometric model by changing its internal data, makes triangulation, calculates inertial properties of the model, builds flat projections of the model and detects collisions of model elements.

C3D Solver is responsible for defining relations between the elements of the geometric model, this permits you to edit the model, build similar models and simulate mechanisms by recalculating variation relations.

C3D Converter permits to share data of the geometric model with other systems.

So, C3D geometric kernel includes geometric constraints solving component C3D Solver and exchange data converter C3D Converter. This is the first distinctive feature of the C3D geometric kernel. Another distinctive feature is the direct access to C3D objects, which permits you to extend functionality by inheriting from C3D objects.

Theoretical Foundations

C3D geometric kernel uses a boundary representation that exactly describes the geometric shape of the modeled object. To describe geometric shapes, C3D uses a set of faces located at the border that separates the internal volume of the modeled object from the rest of the volume. The faces are curved surfaces jointed at their edges. The edges of the faces may have complex shapes. The faces are being created and jointed when a model is being built. Model construction and data management methods of C3D geometric kernel provide this.

Geometric constraints that describe relations between model elements and other conditions are formulated as equations. C3D geometric kernel uses a variational approach to find a solution that satisfies the equations. Variational approach ensures equal rights of all geometric constraints.

Boundary representation uses triangulation to enable construction of polygonal model representation for visualization and geometric calculations. Polygonal objects consist of triangular and quadrilateral plates that approximate the faces and broken lines that approximate the edges. Delaunay triangulation in the plane of parameters of surfaces is used in C3D geometric kernel.

C3D geometric kernel can create NURBS (Non-Uniform Rational B-Spline) copies for curves and surfaces. NURBS objects are used for direct modeling and for data exchange, when there is no direct correspondence between the objects of C3D geometric kernel and the objects in exchange formats.

C3D geometric kernel uses mathematical objects, methods and algorithms described in the book: Geometric Modeling, N. Golovanov. — Charleston: 2015, http://www.amazon.com/Geometric-Modeling-The-mathematics-shapes/dp/1497473195 .

Package

C3D geometric kernel package contains c3d.lib, c3d.dll, libc3d.so and libmath.dylib library files and a set of Include/*.h header files. In Windows, library files were compiled in 32bit/64bit, ISO/Unicode and Debug/Release configurations in VisualStudio 2005, VisualStudio 2008, VisualStudio 2010, VisualStudio 2012, VisualStudio 2013 and VisualStudio 2015 development environments. GCC compiler was used in 64bit, Unicode and Debug/Release configurations in Linux OS. Library files were also compiled in Mac OS (64bit, Unicode and Debug/Release) using Clang compiler. GCC compiler was used in Android OS for architectures armeabi-v7a and arm64-v8a.

Include/*.h header files were used to generate C3D geometric kernel documentation. Header files contain description of C3D geometric kernel objects and methods in Russian and English. C3D geometric kernel objects and methods are also described in this Developer Manual. Changes.txt file contains information on the changes of geometric kernel interface.

The distribution kit contains geometric kernel wrapper that permits to use .NET technology when applications are developed in C#.

Besides C3D geometric kernel, the package also contains test.exe application for Windows that demonstrates the capabilities of C3D geometric kernel, its source code, CMakeLists.txt file to generate application project and a set of files with models. Available the source code of example of using the C3D geometric kernel in the Android OS.

In order to run test.exe, please enter the key and the signature by selecting Help -> License_Key, Signature item in the menu.

Test Application

Ready-to-use Windows-based test application for C3D geometric kernel is stored in Example/Demo folder.

Test_VS2005.sln and Test_VS2005.vcproj files contain C3D geometric kernel test application solution and project respectively for Microsoft VisualStudio 2005.

Test_VS2012.sln, Test_VS2012.vcxproj and Test_VS2012.vcxproj.filters files contain C3D geometric kernel test application solution and project respectively for Microsoft VisualStudio 2012.

To create a project and compile the test application of geometric kernel C3D, it is required to perform the following steps:

1. Create a test folder (for example, TestApp) in any location of your choice.
2. Choose an archive in the «C3D» catalog corresponding to your development environment.
3. Copy <Debug>, <Include>, <Release> folders from the chosen archive to the test folder.
4. Copy <Source> folder from the «Example» archive to the test folder.
5. Make sure that the test folder (TestApp) contains <Debug>, <Include>, <Release> and <Source> subfolders.
6. Install CMake and use «Add CMake to the system PATH for all users» option during installation.
7. Create a project for test application using the following procedure.
   Run CMake to generate a project using CMakeLists.txt file.
   Specify <path_to_testapp>\TestApp\Source folder in «Where is the source code» field.
   Specify <path_to_testapp>\TestApp\Build folder in «Where to build the binaries» field.
   Click **Configure** to make settings for the project.
   Confirm creation of <path_to_testapp>\TestApp\Build folder in «Create Directory» dialog box.
   Specify development environment configuration appropriate to C3D version in «Specify the generator for this project» dialog box.
   Click **Generate** to build project files.
8. Run newly created TestApp\Build\Test.sln test application project in the development environment.
9. In order to activate C3D geometric kernel, before compilation specify the actual key and the signature in test_manager.cpp file to modify EnableMathModules(...) method call in Manager object constructor.
10. After compiling, run newly generated test application test.exe from TestApp\Debug or TestApp\Release folder, respectively.

The above procedure is described in readme.txt file.

Development in .NET Environment

C3D geometric kernel can work in .NET environment. You should use the wrapper included in C3D geometric kernel package to develop applications in .NET environment.

C3D geometric kernel wrapper is NetC3D.dll file that was built using .Framework4.5 platform in 32bit/64bit, Debug/Release configurations and VisualStudio 2012 and in VisualStudio 2013 development environments. The library was compiled with Strong Name signature support.

Please execute the following procedure to use C3D geometric kernel in newly developed C# applications:

1. In C3D geometric kernel package, select NetC3D.dll file with the required configuration: 32bit/64bit or Debug/Release in VisualStudio 2012 or VisualStudio 2013 development environment.
2. Copy c3d.dll file from the same package for the same configuration and development environment: 32bit/64bit, Debug/Release or VisualStudio 2012/VisualStudio 2013 to the folder containing NetC3D.dll.
3. Include NetC3D.dll file into the current project: References->Add Reference->Browse..., then select

NetC3D.dll file.

4. Enter the license key and the signature before calling functions from NetC3D.dll. This can be done as follows:
   ```
   var key = Environment.GetEnvironmentVariable("C3Dkey");
   var signature = Environment.GetEnvironmentVariable("C3Dsignature");
   NetC3D.ToolEnabler.EnableMathModules(key, signature);
   ```
   where C3Dkey and C3Dsignature are environment variables containing the key and the signature.

M.1. METHODS USED TO SOLIDS CONSTRUCTING

The main elements of the geometric model serve the solids. C3D geometric kernel constructs solids that fully or partially describe the surface of the modeled object. The solid can be closed and non-closed. Closed solid doesn't contain boundary edges. It describes the whole surface of the modeled object and the set of its internal points. Non-closed solid contains boundary edges. It describes only a part of surface of the modeled object. Many solids have a simple form and are built on the basis of points, curves and surfaces.

M.1.1. Constructing an Elementary Solid

Method
MbResultType
ElementarySolid (SArray<MbCartPoint3D> & **points**,
ElementaryShellType *solidType*,
const MbSNameMaker & names,
MbSolid*& **result**)
constructs an elementary solid (a sphere, a torus, a cylinder, a cone, a straight parallelepiped, a pyramid or a rounded plate) based on specified points.

Method input parameters are:

- **points** is a set of control points,
- *solidType* is the type of the created solid,
- names is faces namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns an error code from MbResultType enumeration.

This method is declared in action_solid.h file.

points parameter contains the control points used to construct a solid. *solidType* parameter defines the type of the created solid. names parameter is responsible for naming faces of the constructed solid.

The number of required control points depends on the type of created solid. Table M.1.1.1 contains data on the number of control points in **points** set required to construct a solid that belongs to *solidType* type.

Table M.1.1.1.

solidType	**Solid type**	**Number of control points**
et_Sphere	sphere	3 points
et_Torus	torus	3 points
et_Cylinder	cylinder	3 points
et_Cone	cone	3 points
et_Block	block	4 points
et_Wedge	wedge	4 points
et_Plate	plate	4 points
et_Prism	prism	the number of base nodes + 1 point
et_Pyramid	pyramid	the number of base nodes + 1 point

When a sphere is constructed, **points**[0] from the set defines the center of the sphere, **points**[1] defines the direction of **axisZ** in the local coordinate system of the sphere, **points**[2] along with the points mentioned above define the plane of **axisX** and **axisZ** in the local coordinate system of the sphere. The distance between **points**[0] and **points**[2] is defined by the radius of the sphere, see Figure M.1.1.1.

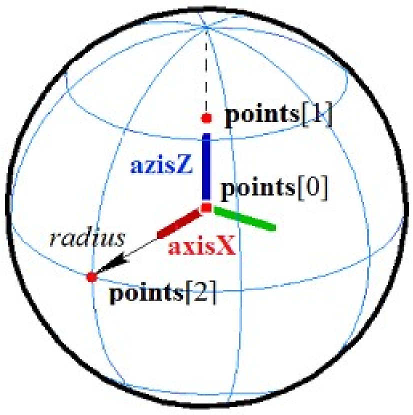


Figure M.1.1.1.

When a torus is constructed, a point from **points**[0] set defines the center of the torus, **points**[1] point defines the direction of **axisX** in torus local coordinate system, **points**[2] point along with the previous points defines the plane of **axisX** and **axisZ** in torus local coordinate system. The distance between **points**[0] and **points**[1] points defines the larger torus radius; the distance between **points**[1] and **points**[2] points defines the smaller torus radius, see Figure M.1.1.2.

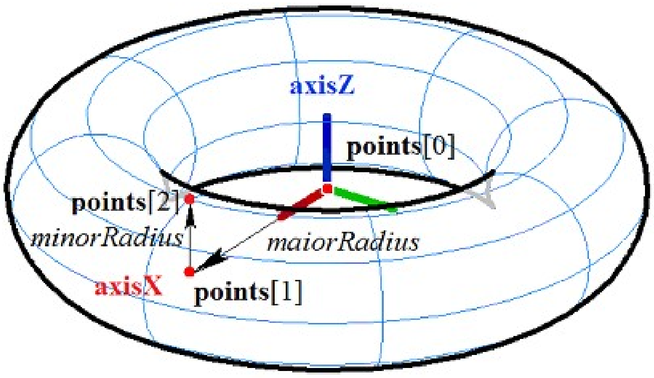


Figure M.1.1.2.

When a cylinder is constructed, a point from **points**[0] set defines the center of cylinder lower base, **points**[1] point defines the center of the upper cylinder base and the direction of **axisZ** in cylinder local coordinate system, **points**[2] point along with the previous points defines the plane of **axisX** and **axisZ** in cylinder local coordinate system. The distance between **points**[0] and **points**[1] points defines cylinder height, the distance from **axisZ** to **points**[2] point defines cylinder radius, see Figure M.1.1.3.

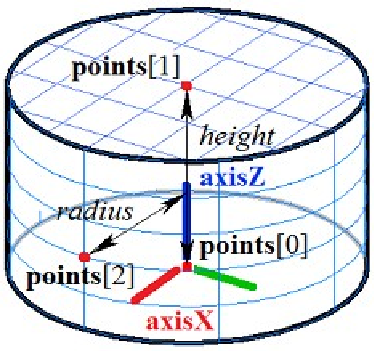


Figure M.1.1.3.

When a cone is constructed, a point from **points**[0] set defines cone vertex, **points**[1] point defines the center of the cone base and **axisZ** direction in cone local coordinate system, **points**[2] point along with the previous points defines the plane of **axisX** and **axisZ** in cone local coordinate system. The distance between **points**[0] and **points**[1] points defines cone height; the cone angle is defined taking into account the fact that **points**[2] point lies on cone lateral surface, see Figure M.1.1.4.

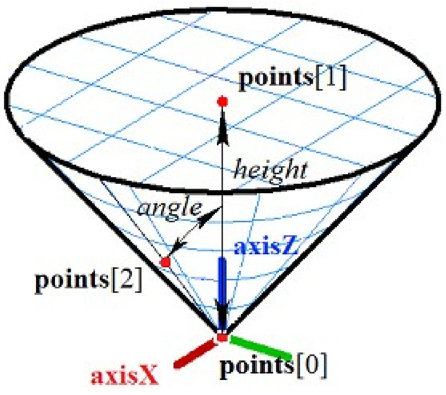


Figure M.1.1.4.

When a rectangular block is constructed, **points**[0] and **points**[1] points define an edge and two vertices of the block, **points**[2] point along with the previous points define the plane of lower block base, block edge that is parallel to **points**[0] edge and **points**[1] edge goes through **points**[2] point, and **points**[3] point defines the plane of the upper base of the block, see Figure M.1.1.5.

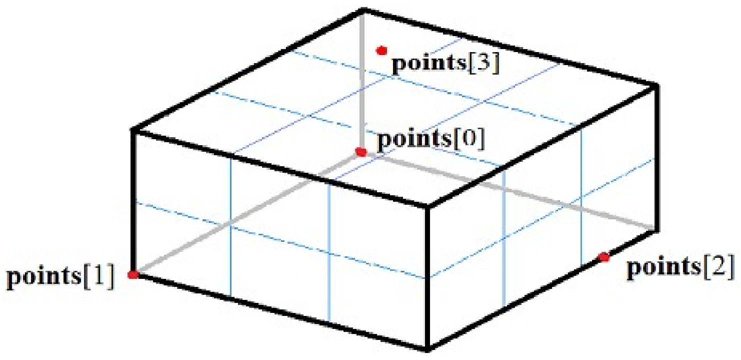


Figure M.1.1.5.

When a rectangular wedge is constructed, **points**[0] and **points**[1] points define an edge and two vertices of the wedge, **points**[2] point along with the previous points defines the plane of wedge lower base and its vertex, wedge edge that is parallel to **points**[0] and **points**[1] edge goes through point **points**[2], and **points**[3] point defines the plane of the upper wedge base, see Figure M.1.1.6.

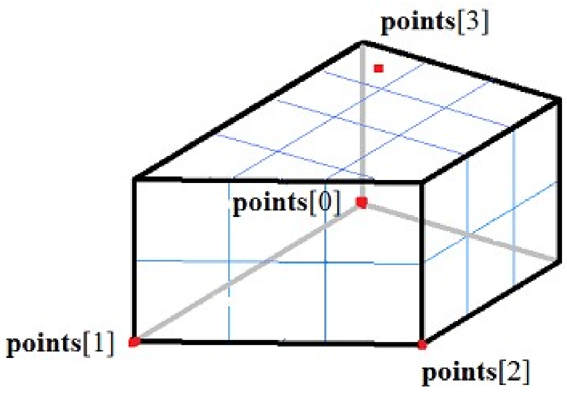


Figure M.1.1.6.

When a rectangular plate with cylindrical ends is constructed, **points**[0] and **points**[1] points define an edge and two vertices of the plate, **points**[2] point along with the previous points defines the plane of plate lower base, plate edge is parallel to **points**[0] and **points**[1] edge goes through **points**[2] point, and **points**[3] point defines the upper plate base plane, see Figure M.1.1.7.

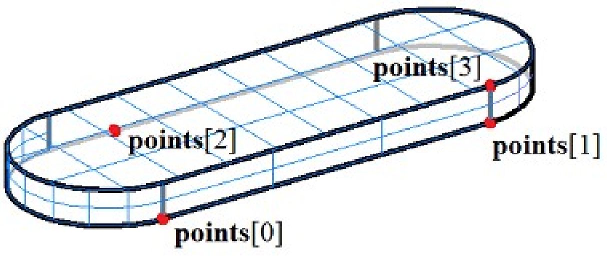


Figure M.1.1.7.

When a right-angle prism with a polygon at the base is constructed, **weightCentre**, **points**[0] and **points**[1] points define the plane of prism lower base, where **weightCentre** is the center of gravity point of the base. **points**[0], **points**[1], ..., **points**[*n*–1] projection of points define the polygonal base, and prism height is defined by the distance from the plane of the lower base to **points**[*n*] last point. In Figure M.1.1.8, you can see a right-angle prism with a pentagonal base.

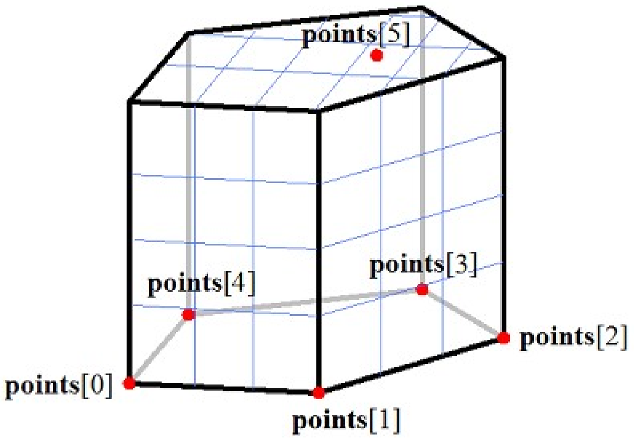


Figure M.1.1.8.

When a pyramid with a polygon at the base is constructed, **weightCentre**, **points**[0] and **points**[1] points define the plane of pyramid lower base, where **weightCentre** is the center of gravity point of the base. **points**[0], **points**[1], ..., **points**[*n*–1] points define the polygonal base, and **points**[*n*] last point defines the top of the pyramid. In Figure M.1.1.9, you can see a pyramid with a pentagonal base.

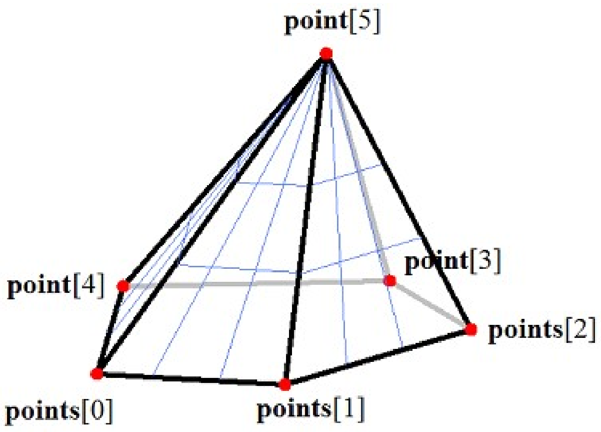


Figure M.1.1.9.

Points from **points** set that define the base of the pyramid or prism may be located at the vertices of a regular polygon. There may be any polygon at the base of a prism or a pyramid.

ElementarySolid method adds the MbElementarySolid constructor to the log of the newly constructed solid. This constructor contains all data required to construct the solid. MbElementarySolid constructor is declared in cr_elementary_solid.h file.

test.exe test application constructs elementary solids using the points specified by «Create->Solid->Elementary->» and «Create->Solid->By Points->» menu commands.

M.1.2. Constructing an Elementary Solid by a Given Surface

Method
MbResultType
ElementarySolid (const MbSurface & **surface**,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs an elementary solid by a given surface.

Method input parameters are:

- **surface** is an elementary surface,
- names is faces namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns an error code from MbResultType enumeration.

This method is declared in action_solid.h file.

surface parameter contains the original surface. names parameter is responsible for naming faces of the constructed solid.

An elementary surface may be represented by MbSphere sphere, MbTorus toroidal surface, MbCylinder cylindrical surface or MbCone conical surface. In Figure M.1.2.1, you can see a spherical surface and a solid that was constructed by it.

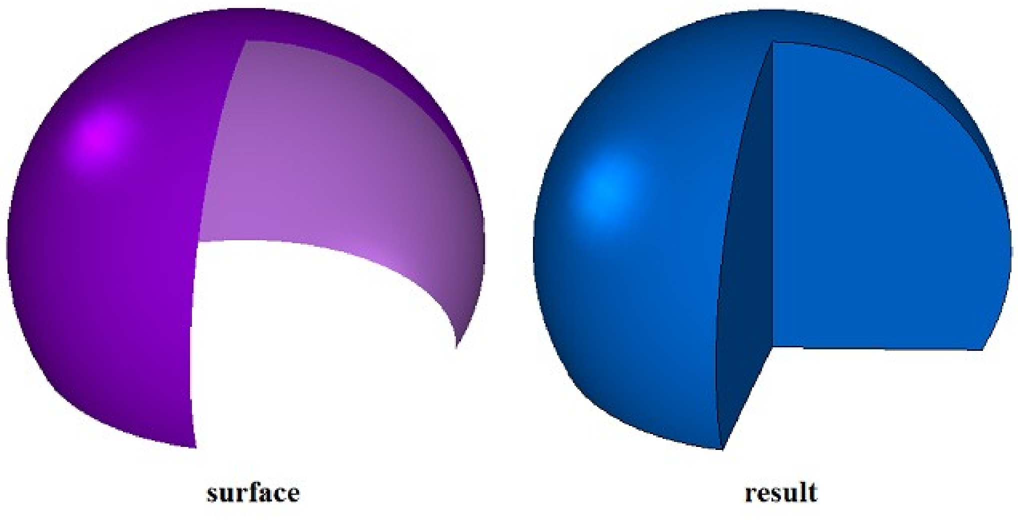


Figure M.1.2.1.

In Figure M.1.2.2, you can see a toroidal surface and a solid that was constructed for it.

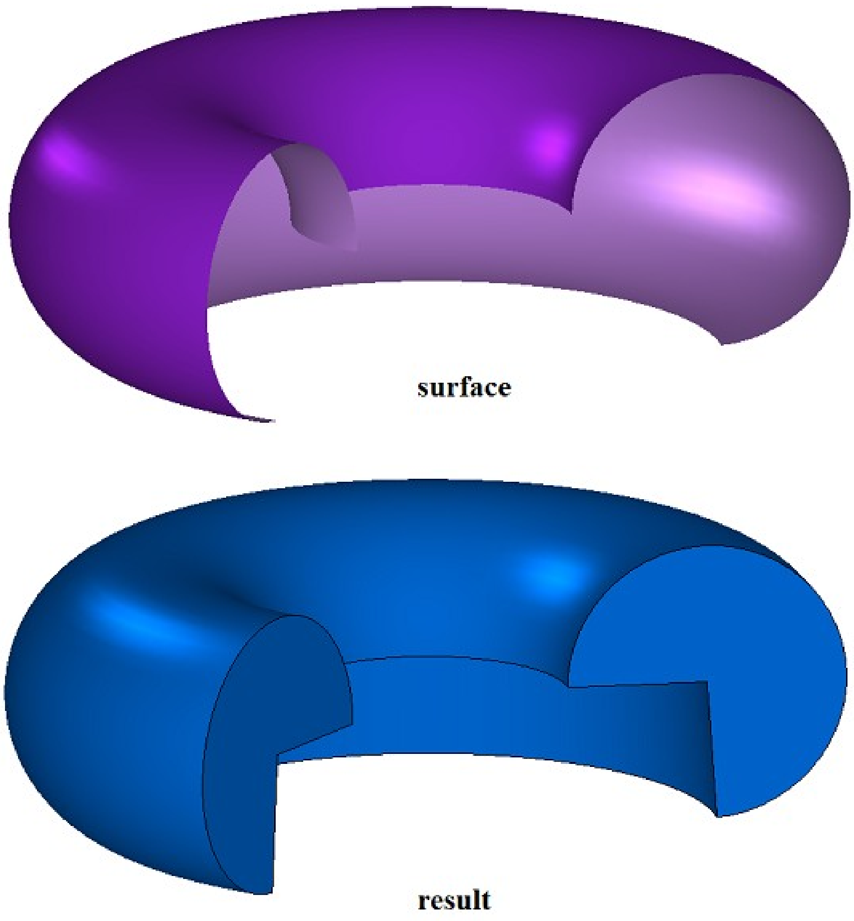


Figure M.1.2.2.

In Figure M.1.2.3, you can see a cylindrical surface and a solid that was constructed for it.

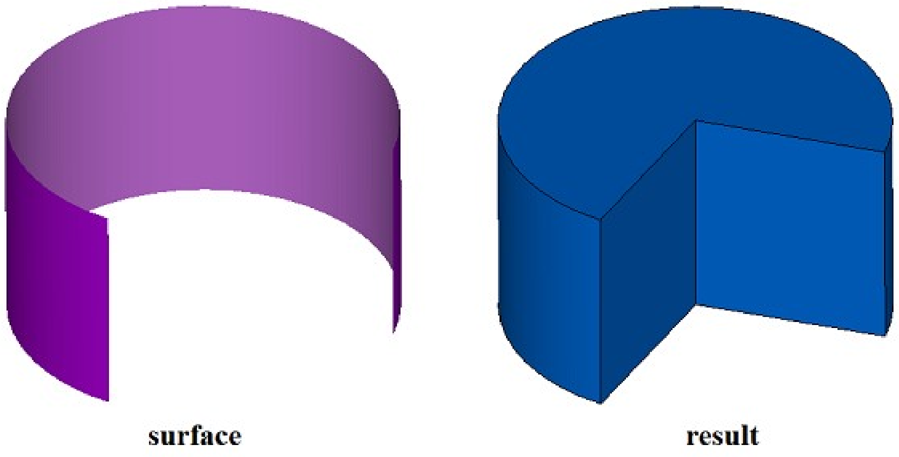


Figure M.1.2.3.

In Figure M.1.2.4, you can see a conical surface and a solid that was constructed for it.

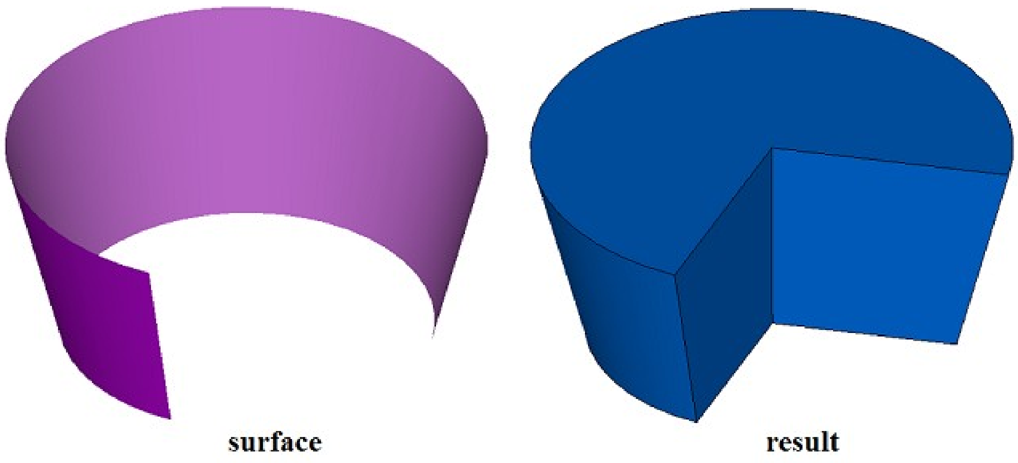


Figure M.1.2.4.

If these are cyclically closed surfaces, then the solids to be constructed for them would have a corresponding form. If the elementary surface does not belong to any of these types, then the method returns rt Error error code.

ElementarySolid method adds MbRevolutionSolid constructor to the log of the newly constructed solid. This constructor contains all data required to construct the solid. MbRevolutionSolid constructor is declared in cr_revolution_solid.h file.

test.exe constructs an elementary solid for a given surface using «Create->Solid->By surface->By elementary surface» menu command.

M.1.3. Constructing an Extrusion Solid

Method
MbResultType
ExtrusionSolid (const MbSweptData & **sweptData**,
const MbVector3D & **direction**,
const MbSolid * **solid**1,
const MbSolid * **solid**2,
bool *checkIntersection*,
ExtrusionValues & *params*,
const MbSNameMaker & names,
PArray<MbSNameMaker> & cnames,
MbSolid *& **result**)
constructs an extrusion solid.

Method input parameters are:

- **sweptData** are data on curve generators,
- **direction** is the extrusion direction,
- **solid1** is used when option «To next object» in the forward direction is selected,
- **solid2** is used when option «To next object» in the backward direction is selected,
- *checkIntersection* is a flag indicating that it is necessary to merge solid1 and solid2 solids subject to checking the intersection,
- *params* are construction parameters,
- names is face namer,
- cnames are namers of curve generator segments.

Method output parameter is **result** constructed solid. If successful, the method returns rt_Success, otherwise it returns an error code from MbResultType listing.

This method is declared in action_solid.h file.

Extrusion solid belongs to the type of motion solids, which are constructed by moving a generating curve along a guiding curve. A line segment is a guiding curve for an extruded solid. An extruded solid is constructed by moving one or more curves along the segment, the direction of which is determined by **direction** vector.

sweptData parameter contains information on generator curves. MbSweptData class and ExtrusionValues structure are described in swept_parameter.h file. Generating curves may be two-dimensional **contours** on **surface** or contours in **contours3D** space. In particular cases, two-dimensional **contours** may be located on a plane. **contours** may have arbitrary orientation. **contours** may be nested with each other. **contours** shouldn't intersect with each other.

A solid can be constructed in the forward direction in respect to **direction** vector, in the backward direction in respect to **direction** vector, as well as in both directions. Construction parameters for each direction are set by MbSweptSide objects.

params parameter contains information on MbSweptSide *side*1 extrusion method in forward direction as well as information on MbSweptSide *side*2 extrusion method in the backward direction. Extrusion in each direction can be executed using three methods. If *way*==sw_scalarValue then extrusion is executed to *scalarValue* length in the direction of *side*1 or *side*2, respectively. If *way*==sw_shell then extrusion is executed to the nearest **solid**1 or **solid**2 object, respectively. If *way*==sw_surface then extrusion is executed to *side*1.**surface** or *side*2.**surface**, respectively, if *side*1.*distance*=0 or *side*2.*distance*=0. If *way*==sw_surface and *distance*!=0 then extrusion is executed to the equidistant surface to *side*1.**surface** or to the equidistant surface to *side*2**surface**, respectively. If *side*1.*rake*!=0 or *side*2.*rake*!=0, then graded extrusion is executed with *side*1.*rake* or *side*2.*rake* grade in the respective direction. *params.thickness*1 parameter defines outward offset from the generator curve, and *params.thickness*2 parameter defines inward offset from the generator curve. *params.shellClosed* parameter controls whether the constructed solid is closed. *params.checkSelfInt* parameter defines the need to check the result of construction for self-intersection. By default *params.checkSelfInt*=false and the check is not performed.

In Figure M.1.3.1, you can see the data used for construction, as well as the scheme to inherit the parameters of constructed extrusion solid (ExtrusionValues & *params*).

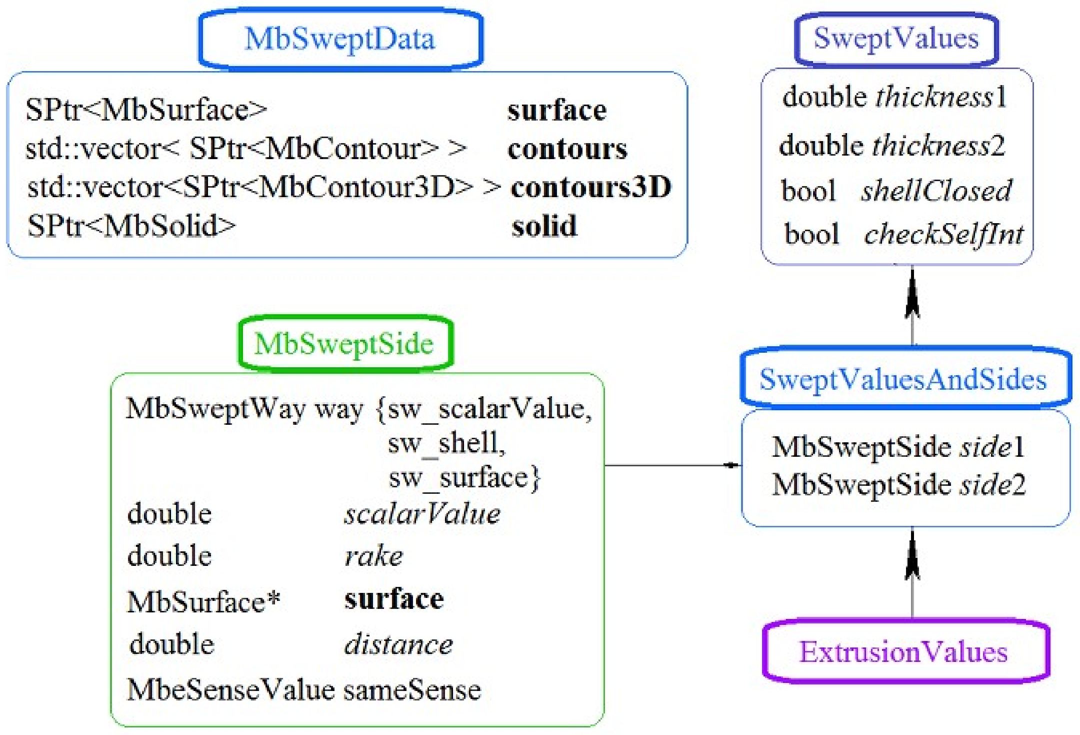


Figure M.1.3.1.

names and cnames parameters are responsible for naming the faces of the newly constructed solid.
In Figure M.1.3.2, you can see a two-dimensional **contour** and **surface** (MbPlane) flat surface.

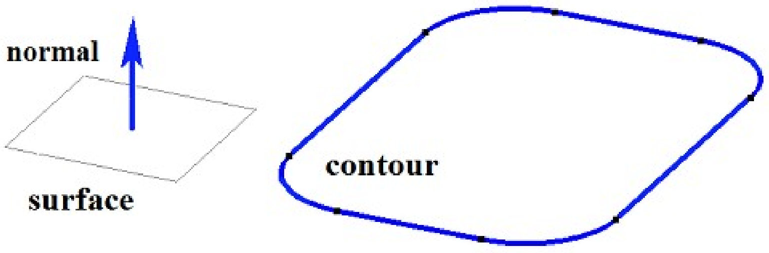


Figure M.1.3.2.

In Figure M.1.3.3, you can see a closed solid that was constructed by using specified parameters to extrude the contour shown in Figure M.1.3.2. Each contour segment has a corresponding face of the solid, its name was taken from the corresponding element of cnames[0] name generator.

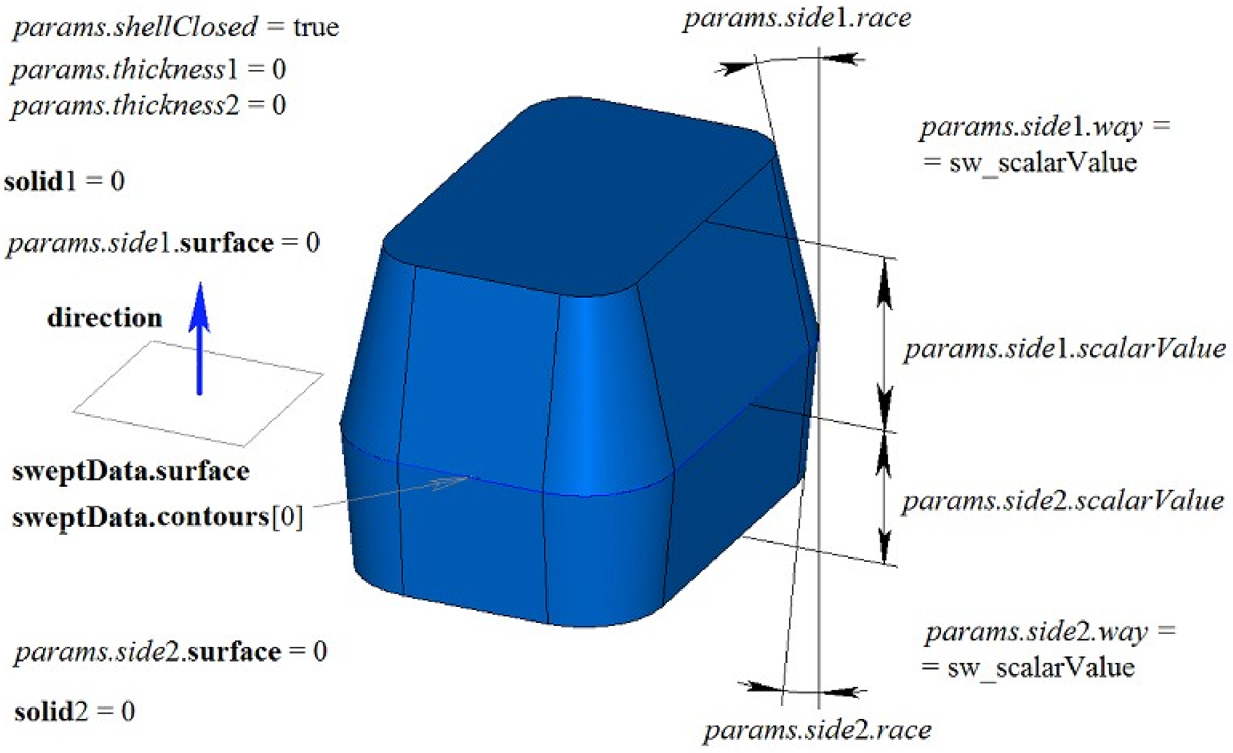


Figure M.1.3.3.

In Figure M.1.3.4, you can see a thin-walled closed solid that was constructed by extrusion based on specified contour parameters shown in Figure M.1.3.2.

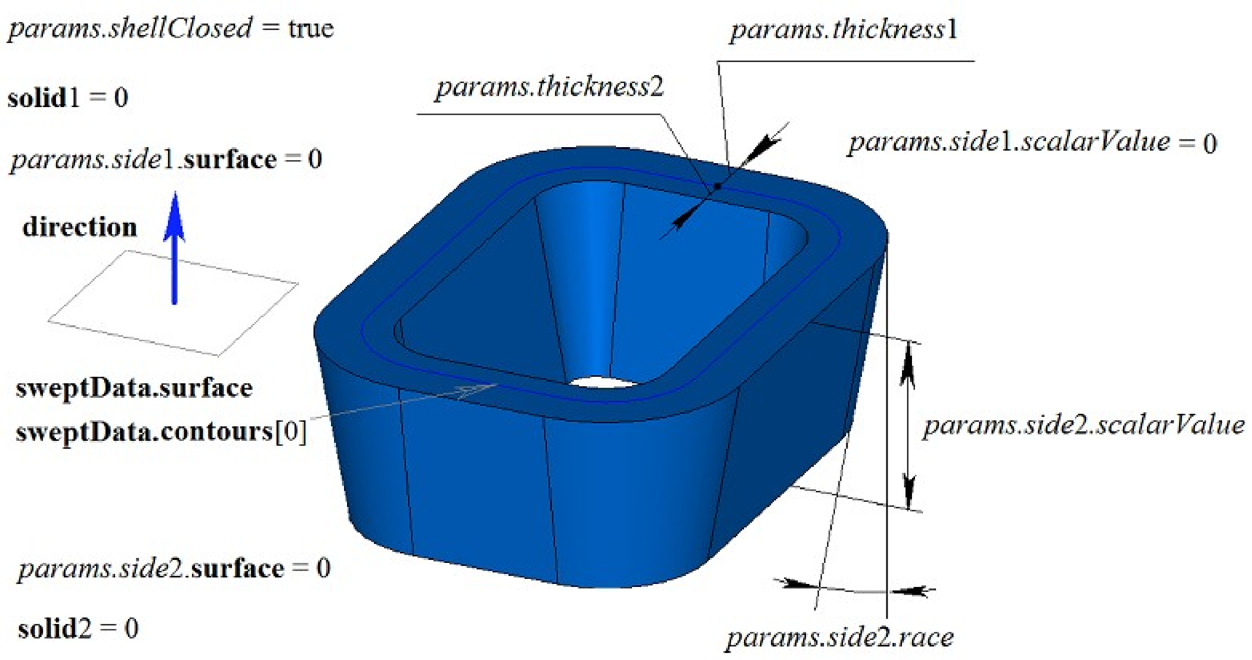


Figure M.1.3.4.

In Figure M.1.3.5, you can see a non-closed solid that was constructed by using extrusion with specified contour parameters shown in Figure M.1.3.2. Parameters used to construct the solid shown in Figure M.1.3.3 differ from parameters used to construct the solid shown in Figure M.1.3.5 only by *params.shellClosed* value.

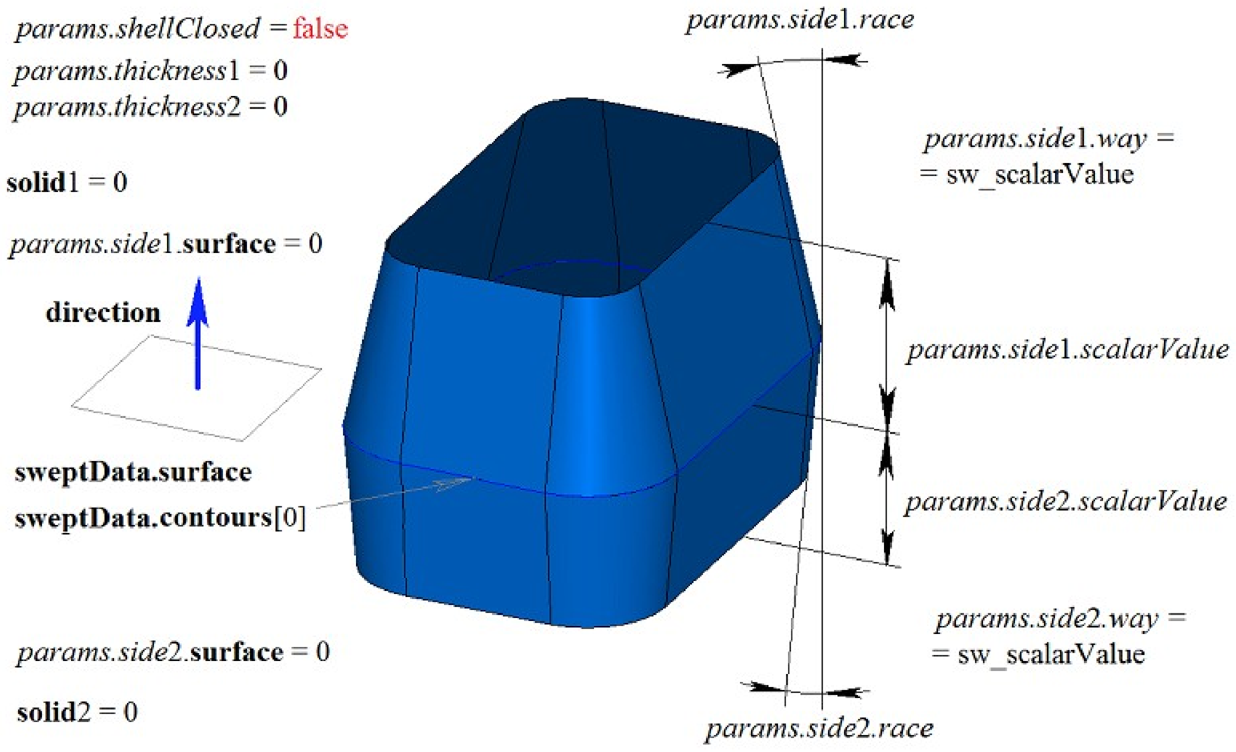


Figure M.1.3.5.

In Figure M.1.3.6, you can see two-dimensional **contour**, flat **surface** (MbPlane) as well as two solids (**solid**1 and **solid**2) that will be used to construct an extruded solid. For construction **solid**1 and **solid**2 solids should completely cover contour motion path in the appropriate direction. In this case, the following parameters should be taken into account: *params.side*1*.rake*, *params.side*2*.rake*, *params.thickness*1, *params.thickness*2.

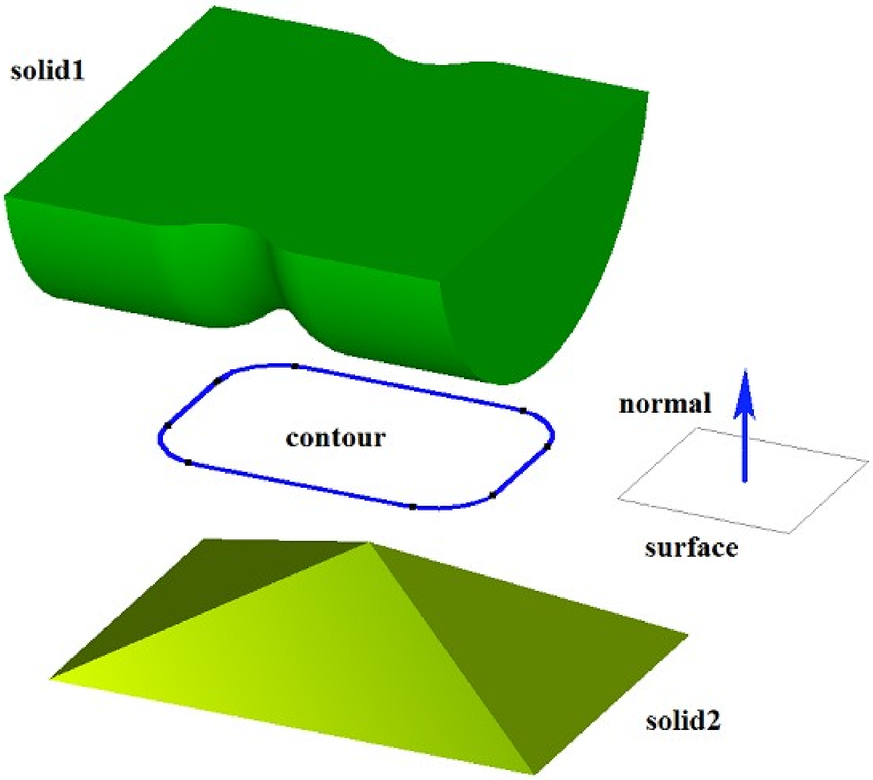


Figure M.1.3.6.

Such construction is executed by extruding the contour to a length exceeding the maximum distance to the specified solid and then subtracting the specified solid from the newly constructed solid.

In Figure M.1.3.7, you can see a solid that was constructed by extruding the contour shown in Figure M.1.3.6 with «To the nearest objects» option selected for **solid**1 and **solid**2 solids.

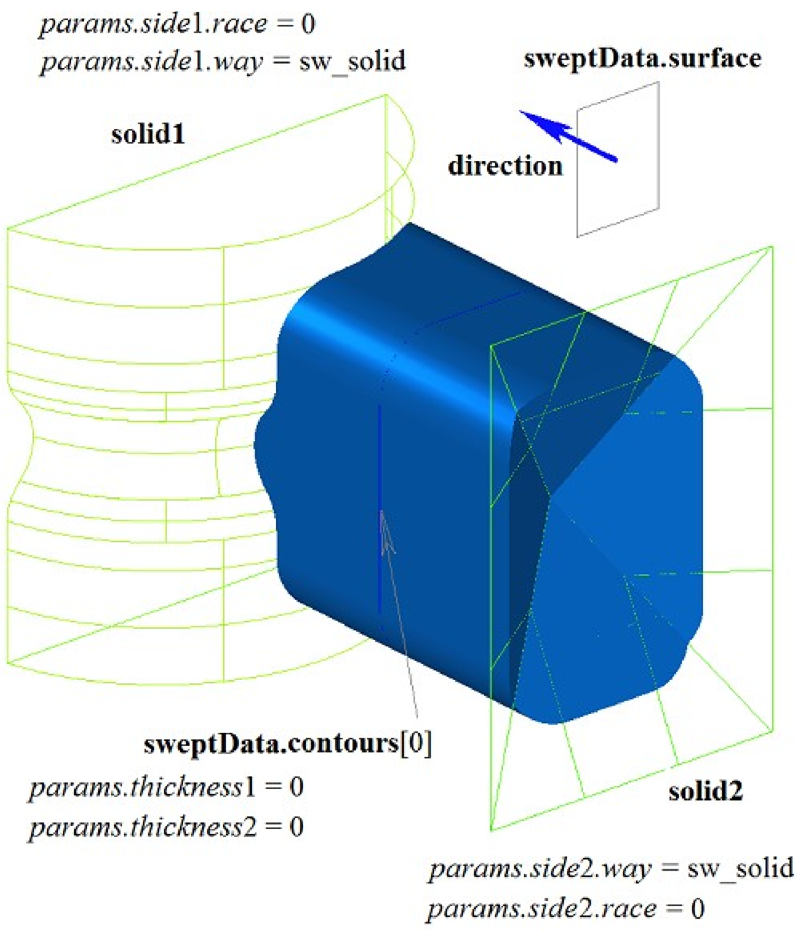


Figure M.1.3.7.

In Figure M.1.3.8, you can see a thin-walled solid with sloping faces constructed by extruding the contour shown in Figure M.1.3.6 with «To the nearest objects» option selected for **solid**1 and **solid**2 solids.

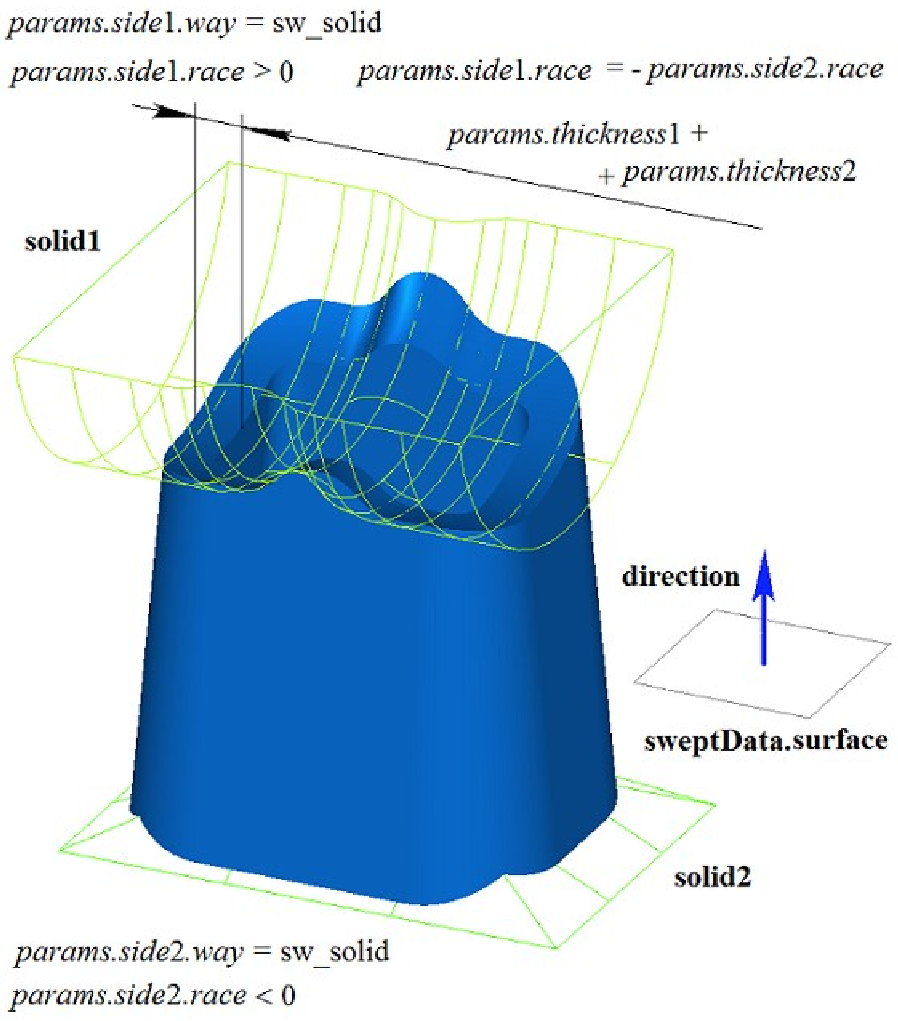


Figure M.1.3.8.

In Figure M.1.3.9, you can see a two-dimensional **contour**, **surface** (MbPlane) flat surface and two surfaces, **surface**1 and **surface**2 (that will be used to construct the extruded solid). For the construction **surface**1 and **surface**2 should completely cover the path of the contour moved in the appropriate direction. In this case, the following parameters should be taken into account: *params.side*1*.rake*, *params.side*2*.rake*, *params.thickness*1, *params.thickness*2. The extruded solid is cut off by the specified surfaces or by the surfaces equidistant to them if *params.side*1*.distance* or *params.side*2*.distance* are not equal to zero.

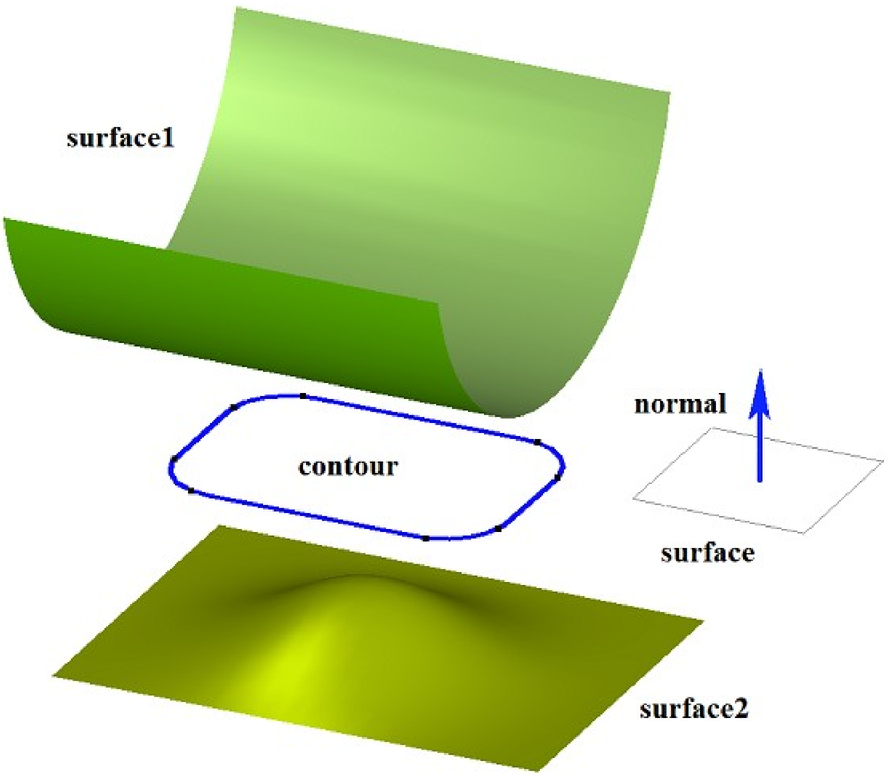


Figure M.1.3.9.

In Figure M.1.3.10, you can see a solid that was constructed by extruding the contour shown in Figure M.1.3.9 with «To the surface» options. **surface**1 and **surface**2 were specified as such surfaces.

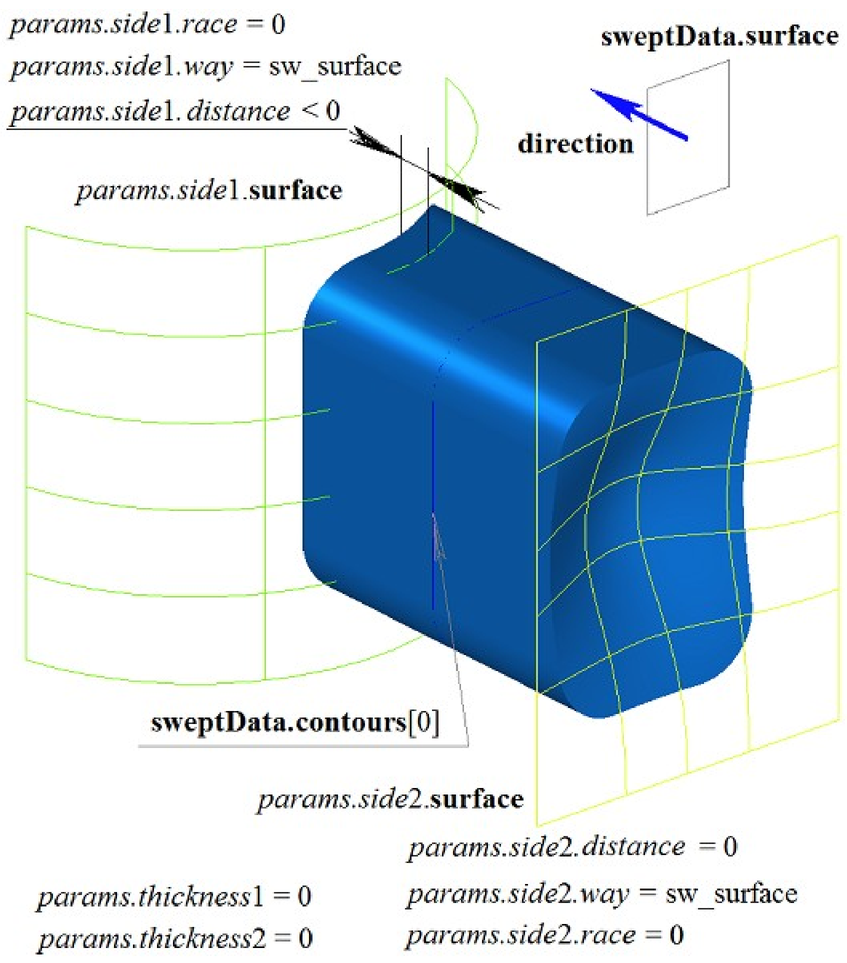


Figure M.1.3.10.

In Figure M.1.3.11, you can see a thin-walled solid with sloping faces that was constructed by extruding the contour shown in Figure M.1.3.9 with «To the surface» options. **surface**1 and **surface**2 were specified as such surfaces.

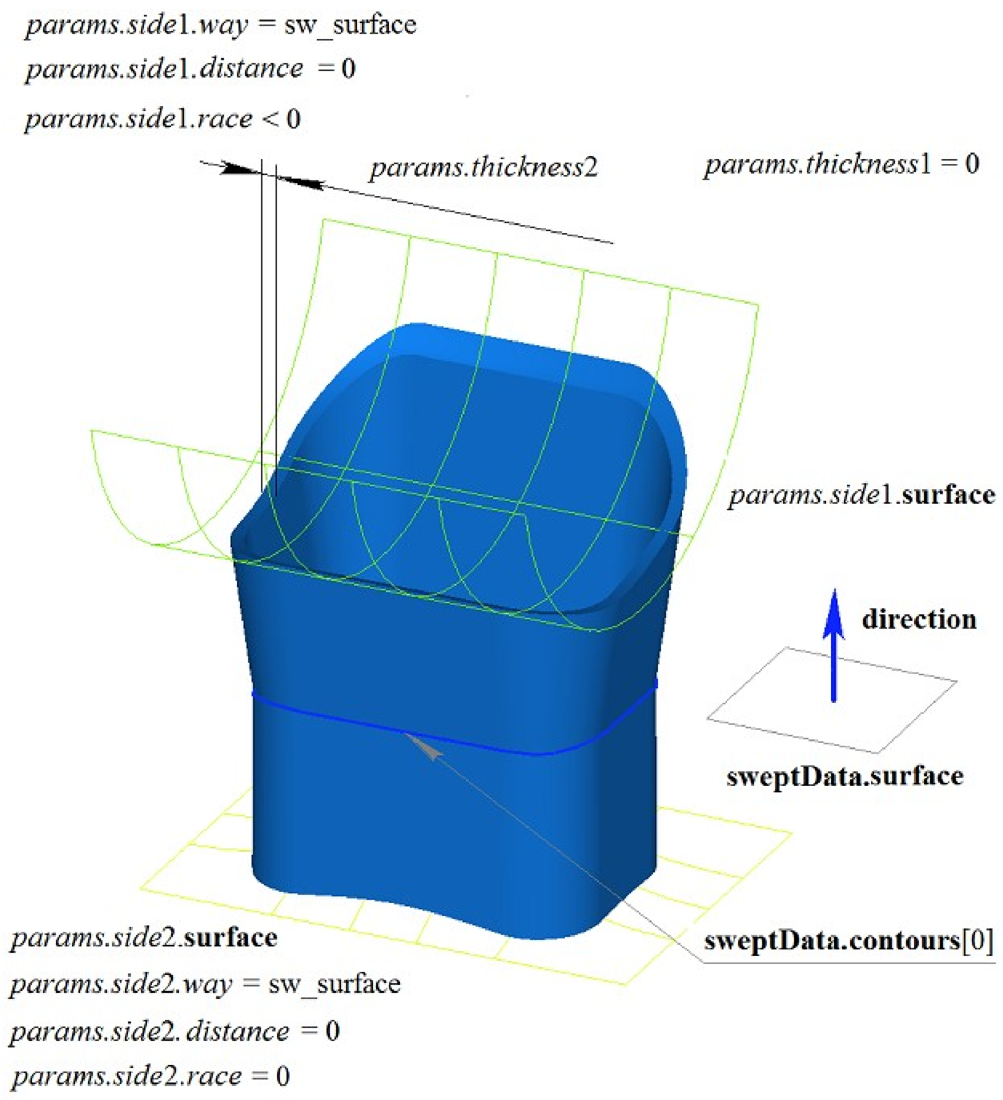


Figure M.1.3.11.

A two-dimensional contour may be drawn on a flat surface or on a curved surface. For example, a solid can be constructed by extruding a contour at a curved surface created from a cycle of one of the faces of the solid solid shown in Figure M.1.3.12.

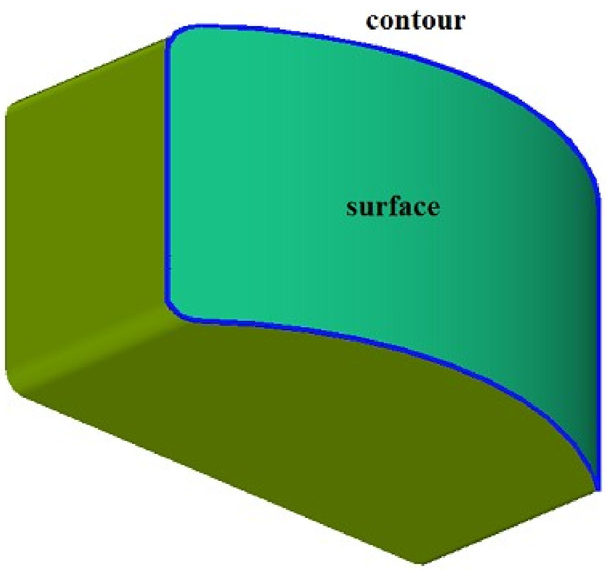


Figure M.1.3.12.

In Figure M.1.3.13, you can see a solid that was constructed by extruding the contour on a curved surface shown in Figure M.1.3.12.

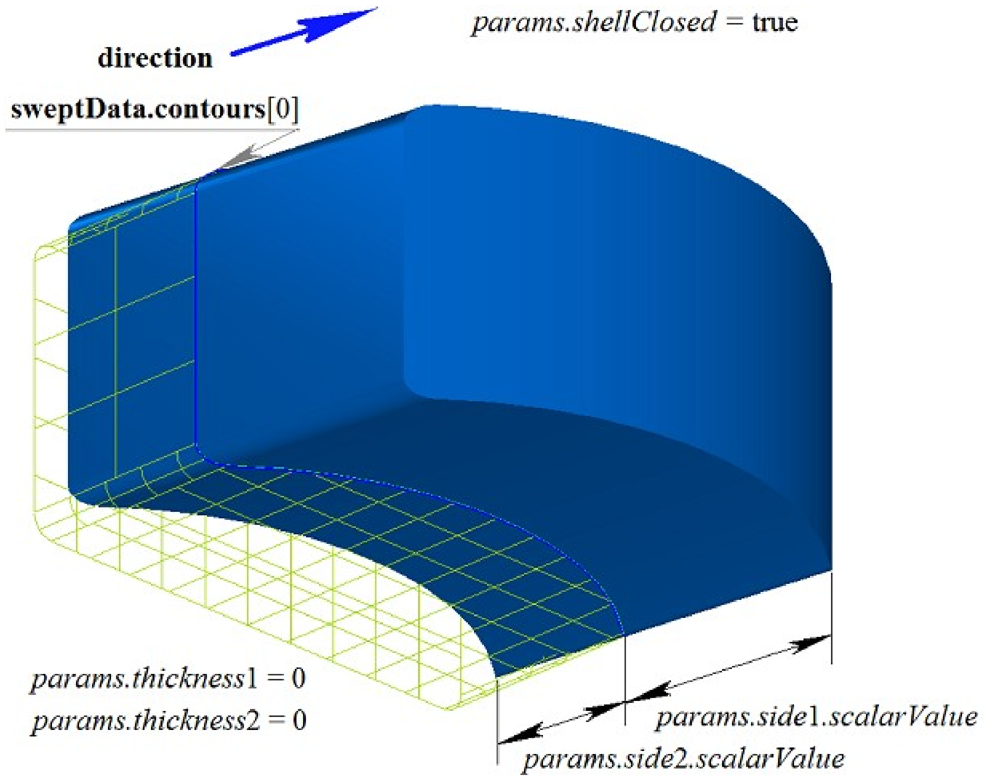


Figure M.1.3.13.

In Figure M.1.3.14, you can see a thin-walled solid that was constructed by extruding the contour on a curved surface shown in Figure M.1.3.12.

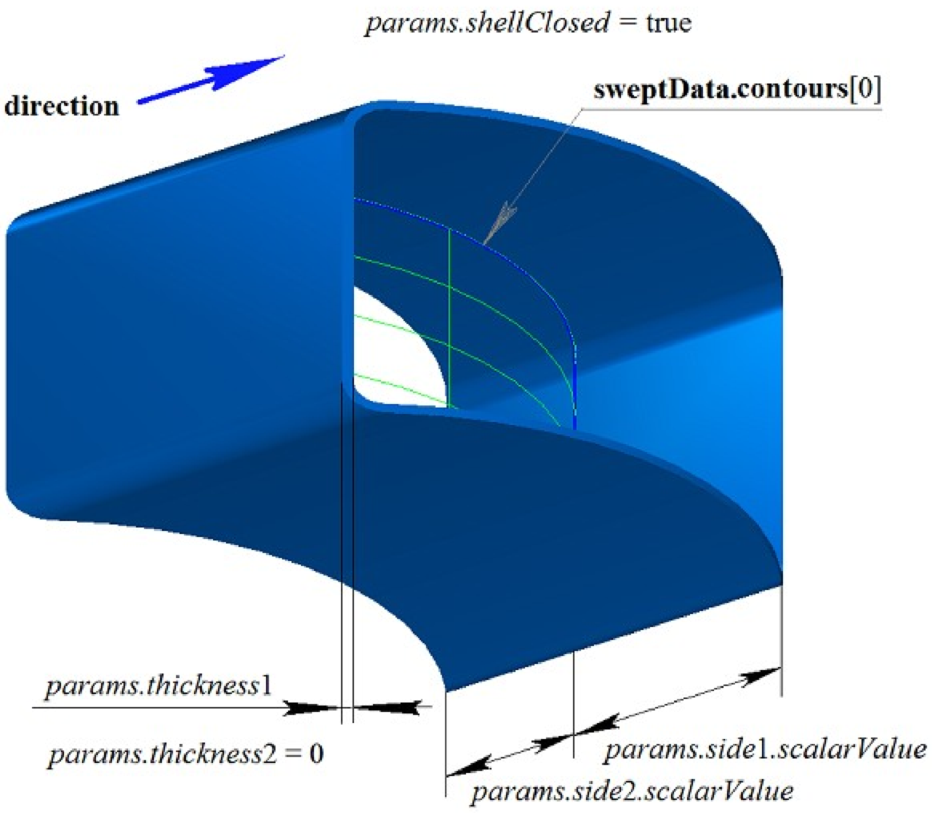


Figure M.1.3.14.

In Figure M.1.3.15, you can see a non-closed solid that was constructed by extruding the contour on a curved surface shown in Figure M.1.3.12.

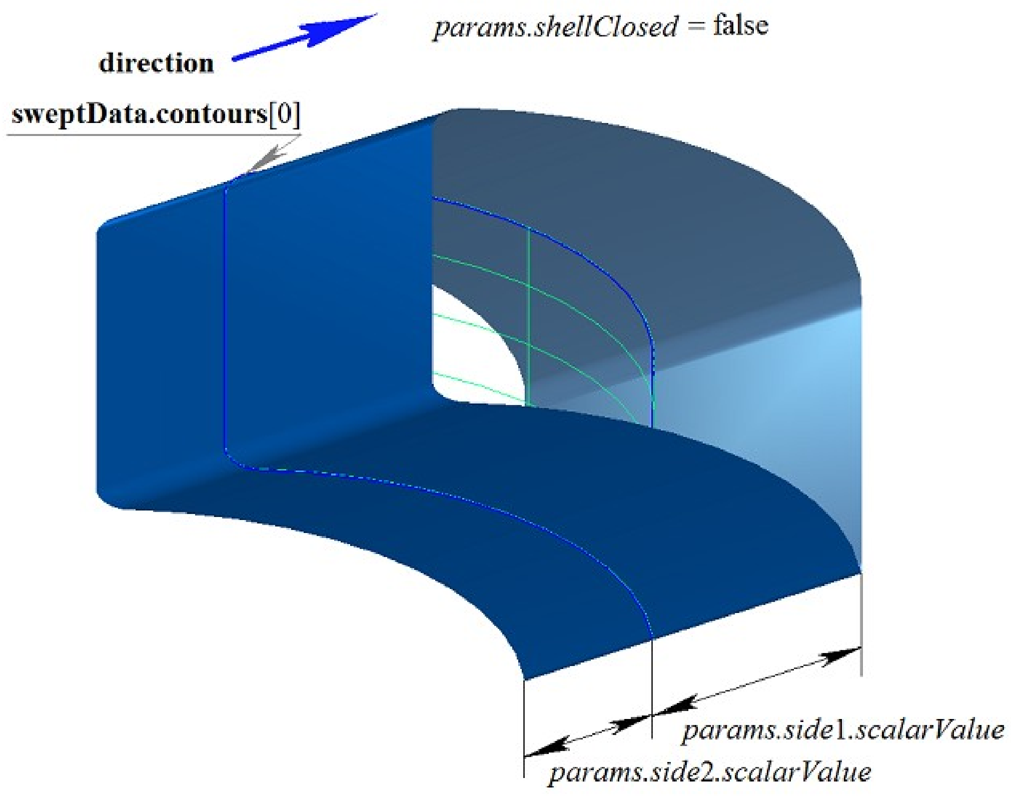


Figure M.1.3.15.

If one surface contains a set of non-intersecting two-dimensional contours, then the considered method defines external and nested internal contours (multilevel nesting can be used). In Figure M.1.3.16, you can see a set of non-intersecting two-dimensional **contours** and **surface** (MbPlane) flat surface.

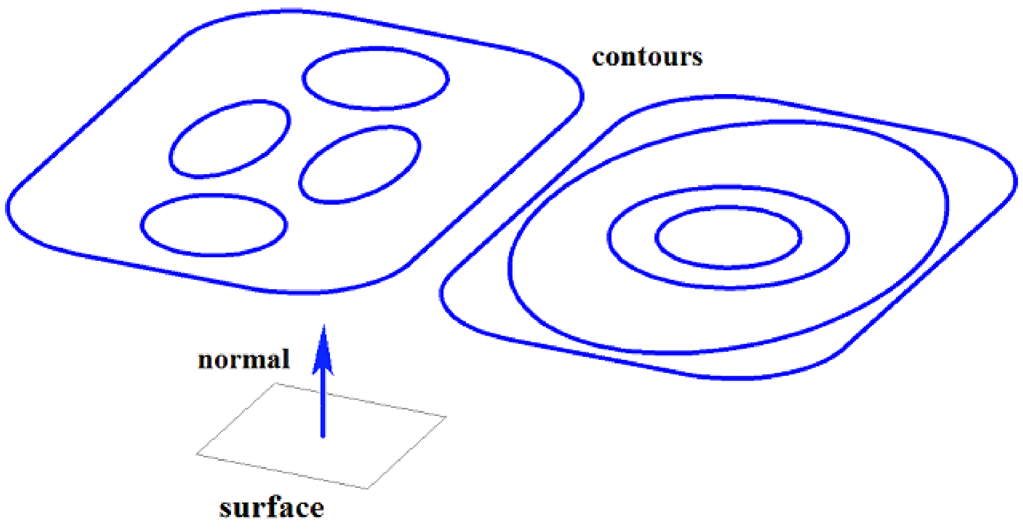


Figure M.1.3.16.

In Figure M.1.3.17, you can see a multi-part closed solid that was constructed by extruding the set of contours shown in Figure M.1.3.16.

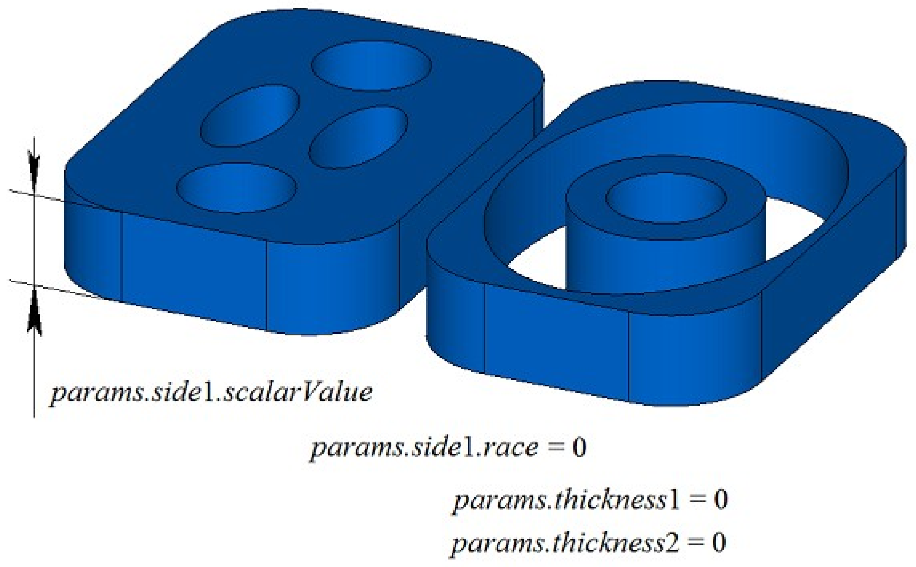


Figure M.1.3.17.

In Figure M.1.3.18, you can see a multi-part closed solid that was constructed by extruding (with a slope) a set of contours shown in Figure M.1.3.16.

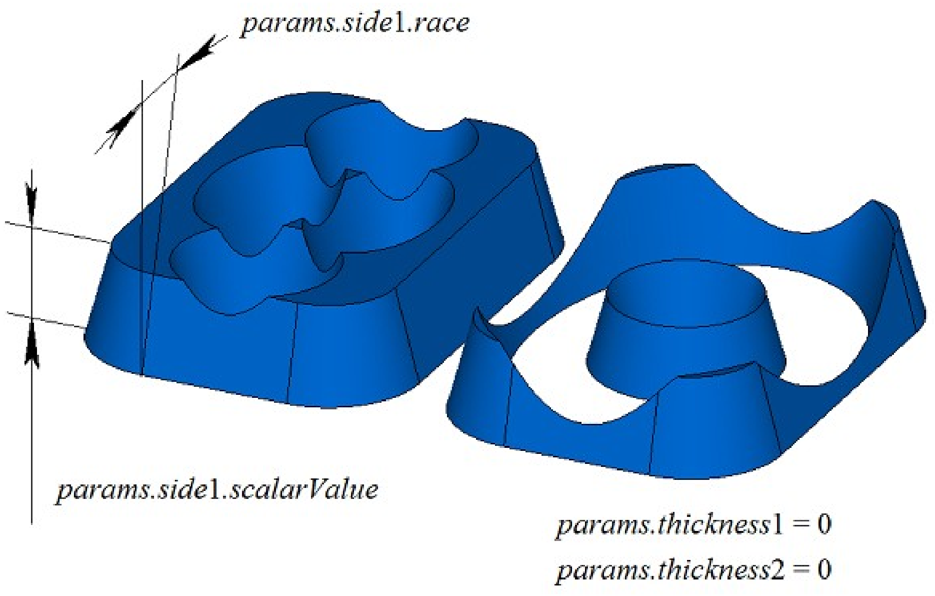


Figure M.1.3.18.

In Figure M.1.3.19 you can see a multi-part thin-walled closed solid that was constructed by extruding the set of contours shown in Figure M.1.3.16.

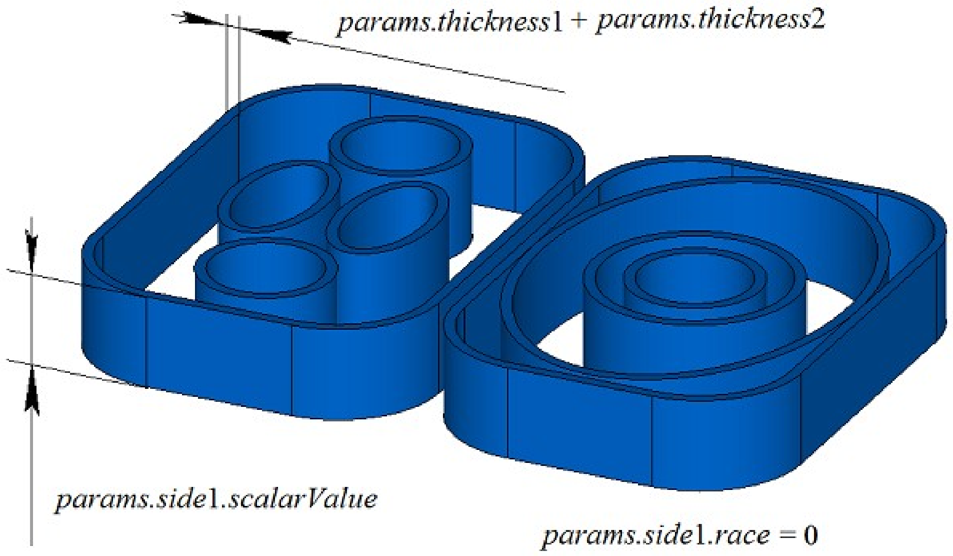


Figure M.1.3.19.

In Figure M.1.3.20, you can see two three-dimensional contours.

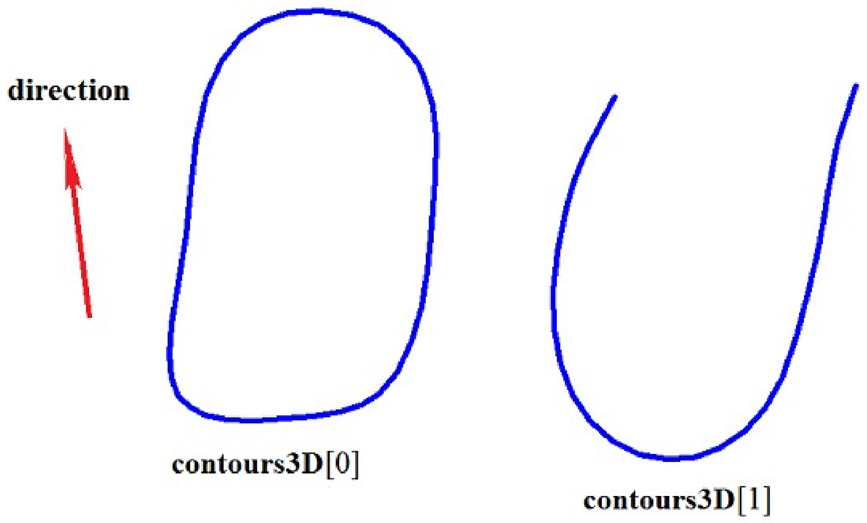


Figure M.1.3.20.

In Figure M.1.3.21, you can see a double-connected thin-walled closed solid that was constructed by extruding three-dimensional contours shown in Figure M.1.3.20.

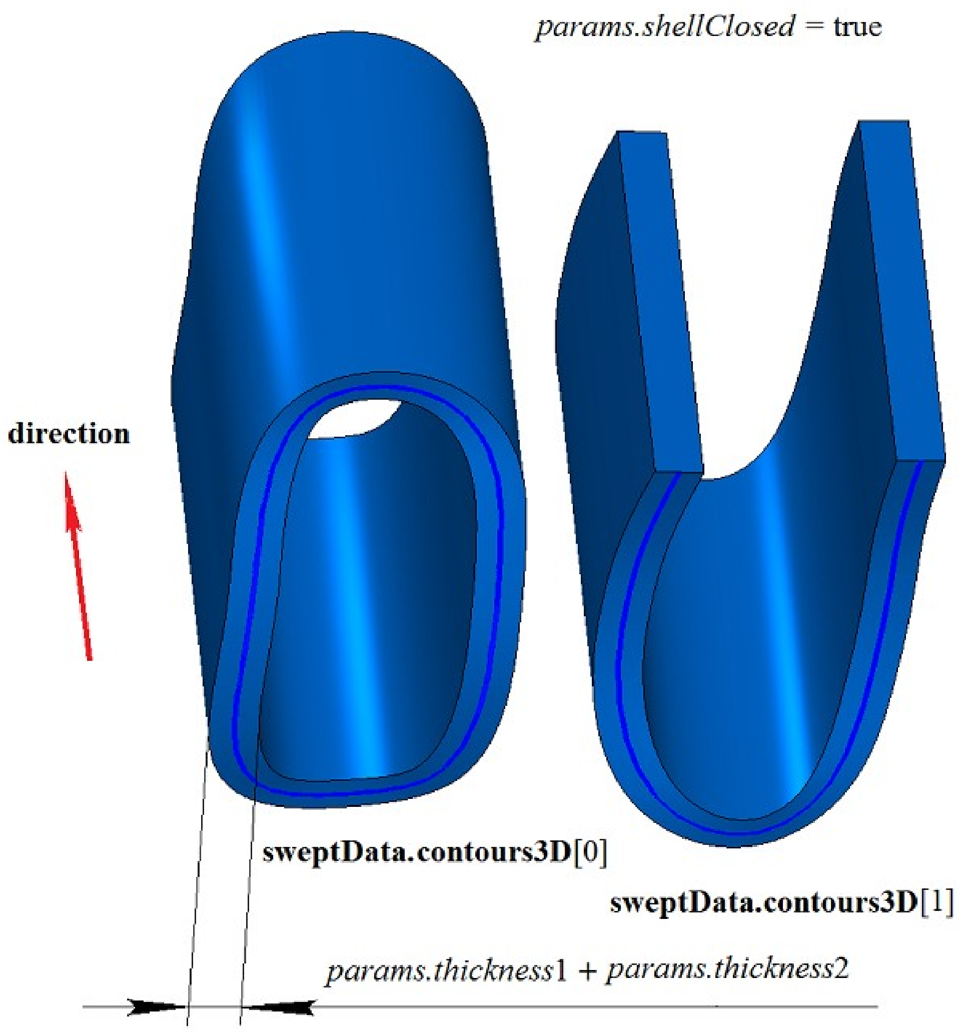

params.shellClosed = true
direction
sweptData.contours3D[0]
sweptData.contours3D[1]
*params.thickness*1 + *params.thickness*2

Figure M.1.3.21.

In Figure M.1.3.22, you can see two non-closed solids that were constructed by extruding three-dimensional contours shown in Figure M.1.3.20. The solids were constructed separately for each contour.

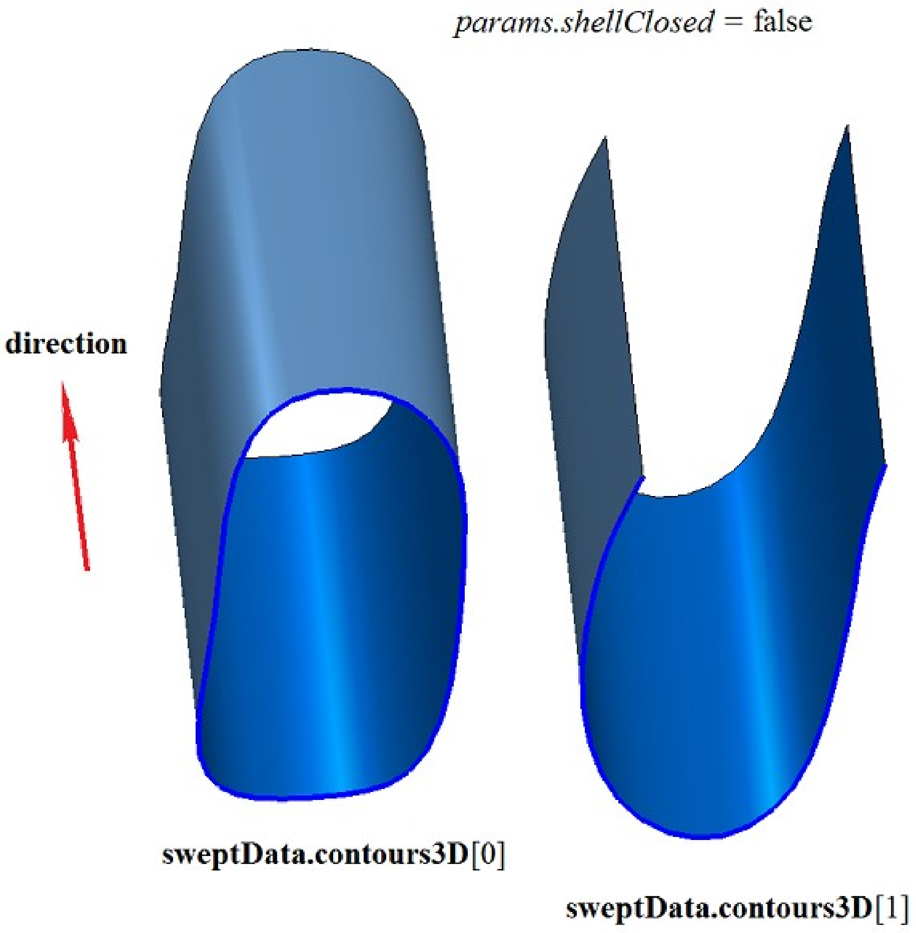

params.shellClosed = false
direction
sweptData.contours3D[0]
sweptData.contours3D[1]

Figure M.1.3.22.

ExtrusionSolid extrusion solid construction method adds MbExtrusionSolid constructor in the log of the newly constructed solid which contains all the necessary data to construct the solid. MbExtrusionSolid constructor is declared in cr_extrusion_solid.h file.

test.exe test application constructs an extruded solid using «Create->Solid->By curves->By extruding a surface curve» and «Create->Solid->By curves->By extruding a 3D curve» menu commands.

M.1.4. Constructing a Revolution Solid

Method
MbResultType
RevolutionSolid (const MbSweptData & **sweptData**,
const MbAxis3D & **axis**,
RevolutionValues & *params*,
const MbSNameMaker & names,
PArray<MbSNameMaker> & cnames,
MbSolid *& **result**)
constructs a revolution solid.

Method input parameters are:

- **sweptData** are data on curve generators,
- **axis** is rotation axis,
- params are construction parameters,
- names is face namer,

- cnames are namers of curve generator segments.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns an error code from MbResultType enumeration. This method is declared in action_solid.h file.

Rotation solid belongs to the type of motion solids which are constructed by moving a curve generator along a guiding curve. A circle or an arc can be a guidng curve for rotation solid. Rotation solid is constructed by rotating one or more curves around **axis**.

sweptData parameter contains information on generator curves. MbSweptData class and RevolutionValues structure are described in swept_parameter.h file. Generating curves may be two-dimensional **contours** on **surface** or contours in **contours3D** space. In particular cases, two-dimensional **contours** may be located on a plane. **contours** may have arbitrary orientation. **contours** may be nested with each other. **contours** shouldn't intersect with each other.

Curves can be rotated in forward direction about **axis**, in backward direction about **axis**, and in both directions. The rotation in forward direction is counterclockwise when looking toward the **axis**. Construction parameters for each direction are set by MbSweptSide objects.

params parameter contains information on rotation method in forward direction MbSweptSide *side*1 and information on rotation method in backward direction MbSweptSide *side*2. Rotation in each direction can be executed in two ways. If *way*==sw_scalarValue, then rotation is executed about *scalarValue* angle in direction *side*1 or *side*2, respectively. If *way*==sw_surface, then rotation is executed about *side*1.**surface** or *side*2.**surface** angle, respectively, if *side*1.*distance*=0 or *side*2.*distance*=0. If *way*==sw_surface and *distance*!=0, then rotation is executed about to equidistant surface to *side*1.**surface** or to equidistant surface to *side*2.**surface**, respectively. *params.thickness*1 and *params.thickness*2 parameters define the wall thickness of thin-walled solid. *params.thickness*1 parameter defines outward offset from the generator curve, and *params.thickness*2 parameter defines inward offset from the generator curve. *params.shellClosed* parameter controls whether the constructed solid is closed. *params.checkSelfInt* parameter defines the need to check the result of construction for self-intersection. By default, *params.checkSelfInt*=false and the check is not performed. *params.shape* parameter controls the shape of the constructed solid. If *params.shape*=1, then the constructed solid has torus topology. If *params.shape*=0, then the solid has sphere topology.

In Figure M.1.4.1, you can see the data used for construction, as well as parameters inheritance scheme for constructed revolvution solid (RevolutionValues & *params*).

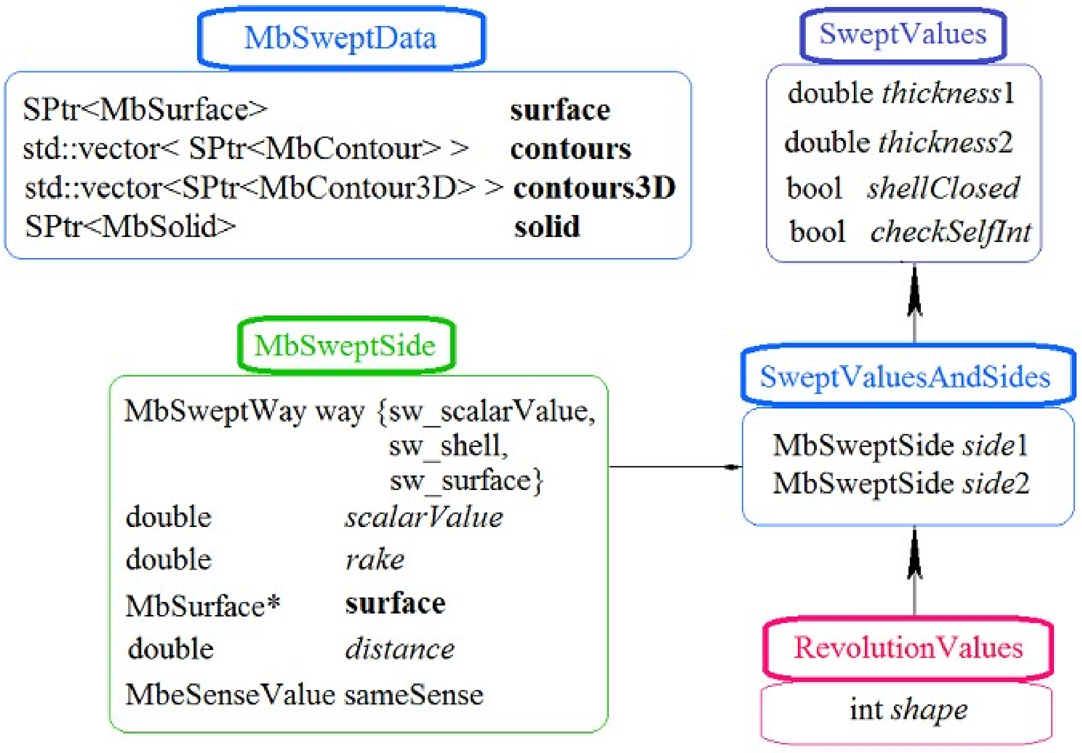


Figure M.1.4.1.

names and cnames parameters are responsible for naming the faces of the newly constructed solid.
In Figure M.1.4.2, you can see a two-dimensional **contour**, a flat **surface** (MbPlane) and a rotation **axis**.

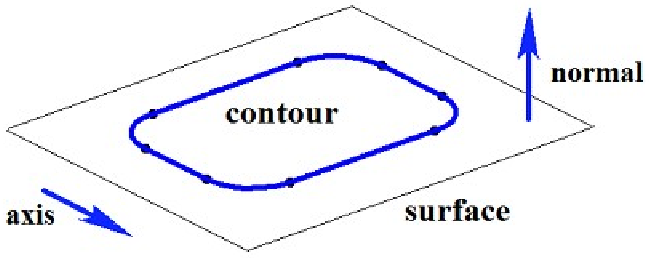


Figure M.1.4.2.

In Figure M.1.4.3, you can see a closed solid that was constructed by rotation using specified parameters of the contour shown in Figure M.1.4.2. Each contour segment has a corresponding face of the solid, its name was taken from the corresponding element of cnames[0] name generator.

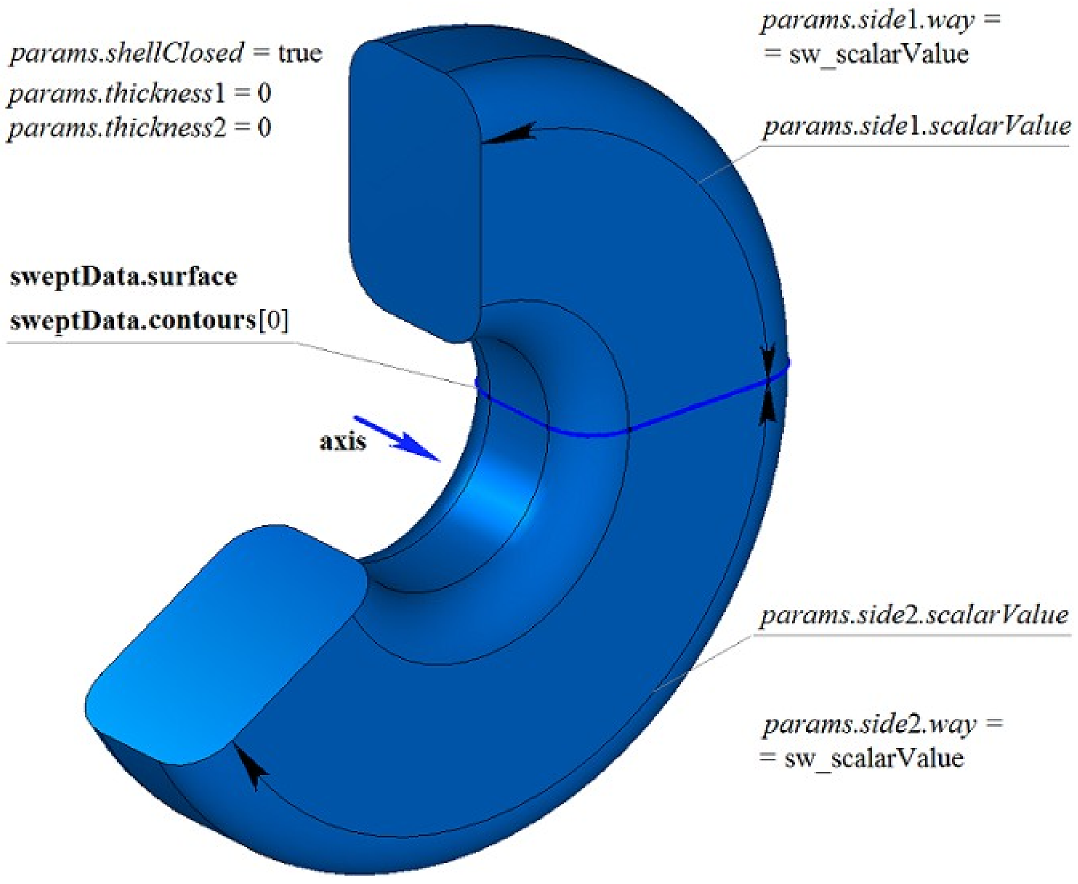


Figure M.1.4.3.

In Figure M.1.4.4, you can see a closed thin-walled solid that was constructed by using rotation for specified parameters of the contour shown in Figure M.1.4.2.

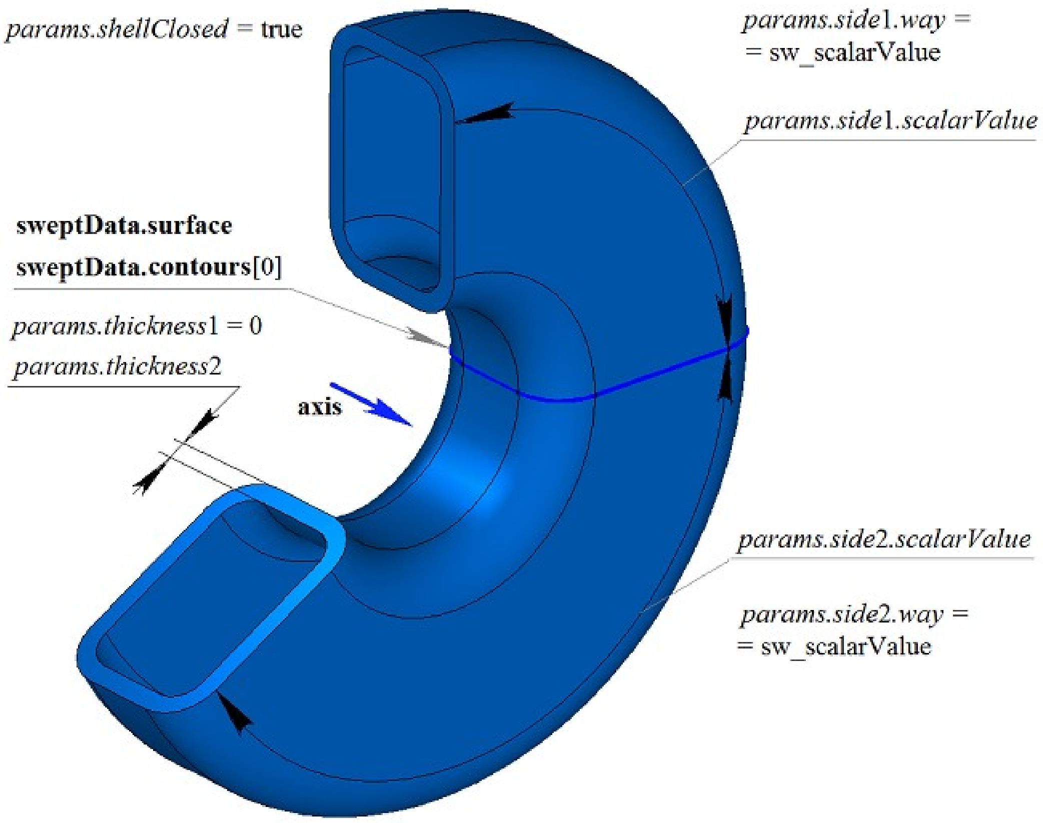


Figure M.1.4.4.

In Figure M.1.4.5, you can see a non-closed solid that was constructed by rotation using the specified parameters of the contour shown in Figure M.1.4.2. Parameters used to construct the solid shown in Figure M.1.4.3, are not the same as the parameters for constructing the solid shown in Figure M.1.4.5, but the only diffrence is *params.shellClosed* value.

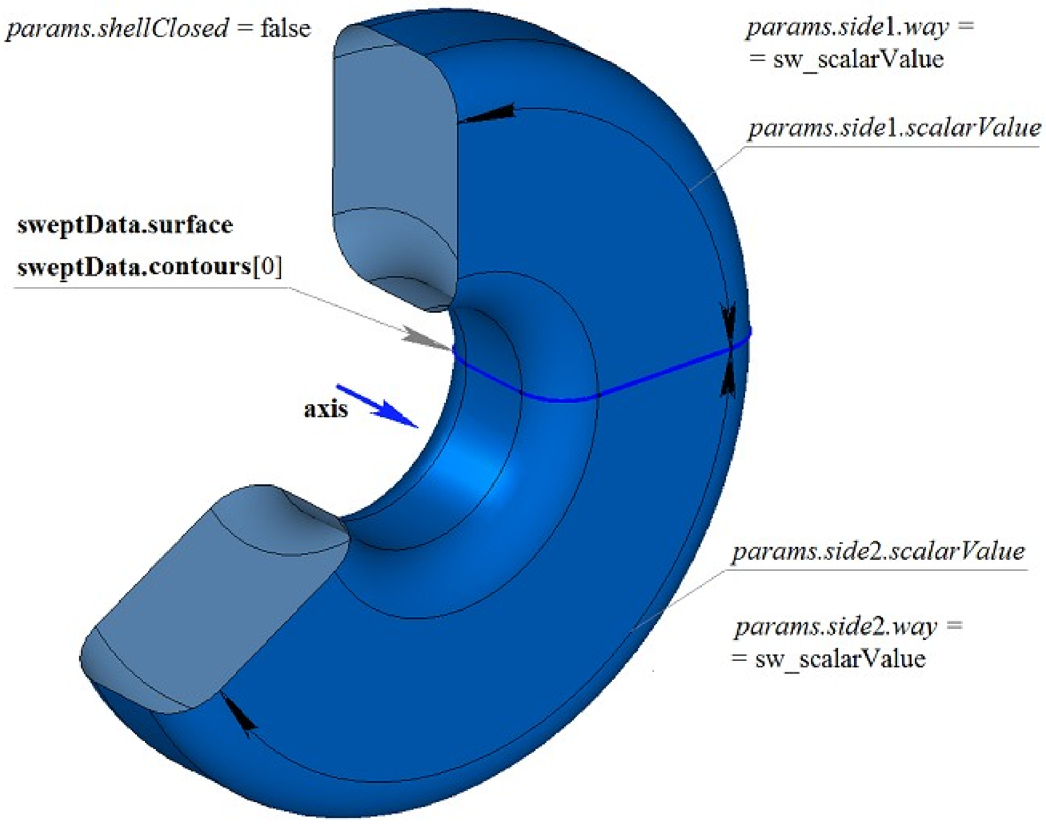

params.shellClosed = false
params.side1.way = = sw_scalarValue
params.side1.scalarValue
sweptData.surface
sweptData.contours[0]
axis
params.side2.scalarValue
params.side2.way = = sw_scalarValue

Figure M.1.4.5.

In Figure M.1.4.6, you can see a two-dimensional **contour**, a flat **surface** (MbPlane) and two surfaces (**surface**1 и **surface**2) that will be used to construct a revolution solid. For the construction **surface**1 and **surface**2 should completely cover the path of the contour moved in the appropriate direction. The following parameters should be taken into account: *params.thickness*1, *params.thickness*2. A revolution solid is cut off by specified or equidistant surfaces if *params.side*1*.distance* or *params.side*2*.distance* are not equal to zero.

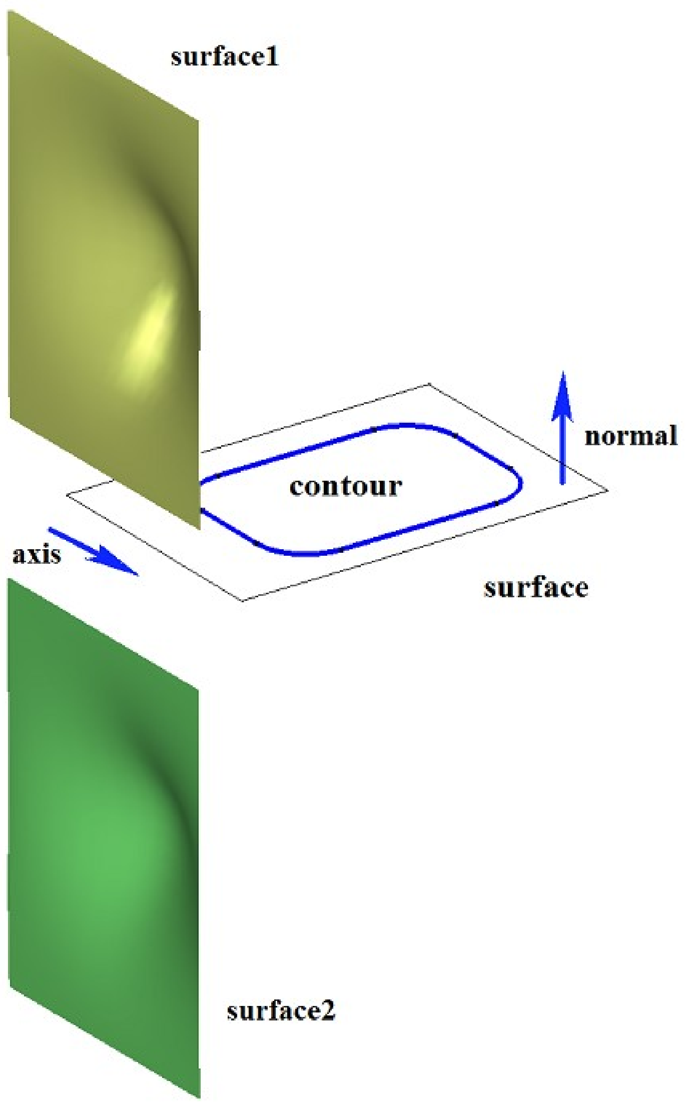


Figure M.1.4.6.

In Figure M.1.4.7, you can see a solid that was constructed by rotating the contour shown in Figure M.1.4.6 with selected options «To the surface» (**surface**1 and **surface**2).

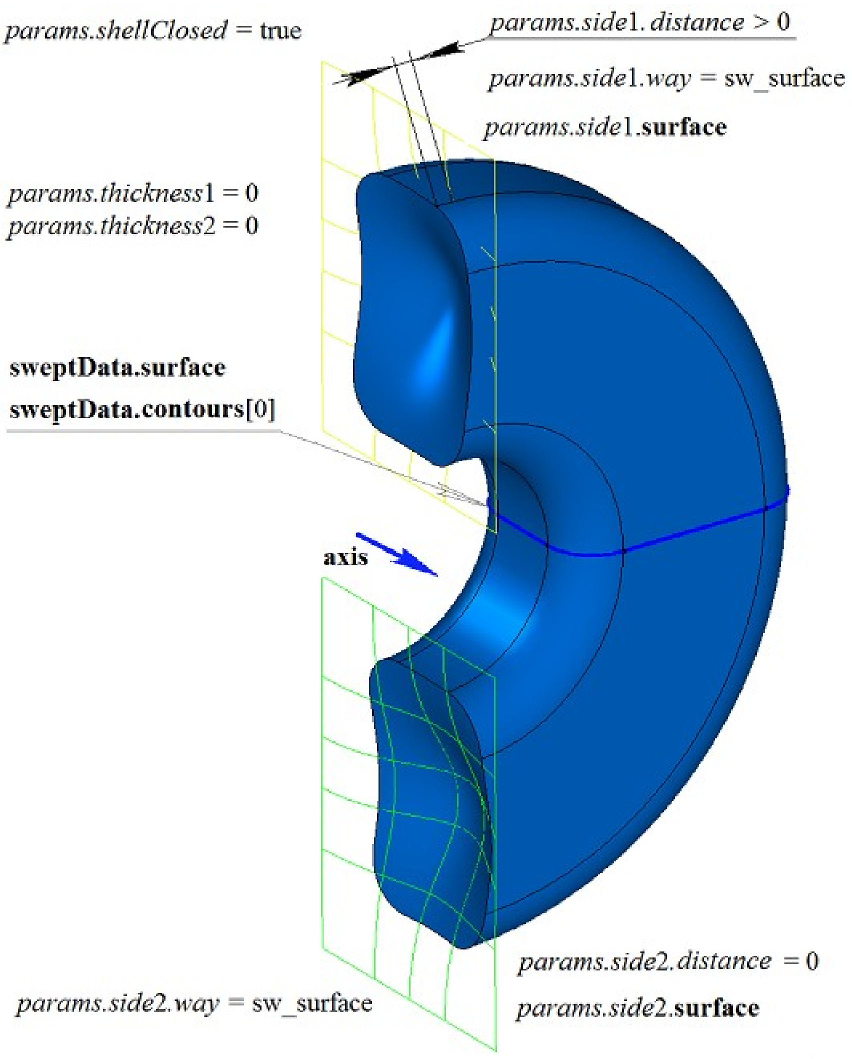

params.shellClosed = true
params.side1.distance > 0
params.side1.way = sw_surface
params.side1.surface
params.thickness1 = 0
params.thickness2 = 0
sweptData.surface
sweptData.contours[0]
axis
params.side2.distance = 0
params.side2.way = sw_surface
params.side2.surface

Figure M.1.4.7.

In Figure M.1.4.8, you can see a thin walled-solid that was constructed by rotating the contour shown in Figure M.1.4.6 with selected options «To the surface» (**surface**1 and **surface**2).

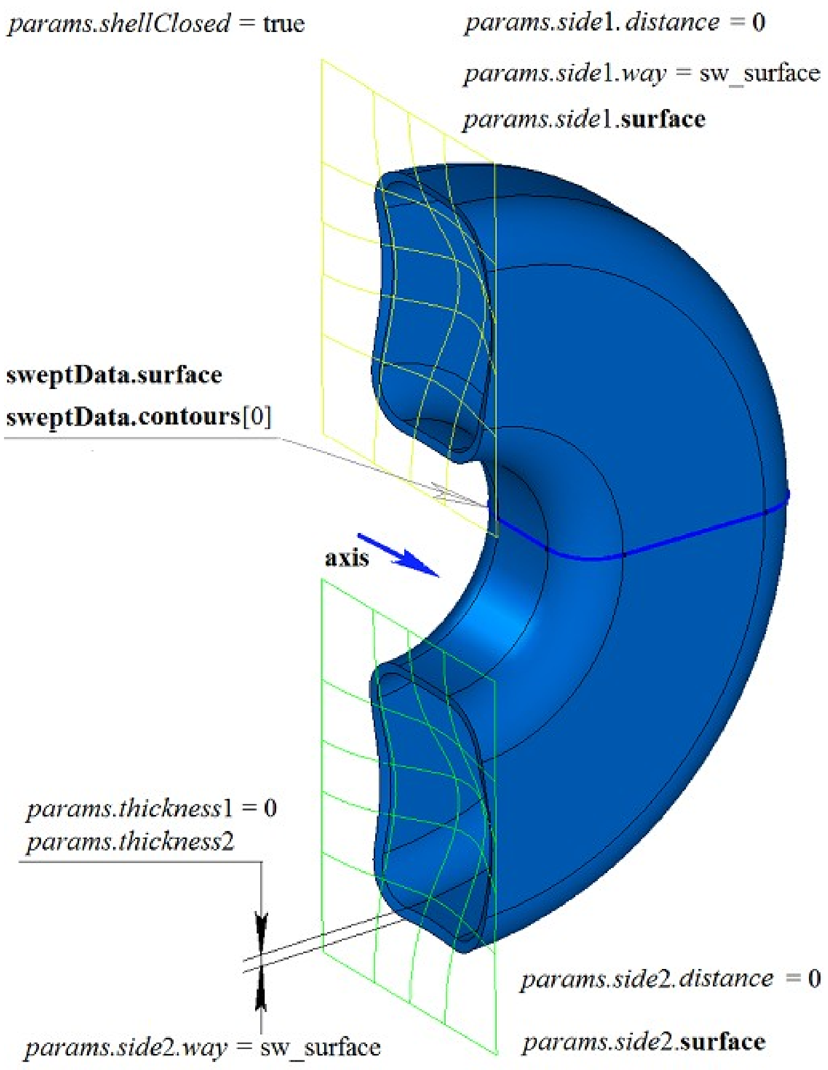

params.shellClosed = true
params.side1.distance = 0
params.side1.way = sw_surface
params.side1.surface
sweptData.surface
sweptData.contours[0]
axis
params.thickness1 = 0
params.thickness2
params.side2.distance = 0
params.side2.way = sw_surface
params.side2.surface

Figure M.1.4.8.

A two-dimensional contour may be drawn on a flat surface or on a curved surface. For example, you can construct a solid by rotating a contour on a curved surface that was created using a cycle for one of the faces of the revolution solid shown in Figure M.1.4.9.

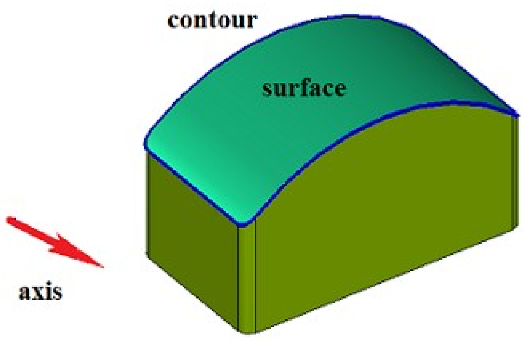


Figure M.1.4.9

In Figure M.1.4.10, you can see a solid that was constructed by rotating the contour on the curved surface shown in Figure M.1.4.9.

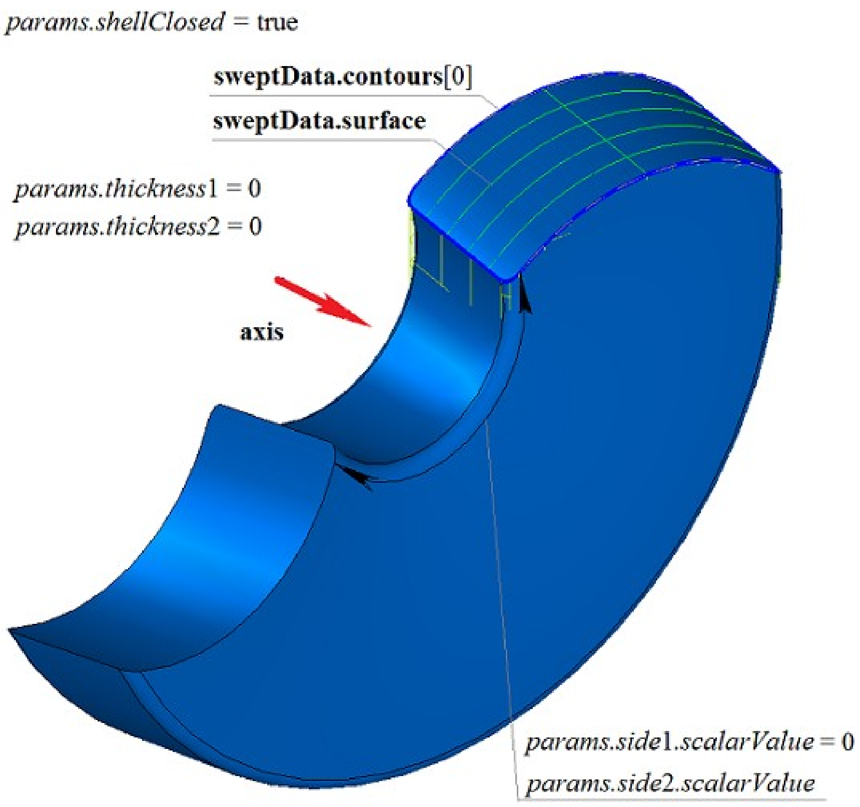


Figure M.1.4.10.

In Figure M.1.4.11, you can see a thin-walled solid that was constructed by rotating the contour on the curved surface shown in Figure M.1.4.9.

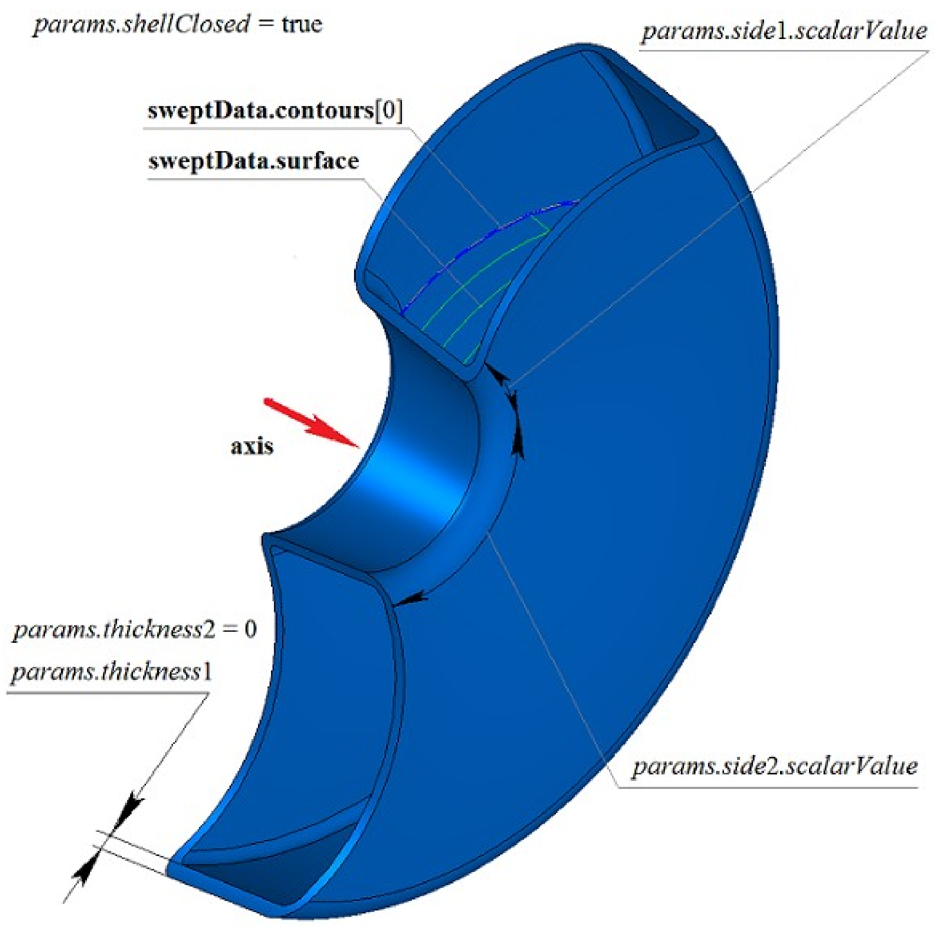


Figure M.1.4.11.

In Figure M.1.4.12, you can see a non-closed solid that was constructed by rotating the contour on the curved surface shown in Figure M.1.4.9.

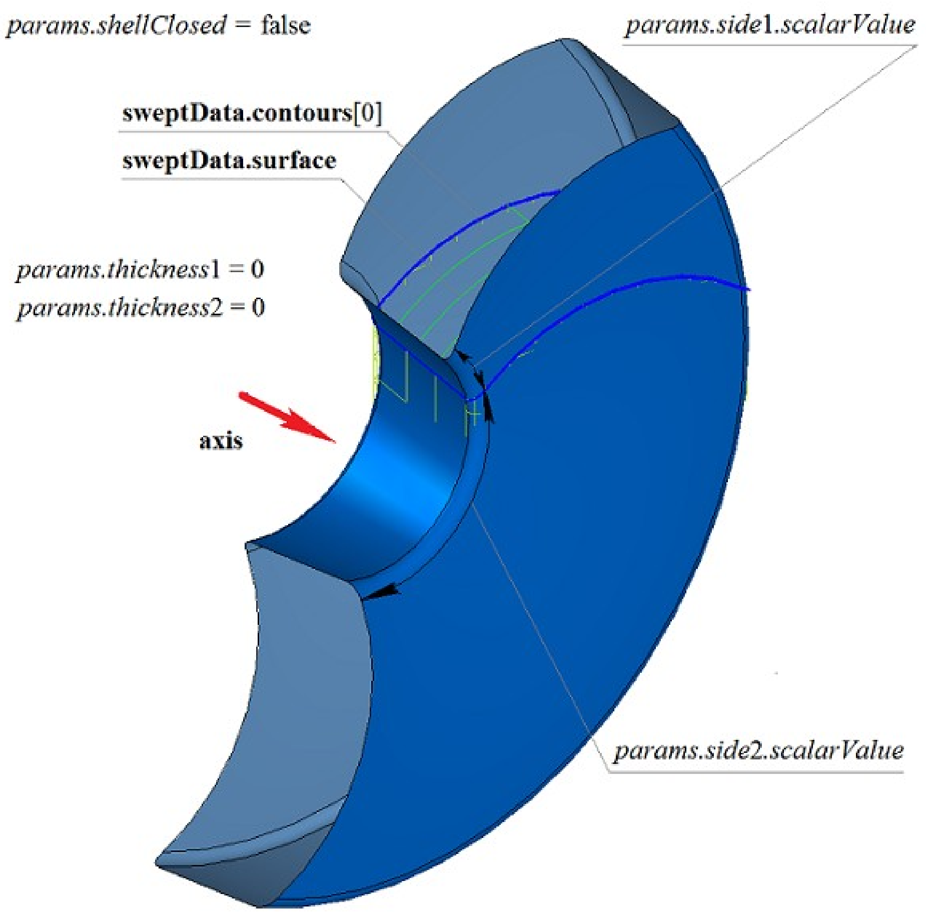


Figure M.1.4.12.

If one surface contains a set of non-intersecting two-dimensional contours, then the considered method defines external and nested internal contours (multilevel nesting can be used). In Figure M.1.4.13, you can see a set of non-intersecting two-dimensional **contours** and a flat **surface** (MbPlane).

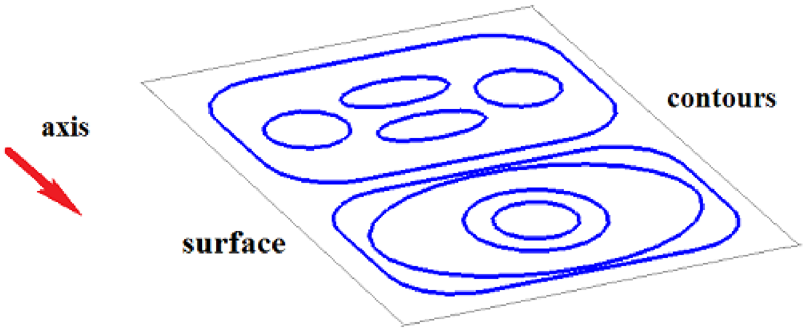


Figure M.1.4.13.

In Figure M.1.4.14, you can see a multi-part closed solid that was constructed by rotating the set of contours shown in Figure M.1.4.13.

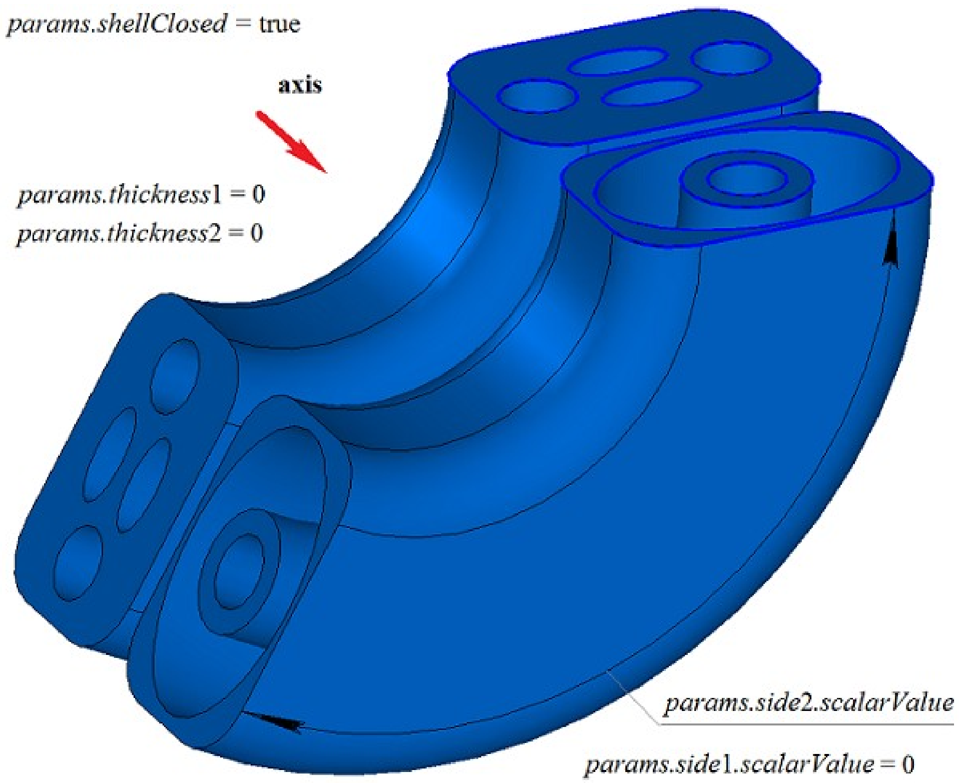


Figure M.1.4.14.

In Figure M.1.4.15, you can see a multi-part thin-walled closed solid that was constructed by rotating the set of contours shown in Figure M.1.4.13.

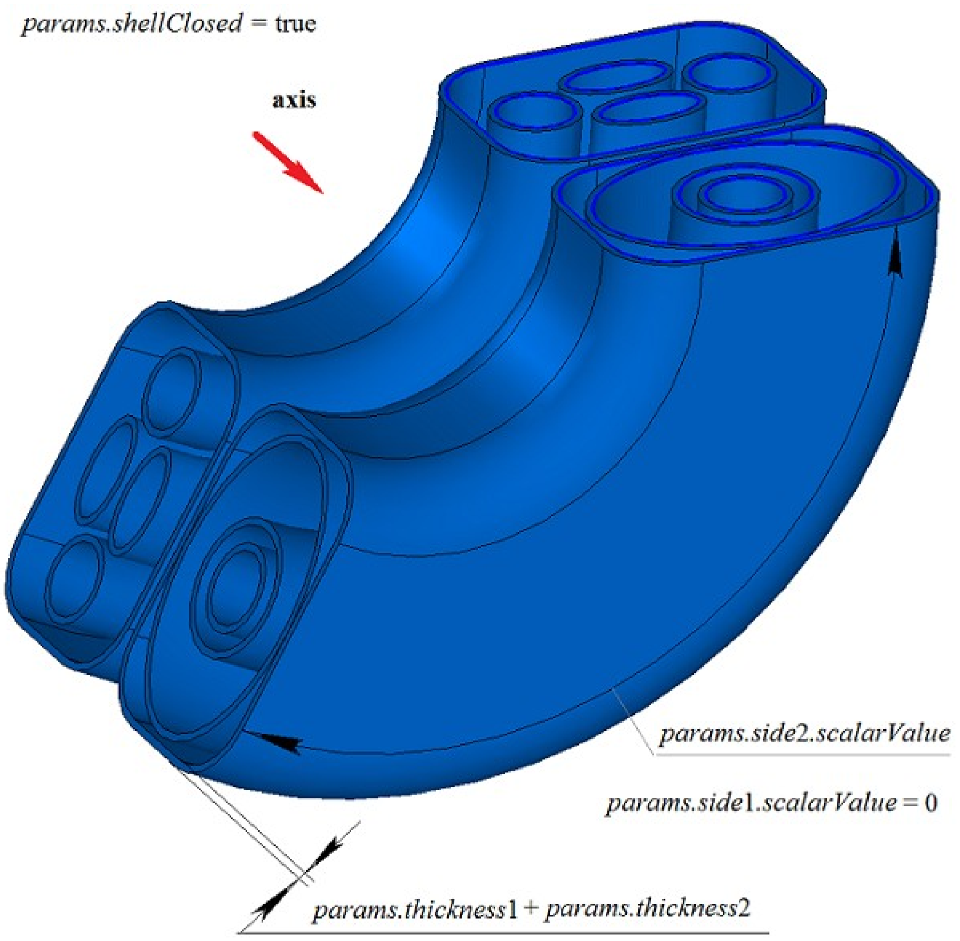

params.shellClosed = true
axis
params.side2.scalarValue
params.side1.scalarValue = 0
params.thickness1 + params.thickness2

Figure M.1.4.15.

In Figure M.1.4.16, you can see two three-dimensional contours.

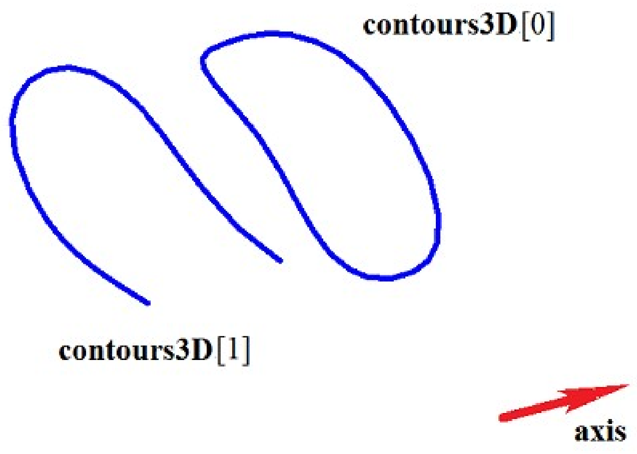


Figure M.1.4.16.

In Figure M.1.4.17, you can see a doubly-connected thin-walled closed solid that was constructed by rotating the three-dimensional contours shown in Figure M.1.4.16.

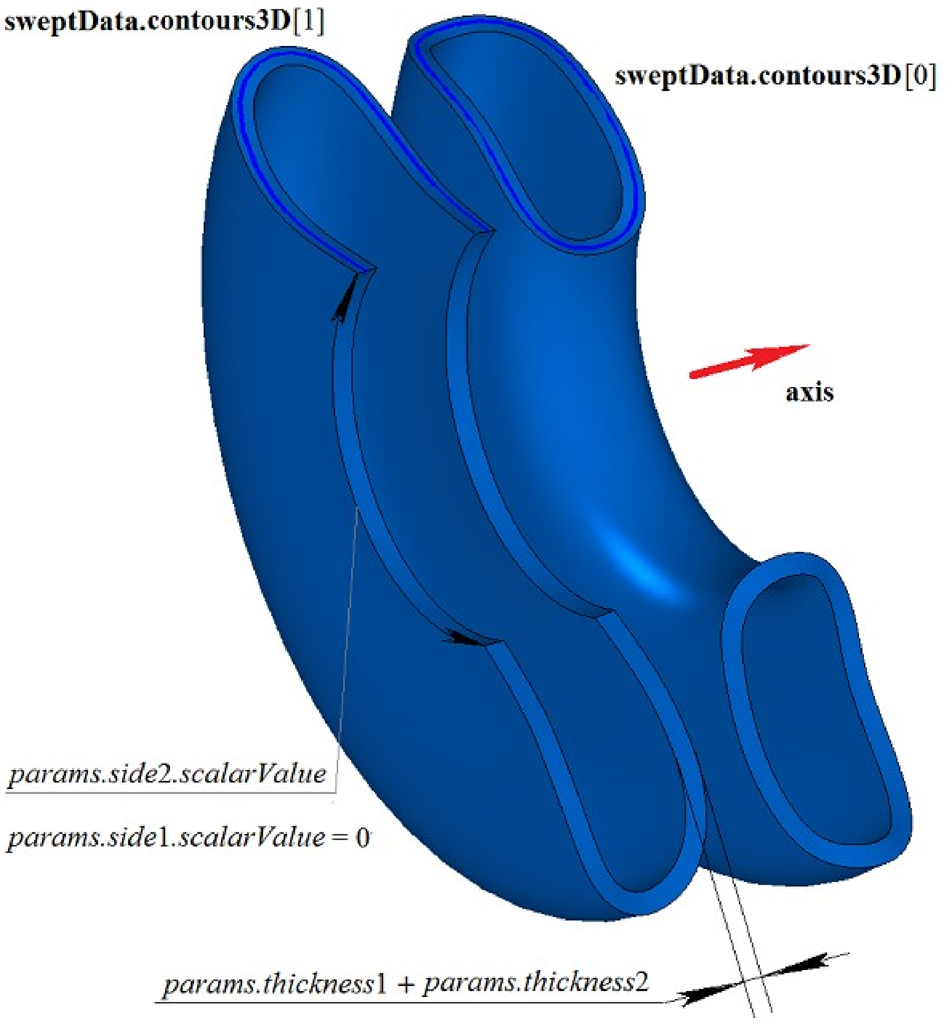


Figure M.1.4.17.

In Figure M.1.4.18, you can see two non-closed solids that were constructed by rotating three-dimensional contours shown in Figure M.1.4.16.

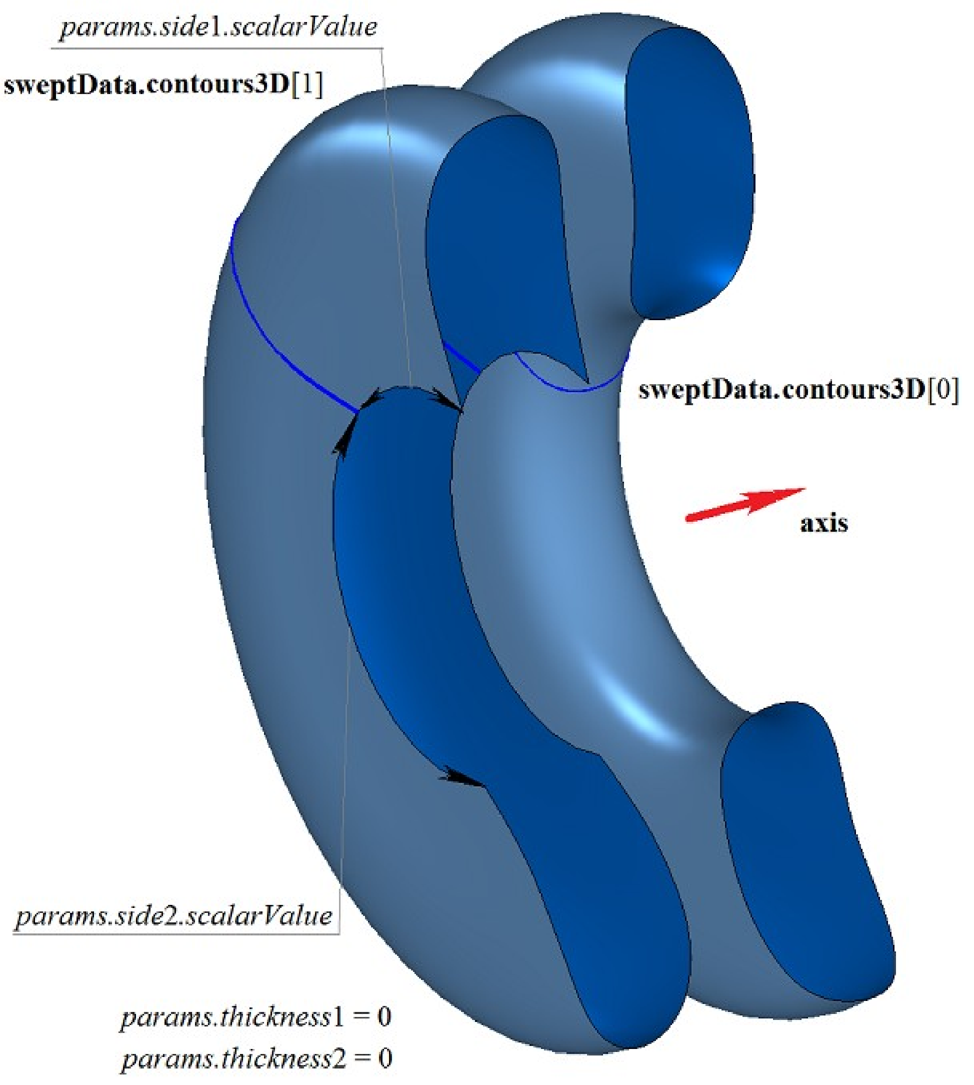


Figure M.1.4.18.

RevolutionSolid method that is used to construct a revolution solid adds MbRevolutionSolid constructor to the log of the newly constructed solid. This constructor contains all data required to construct the solid. MbRevolutionSolid constructor is declared in cr_revolution_solid.h file.

test.exe test application constructs a revolution solid using «Create->Solid->By curves->By rotating a surface curve» and «Create->Solid->By curves->By rotating a 3D curve» menu commands.

M.1.5. Constructing a Swept Solid

Method

MbResultType
EvolutionSolid (const MbSweptData & **sweptData**,
const MbCurve3D & **spine**,
EvolutionValues & *params*,
const MbSNameMaker & names,
const MbSNameMaker & cnames,
const MbSNameMaker & snames,
MbSolid *& **result**)

constructs a swept solid by moving a curve generator along a guiding curve.
Method input parameters are:

- **place** is generating contour local coordinate system,
- **contour** is generating contour,
- **spine** is guiding curve,
- *params* are construction parameters,
- names is face namer,
- cnames is generator namer,
- snames is guiding line namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns an error code from MbResultType enumeration. This method is declared in action_solid.h file.

A swept solid is a general case of movement solids, which are constructed by moving a generator curve along the guiding curve. Arbitrary curve can be used as a guiding curve for a swept solid.

sweptData parameter contains information on generator curves. MbSweptData class and EvolutionValues structure are described in swept_parameter.h file. Generating curves may be two-dimensional **contours** on **surface** or contours in **contours3D** space. In particular cases, two-dimensional **contours** may be located on a plane. **contours** may have arbitrary orientation. **contours** may be nested with each other. **contours** shouldn't intersect with each other.

Generator curves are moved along **spine** guiding curve. *params* parameter contains information about movement mode, presence of solid walls and their thickness and data whether the constructed solid is closed. *params.thickness*1 and *params.thickness*2 parameters define wall thickness of the constructed thin-walled solid. *params.thickness*1 parameter defines outward offset from the generator curve, and *params.thickness*2 parameter defines inward offset from the generator curve. *params.shellClosed* parameter controls whether the constructed solid is closed. *params.checkSelfInt* parameter defines the need to check the result of construction for self-intersection. By default, *params.checkSelfInt*=false, the check is not performed and the method permits you to construct self-intersecting solids. The movement can be performed in three ways. Movement mode is defined by *params.parallel* parameter. If *params.parallel*=0, then the movement of generator curves is coplanar. If *params.parallel*=1, then moving generator curves maintain their position in the local coordinate system, which is tangent to the generator curve. If *params.parallel*=2, then before movement, generator curves are transferred to a plane perpendicular to the starting end of the guiding curve, and subsequently they maintain their position in the local coordinate system, which is tangent to the generation curve.

In Figure M.1.5.1, you can see the data used for construction, as well as parameters inheritance scheme for constructed swept solid ExtrusionValues & *params*.

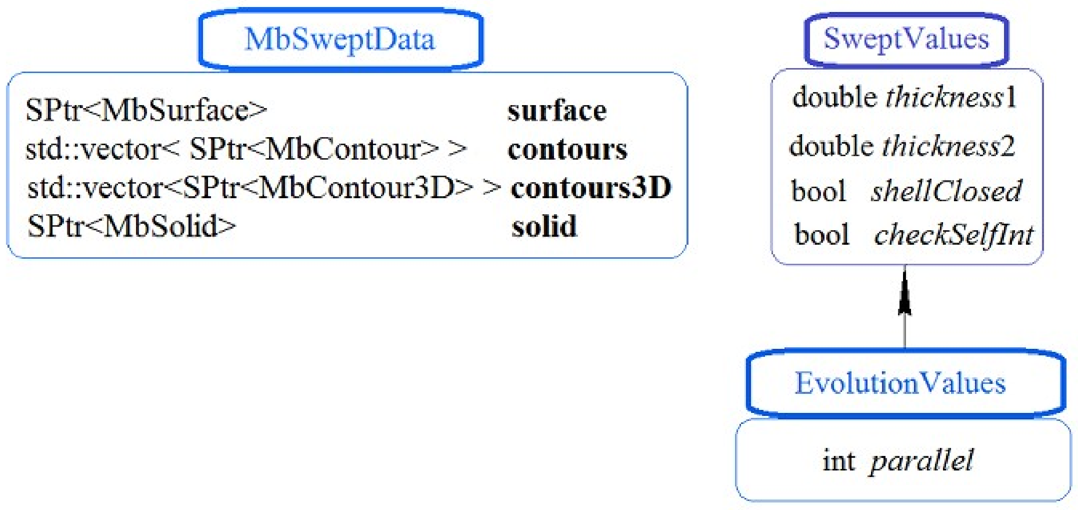


Figure M.1.5.1.

names, cnames и snames parameters are responsible for naming the faces of the newly constructed solid.

In Figure M.1.5.2, you can see a two-dimensional **contour**, flat **surface** (MbPlane) and **spine** guiding curve.

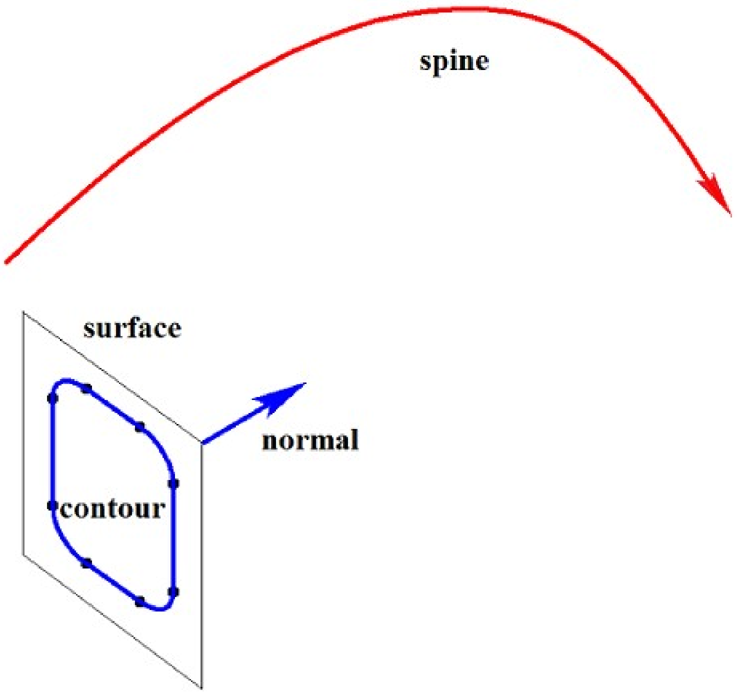


Figure M.1.5.2.

In Figure M.1.5.3, you can see a swept solid that was constructed by moving the contour along the guiding curve shown in Figure M.1.5.2. The method of moving is determined by *params.parallel*=0 parameter, in this case the planes of solid ends remain parallel.

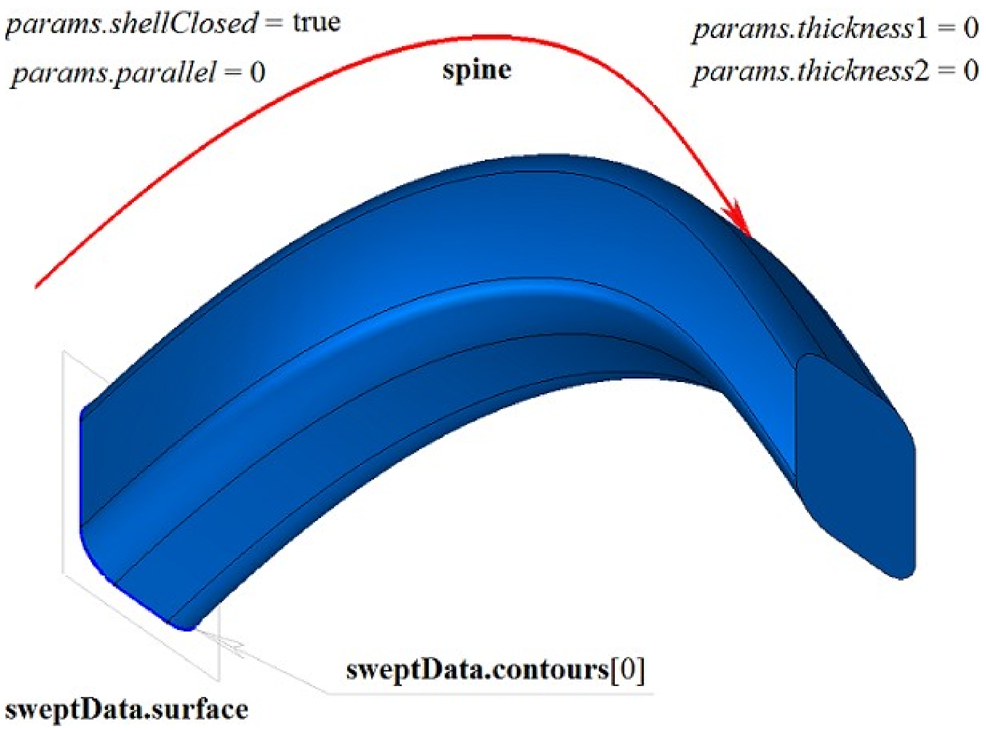


Figure M.1.5.3.

In Figure M.1.5.4, you can see a swept solid that was constructed by moving a contour along the guiding curve shown in Figure M.1.5.2, using the method defined by *params.parallel*=1 parameter. In this case, the plane of the solid end edge keeps its position relative to the end of the guiding curve according to the position of the start edge of the plane relative to the end of the guiding curve.

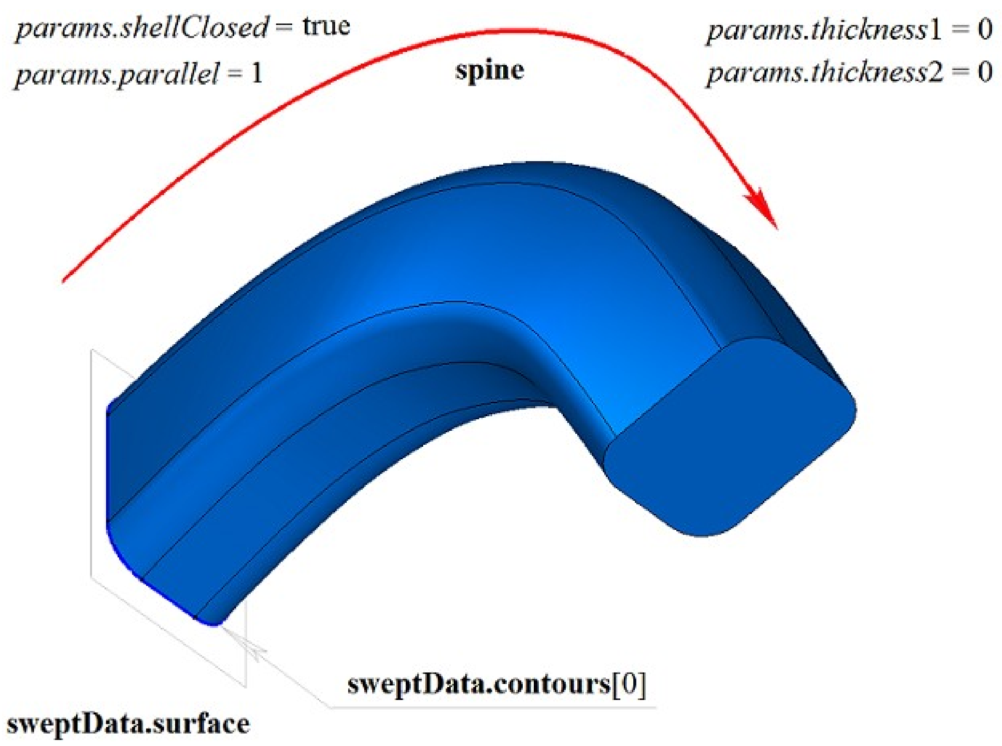


Figure M.1.5.4.

In Figure M.1.5.5, you can see a swept solid that was constructed by moving the contour along the guiding curve shown in Figure M.1.5.2. The method of moving is determined by *params.parallel*=2 parameter, in this case the planes of solid ends remain perpendicular to the guiding curve at its beginning and end.

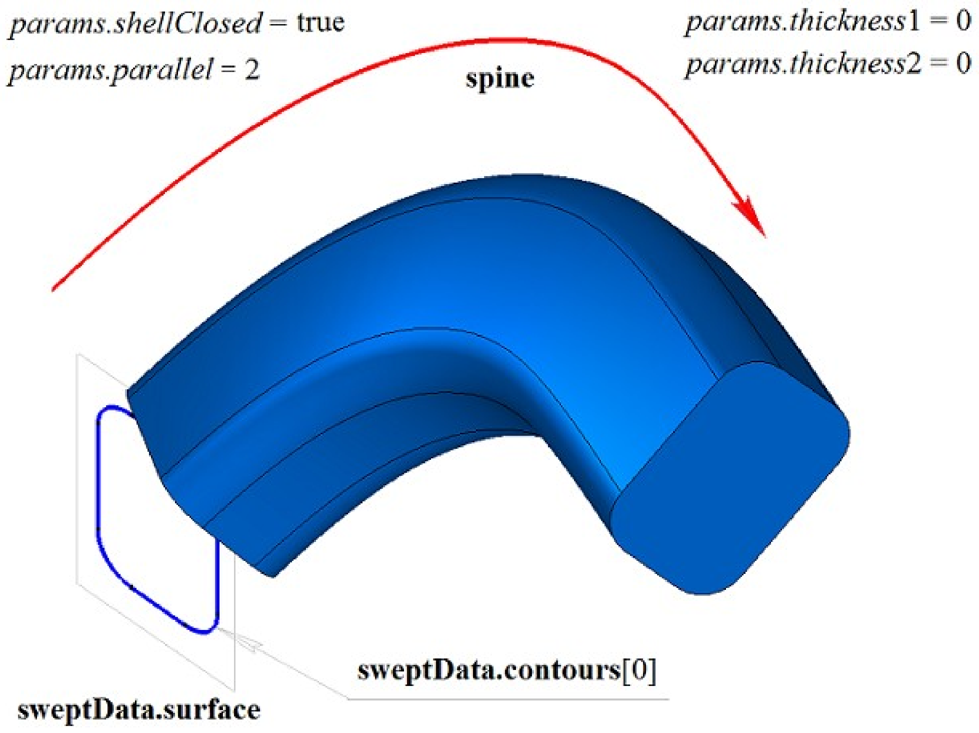


Figure M.1.5.5.

In Figure M.1.5.6, you can see a closed thin-walled swept solid that was constructed by moving the contour along the guiding curve shown in Figure M.1.5.2. The method of moving is determined by *params.parallel*=1 parameter. Each contour segment has a corresponding solid face, its name is taken from the corresponding element of cnames[0] name generator.

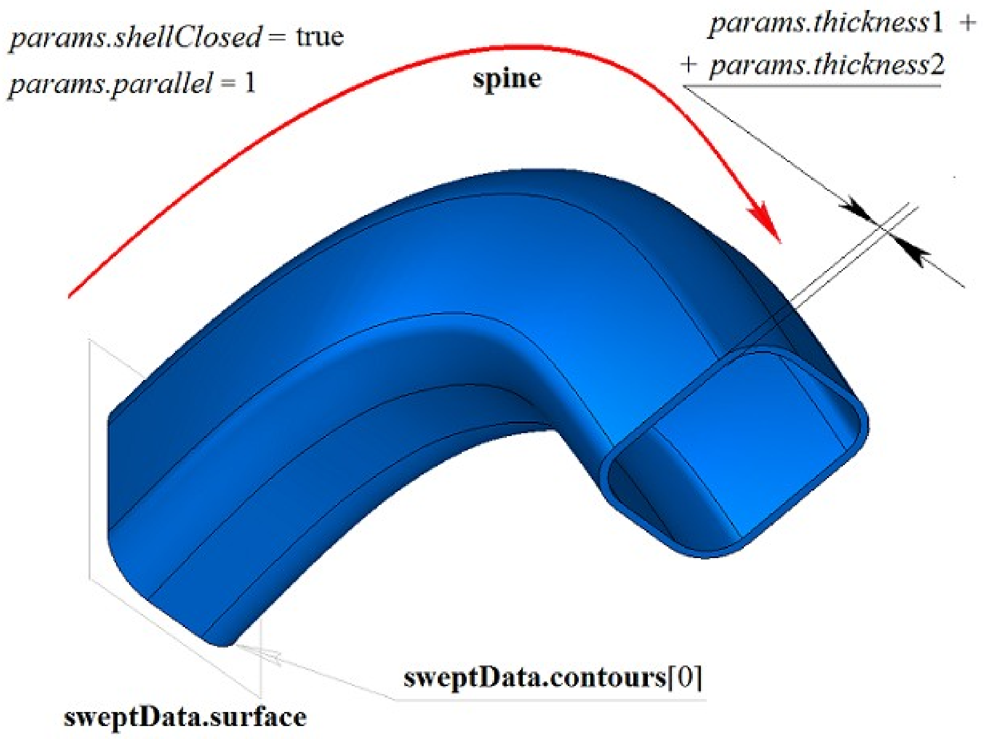


Figure M.1.5.6.

In Figure M.1.5.7, you can see a non-closed swept solid that was constructed by moving the contour along the guiding curve shown in Figure M.1.5.2. The method of moving is determined by *params.parallel*=1 parameter. Parameters used to construct the solid shown in Figure M.1.5.4 are not the same as the parameters for constructing the solid shown in Figure M.1.5.7, the only difference is the value of *params.shellClosed*=false.

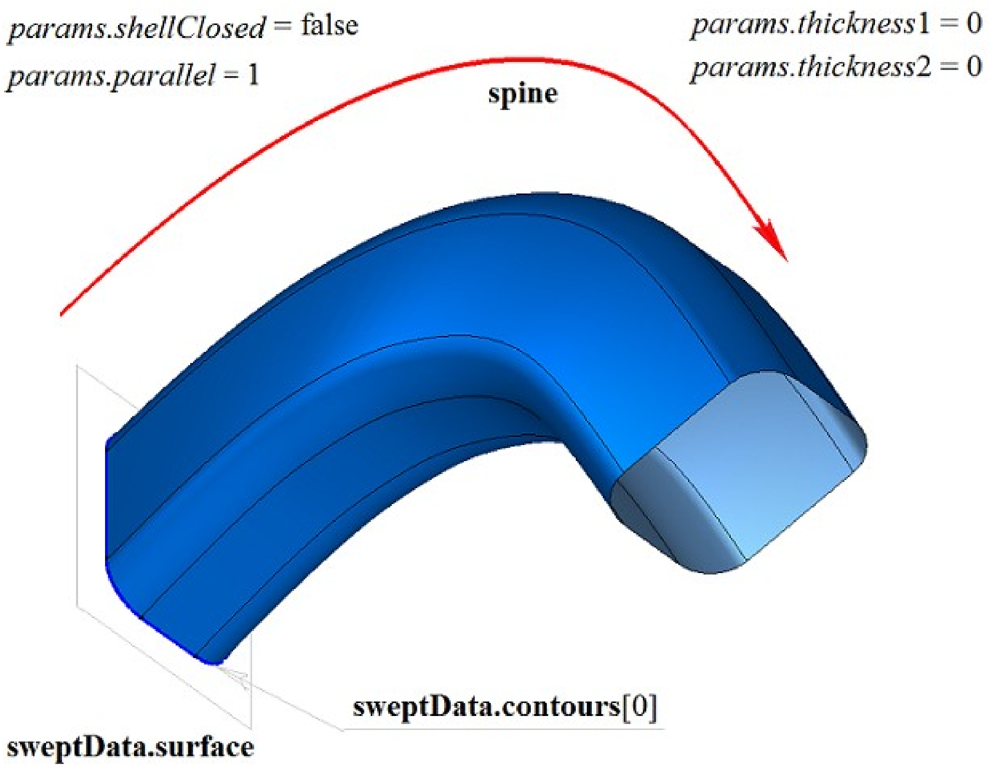


Figure M.1.5.7.

A two-dimensional contour may be drawn on a flat surface or on a curved surface. For example, a solid can be constructed by moving contours on a curved surface. The contours are to be created using the cycles of one solid solid face as shown in Figure M.1.5.8.

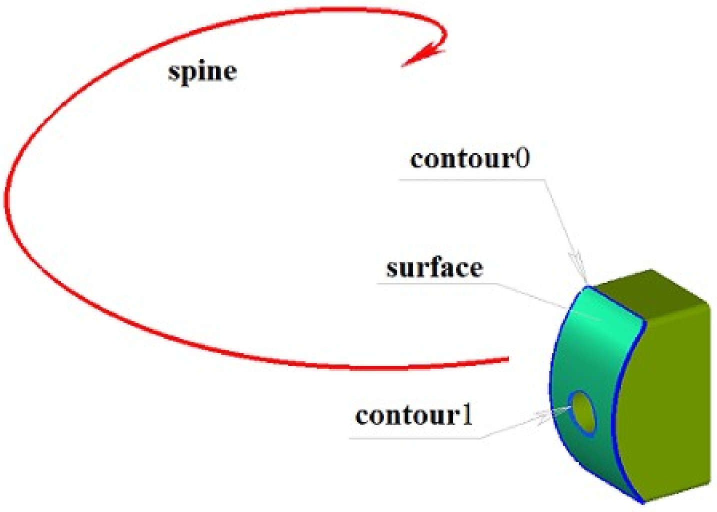


Figure M.1.5.8.

In Figure M.1.5.9, you can see a swept solid that was constructed by moving two contours on a curved surface along the guiding curves shown in Figure M.1.5.8. The method of moving the contours is determined by *params.parallel*=1.

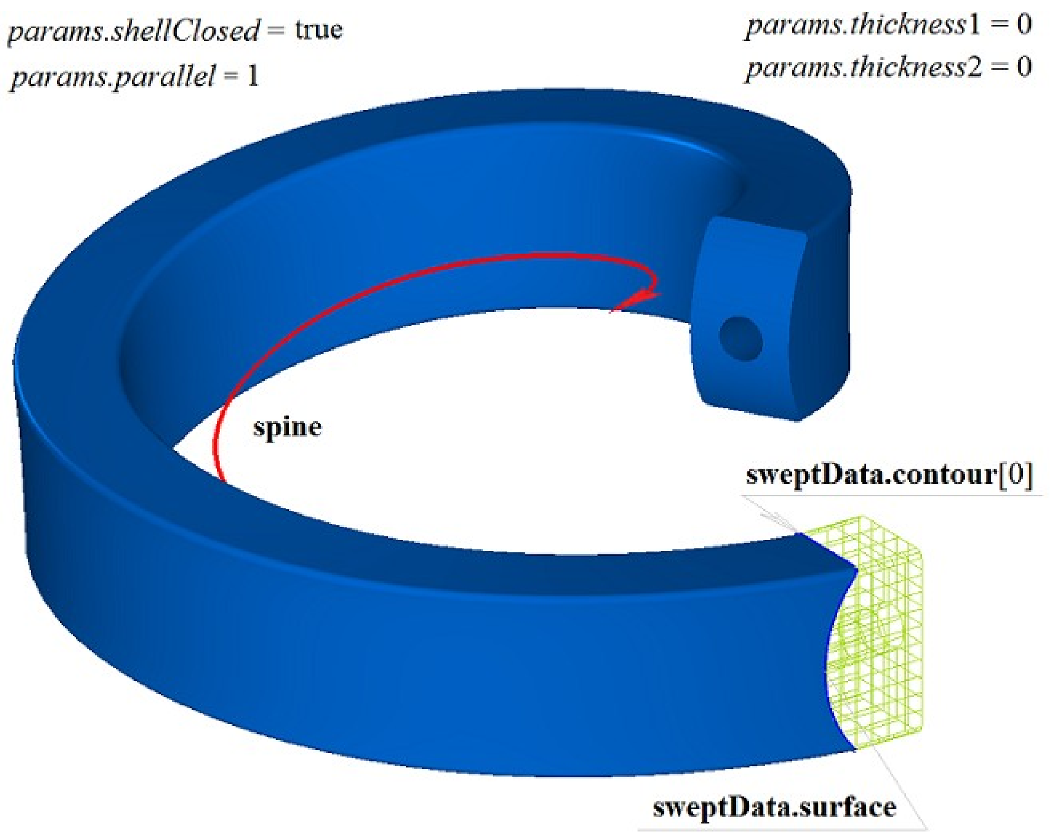


Figure M.1.5.9.

In Figure M.1.5.10, you can see a doubly-connected thin-walled swept solid that was constructed by moving two contours on a curved surface along the guiding curves shown in Figure M.1.5.9.

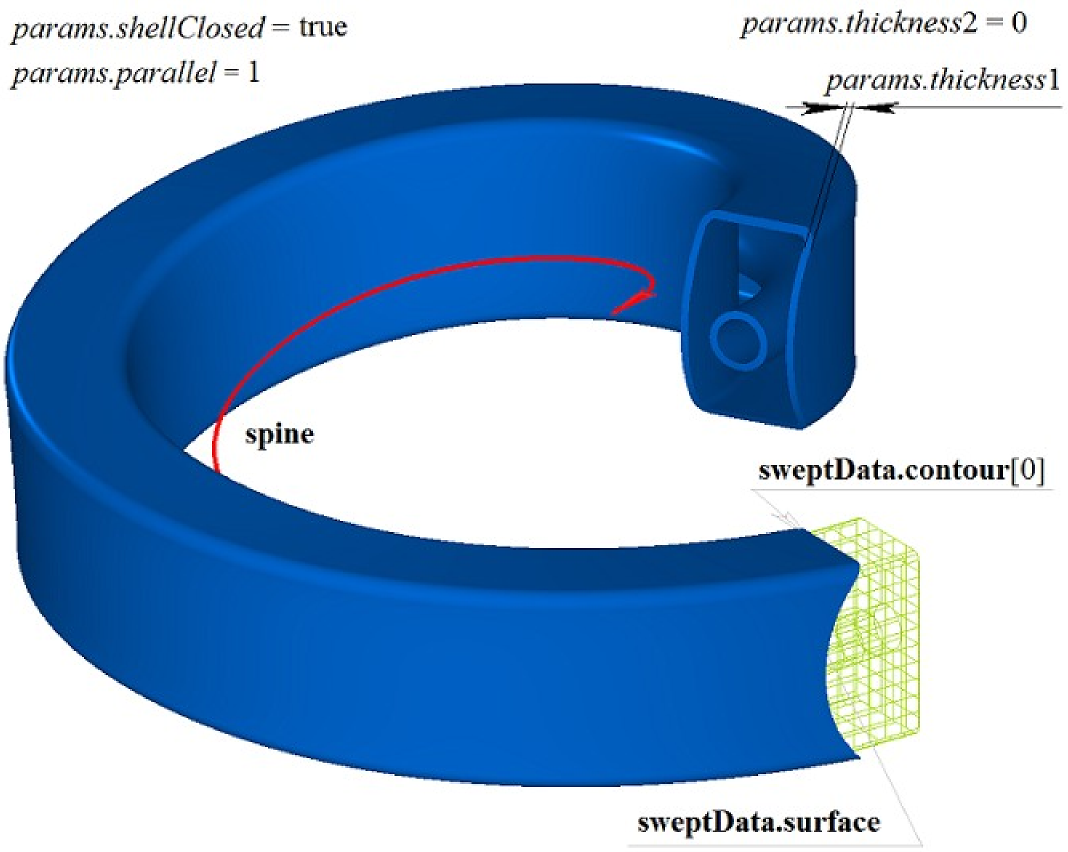


Figure M.1.5.10.

In Figure M.1.5.11, you can see a doubly-connected non-closed swept solid that was constructed by moving two contours on a curved surface along the guiding curve shown in Figure M.1.5.9.

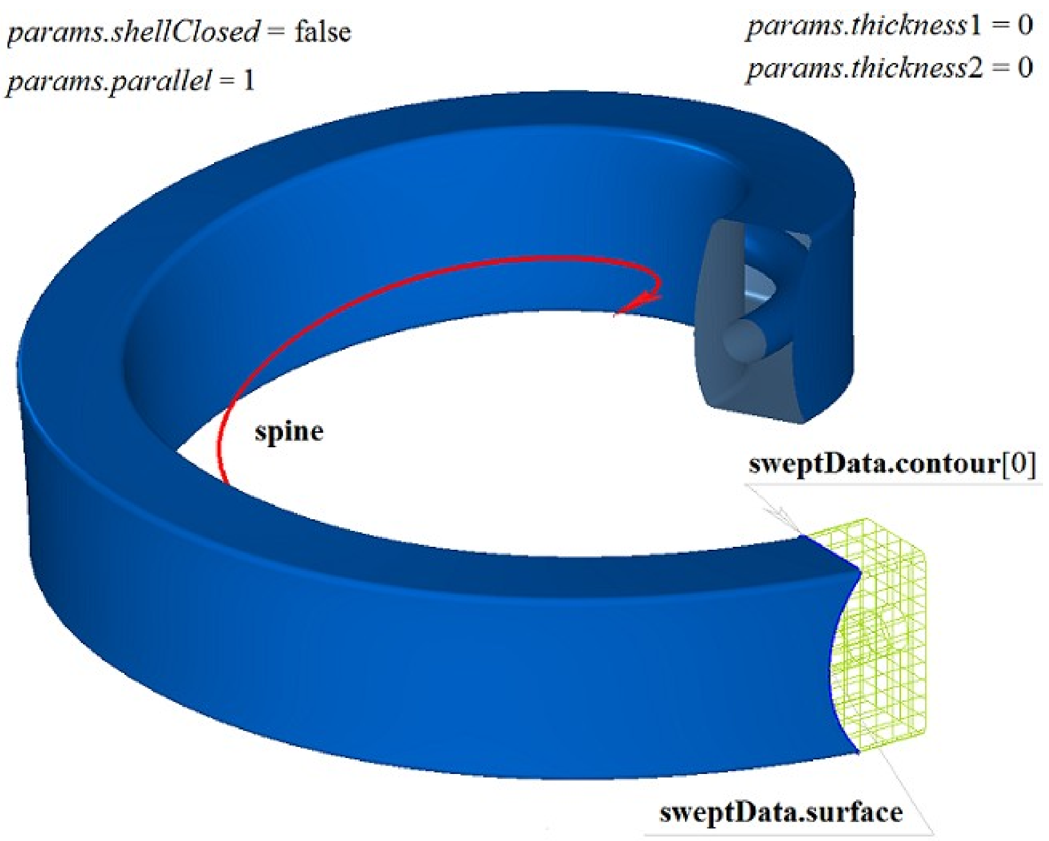


Figure M.1.5.11.

If one surface contains a set of non-intersecting two-dimensional contours then the considered method defines external and nested internal contours (multilevel nesting can be used). In Figure M.1.4.12, you can see a set of non-intersecting two-dimensional **contours**, a flat **surface** (MbPlane) and **spine** guiding curve.

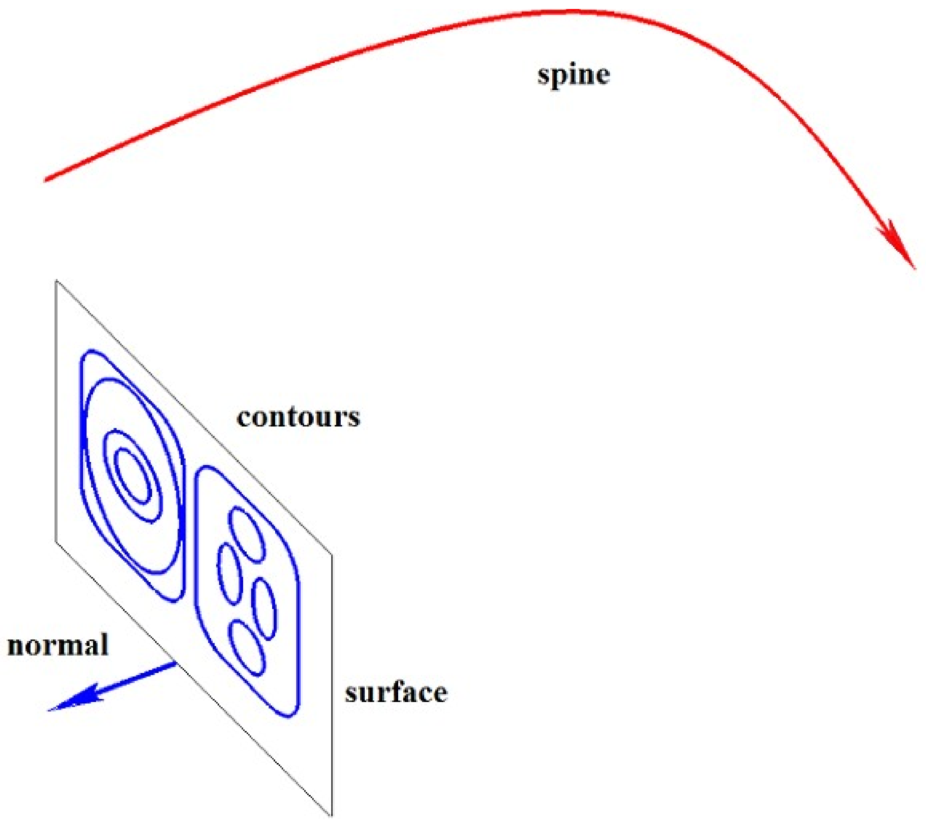


Figure M.1.5.12.

In Figure M.1.5.13, you can see a multi-part multiply-connected swept solid that was constructed by moving a set of flat contours along the guiding curve shown in Figure M.1.5.12. The contours should not intersect, but they can be nested several times.

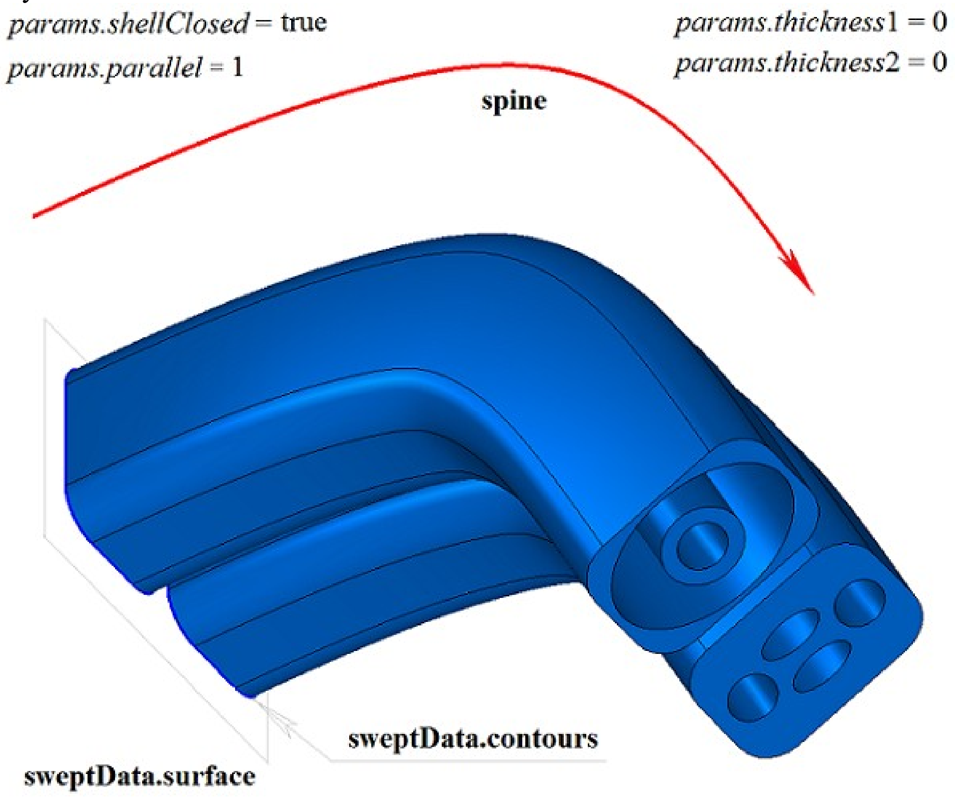


Figure M.1.5.13.

In Figure M.1.5.14, you can see a multi-part multiply-connected thin-walled swept solid that was constructed by moving a set of flat contours along the guiding curve shown in Figure M.1.5.12.

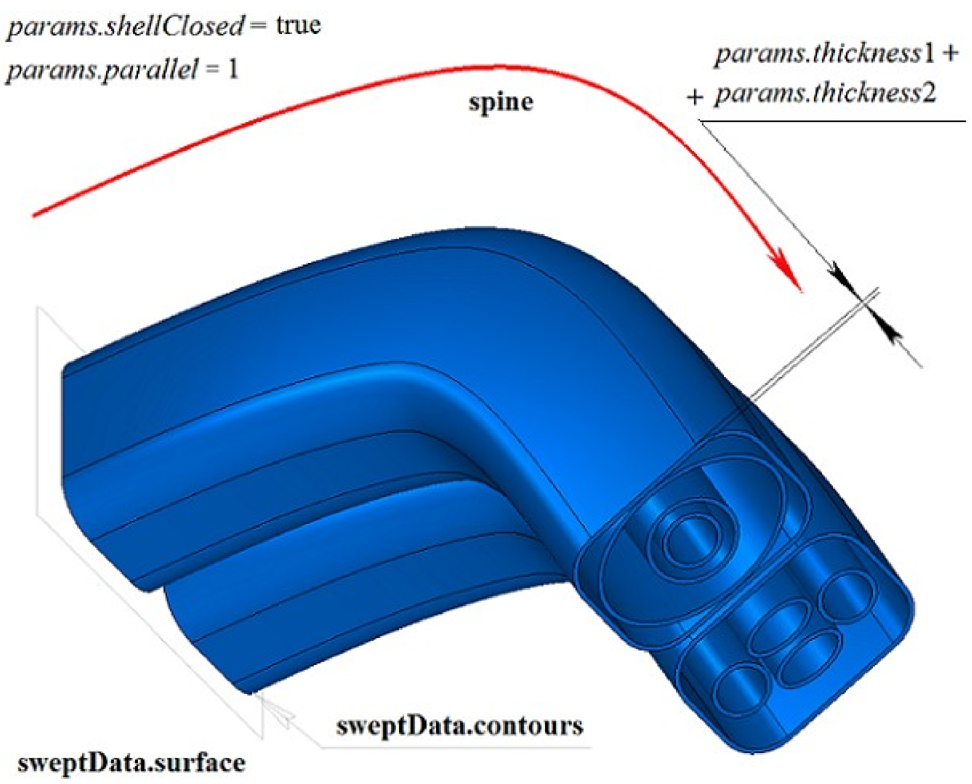


Figure M.1.5.14.

In Figure M.1.5.15, you can see a non-closed multi-part swept solid that was constructed by moving the set of flat contours along the guiding curve shown in Figure M.1.5.12. When a non-closed swept solid is constructed, the contours should not be nested.

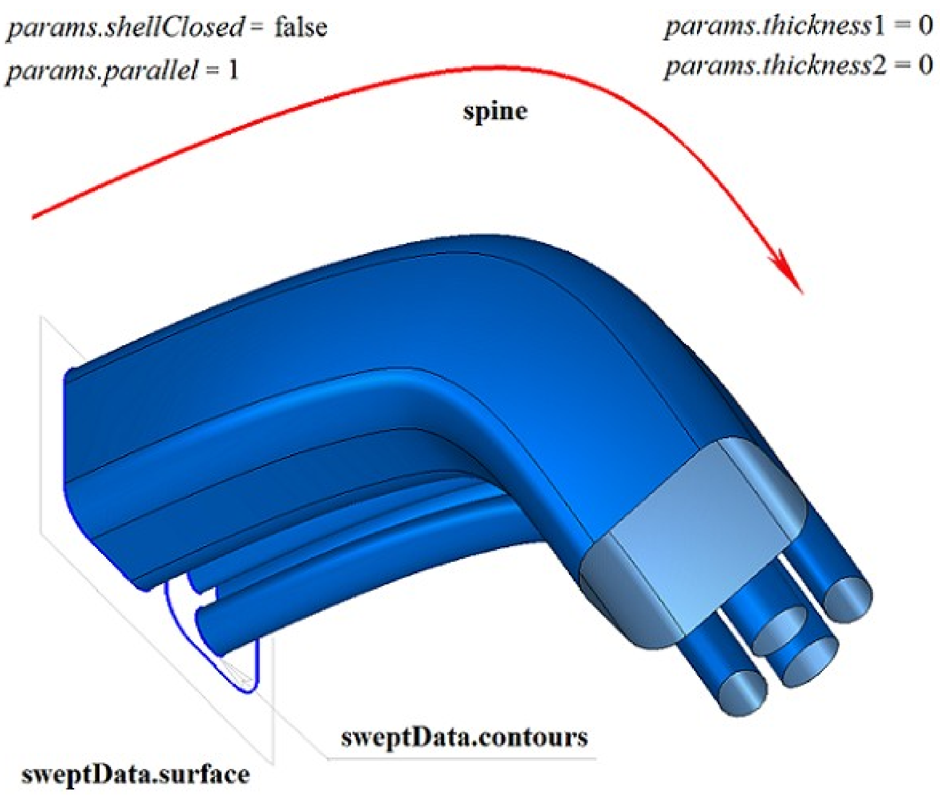


Figure M.1.5.15.

In Figure M.1.5.16, you can see two three-dimensional contours (**contour3D** 0 and **contour3D** 1) and **spine** guiding curve that will be used to construct swept solids.

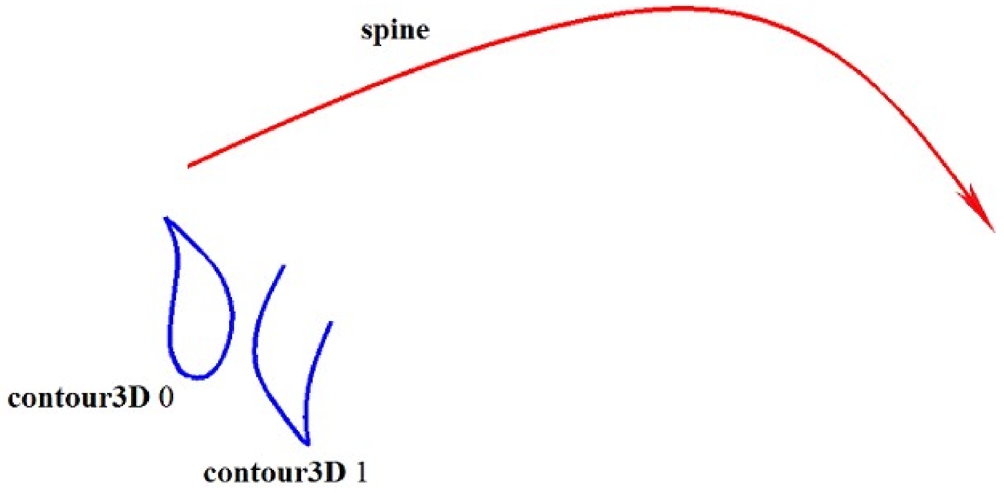


Figure M.1.5.16.

In Figure M.1.5.17, you can see a closed doubly-connected thin wall swept solid that was constructed by moving the three-dimensional contours along the guiding curve (please see Figure M.1.5.16).

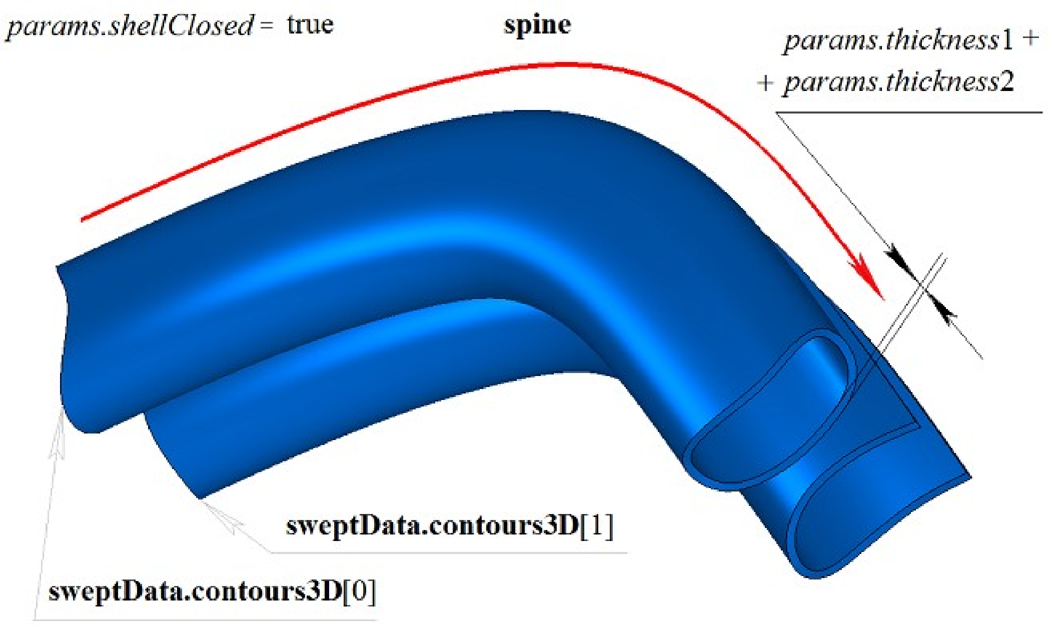


Figure M.1.5.17.

In Figure M.1.5.18, you can see two non-closed solids that were constructed by moving three-dimensional contours along the guiding curve (please see Figure M.1.5.16).

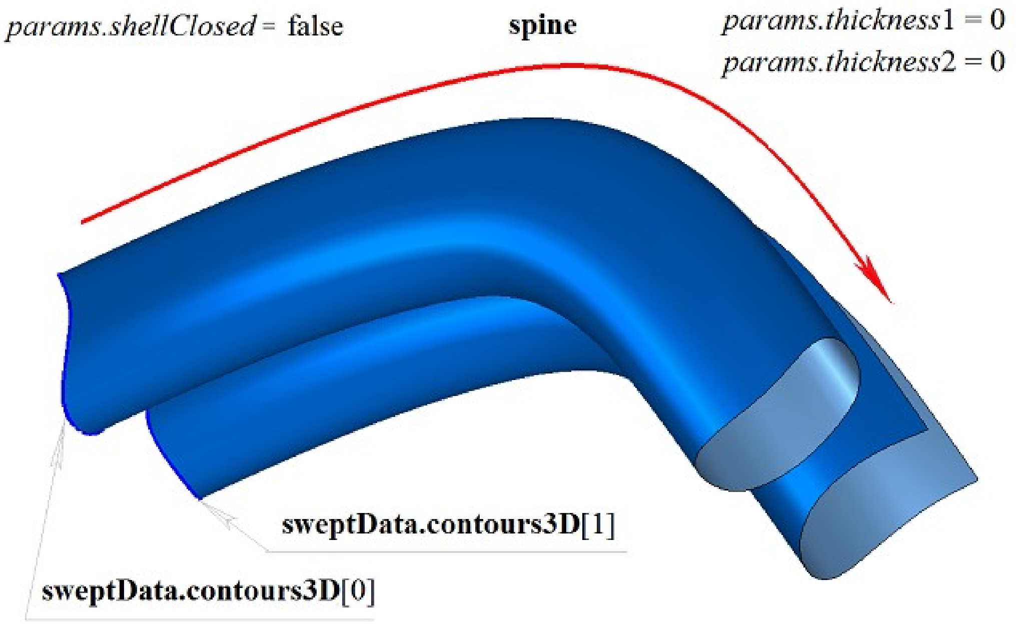


Figure M.1.5.18.

EvolutionSolid method that is used to construct a swept solid adds MbEvolutionSolid constructor to the log of the newly constructed solid, which contains all necessary data to construct the solid. MbEvolutionSolid constructor is declared in cr_evolution_solid.h.

test.exe constructs a swept solid by «Create->Solid->By curves->By moving curves» menu command.

M.1.6. Constructing a Solid by Flat Sections

Method
MbResultType
LoftedSolid (SArray<MbPlacement3D> & **places**,
RPArray<MbContour> & **contours**,
const MbCurve3D * **spine**,
LoftedValues & params,
SArray<MbCartPoint3D> * **points**,
const MbSNameMaker & names,
PArray<MbSNameMaker> & snames,
MbSolid *& **result**)
constructs a solid based on flat sections.

Method input parameters are:

- **places** is the set of local coordinate systems of generating contours,
- **contours** is the set of generating contours,
- **spine** is the guiding curve (it may be missing),
- params are construction parameters,
- **points** are a set of control points (it may be missing),
- names is face namer,
- snames are namers of generating contours.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns an error code from MbResultType enumeration.

This method is declared in action_solid.h file.

The surface of the newly constructed solid contains all the flat curves defining the solid. **places** set contains local coordinate systems, two-dimensional **contours** lie in their XY plane. **places** and **contours** sets are aligned by index: **contours**[*i*] is located in XY plane of **places**[*i*] local coordinate system. **contours** may have arbitrary orientation. If all contours **contours** are closed, then beginnings of local coordinate system are changed so that the beginnings should located as close as possible to each other in order to prevent twisting of the surfaces. **points** control points permit you to change the joining points in the curves of the set of contours. If **points** set is not empty then it should be aligned with **places** and **contours** sets. **spine** guiding curve can be used to control solid shape between sections. Arbitrary curves can be used as guiding curve for a solid.

params parameter contains information about movement mode, presence of solid walls and their thickness and data whether the constructed solid is closed. *params.thickness*1 and *params.thickness*2 parameters define wall thickness of the constructed thin-walled solid. *params.thickness*1 parameter defines outward offset from the generator curve, and *params.thickness*2 parameter defines inward offset from the generator curve. *params.shellClosed* parameter controls whether the constructed solid is closed. *params.checkSelfInt* parameter defines the need to check the result of construction for self-intersection. By default *params.checkSelfInt*=false and the check is not performed. *params.closed* parameter controls the presence of edges of the solid. If *params.closed*=true then there are no edges and the solid has torus topology. *params.***vector**1 and *params.***vector**2 vectors define the direction of the solid in the area of the start and end edges. For example, they permit you to define the direction of the solid in the area of the edges orthogonal to edge planes. By default, *params.***vector**1 and *params.***vector**2 vectors are equal to zero.

In Figure M.1.6.1, you can see the data used for construction and parameters inheritance scheme for a constructed solid by LoftedValues & *params* flat sections.

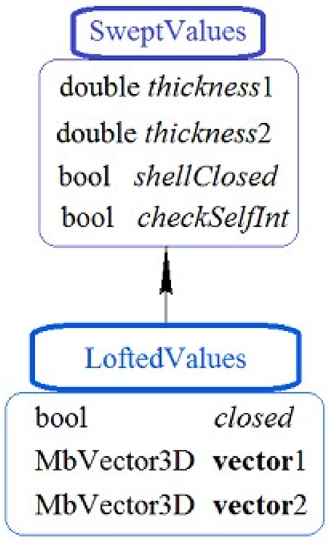


Figure M.1.6.1.

names and snames parameters are responsible for naming faces of the newly constructed solid.

In Figure M.1.6.2, you can see a set of two-dimensional **contours** and their local coordinate systems (**places**). Arrows indicate the directions of normals in local coordinate systems.

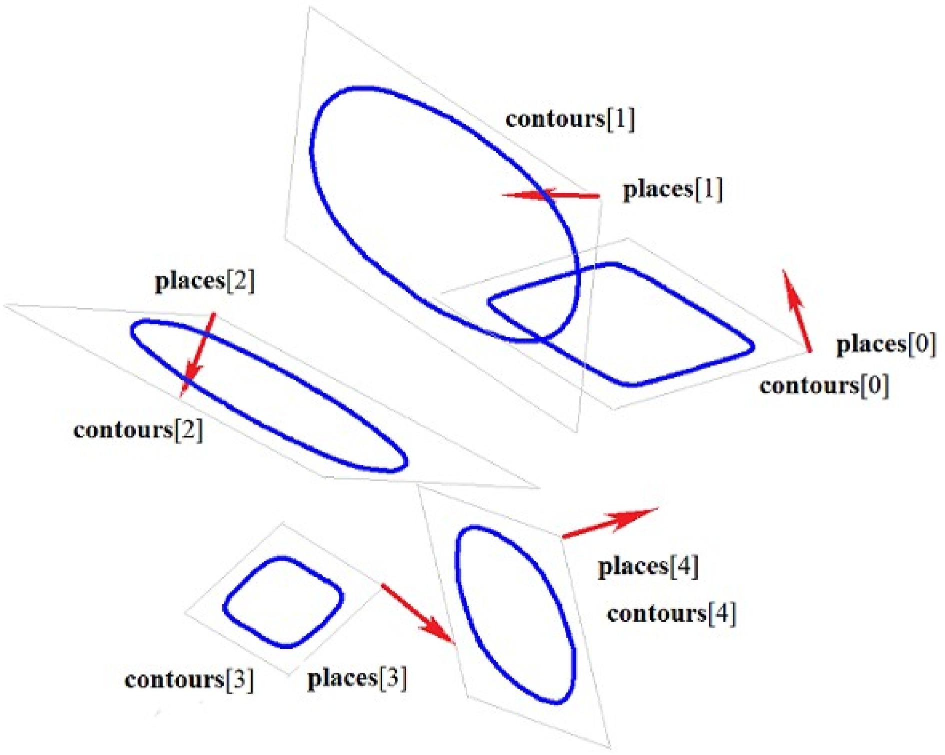


Figure M.1.6.2.

In Figure M.1.6.3, you can see a solid that was constructed by flat sections shown in Figure M.1.6.2 according to the specified directions of the normals at the edges if *params.closed*=false.

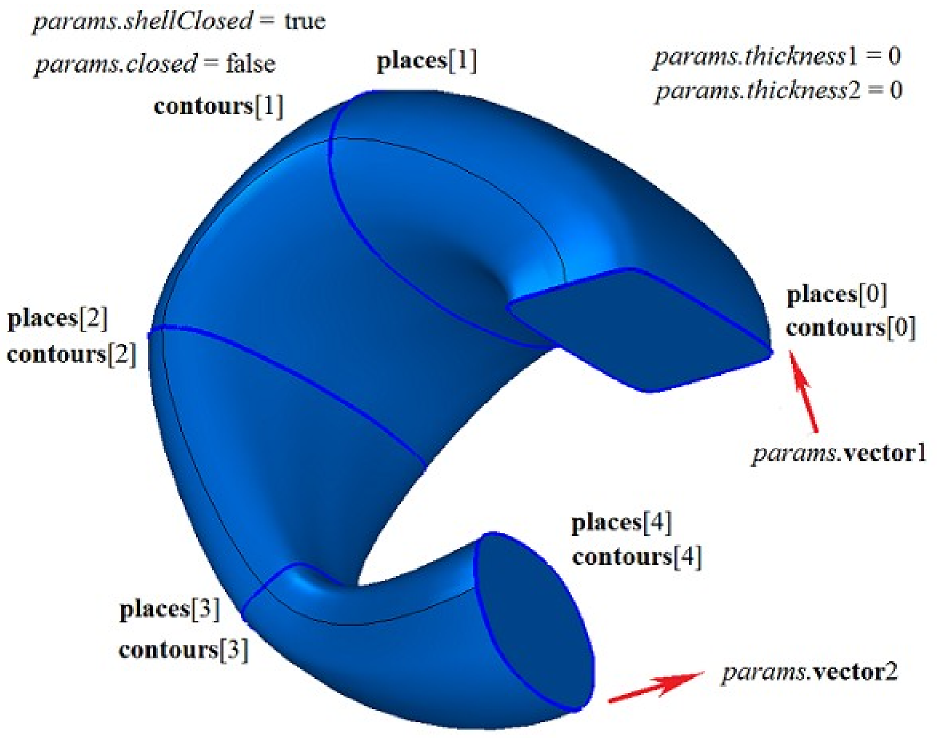


Figure M.1.6.3.

In Figure M.1.6.4, you can see a solid that was constructed by flat sections shown in Figure M.1.6.2 if *params.closed*=true. There are no edges; the solid has torus topology.

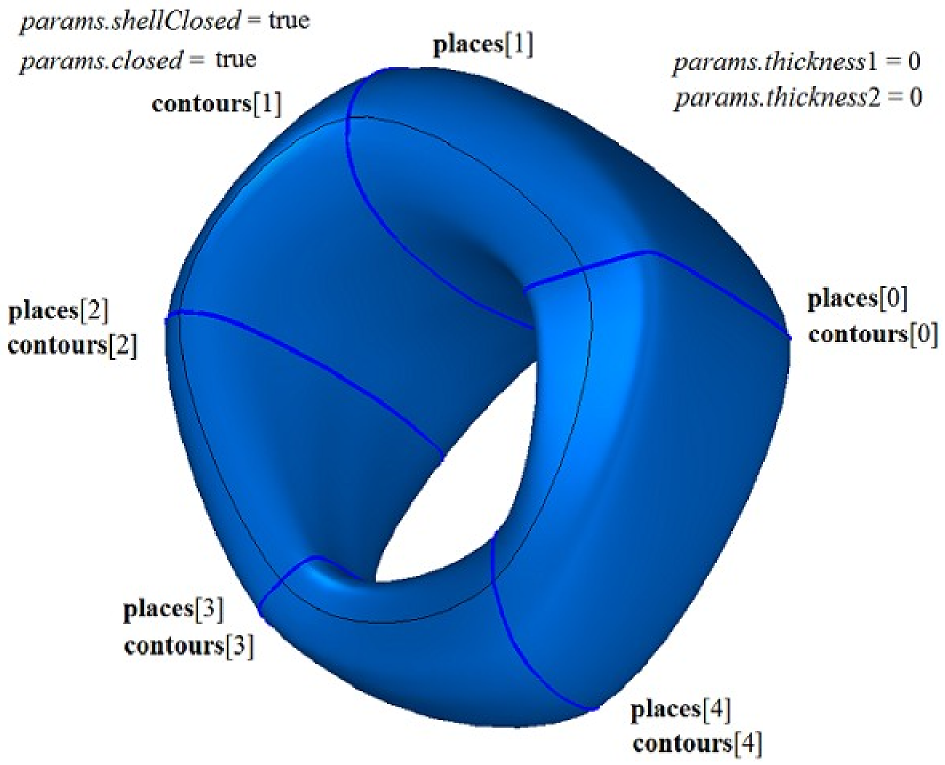


Figure M.1.6.4.

In Figure M.1.6.5, you can see a thin-walled solid that was constructed by flat sections shown in Figure M.1.6.2 without determination of normals at the edges if *params.closed*=false.

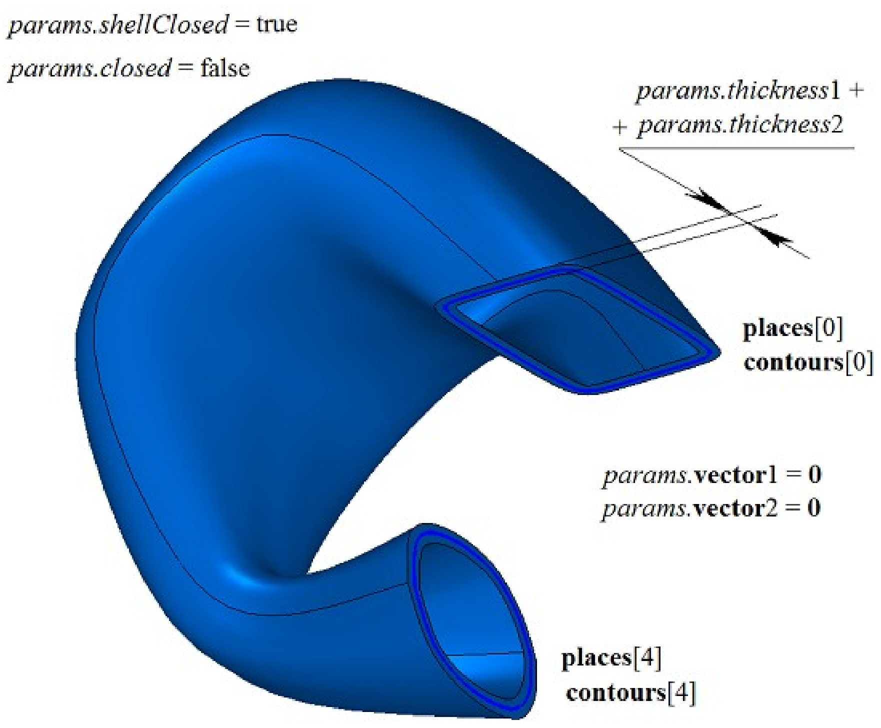


Figure M.1.6.5.

In Figure M.1.6.6, you can see a thin-walled solid that was constructed by flat sections shown in Figure M.1.6.2 according to specified normals at the edges if *params.closed*=false.

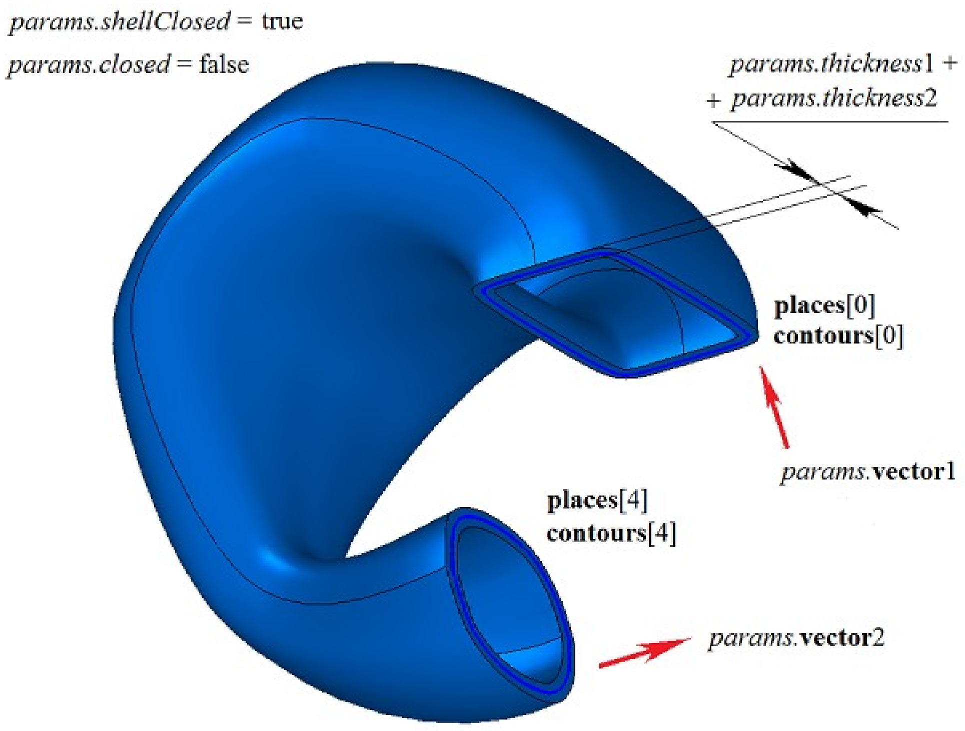


Figure M.1.6.6.

In Figure M.1.6.7, you can see a thin-walled solid that was constructed by non-closed flat contours with not defined normals at the edges if *params.closed*=false. *params.thickness*1 and *params.thickness*2 parameters should not be equal to zero.

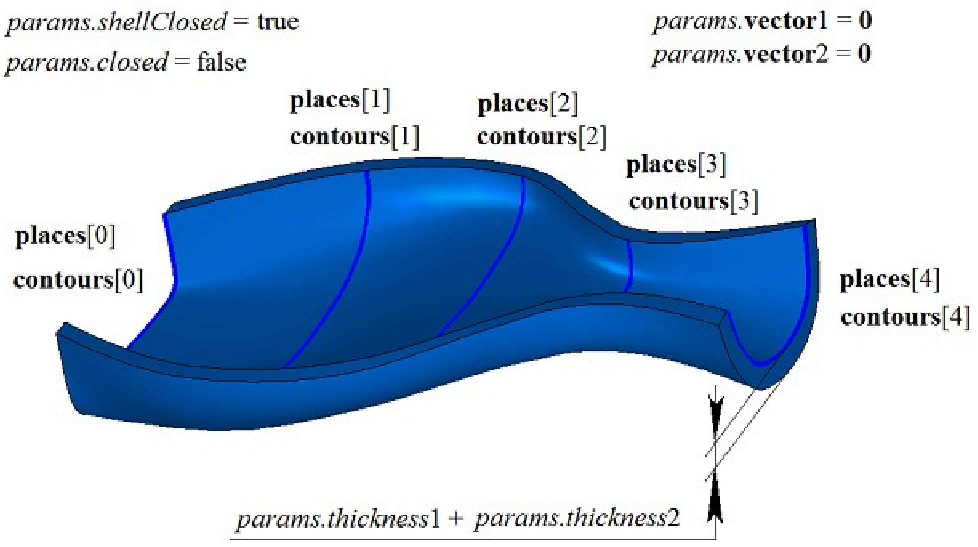


Figure M.1.6.7.

In Figure M.1.6.8, you can see a non-closed solid constructed to non-closed flat contours without normals at the edges if *params.closed*=false. *params.thickness*1 and *params.thickness*2 parameters may be equal to zero.

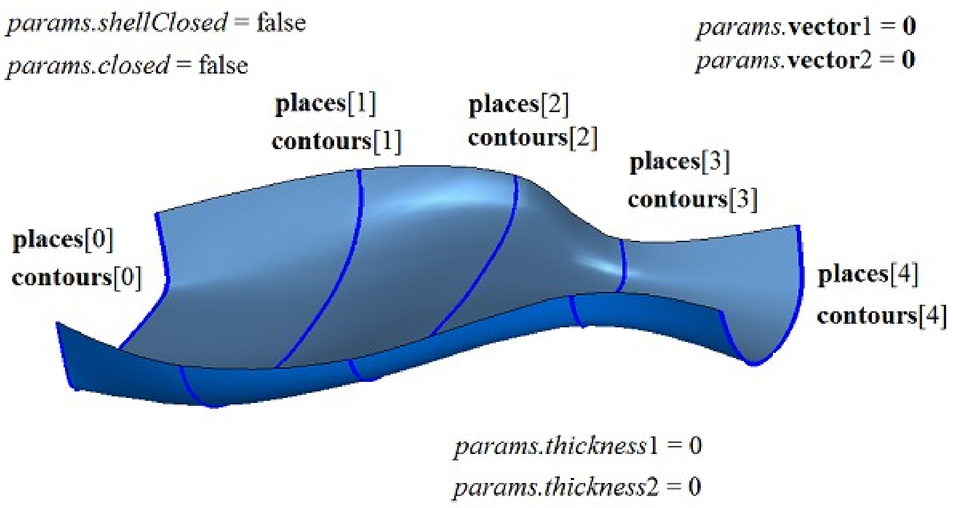


Figure M.1.6.8.

If flat contours have unequal quantities of segments, then some segments are divided so that the quantity of segments in all **contous** contour sets should be the same. In Figure M.1.6.9, you can see three contours that have unequal quantities of segments.

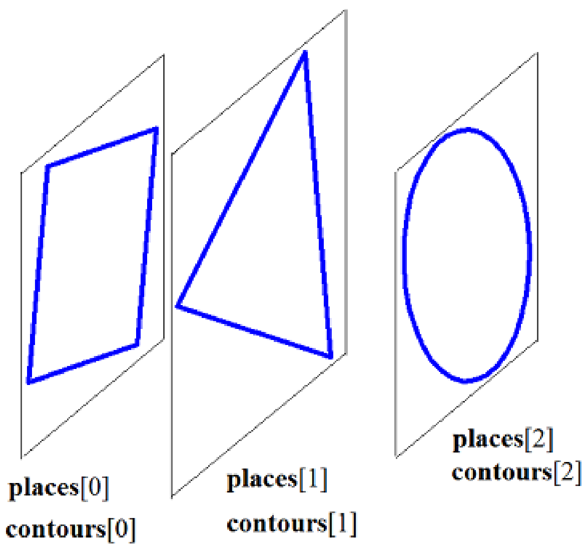


Figure M.1.6.9.

In Figure M.1.6.10, you can see a solid that was constructed by these contours: one segment of triangular contour is divided into two segments, and a circle is divided into four arcs.

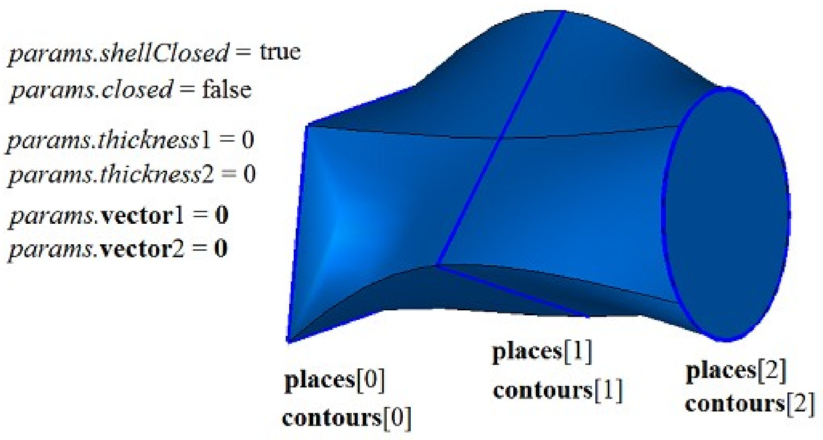


Figure M.1.6.10.

points control points permit you to define the position of edges connecting vertices of different contours in the set. **points**[*i*] indicate the positions of the joints between the segments of different contours of the set that should be connected by edges. To demonstrate the use of **points** control points, let's construct a solid by flat sections shown in Figure M.1.6.11.

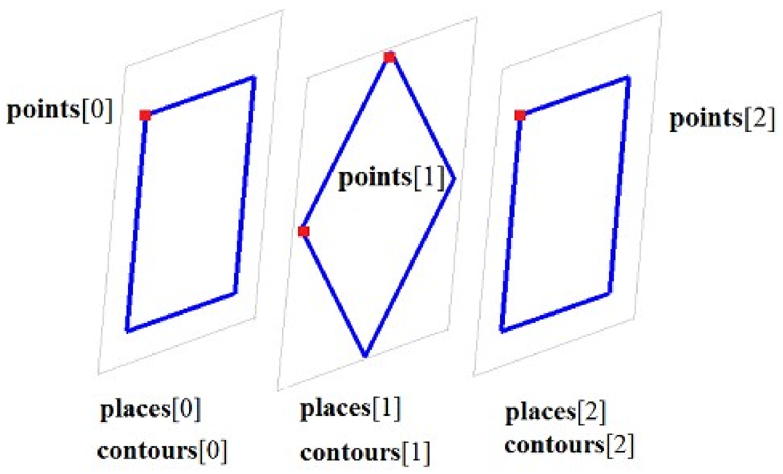


Figure M.1.6.11.

In Figure M.1.6.12 and M.1.6.13, you can see solids that were constructed by flat sections shown in Figure M.1.6.11 according to different **points** control points.

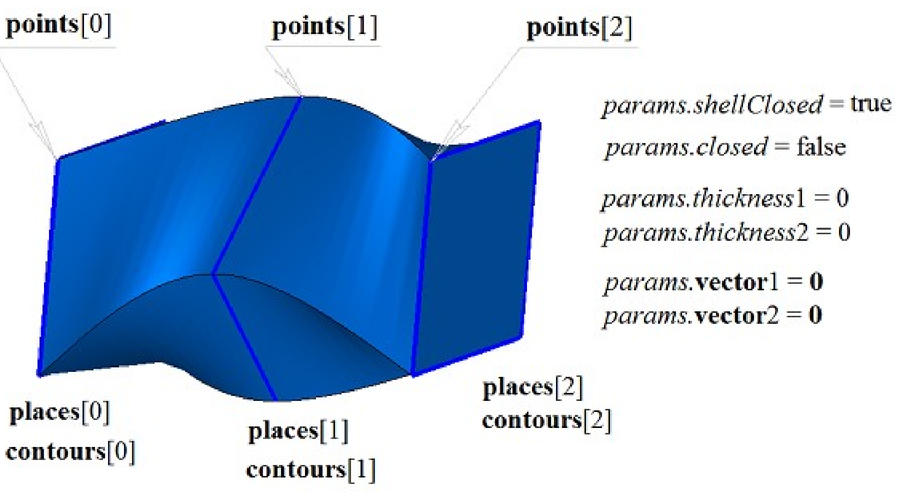


Figure M.1.6.12.

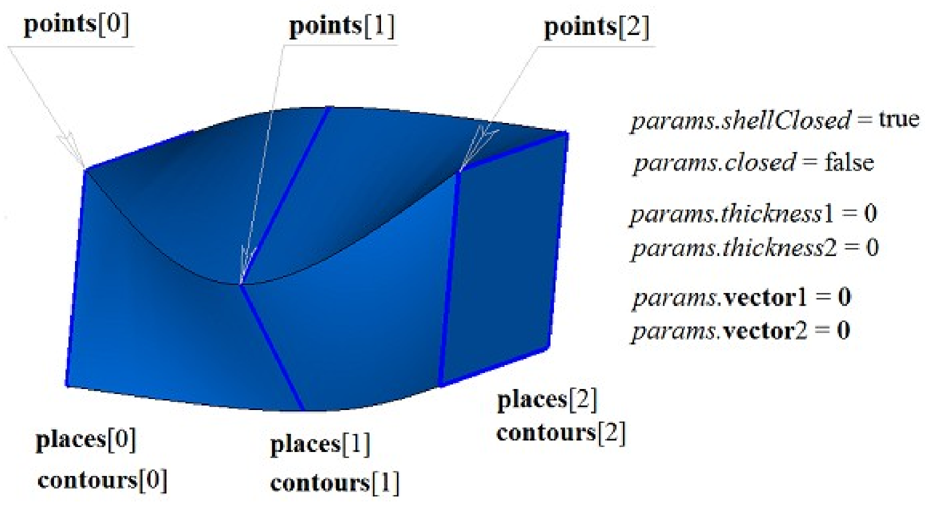


Figure M.1.6.13.

In Figure M.1.6.14, you can see two two-dimensional contours and **spine** curve that would be a guiding curve when a solid would be constructed by flat sections with a guiding curve.

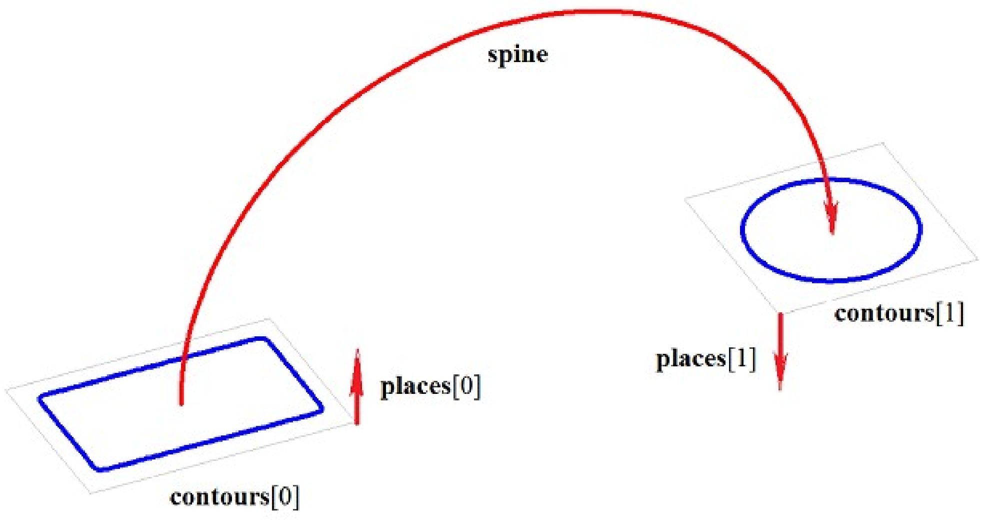


Figure M.1.6.14.

In Figure M.1.6.15, you can see a solid that was constructed by flat sections and a guiding curve shown in Figure M.1.6.14.

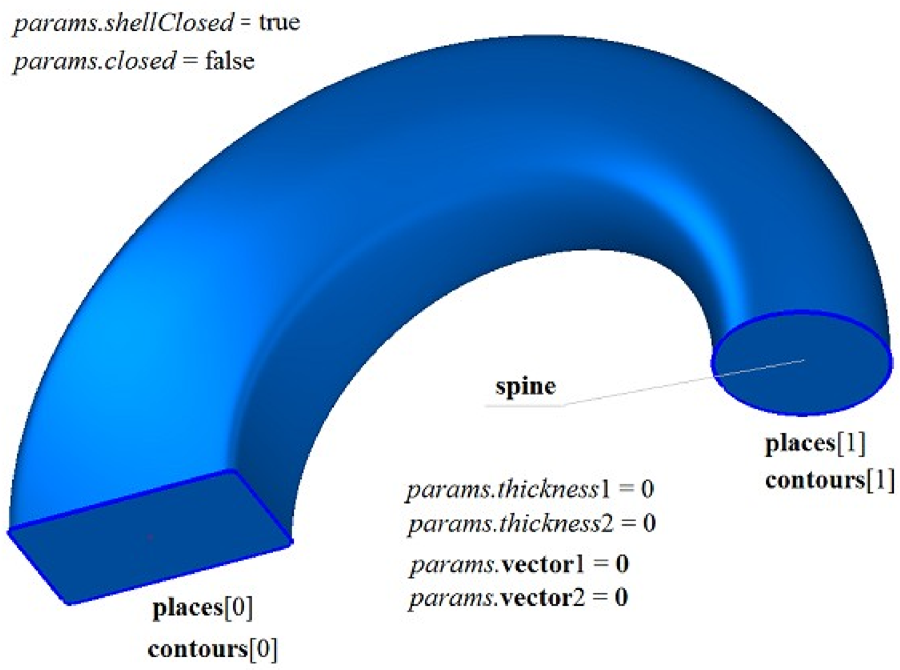


Figure M.1.6.15.

In Figure M.1.6.16, you can see a thin-walled solid that was constructed by flat sections and the guiding curve shown in Figure M.1.6.14.

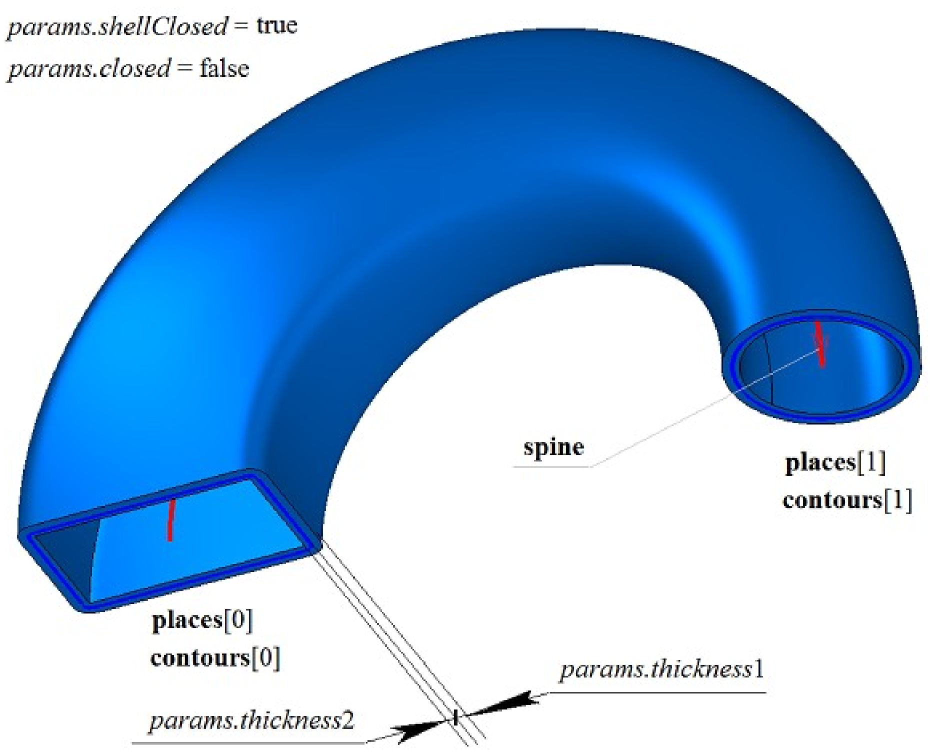


Figure M.1.6.16.

In Figure M.1.6.17, you can see a non-closed solid that was constructed by flat sections and the guiding curve shown in Figure M.1.6.14.

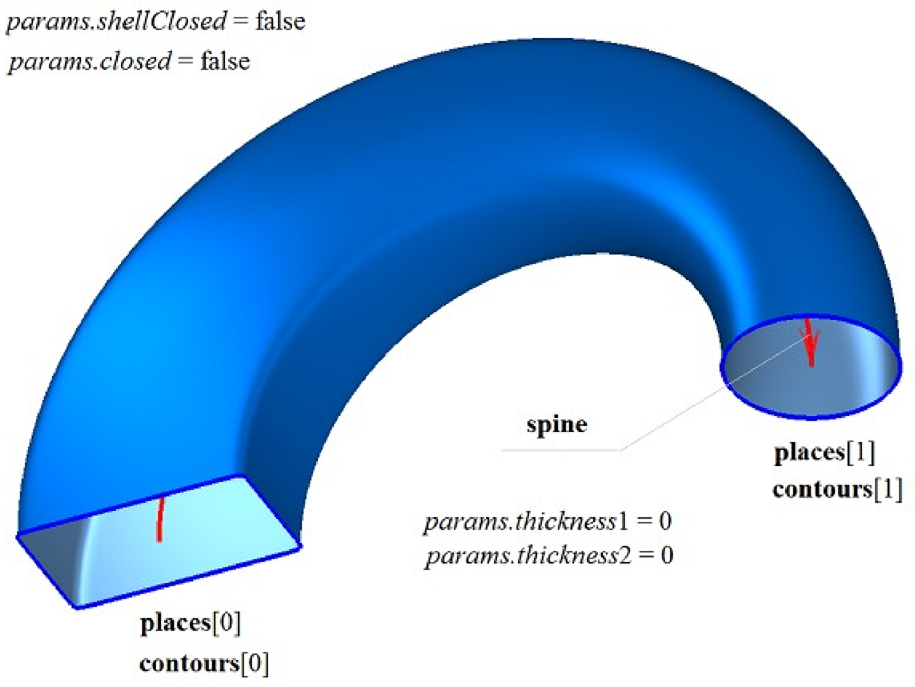


Figure M.1.6.17.

LoftedSolid method constructs a solid by flat sections; it adds MbLoftedSolid constructor in the log of the newly constructed solid. This constructor contains all data required to construct a solid. MbLoftedSolid constructor is declared in cr_lofted_solid.h file.

test.exe test application constructs a solid by flat sections using «Create->Solid->By curves->By sections»; «Create->Solid->By curves->By sections with a guiding curve»; «Create->Solid->By curves->By sections»; and «Create->Shell->By curves->By sections with a guiding curve» menu commands.

M.1.7. Creating a Solid by a Specified Set of Faces

Method
MbSolid *
CreateSolid (MbFaceShell & **faceSet**,
const MbSNameMaker & names)
creates a solid with the specified set of faces without construction history.

Method input parameters are:

- **faceSet** is the set of faces,
- names is faces namer.

If successful, the method returns the newly constructed solid, otherwise it returns zero.

This method is declared in action_solid.h file.

faceSet parameter contains the initial set of faces for a solid. names parameter is responsible for naming faces of the constructed solid.

In Figure M.1.7.1, you can see a solid that was constructed by a set of faces.

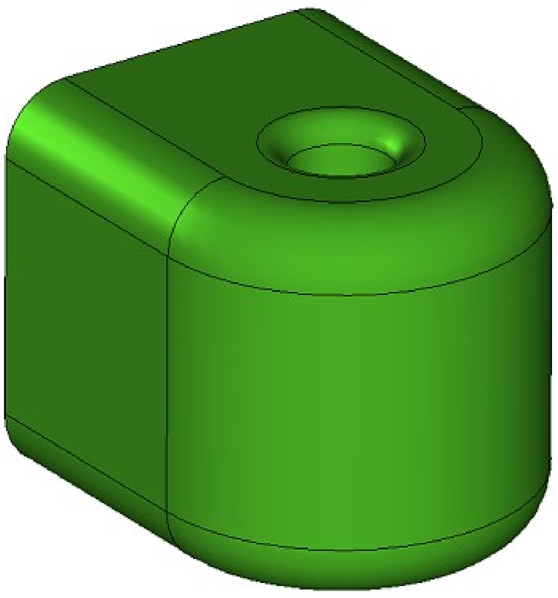


Figure M.1.7.1.

This method gives names to unnamed faces, edges ribs and vertices and then it creates a solid for the specified set of faces. This method doesn't check or construct anything. If the set of faces contains boundary ribs, this method constructs a non-closed solid. **CreateSolid** method adds MbSimpleCreator simple constructor to the log of the newly constructed solid. This constructor is declared in cr_simple_creator.h file.

M2. OPERATIONS ON SOLIDS

One of approaches to construction of solids in geometrical modelling is similar to making a modeled object. First, simple solids are constructed and then a set of actions is executed in order to construct more complex solids from simple solids. More complex solids are constructed by executing operations with previously constructed solids. All operations are recorded in a construction log. For closed and non-closed solids the same operations can lead to different results.

M.2.1. Boolean Operation on Solids

Method
MbResultType
BooleanResult (MbSolid & **solid1**,
MbeCopyMode *sameShell*1,
MbSolid & **solid2**,
MbeCopyMode *sameShell*2,
OperationType *oType*,
const MbSNameMaker & names,
bool *mergeFaces*,
bool *closed*,
MbSolid *& **result**)
constructs a new solid by executing a Boolean operation on two specified solids.

Input parameters of the method are as follows:

- **solid1** is the first solid for the Boolean operation,
- *sameShell*1 is copying method for the first solid,
- **solid2** is the second solid for the Boolean operation,
- *sameShell*2 is copying method for the second solid,
- *oType* is Boolean operation type: bo_Union means merging of the solids,
  bo_Intersect means intersection of the solids,
  bo_Difference means subtraction of the solids,
- names is a namer used for versioning,
- *mergeFaces* indicates whether similar faces should be merged,
- *closed* indicates whether it is required to verify closedness of constructed solid.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration. The method is declared in action_solid.h file.

The method executes merge, intersect or subtract operations on points of two solids (**solid1** and **solid2**). *sameShell*1 and *sameShell*2 parameters control transfer of faces, edges and vertices of **solid1** and **solid2** original solids to **result** constructed solid.

*sameShell*1 and *sameShell*2 parameters may take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

oType (OperationType) parameter defines Boolean operation type; it takes one of the following three values: bo_Union, bo_Intersect, bo_Difference. If *oType*=bo_Union, then the method merges **solid1** and **solid2** solids; if *oType*=bo_Intersect, then the method intersects **solid1** and **solid2** solids; if *oType*=bo_Difference, then the method subtracts **solid2** solid from **solid1** solid.

names parameter is used to version the Boolean operation.

mergeFaces parameter controls merging of similar faces. If *mergeFaces*==false, then similar faces are not merged.

closed parameter is used only for nonclosed solids and it informs the operation whether it is required to check the result for closedness. For non-closed solids, Boolean operation is executed by **BooleanShell** method described in item M.2.2. Boolean Operation on Non-Closed Solids.

In Fig. M.2.1.1, **solid1** and **solid2** original operand solids are shown.

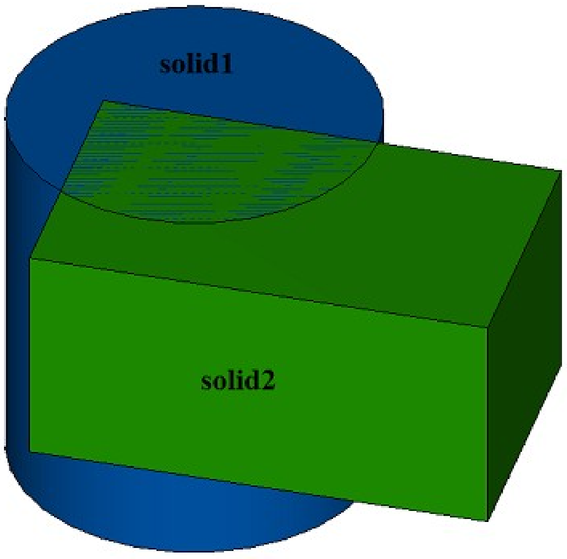
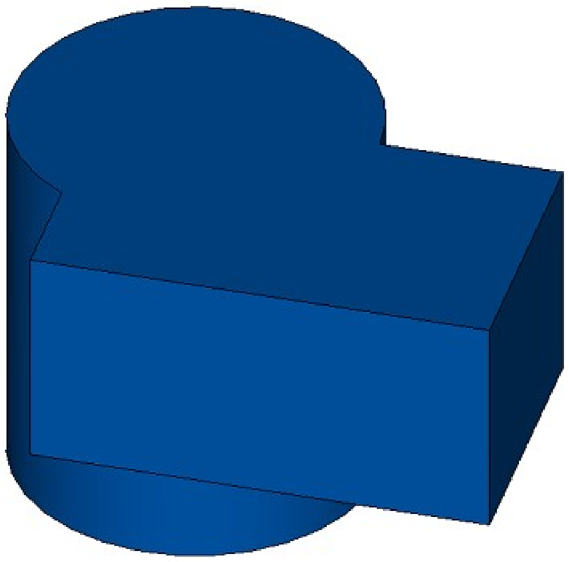


Fig. M.2.1.1.

In Fig. M.2.1.2, you can see the result of Boolean operation that merges **solid1** and **solid2** solids shown in Figure M.2.1.1.

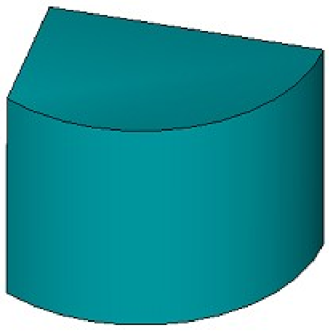


oType = bo_Union

solid1 + solid2

Fig. M.2.1.2.

In Fig. M.2.1.3, you can see the result of Boolean operation that intersects **solid1** and **solid2** solids shown in Figure M.2.1.1.

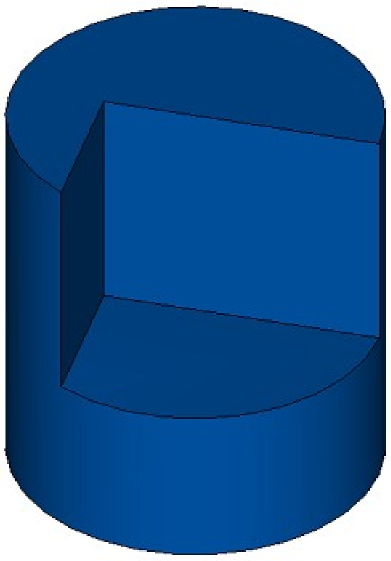


oType = bo_Intersect

solid1&solid2

Fig. M.2.1.3.

In Fig. M.2.1.4, you can see the result of Boolean operation that substracts **solid2** solid from **solid1** solid; they are shown in Figure M.2.1.1.



oType = bo_Difference

solid1 - solid2

Fig. M.2.1.4.

In Fig. M.2.1.5, you can see the result of Boolean operation that subtracts **solid1** solid from **solid2** solid; two original solids are shown in Figure M.2.1.1.

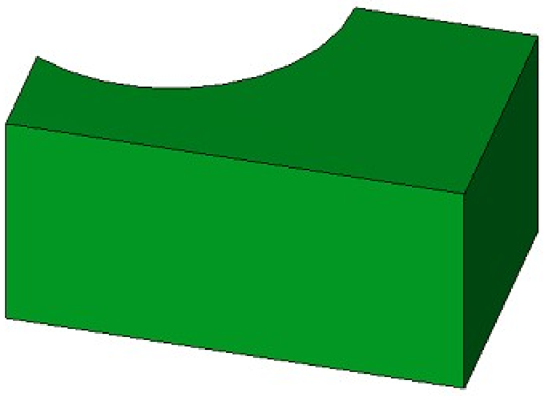


Fig. M.2.1.5.

In order to demonstrate the use of *mergeFaces* parameter, let's analyze Boolean operations executed on **solid1** and **solid2** original solids shown in Figure M.2.1.6.

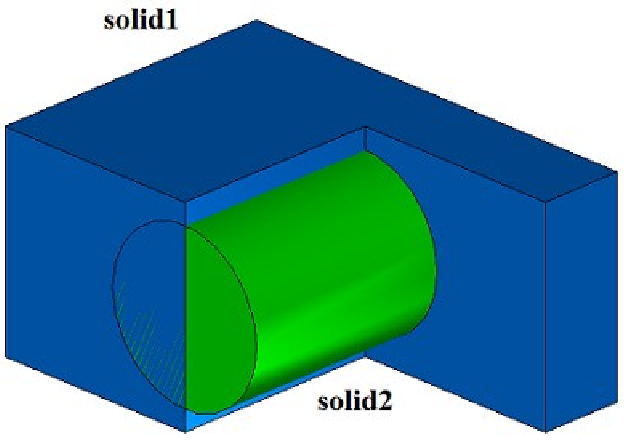


Fig. M.2.1.6.

In Fig. M.2.1.7, you can see **result** solid that has was constructed by merging **solid1** and **solid2** solids, when the method was used with *mergeFaces*==true. In Fig. M.2.1.8, you can see **result** solid that was constructed by merging **solid1** and **solid2** solids, when the method was used with *mergeFaces*==false parameter. Coinciding faces are not merged in Figure M.2.1.8.

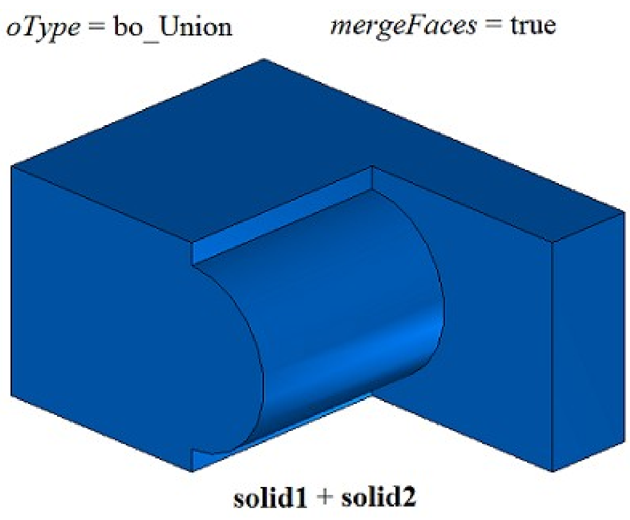


Fig. M.2.1.7.

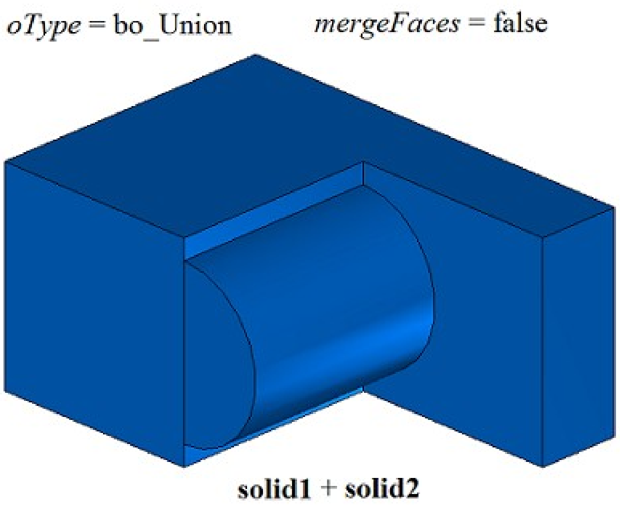


Fig. M.2.1.8.

In Fig. M.2.1.9, you can see **result** solid that was constructed by subtracting **solid2** solid from **solid1** solid, when the method concerned was used with *closed*== true parameter. In Fig. M.2.1.10, you can see **result** solid that was constructed by subtracting **solid2** solid from **solid1** solid when the method was used with *closed*==false parameter. Coinciding faces are not merged in Figure M.2.1.10.

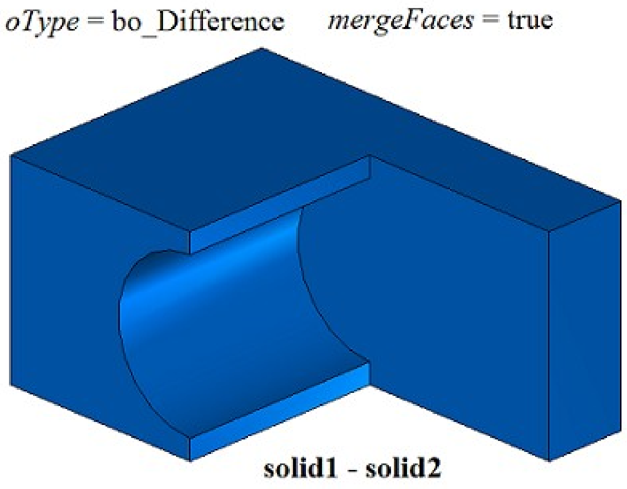


Fig. M.2.1.9.

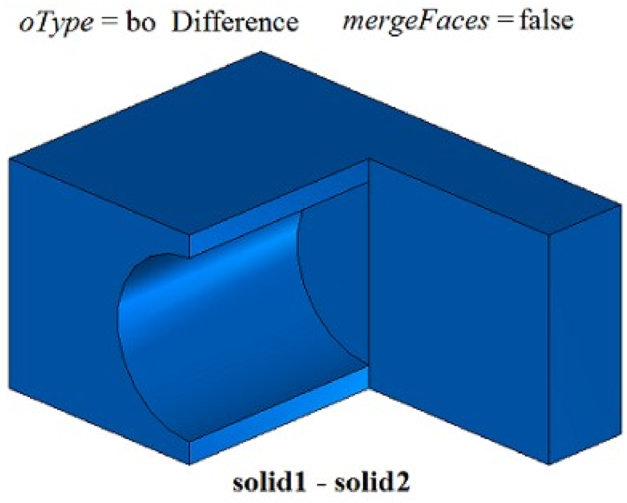


Fig. M.2.1.10.

Method
MbResultType
BooleanSolid (MbSolid & **solid1**,
MbeCopyMode *sameShell*1,
MbSolid & **solid2**,
MbeCopyMode *sameShell*2,
OperationType *oType*,
const MbSNameMaker & names,
MbSolid *& **result**)
executes the same actions as **BooleanResult** method when *mergeFaces*==true and *closed*==true. **BooleanSolid** method is applicable to closed solids only.

BooleanResult and **BooleanSolid** methods add MbBooleanSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbBooleanSolid constructor is declared in cr_boolean_solid.h file.

test.exe test application executes Boolean operations on solids using New ->Solid -> By Gluing to Solid -> a Solid, New ->Solid -> By cutting from Solid ->Solid, New ->Solid -> Intersecting with Solid ->Solid menu commands.

M.2.2. Boolean Operation on Non-Closed Solids

Method
MbResultType
BooleanShell (MbSolid & **solid1**,
MbeCopyMode *sameShell*1,
MbSolid & **solid2**,
MbeCopyMode *sameShell*2,
OperationType *oType*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a new solid by executing a Boolean operation on two given non-closed solids.

Input parameters of the method are as follows:

- **solid1** is the first solid for the Boolean operation,
- *sameShell*1 is copying method for the first solid,
- **solid2** is the second solid for the Boolean operation,
- *sameShell*2 is copying method for the second solid,
- *oType* is Boolean operation type: bo_Variety is union of solids,
  bo_Internal is intersection of solids,
  bo_External is subtraction of solids,
- names is a namer used used for versioning.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

The method executes merging, intersecting and subtractiom operations on points of two nonclosed solids (**solid1** and **solid2**). *sameShell*1 and *sameShell*2 parameters control transfer of faces, edges and vertices of **solid1** and **solid2** original solids to **result** constructed solid.

*sameShell*1 and *sameShell*2 parameters may take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

*oType (*OperationType) parameter defines Boolean operation type; it takes one of the following three values: bo_Variety, bo_Internal, bo_External. If *oType*=bo_Variety, then the method merges surfaces of **solid1** and **solid2** solids; if *oType*=bo_Internal, then the method intersects surfaces of **solid1** and **solid2** solids; if *oType*=bo_External, then the method subtracts **solid2** solid from **solid1** solid. names parameter is used to version the Boolean operation.

In Fig. M.2.2.1, you can see **solid1** and **solid2** original non-closed solids.

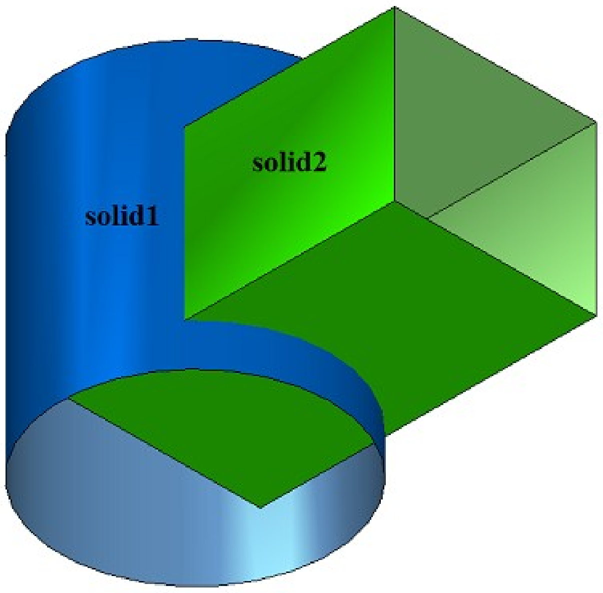
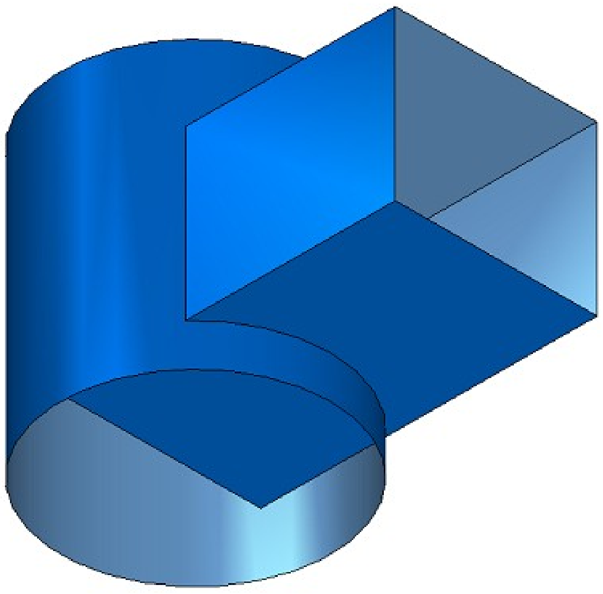


Fig. M.2.2.1.

In Fig. M.2.2.2, you can see the result of Boolean operation that merges **solid1** and **solid2** non-closed solids shown in Figure M.2.2.1.

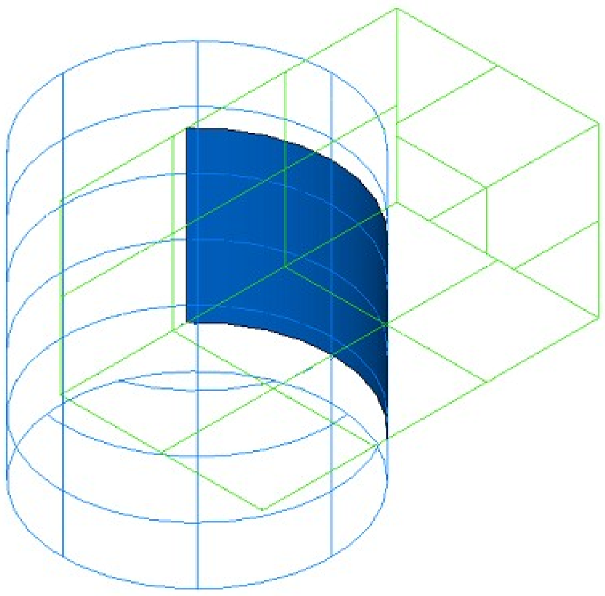


oType = bo_Variety

solid1 + solid2

Fig. M.2.2.2.

In Fig. M.2.2.3, you can see the result of non-closed solids **solid1** and **solid2** truncation Boolean operation cutting shown in Figure M.2.2.1.

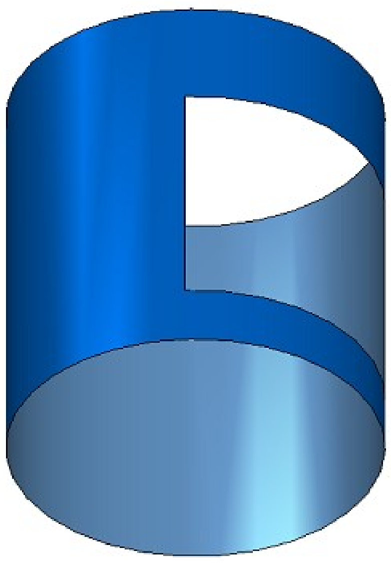


oType = bo_Internal

solid1&solid2

Fig. M.2.2.3.

In Fig. M.2.2.4, you can see the result of Boolean operation that subtracts **solid2** non-closed solid from **solid1** non-closed solid shown in Figure M.2.2.1.



oType = bo_External

solid1 - solid2

Fig. M.2.2.4.

In Fig. M.2.2.5, you can see the result of Boolean operation that subtracts **solid1** non-closed solid from **solid2** non-closed solid shown in Figure M.2.2.1.

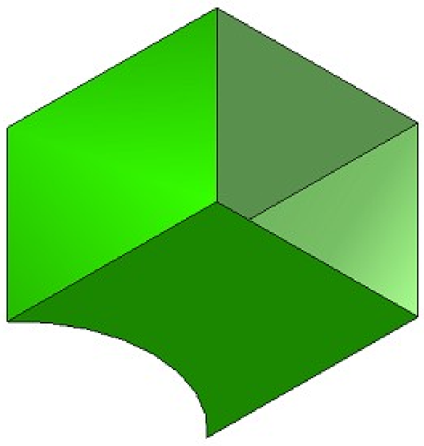


Fig. M.2.2.5.

This method works with non-closed solids, but the second operand may be a closed solid. The method executes a Boolean operation having the same name on a set of points on the surfaces of the solids.

BooleanShell method adds MbBooleanSolid constructor in a log of newly constructed solid that contains all data required to execute the operation. MbBooleanSolid constructor is declared in cr_boolean_solid.h file.

Test.exe test application executes Boolean operations on solids using New -> Shell -> On Base of Shell -> By Merging with Shell, New -> Shell -> On Base of Shell -> By subtracting Shell, New -> Shell -> On Base of Shell -> By Limiting by Shell menu commands.

M.2.3. Boolean Operation on Extrusion Solid

Method
MbResultType
ExtrusionResult (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbSweptData & **sweptData**,
const MbVector3D & **direction**,
ExtrusionValues & *params*,
OperationType *oType*,
const MbSNameMaker & names,
PArray<MbSNameMaker> & snames,
MbSolid *& **result**)
constructs an extruded solid and executes Boolean operation on the given solid using the constructed solid.

Input parameters of the method are as follows:

- **solid** is a solid given for Boolean operation,
- *sameShell* is copying version for the given solid,
- **sweptData** contains data on generating curves for construction of extruded solid,
- **direction** is extrusion direction,
- *params* are construction parameters,
- *oType* is Boolean operation type: bo_Union means merging of the solids,
  bo_Intersect means intersection of the solids,
  bo_Difference is subtraction of the solids,
- names is operation namer,
- snames are namers for extruded solid faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.The method is declared in action_solid.h file.

This method executes successive merging of the following two methods: **ExtrusionSolid** method that constructs a solid by extruding **sweptData** curves according to given *params* parameters in **direction** and **BooleanSolid** method that executes *oType* Boolean operation on **solid** solid that was constructed on the previous step. **ExtrusionSolid** method is described in item M.1.3. Constructing an Extrusion Solid, and **BooleanSolid** method is described in item M.2.1. Boolean Operation on Solids. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

oType (OperationType) parameter defines Boolean operation type; it takes one of the following three values: bo_Union, bo_Intersect, bo_Difference. If *oType*=bo_Union then the method merges **solid** solid and the extruded solid; if *oType*=bo_Intersect, then the method intersects **solid** solid and the extruded solid; if *oType*=bo_Difference, then the method subtracts the extruded solid from **solid** solid. names and snames parameters provide naming of the faces for newly constructed solid.

If the solid is constructed by extruding curves, then **ExtrusionResult** method provides the same capabilities as **ExtrusionSolid** method: extruded curves may be located in a plane (Figure M.1.3.2), at a curved surface (Figure M.1.3.12) or in space (Figure M.1.3.20). Extrusion may be executed in forward, backward or both directions; with slope or aclinal faces newly constructed solid may completely fill closed curves (Figure M.1.3.13) or have a thin wall (Figure M.1.3.14). We shall not repeat the description of all features of the method, we shall rather focus on some features associated with Boolean operations.

In Fig. M.2.3.1, you can see **solid** solid, a generating curve included in **sweptData** data and extrusion **direction**.

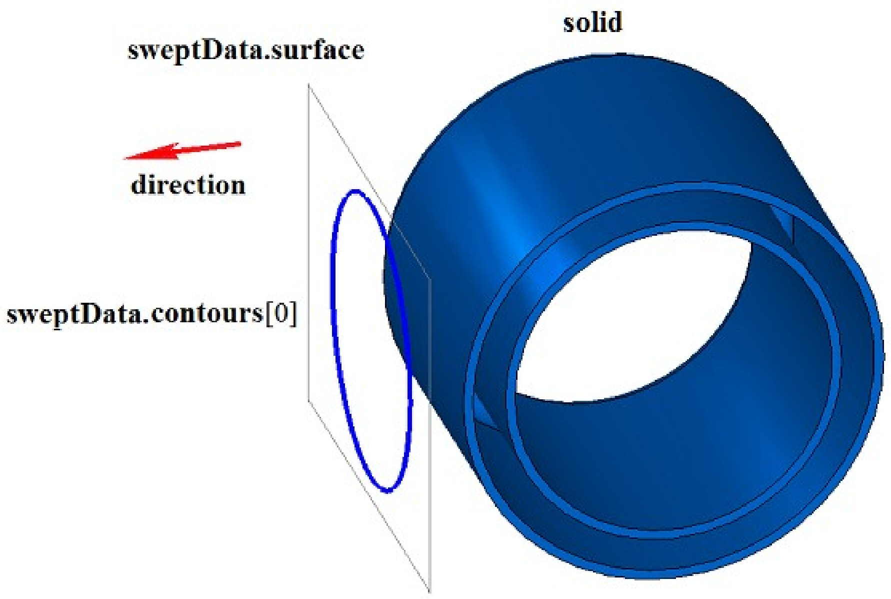


Fig. M.2.3.1.

In Fig. M.2.3.2, you can see the result of the Boolean operation that merges **solid** solid and a thin-walled solid received by extruding **sweptData** generating curve according to **direction** shown in Figure M.2.3.1 for predefined distance.

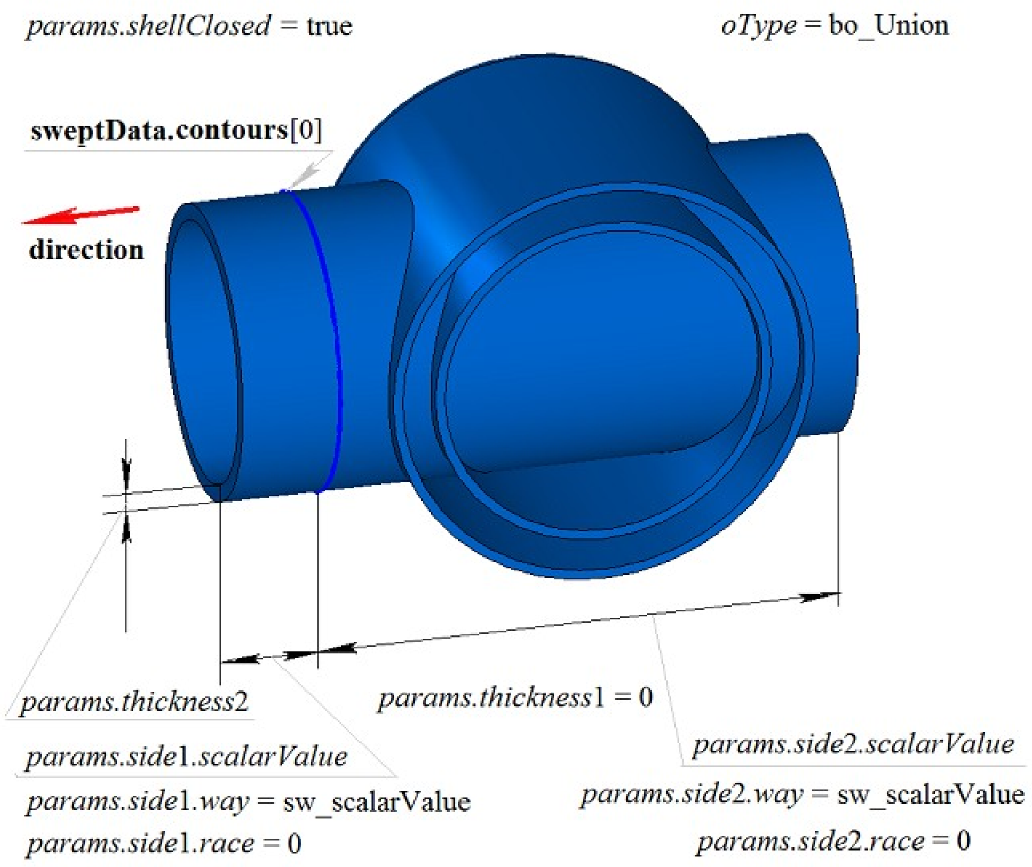


Fig. M.2.3.2.

In Fig. M.2.3.3, you can see the result of Boolean operation that merges **solid** solid and a solid received by extrusion of **sweptData** generating curve according to **direction** shown in Figure M.2.3.1. Generating curve was extruded in backward direction without a slope with «To Nearest Objects» option (*params.side*2.*way*=sw_shell).

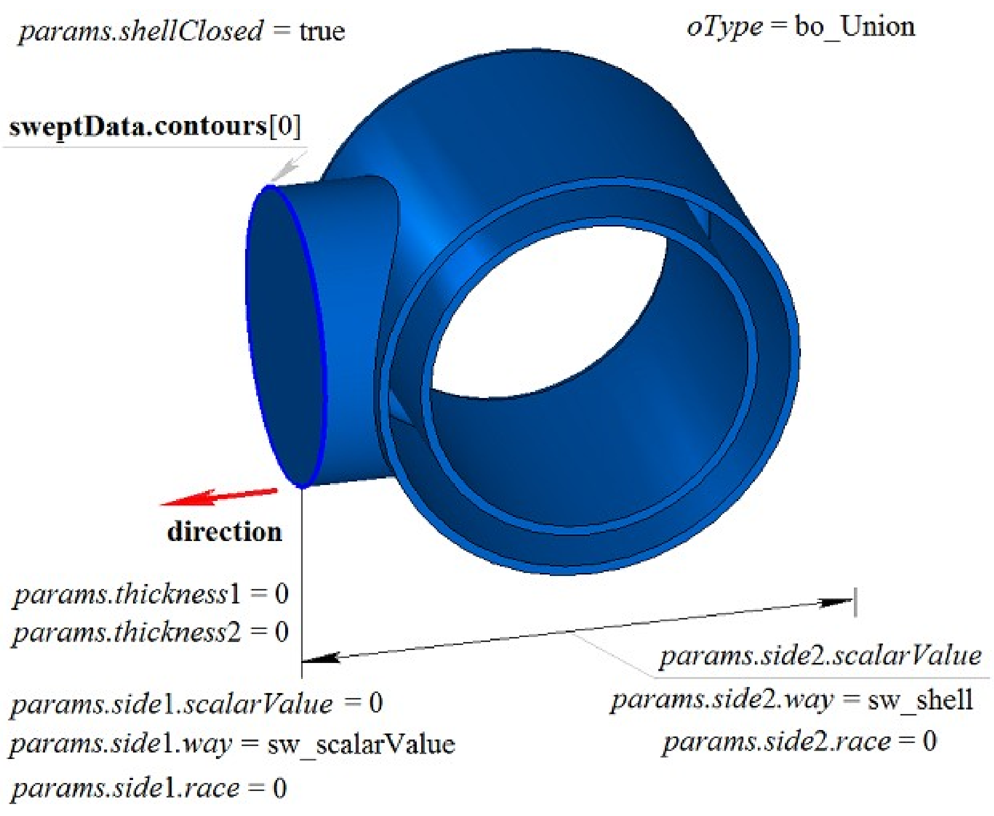


Fig. M.2.3.3.

In Fig. M.2.3.4, you can see the result of the Boolean operation that subtracts a solid received by extruding **sweptData** generating curve from **solid** solid according to **direction** shown in Figure M.2.3.1.

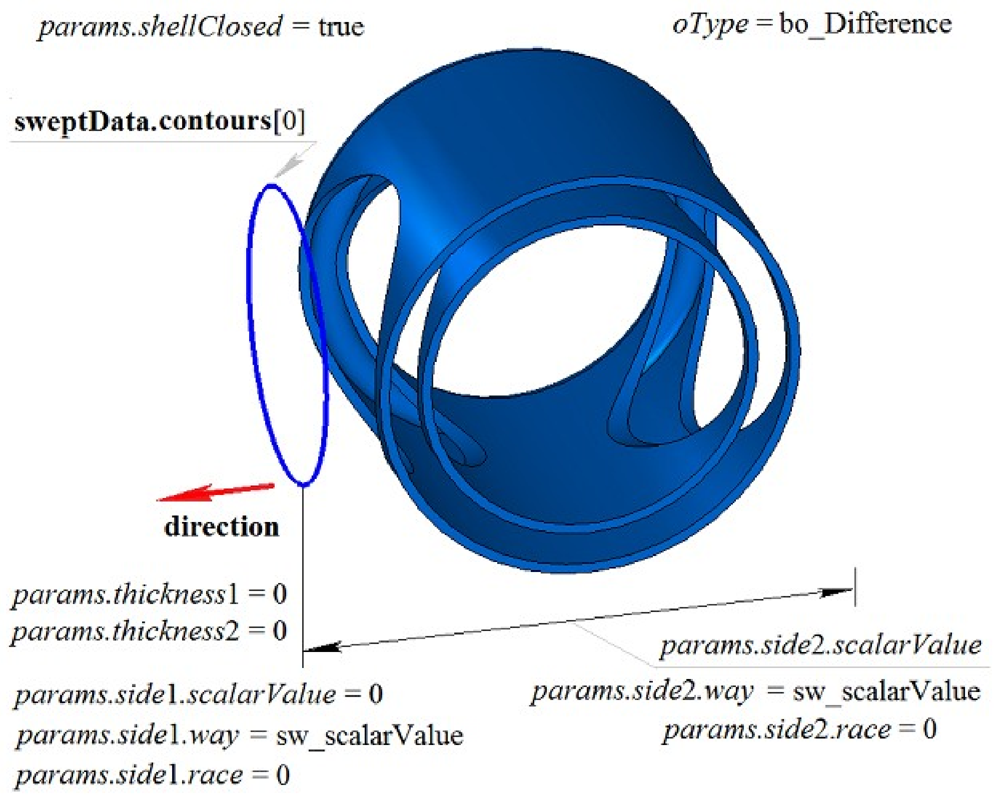


Fig. M.2.3.4.

In Fig. M.2.3.5, you can see the result of Boolean operation that subtracts a solid received by extrusion of **sweptData** generating curve from **solid** solid according to **direction** shown in Figure M.2.3.1. Generating curve was extruded in backward direction without a slope with «To Nearest Objects» option (*params.side*2.*way*=sw_shell).

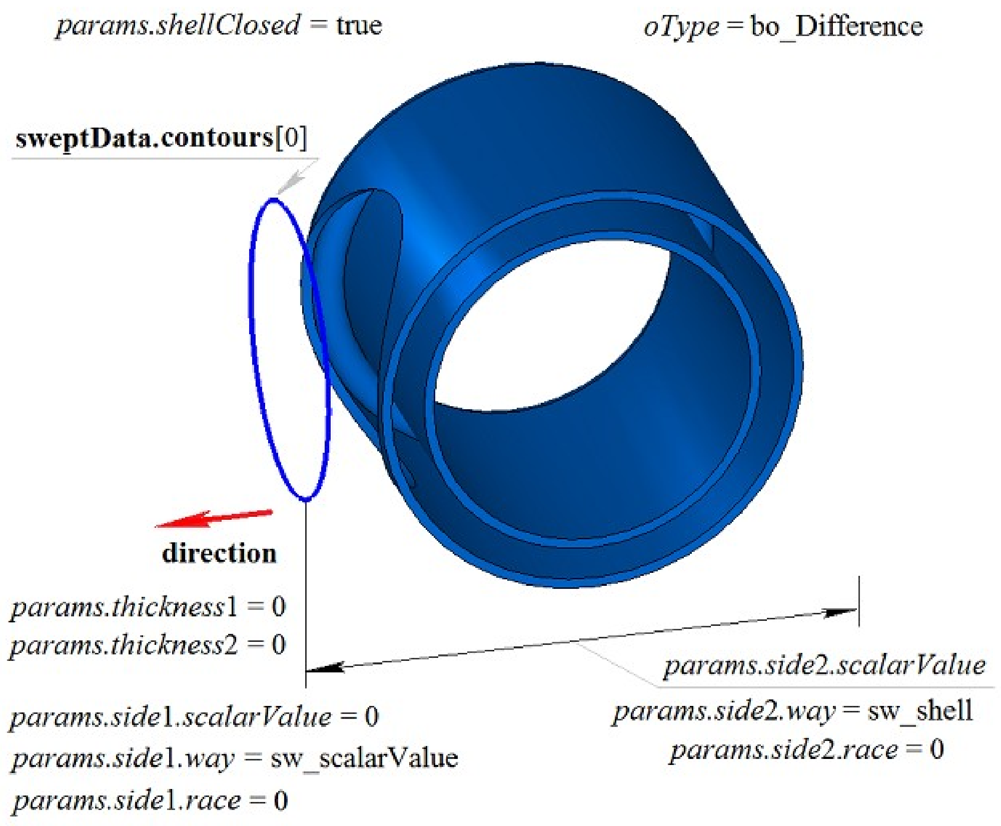

params.shellClosed = true
oType = bo_Difference
sweptData.contours[0]
direction
*params.thickness*1 = 0
*params.thickness*2 = 0
params.side2.scalarValue
*params.side*1*.scalarValue* = 0
*params.side*1*.way* = sw_scalarValue
*params.side*1*.race* = 0
*params.side*2*.way* = sw_shell
*params.side*2*.race* = 0

Fig. M.2.3.5.

In Fig. M.2.3.6, you can see the result of Boolean operation that intersects **solid** solid and a solid received by extrusion of **sweptData** generating curve according to **direction** shown in Figure M.2.3.1.

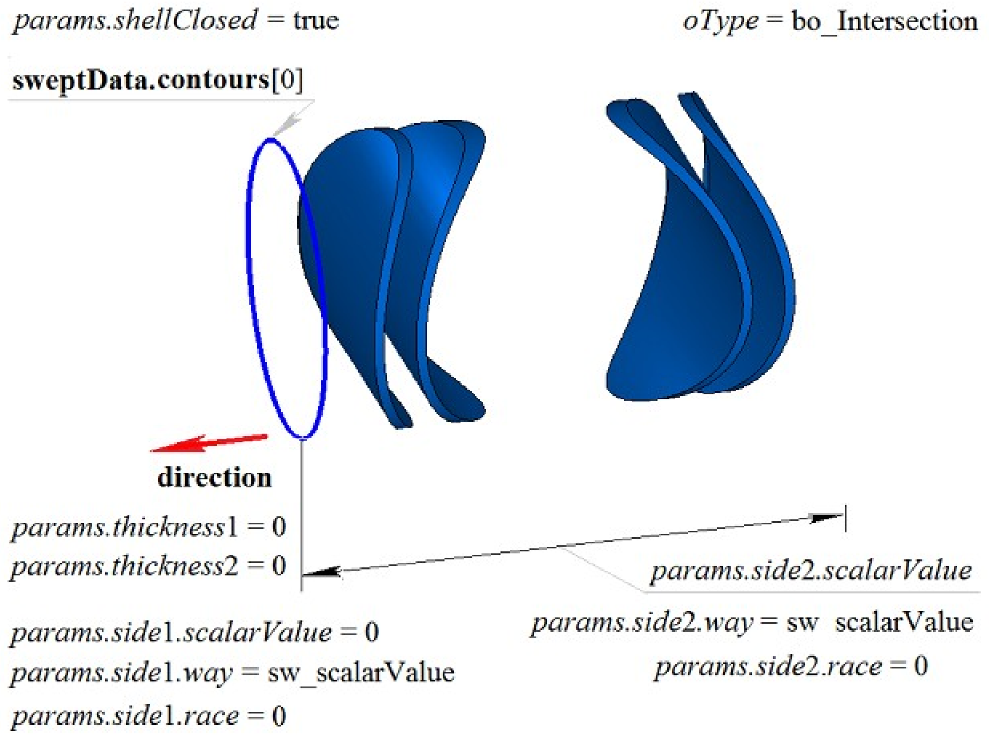


Fig. M.2.3.6.

In Fig. M.2.3.7, you can see the result of the Boolean operation that intersects **solid** solid and a solid received by extruding **sweptData** generating curve according to **direction** shown in Figure M.2.3.1. The generating curve was extruded in the backward direction without a slope with«To Nearest Objects» option.

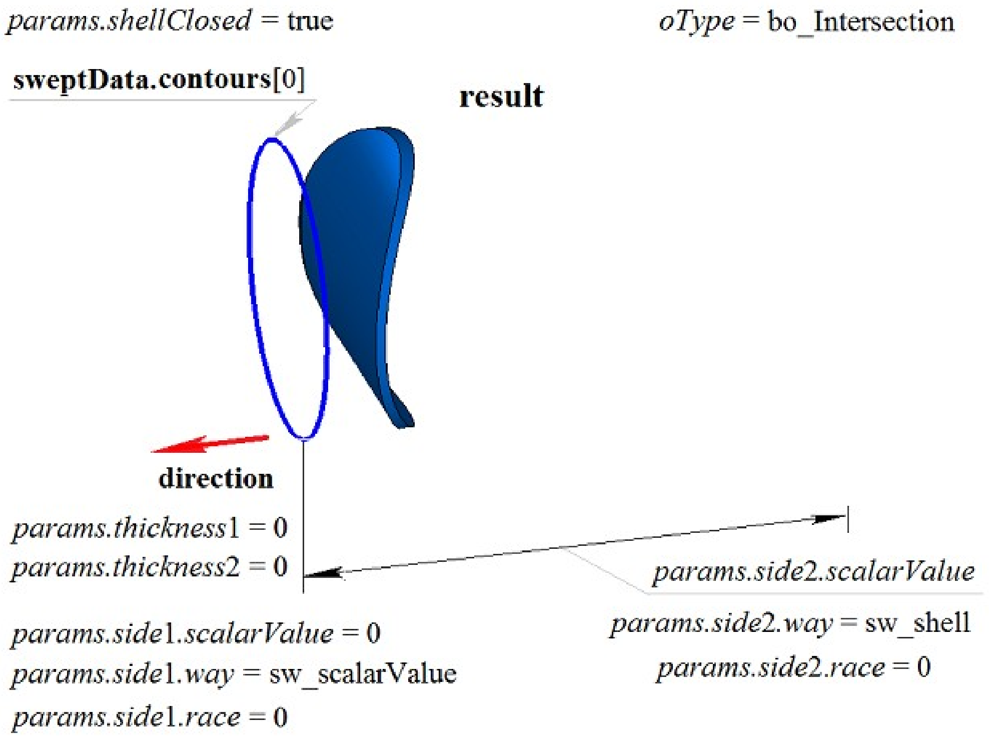


Fig. M.2.3.7.

ExtrusionResult method adds MbExtrusionSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbExtrusionSolid constructor is declared in cr_extrusion_solid.h file.

test.exe test application executes Boolean operations on the solid recieved by extruding a solid using New ->Solid -> By Gluing to Solid -> By Extruding Curve, New ->Solid -> Cut from Solid -> By Extruding Curve, New ->Solid -> Intersection with Other Solid -> By Extruding Curve menu commands.

M.2.4. Boolean Operation on Revolution Solid

Method
MbResultType
RevolutionResult (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbSweptData & **sweptData**,
const MbAxis3D & **axis**,
RevolutionValues & *params*,
OperationType *oType*,
const MbSNameMaker & names,
PArray<MbSNameMaker> & snames,
MbSolid *& **result**)
constructs a rotation solid and executes a Boolean operation of determined solid with constructed solid.

Input parameters of the method are as follows:

- **solid** is a solid given for Boolean operation,
- *sameShell* is copying version for the given solid,
- **sweptData** contains data on generating curves for construction of extruded solid,
- **axis** is a rotation axis,
- *params* are construction parameters,
- *oType* is Boolean operation type: bo_Union means merging of the solids,
  bo_Intersect means intersection of the solids,
  bo_Difference means subtraction of the solids,
- names is operation namer,
- snames are namers of rotation solid faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.The method is declared in action_solid.h file.

This method executes successive merging of the following two methods: **RevolutionSolid** method that constructs a solid by extruding **sweptData** curves according to *params* parameters in **direction**, and **BooleanSolid** method that executes *oType* Boolean operation on **solid** solid that was constructed in the previous step. **RevolutionSolid** method is described in item M.1.4. Constructing a Revolution Solid, and **BooleanSolid** method is described in item M.2.1. Boolean Operation on Solids. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

oType (OperationType) parameter defines Boolean operation type; it takes one of the following three values: bo_Union, bo_Intersect, bo_Difference. If *oType*=bo_Union, then the method merges **solid** solid and a revolution solid; if *oType*=bo_Intersect, then the method intersects **solid** solid and a revolution solid; if *oType*=bo_Difference, then the method subtracts a revolution solid from **solid** solid. names and snames parameters provide naming of the faces for newly constructed solid.

When a solid is constructed by rotating curves, **RevolutionResult** method provides the same possibilities as **RevolutionSolid** method: rotated curves may be located in a plane (Figure M.1.4.2), in a curved surface (Figure M.1.4.9) or in space (Figure M.1.4.16); it may be rotated in forward direction, backward direction or both directions; newly constructed solid may completely fill closed curves (Figure M.1.4.10) or it may have a thin wall (Figure M.1.4.11). We shall not repeat the description of all features of the method, we shall rather focus on some features associated with Boolean operations.

In Fig. M.2.4.1, you can see **solid** solid, generating curve included in **sweptData** data and **axis** rotation axis.

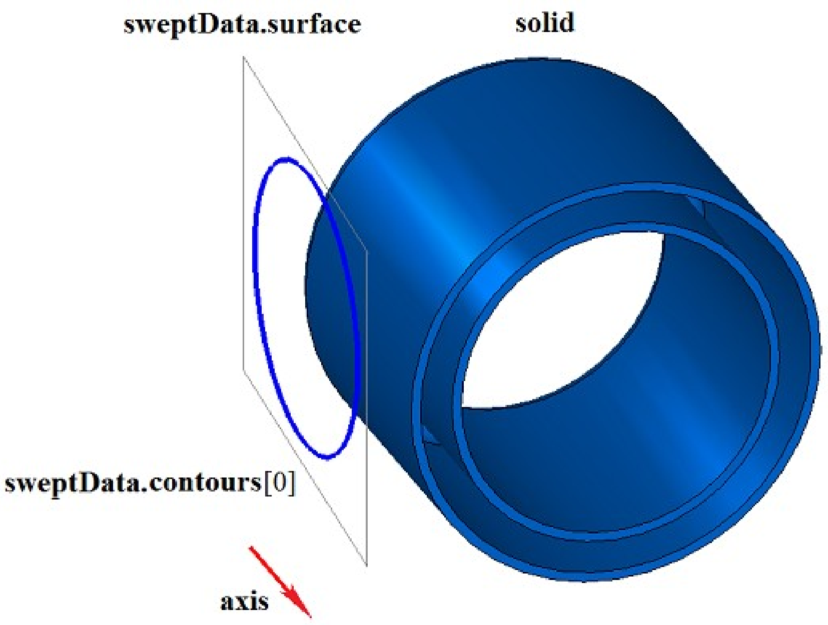


Fig. M.2.4.1.

In Fig. M.2.4.2, you can see the result of Boolean operation that merges **solid** solid and a thin-walled solid received by rotation of **sweptData** curve around **axis** shown in Figure M.2.4.1.

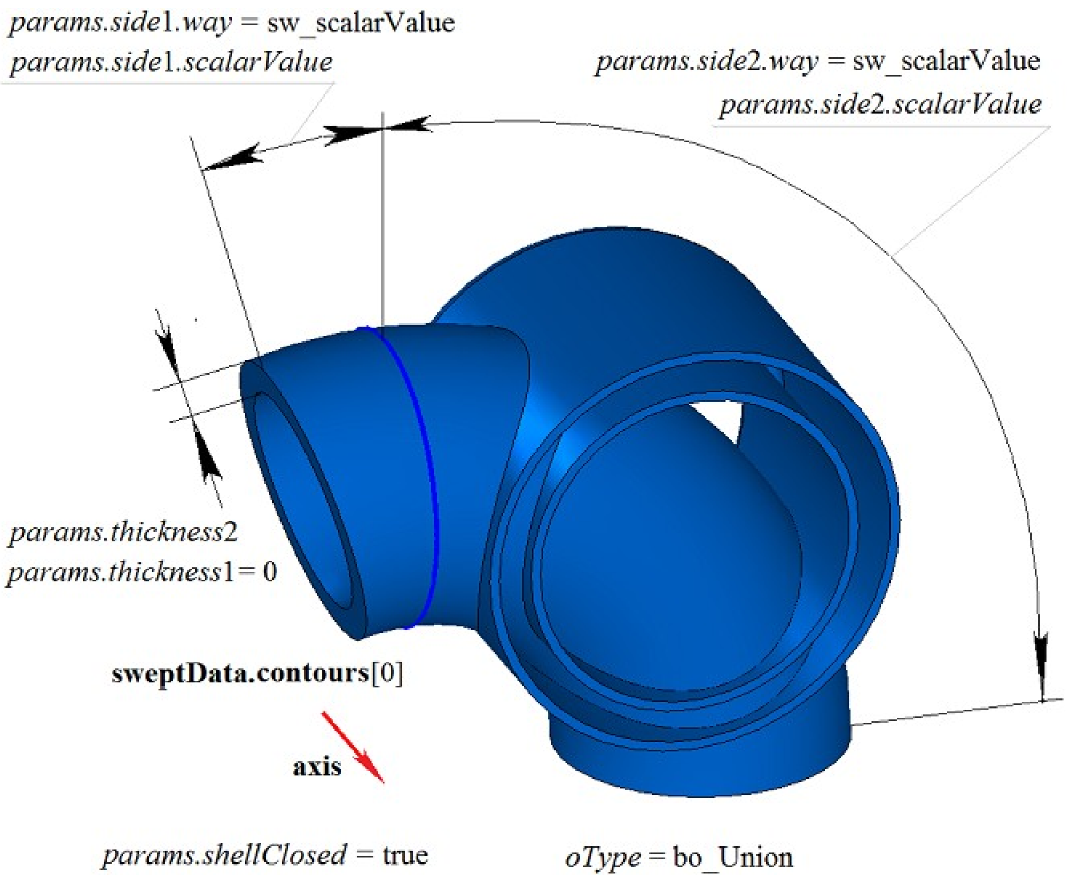
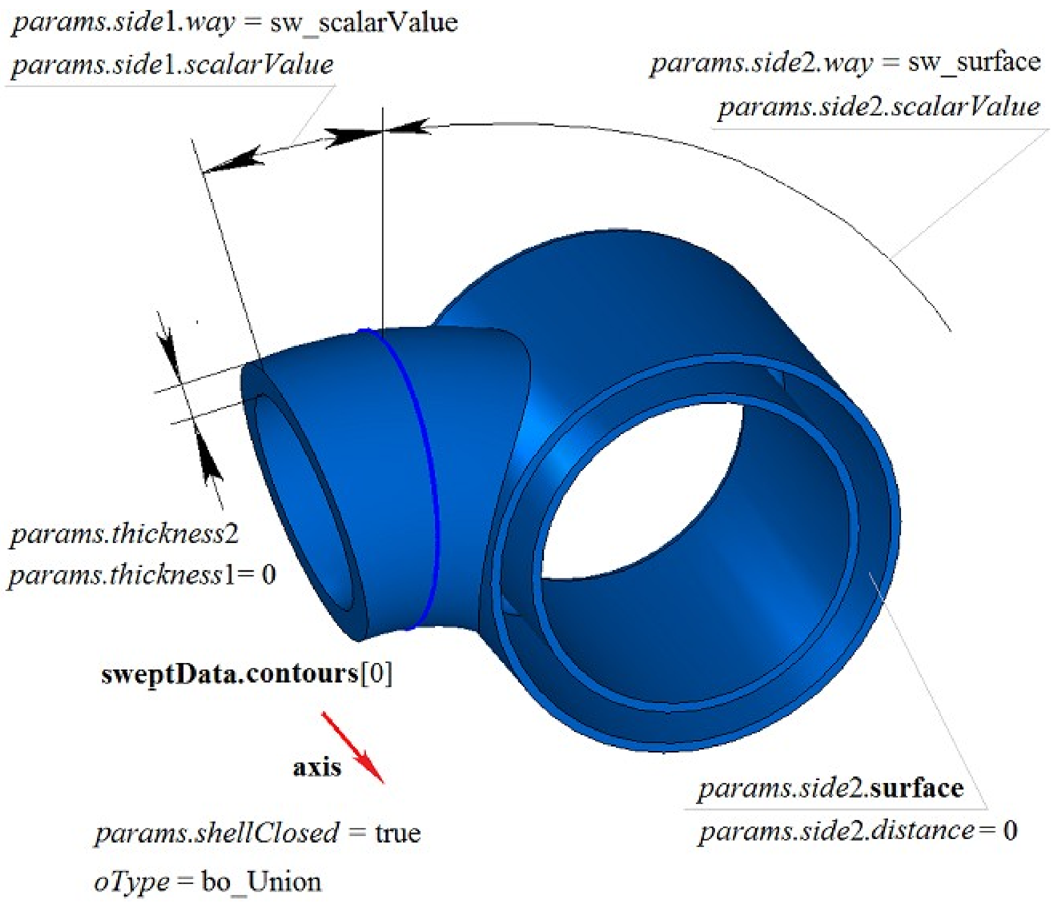


Fig. M.2.4.2.

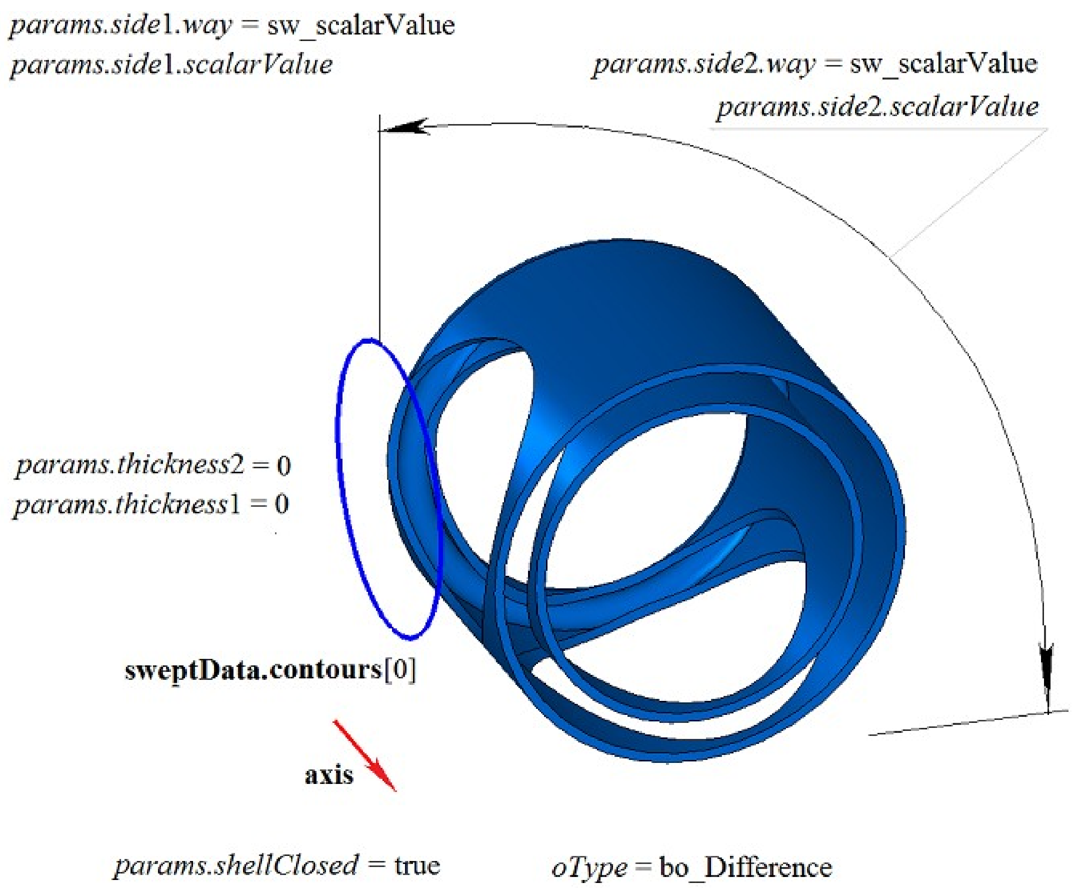
In Fig. M.2.4.3, you can see the result of Boolean operation that merges **solid** solid and a thin-walled solid received by rotation of **sweptData** curve around **axis** shown in Figure M.2.4.1. Generating curve is rotated in backward direction with To Surface option (*params.side*2.*way*=sw_surface).



*params.side*1.*way* = sw_scalarValue
*params.side*1.*scalarValue*
*params.side*2.*way* = sw_surface
*params.side*2.*scalarValue*
*params.thickness*2
*params.thickness*1= 0
sweptData.contours[0]
axis
*params.side*2.**surface**
*params.side*2.*distance*= 0
params.shellClosed = true
oType = bo_Union

Fig. M.2.4.3.

In Fig. M.2.4.4, you can see the result of Boolean operation that subtracts a solid received by rotating **sweptData** generating curve around **axis** from **solid** solid shown in Figure M.2.4.1.



*params.side*1.*way* = sw_scalarValue
*params.side*1.*scalarValue*
*params.side*2.*way* = sw_scalarValue
*params.side*2.*scalarValue*
*params.thickness*2 = 0
*params.thickness*1 = 0
sweptData.contours[0]
axis
params.shellClosed = true
oType = bo_Difference

Fig. M.2.4.4.

In Fig. M.2.4.5, you can see the result of Boolean operation that subtracts a solid received by rotating **sweptData** generating curve around **axis** from **solid** solid shown in Figure M.2.4.1. Generating curve is rotated in backward direction with To Surface option (*params.side*2.*way*=sw_surface).

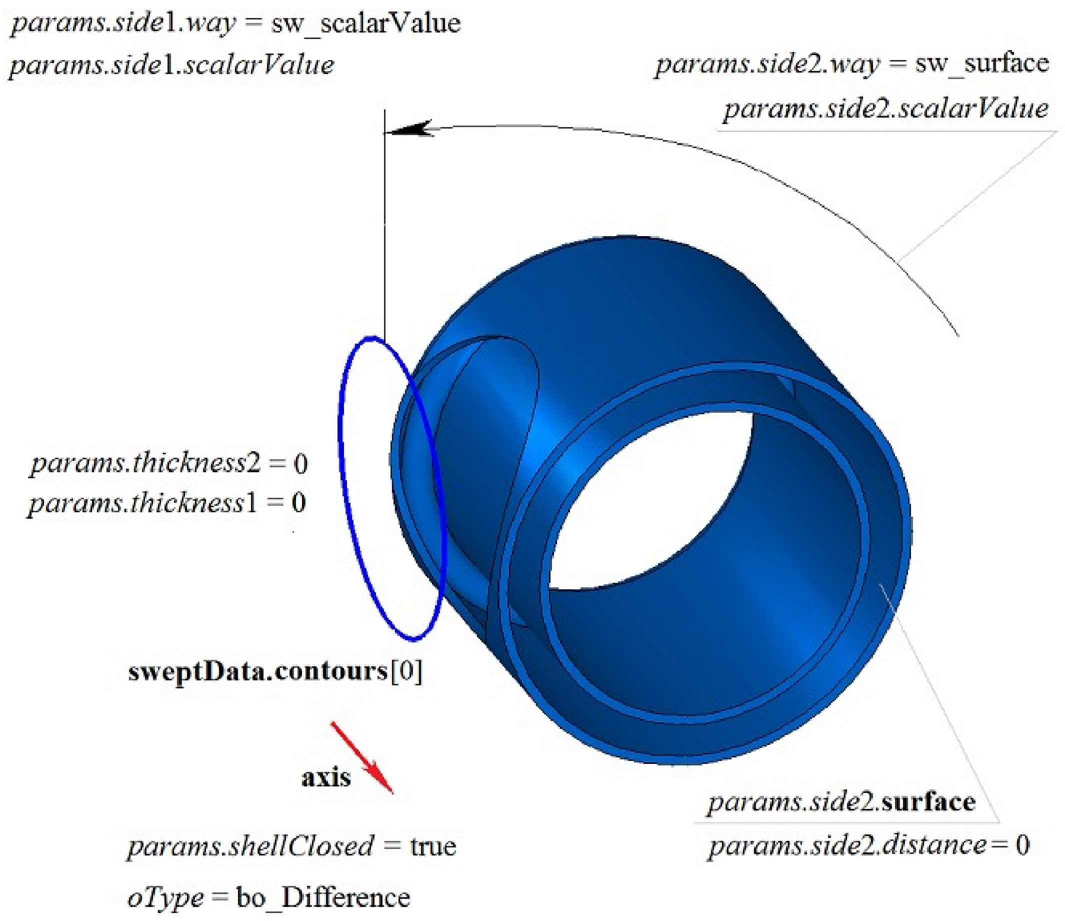


Fig. M.2.4.5.

In Fig. M.2.4.6, you can see the result of Boolean operation that intersects **solid** solid and a solid received by rotating **sweptData** generating curve around **axis** shown in Figure M.2.4.1.

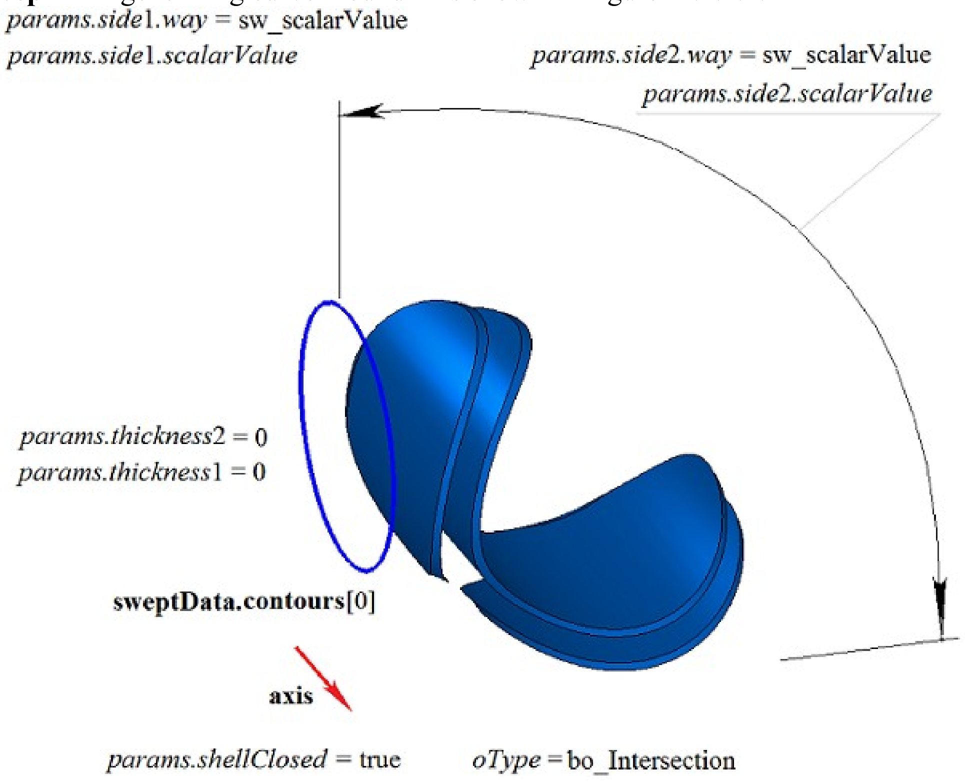


Fig. M.2.4.6.

In Fig. M.2.4.7, you can see the result of Boolean operation that intersects **solid** solid and a solid received by rotating **sweptData** generating curve around **axis** shown in Figure M.2.4.1. Generating curve is rotated in backward direction with To Surface option (*params.side*2.*way*=sw_surface).

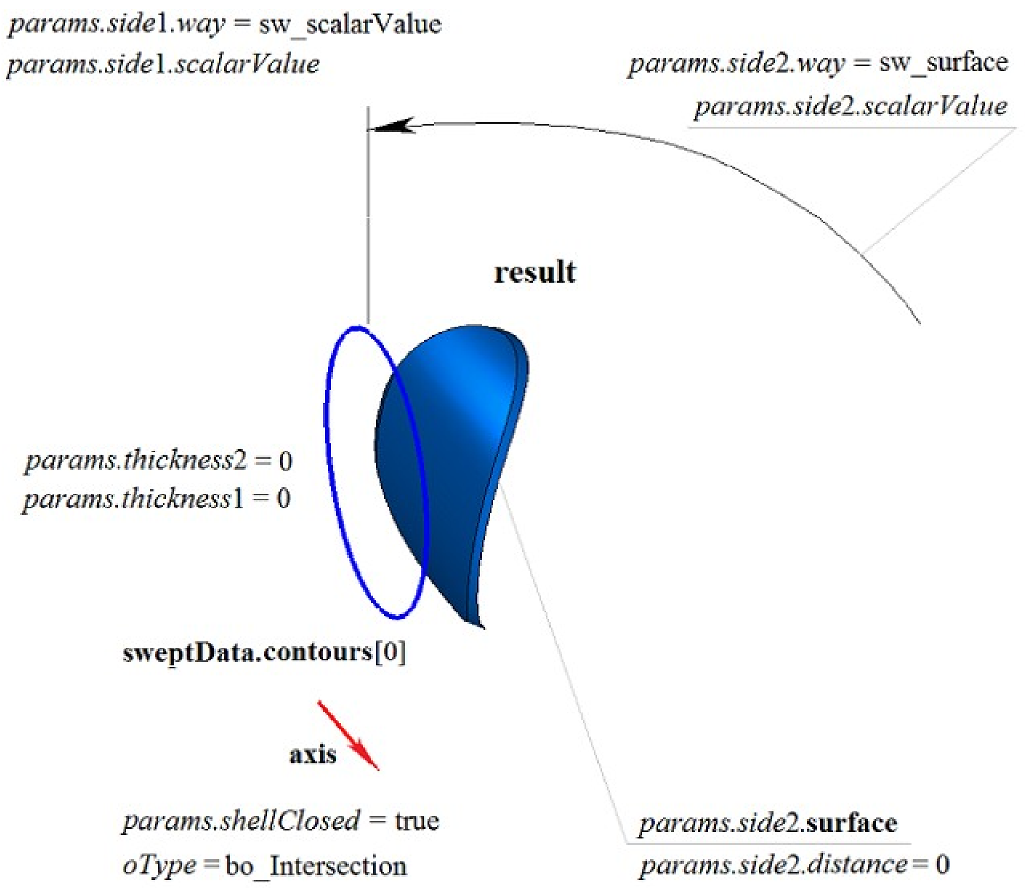


Fig. M.2.4.7.

RevolutionResult method adds MbRevolutionSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbRevolutionSolid constructor is declared in cr_revolution_solid file.

test.exe test application executes a Boolean operation on constructed solid using New ->Solid -> Attach to Other Solid -> Curve Rotation, New ->Solid -> Cut from Other Solid -> Curve Rotation, New ->Solid -> Intersection with Other Solid -> Curve Rotation menu commands.

M.2.5. Boolean Operation on Swept Solid

Method
MbResultType
EvolutionResult (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbSweptData & **sweptData**,
const MbCurve3D & **spine**,
EvolutionValues & *params*,
OperationType *oType*,
const MbSNameMaker & names,
PArray<MbSNameMaker> & cnames,
const MbSNameMaker & snames,
MbSolid *& **result**)
constructs a swept solid and executes a Boolean operation on newly given solid with newly constructed

solid.

Input parameters of the method are as follows:

- **solid** is a solid given for Boolean operation,
- *sameShell* is copying version for the given solid,
- **sweptData** contains data on generating curves for construction of extruded solid,
- **spine** is guiding curve,
- *params* are construction parameters,
- *oType* is Boolean operation type: bo_Union means merging of the solids,
  bo_Intersect means intersection of the solids,
  bo_Difference means subtraction of the solids,
- names is face namer,
- cnames are namers of swept solid faces,
- snames is guiding line namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method executes successive merging of the following two methods: **EvolutionSolid** method that constructs a solid by moving **sweptData** curves along **spine** guiding curve using set *params* parameters and **BooleanSolid** method that executes *oType* Bollean operation on **solid** solid that was constructed in the previous step. **EvolutionSolid** method is described in item M.1.5. Constructing a Swept Solid, and **BooleanSolid** method is described in item M.2.1. Boolean Operation on Solids. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item .O.7.9. Copying a Set of Faces

*oType (*OperationType) parameter defines Boolean operation type; it takes one of the following three values: bo_Union, bo_Intersect, bo_Difference. If *oType*=bo_Union, then the method merges **solid** solid and the swept solid; if *oType*=bo_Intersect, then the method intersects **solid** solid and the swept solid; if *oType*=bo_Difference, then the method subtracts the swept solid from **solid** solid. names, cnames and snames parameters provide face naming for newly constructed solid.

When **EvolutionResult** method constructs solids by moving curves, it provides the same possibilities as **EvolutionSolid** method: guiding curves may be located in a plane (Figure M.1.5.2), in a curved surface (Figure M.1.5.8), or in space (Figure M.1.5.16); the solid may completely fill closed curves (Figure M.1.5.9) or it may have a thin wall (Figure M.1.5.10). We shall not repeat the description of all features of the method, we shall rather focus on some features associated with Boolean operations.

In Fig. M.2.5.1, you can see **solid** solid, a generating curve included in **sweptData** data and **spine** guiding curve.

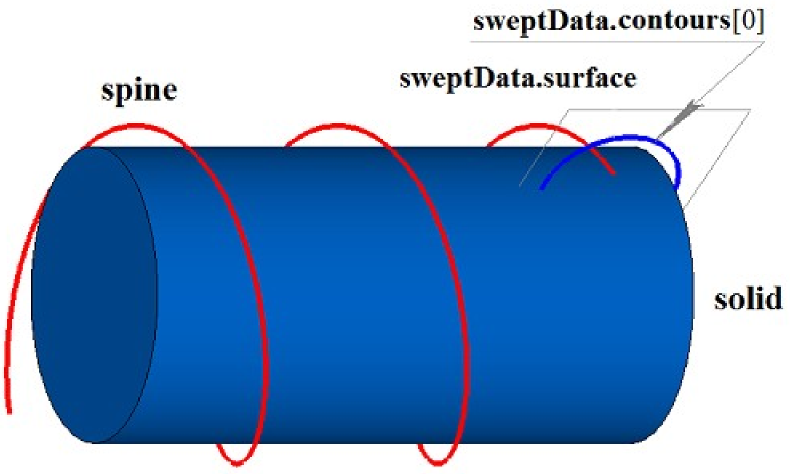


Fig. M.2.5.1.

In Fig. M.2.5.2, you can see the result of the Boolean operation that merges **solid** and the solid received by moving **sweptData** generating curve along **spine** guiding curve shown in Figure M.2.5.1.

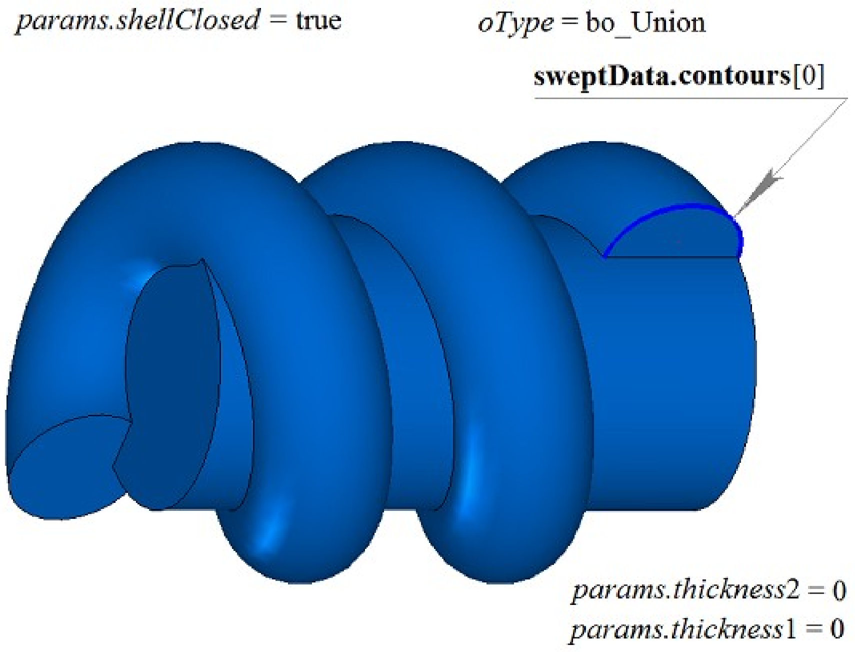


Fig. M.2.5.2.

In Fig. M.2.5.3, you can see the result of Boolean operation that subtracts the solid received by moving **sweptData** generating curve along **spine** guiding curve from **solid** solid shown in Figure M.2.5.1.

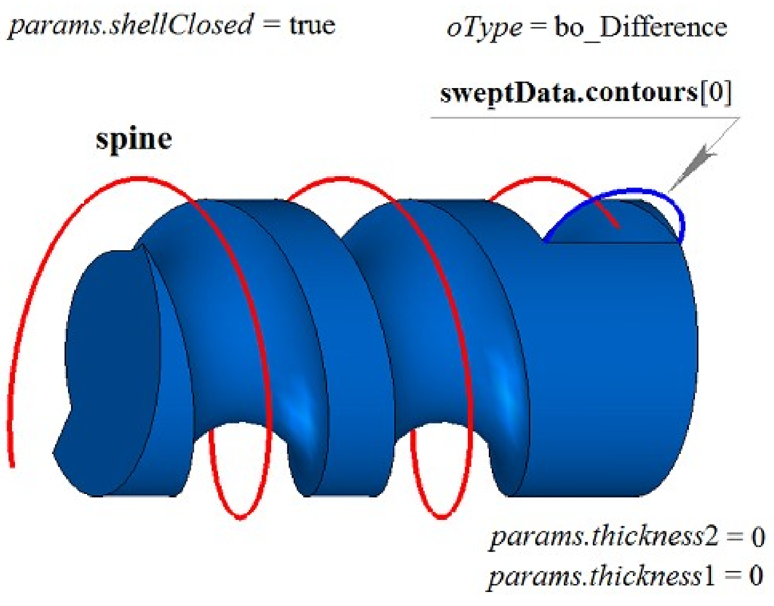


Fig. M.2.5.3.

In Fig. M.2.5.4, you can see the result of Boolean operation that merges **solid** solid and the solid received by moving **sweptData** generating curve along **spine** guiding curve shown in Figure M.2.5.1.

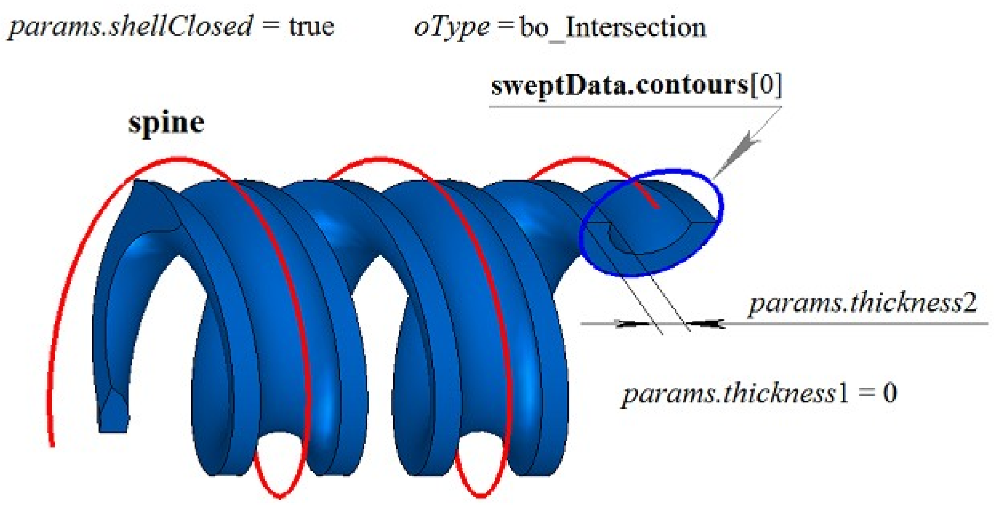


Fig. M.2.5.4.

EvolutionResult method adds MbEvolutionSolid constructor to the log of newly constructed solid that contains all data required to execute the operation. MbEvolutionSolid constructor is declared in cr_evolution_solid.h file.

test.exe test application executes Boolean operations on constructed swept solid using New ->Solid -> Attach to Other Solid -> By Moving Curve, New ->Solid -> Cut from Other Solid -> By Moving a Curve, New ->Solid -> By Intersection with Other Solid -> By Moving a Curve menu commands.

M.2.6. Boolean Operation with a Solid Constructed on Base of Flat Sections

Method

MbResultType
LoftedResult (MbSolid & **solid**,
MbeCopyMode *sameShell*,
SArray<MbPlacement3D> & **places**,
RPArray<MbContour> & **contours**,
const MbCurve3D * **spine**,
LoftedValues & *params*,
OperationType *oType*,
Sarray<MbCartPoint3D> * **points**,
const MbSNameMaker & names,
PArray<MbSNameMaker> & snames,
MbSolid *& **result**)

constructs a solid based on flat sections and executes Boolean operation on the specified solid with newly constructed solid.

Input parameters of the method are as follows:

- **solid** is a solid given for Boolean operation,
- *sameShell* is copying version for the given solid,
- **places** is a set of local coordinate systems for generating contours,
- **contours** is a set of generating contours,
- **spine** is a guiding curve (it may be missing),
- *params* are construction parameters,
- *oType* is Boolean operation type: bo_Union means merging of the solids,
  bo_Intersect means intersection of the solids,
  bo_Difference means subtraction of the solids,
- **points** is a set of control points (it may be missing),
- names is face namer,
- snames are namers of generating contours.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method executes successive merging of the following two methods: **LoftedSolid** method that constructs a solid based on **contours** flat sections at **places** planes taking into account *params* parameters and **BooleanSolid** method that executes *oType* Boolean operation of **solid** solid that was constructed in the previous step. **LoftedSolid** method is described in item M.1.6. Constructing a Solid by Flat Sections, and **BooleanSolid** method is described in item M.2.1. Boolean Operation on Solids. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

oType (OperationType) parameter defines Boolean operation type; it takes one of the following three values: bo_Union, bo_Intersect, bo_Difference. If *oType*=bo_Union then the method merges **solid** solid and the swept solid; if *oType*=bo_Intersect, then the method intersects **solid** solid and the swept solid; if *oType*=bo_Difference, then the method subtracts the swept solid from **solid** solid. names, cnames and snames parameters provide face naming for newly constructed solid.

When **LoftedResult** method constructs solids based on flat sections, it provides the same possibilities as **LoftedSolid** method: constructed solid may be built as non-closed (Figure M.1.6.3) or as closed one (Figure M.1.6.4); the solid may have various shapes near the ends (Figure M.1.6.5 and Figure M.1.6.6); the solid may completely fill the closed curves (Figure M.1.6.3) or it may have a thin wall (Figure M.1.6.6); solid shape in sections can be controlled by a guiding line (Figure M.1.6.15 and Figure M.1.6.16). We shall not repeat the description of all features of the method, we shall rather focus on some features associated with Boolean operations.

In Fig. M.2.6.1, you can see **solid** solid and closed guiding curves.

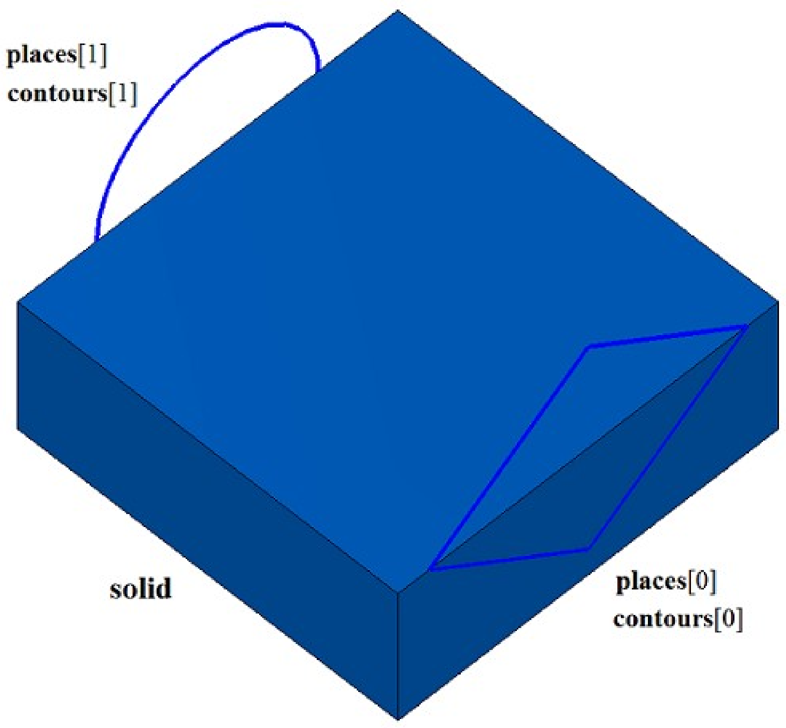


Fig. M.2.6.1.

In Fig. M.2.6.2, you can see the result of Boolean operation that merges **solid** solid and the solid that was constructed based on **contours** flat sections shown in Figure M.2.6.1.

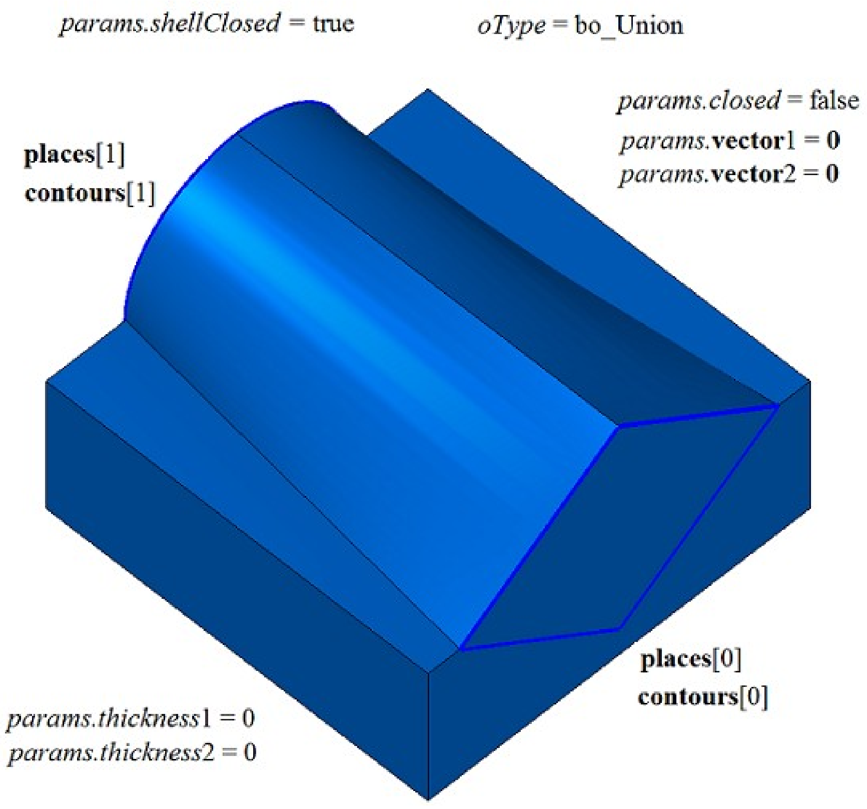


Fig. M.2.6.2.

In Fig. M.2.6.3, you can see the result of Boolean operation that subtracts the solid that was constructed based on **contours** flat sections from **solid** solid shown in Figure M.2.6.1.

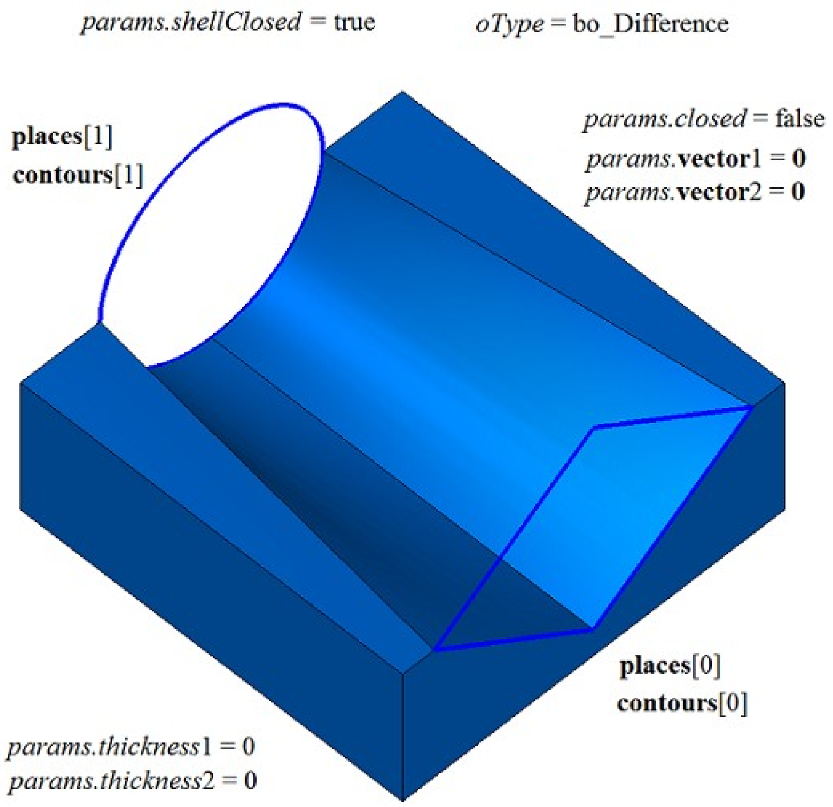


Fig. M.2.6.3.

In Fig. M.2.6.4, you can see the result of the Boolean operation that merges **solid** solid and the solid that was constructed based on **contours** flat sections shown in Figure M.2.6.1.

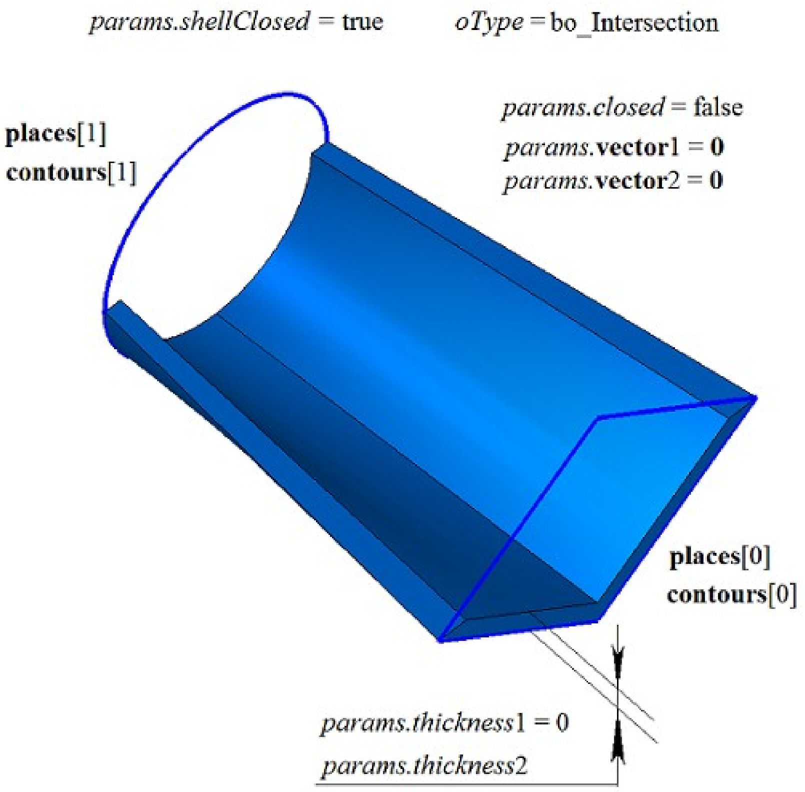


Fig. M.2.6.4.

LoftedResult method adds MbLoftedSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbLoftedSolid constructor is declared in cr_lofted_solid.h file.

test.exe test application executes a Boolean operation with the solid constructed based on flat sections using New ->Solid -> Attach to Other Solid -> By Sections, New ->Solid -> Attach to Other Solid -> By Sections and Generating Curve, New ->Solid -> Cut from Other Solid -> By Sections, New ->Solid -> Cut from Other Solid -> By Sections and Generating Curve, New ->Solid -> Intersection with Other Solid -> By Sections, New ->Solid -> Intersection with Other Solid -> By Sections menu commands.

M.2.7. Cutting a Solid by a Surface

Method
MbResultType
SolidCutting (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbSurface & **surface**,
int *part*,
const MbSNameMaker & names,
bool *closed*,
MbSolid *& **result**)
cuts off part of the solid by a surface that intersects it.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **surface** is the intersecting surface,
- *part* is a part of the solid that should be kept:
  if *part* = +1 then the part of the solid above the surface should be kept,
  if *part* = 0 then all parts of the solid should be kept,
  if *part* = –1 then the part of the solid under the surface should be kept,
- names is cut face namer,
- *closed* is a flag indicating whether the solid is closed in the operation:
  true means that the solid is considered to be closed,
  false means that the solid is considered to be non-closed.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

The method constructs a non-closed shell with one face based on cutting **surface** and it executes a Boolean operation that intersects **solid** original solid with a non-closed shell. To execute the operation, the cutting surface should fully intersect the original solid. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

part parameter defines the part of **solid** original solid to be kept: if *part*=+1, then the part located above the surface will be kept (on the other side, which is directed normal to the surface); if *part*=–1, then the part located under the surface will be kept; if part=0, then the both parts of the solid will be kept. names parameter is used to name the faces of the constructed solid. *closed* parameter defines whether **solid** original solid is closed or non-closed.

If *closed*=true, then the operation is executed on a set of points inside the solid and on its surface. If *closed*=false, then the operation is executed on a set of points located on solid surface.

In Fig. M.2.7.1, you can see **solid** original solid and cutting surface.

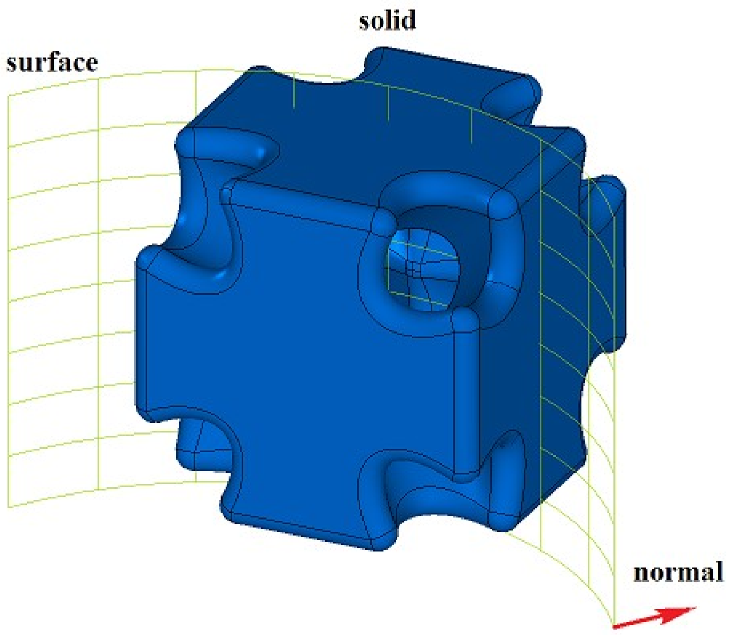


Fig. M.2.7.1.

In Fig. M.2.7.2, you can see **result** constructed solid if *part*=+1 and *closed*=true. In Fig. M.2.7.3, you can see **result** constructed solid if *part*=–1 and *closed*=true.

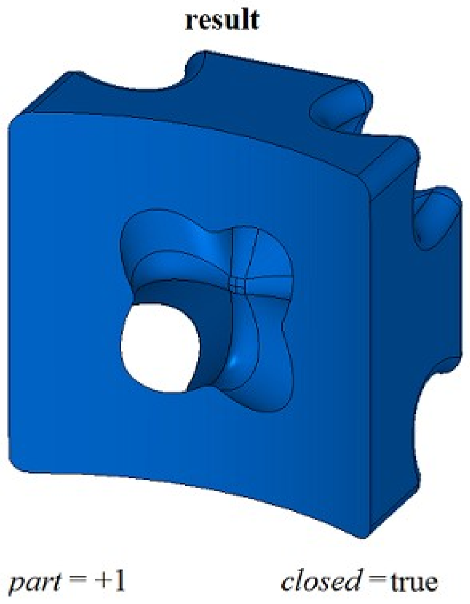


Fig. M.2.7.2.

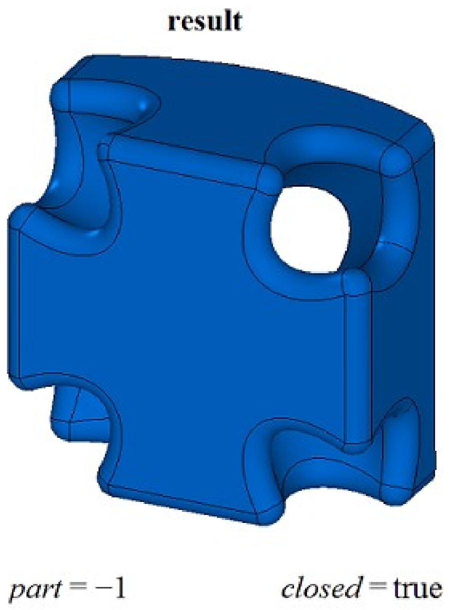


Fig. M.2.7.3.

In Fig. M.2.7.4, you can see **result** constructed solid if *part*=+1 and *closed*=false.

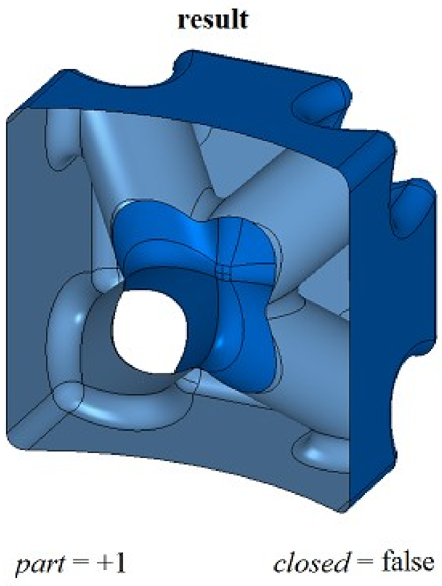


Fig. M.2.7.4.

If *part*=0, then method constructs **result** solid which contains all cutted parts of initial solid. Method **DetachParts** or **CreateParts** can to detach part of **result** solid. Methods **DetachParts** and **CreateParts** are described in item M.2.19. Divide a Solid to Disconnected Parts

Method
MbResultType
SolidCutting (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbSurface & **surface**,
const MbSNameMaker & names,
bool *closed*,
RPArray<MbSolid> & **result**)
constructs all parts of initial solid if *part*=0.

Method has the same parameters besides *part*. In Fig. M.2.7.5, you can see **result** constructed solids if *closed*=true.

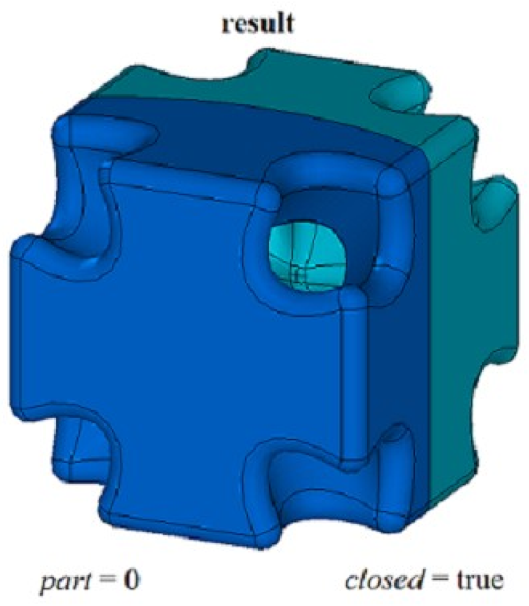


Fig. M.2.7.5.

SolidCutting methods adds MbCuttingSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbCuttingSolid constructor is declared in cr_cutting_solid.h file.

test.exe test application cuts solid with a surface using New ->Solid -> Based on Solid -> Cut with Surface and New -> Shell -> Based on Shell -> Cut with Surface menu commands.

M.2.8. Cutting a Solid by a Flat Contour

Method
MbResultType
SolidCutting (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **place**,
const MbContour & **contour**,
const MbVector3D & **direction**,
int *part*,
const MbSNameMaker & names,
bool *closed*,
MbSolid *& **result**)
cuts off a part of a solid (constructed by extruding a flat contour) by a surface that intersects the solid.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **place** is a local coordinate system of generating contour,
- **contour** is the generating contour,
- **direction** is extrusion direction of the generating contour,
- *part* is a part of the solid that should be kept:
  if *part* = +1, then the part of the solid above the surface should be kept,
  if *part* = 0, then all parts of the solid should be kept,
  if *part* = –1, then the part of the solid under the surface should be kept,
- names is cut face namer,
- *closed* is a flag indicating whether the solid is closed in the operation:
  true means that the solid is considered to be closed,

false means that the solid is considered to be non-closed.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method constructs a non-closed shell by extruding two-dimensional **contour** in **direction** of XY plane of **place** local coordinate system, and executes Boolean operation that intersects **solid** original solid with non-closed shell. If **direction** vector is equal to zero, then the contour is extruded along **place.axisZ** vector. To execute the operation, the cutting contour should fully intersect with the projection of the original solid in XY plane in **place** local coordinate system in the direction of extrusion vector. Contour extrusion length is calculated so that non-closed shell would fully intersect the original solid. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in itemO.7.9. Copying a Set of Faces.

part parameter defines the part of **solid** original solid to be kept: If *part*=+1, then the part of the solid to the right of the contour is to be kept; if *part*=–1, then the part of the solid to the left of the contour is to be kept (as viewed along the contour towards **place.axisZ**). names parameter is used to name the faces of the constructed solid. *closed* parameter defines whether **solid** original solid is closed or non-closed.

If *closed*=true, then the operation is executed on a set of points inside the solid and on its surface. If *closed*=false, then the operation is executed on a set of points located on solid surface.

In Fig. M.2.8.1, you can see **solid** original solid, **contour** cutting contour and XY plane of **place** local coordinate system.

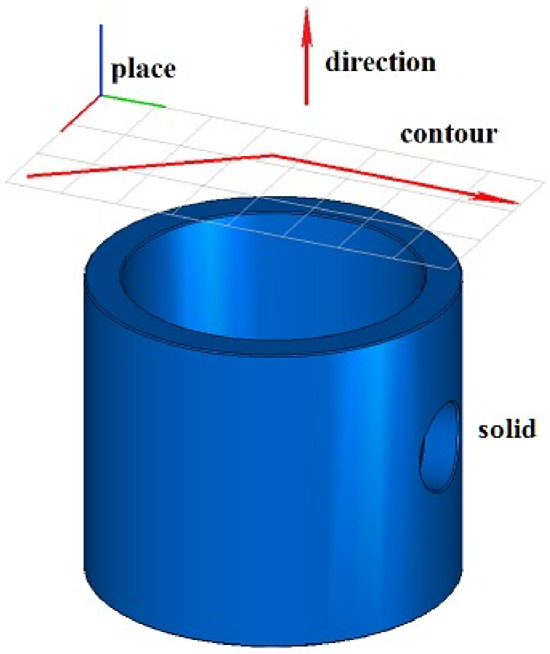
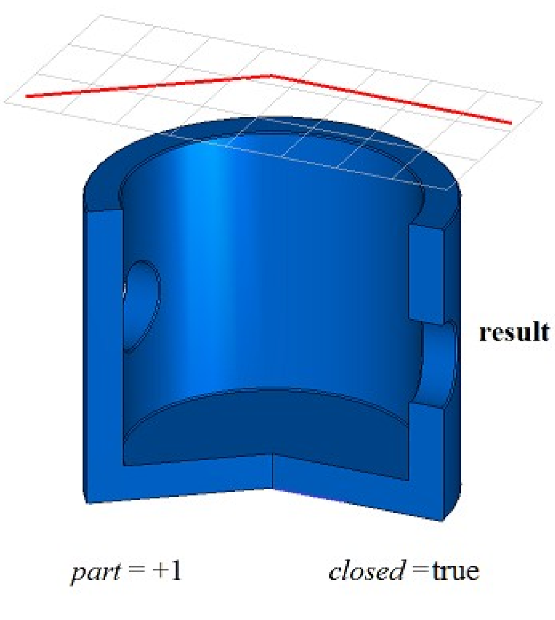


Fig. M.2.8.1.

In Fig. M.2.8.2, you can see **result** constructed solid when *part*=+1 and *closed*=true. In Fig. M.2.8.3, you can see **result** constructed solid, when *part*=–1 and *closed*=true.



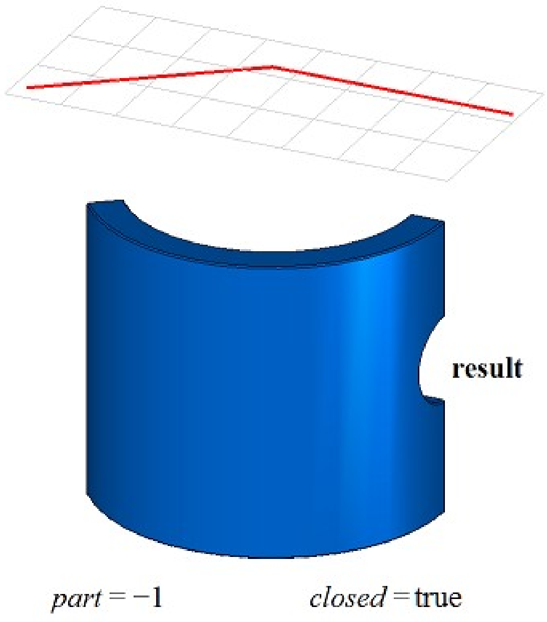


Fig. M.2.8.2. *Fig. M.2.8.3.*

In Fig. M.2.8.4, you can see **result** constructed solid, when *part*=+1 and *closed*=false.

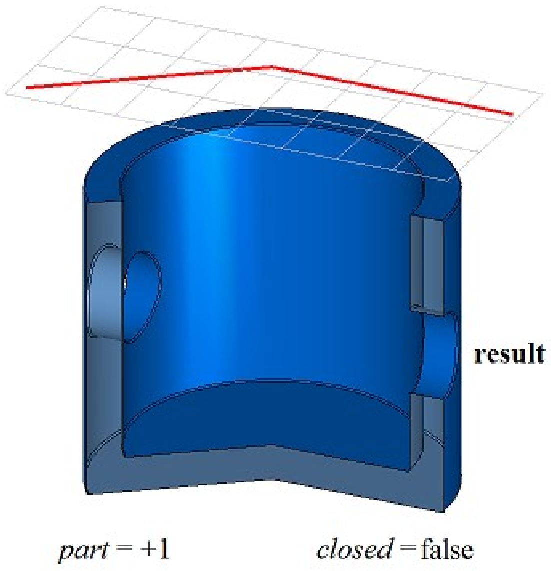


Fig. M.2.8.4.

If *part* = 0, then method constructs **result** solid which contains all cutted parts of initial solid. Method **DetachParts** or **CreateParts** can to detach part of **result** solid. Methods **DetachParts** and **CreateParts** are described in item M.2.19. Divide a Solid to Disconnected Parts.

Method
MbResultType
SolidCutting (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **place**,
const MbContour & **contour**,
const MbVector3D & **direction**,
const MbSNameMaker & names,

bool *closed*,
RPArray<MbSolid> & **result**)
constructs all parts of initial solid if *part*=0. Method has the same parameters besides *part*. In Fig. M.2.8.5, you can see **result** constructed solids if *closed*=true.

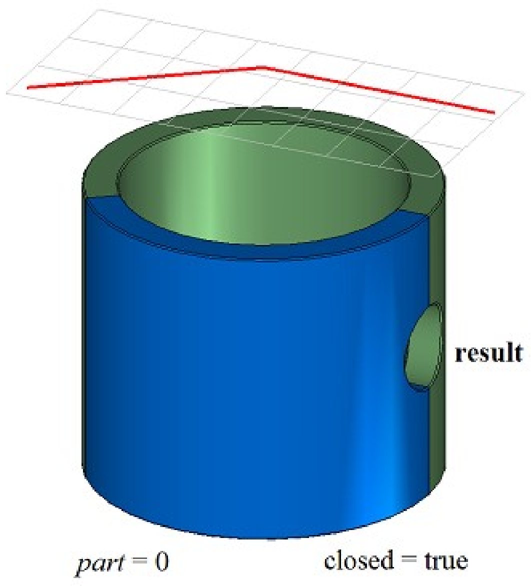


Fig. M.2.8.5.

SolidCutting methods adds MbCuttingSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbCuttingSolid constructor is declared in cr_cutting_solid.h file.

test.exe test application cuts a solid with a surface using New ->Solid -> Based on Solid -> Cut with Curve and New -> Shell -> Based on Shell -> Cut with Curve menu commands.

M.2.9. Constructing a Symmetrical Solid

Method
MbResultType
SymmetrySolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **place**,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a symmetrical solid with a given symmetry plane.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **place** is a local coordinate system, its XY plane is a symmetry plane,
- names is cut face namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

The method constructs a symmetrical solid with a specified symmetry plane as follows. **solid** original solid is cut by XY plane of **place** local coordinate system; the part of the original solid located below the cutting plane is taken; a mirrored copy of the selected part of the original solid is constructed and merged with the selected part of the original solid. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

In Fig. M.2.9.1, you can see **solid** original solid and **place** symmetry plane.

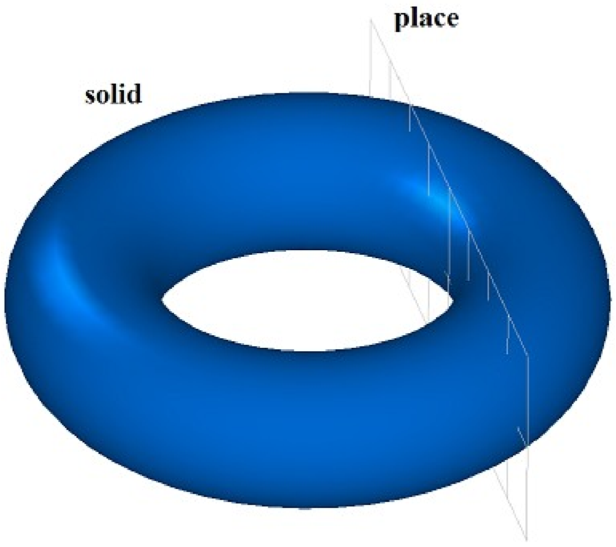


Fig. M.2.9.1.

In Fig. M.2.9.2, you can see **result** constructed solid.

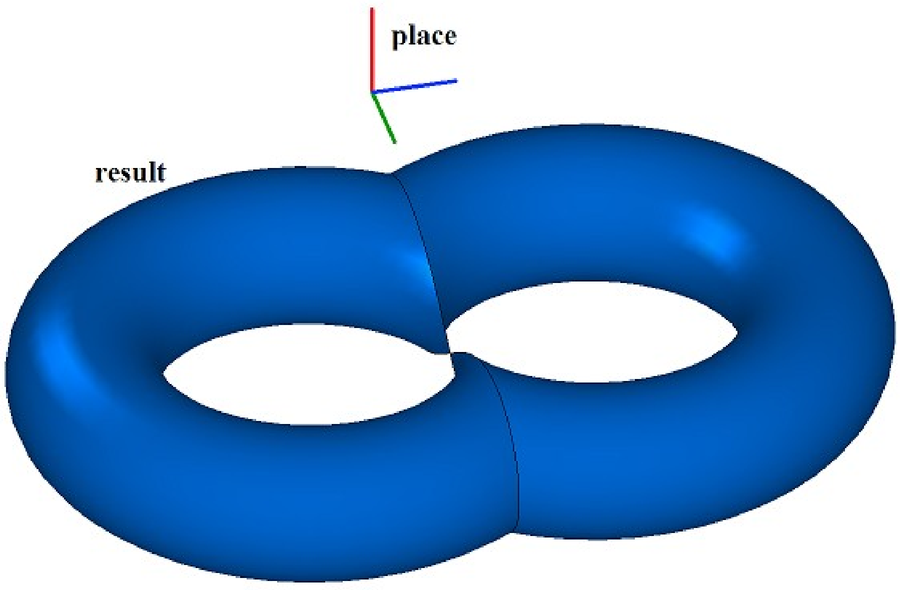


Fig. M.2.9.2.

In Fig. M.2.9.3, you can see **result** solid constructed for symmetry plane with an opposite normal.

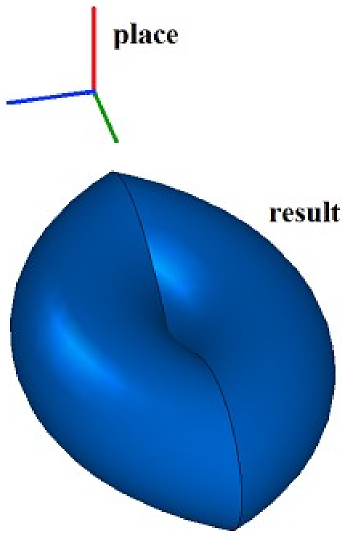


Fig. M.2.9.3.

If **solid** original solid does not touch XY plane of **place** local coordinate system, then the construction is not executed. In the latter case you can use **MirrorSolid** method to construct a symmetrical solid.

SymmetrySolid method adds MbSymmetrySolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbSymmetrySolid constructor is declared in cr_symmetry_solid.h file.

test.exe test application constructs a symmetrical solid using New ->Solid -> Based on Solid -> Symmetrical menu command.

M.2.10. Rounding-off Solid Edges

Method
MbResultType
FilletSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
RPArray<MbCurveEdge> & **edges**,
RPArray<MbFace> & **bounds**,
const SmoothValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
rounds off specified edges in a copy of the original solid.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **edges** is a set of rounded-off edges.
- **bounds** is a set of faces used to cut rounded-off edges (the set may be empty),
- *params* are construction parameters,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

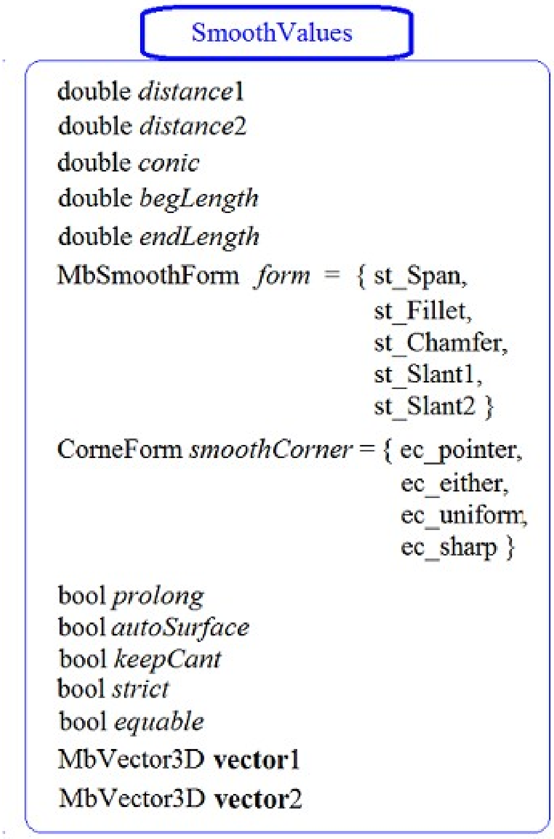
This method replaces specified edges of the original solid with rounded-off faces in order to ensure smooth mating of adjacent faces of specified edges. When edges are rounded-off, the mating faces may have a shape of circle arc, ellipse, hyperbola or parabola in a cross-section.

solid parameter contains the original solid, its edges should be processed. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** resulting solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

edges parameter contains processed edges of **solid** solid. **bounds** parameter contains faces of **solid** solid that should be used to trim rounding-off in an ambiguous situation. names parameter provides naming of mating faces.

params rounding-off parameters contain data on the form and mating method of adjacent faces of processed edges, please see Figure M.2.10.1. SmoothValues class is described in shell_parameter.h file.



```
SmoothValues

double distance1
double distance2
double conic
double begLength
double endLength
MbSmoothForm  form  =  { st_Span,
                         st_Fillet,
                         st_Chamfer,
                         st_Slant1,
                         st_Slant2 }
CorneForm smoothCorner = { ec_pointer,
                           ec_either,
                           ec_uniform,
                           ec_sharp }
bool prolong
bool autoSurface
bool keepCant
bool strict
bool equable
MbVector3D vector1
MbVector3D vector2
```

Fig. M.2.10.1.

params input parameter contains the following data:

- *distance*1 is the first rounding-off radius,
- *distance*2 is the second rounding-off radius,
- *conic* is a shape coefficient of the mating surface,
- *begLength* is the distance from the starting vertex to mating end point (a negative value means that end point is missing),
- *endLength* is the distance from end vertex to mating end point (a negative value means that the end point is missing),
- *form* is mating type from an enumeration MbeSmoothForm,
- *smoothCorner* is rounding-off method for suitcase corners,
- *prolong* is a flag indicating that rounding-off is continued at tangent edges,
- *autoSurface* is a flag for automatic edge-keeping determination,
- *keepCant* is a flag for edge keeping,
- *strict* is construction "strictness" flag: if it is equal to false, then try to round-off everything that is possible,
- *equable* is a flag for insertion of a toroidal surface at joining corners of a mating surface,
- **vector**1 is a vector of normal to the mating end plane in the beginning,
- created **vector**2 is a vector of normal to the mating end plane in the end.

form parameter defines rounding-off type. If *form* is equal to st_Fillet or st_Span, then the edges are rounded-off; any other value of *form* is not used by this method. If *form*=st_Fillet, then the method constructs a rounding-off surface with predetermined radii that define *distance*1 and *distance*2 parameters. In Fig. M.2.10.2, you can see a rounding-off with specified edge radii; this edge joins two cylindrical surfaces.

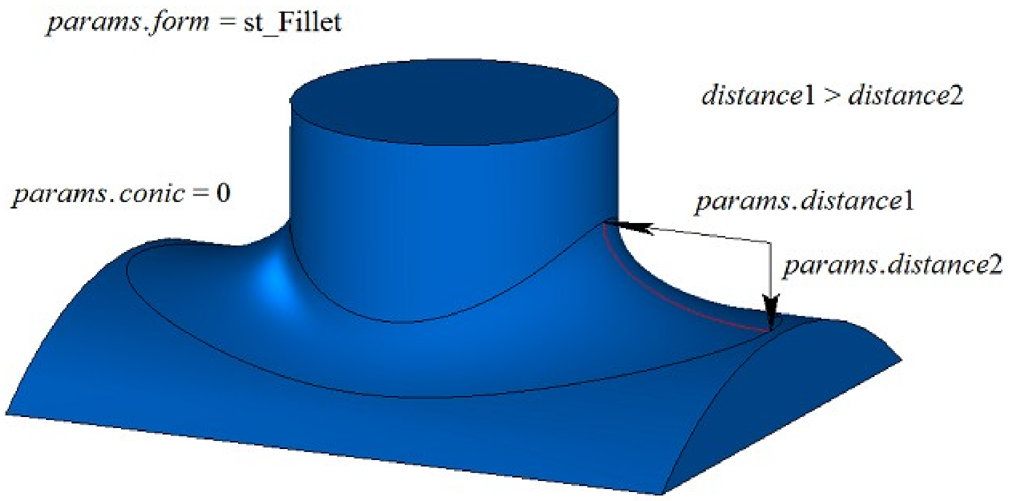


Fig. M.2.10.2.

If *distance*1=*distance*2 and *conic*=0, then the rounding-off surface is constructed by moving a sphere that touches two adjacent faces of rounded-off edge. Reference edges of mating faces are located at contact points of the sphere and corresponding adjacent face. A cross-section of mating face is a circular arc. In Fig. M.2.10.3, you can see a fillet with specified equal edge radii; this edge joins two cylindrical surfaces.

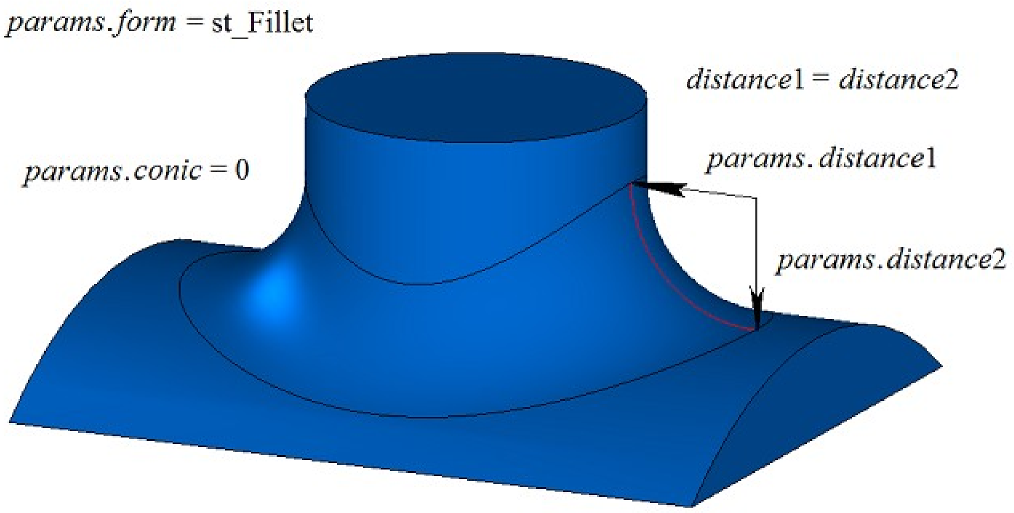


Fig. M.2.10.3.

conic coefficient defines the shape of rounding-off surface. If *conic*=0 (_ARC_ macros), then the section of the mating surface is a circular arc or an ellipse with predetermined radii. Shape coefficient can be equal to zero or it can range from 0.05 to 0.95. If *conic*=0.5, then rounding-off face cross-section is a parabolic arc. If *conic*>0.5, then rounding-off face cross-section is a hyperbolic arc. If *conic*<0.5, then rounding-off face cross-section is an elliptical arc. In Fig. M.2.10.4 and M.2.10.5, you can see rounding-offs with equal radii of the edge joining two cylindrical surfaces with different shape coefficients.

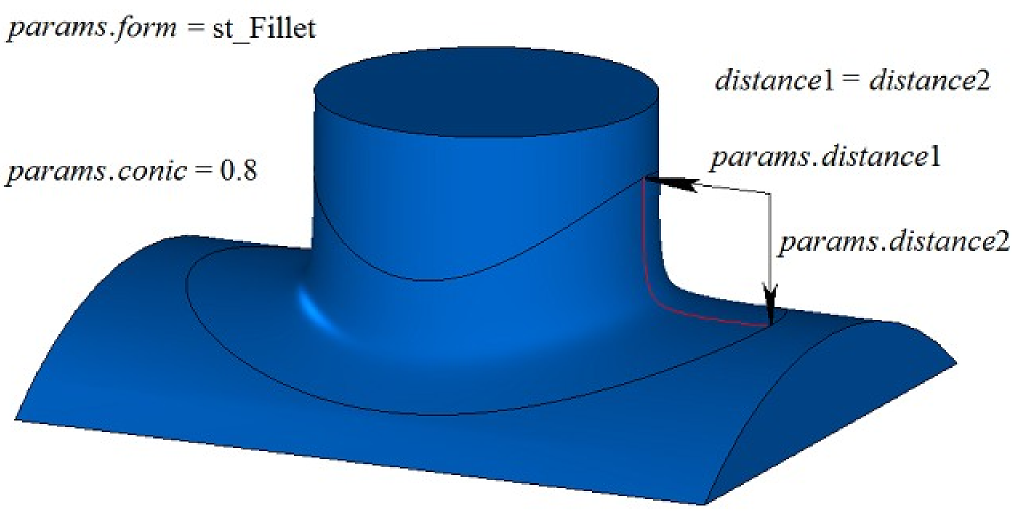


Fig. M.2.10.4.

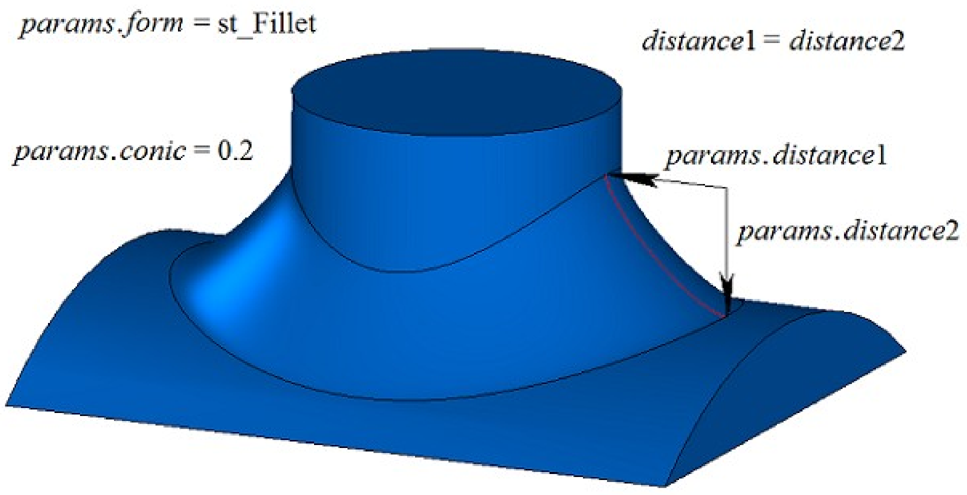


Fig. M.2.10.5.

If *form*=st_Span, then the method constructs a rounding-off surface with a specified chord. *distance*1 and *distance*2 are equal, they determine the distance between the reference edges of the mating face. Rounding-off face cross-section is a circular arc. In general case, arc radii are different in every rounding-off face cross-section and *distance*1 and *distance*2 parameters are equal to the chord of the circular arc. In Fig. M.2.10.6, you can see a rounding-off with specified edge chord; the edge joins two cylindrical surfaces.

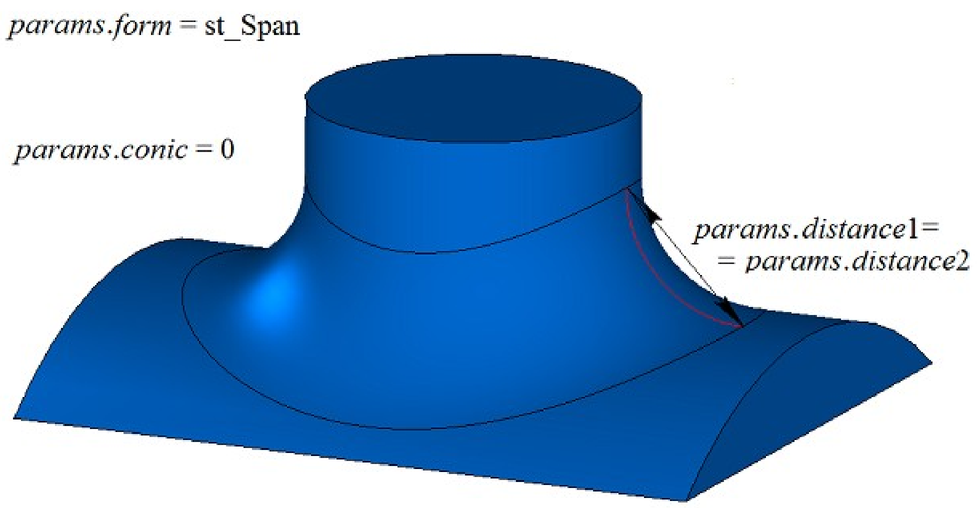


Fig. M.2.10.6.

In Fig. M.2.10.6, you can see a rounding-off with specified edge chord having non-zero shape coefficient; the edge joins two cylindrical surfaces.

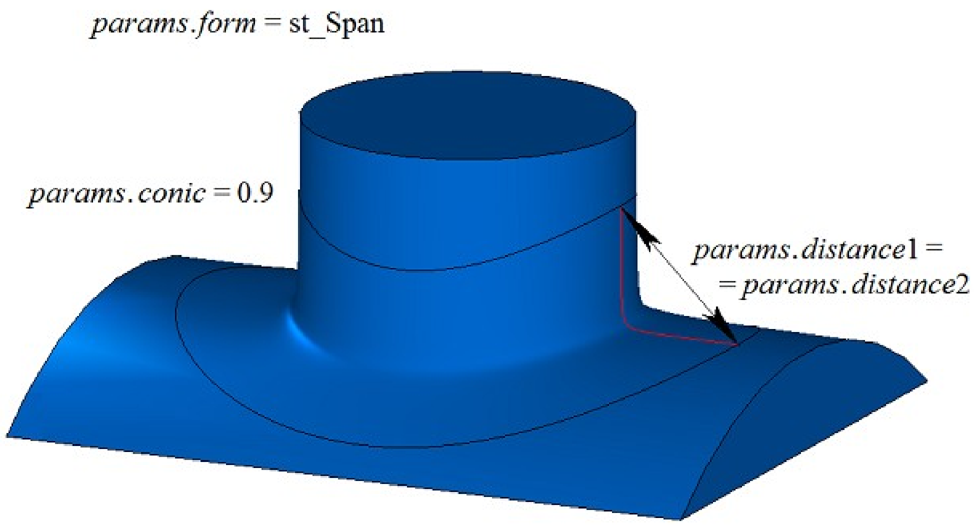
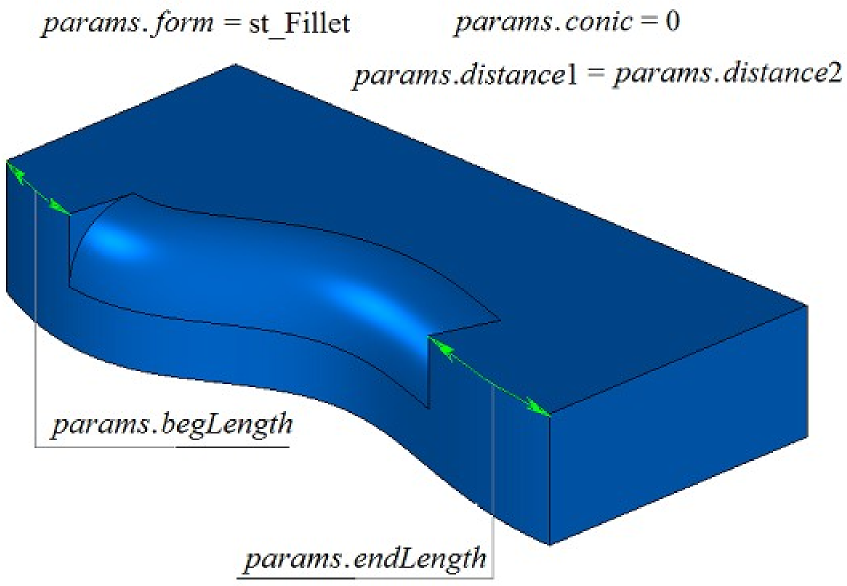


Fig. M.2.10.7.

In Fig. M.2.10.8, you can see an example of rounding-off stop located *begLength* away from start vertex and *endLength* away from end vertex. If there is no need to stop mating, then *begLength* and *endLength* should take negative values. By default, the stop of the fillet edges is perpendicular to the fillet edge. You can override the behavior stops fillets using vector **vector**1 as the normal stopping faces at the beginning of the pairing and the vector **vector**2 as the normal stopping faces at the end of the pairing.



$params.form$ = st_Fillet
$params.conic = 0$
$params.distance1 = params.distance2$
$params.begLength$
$params.endLength$

Fig. M.2.10.8.

Let's look at the solid shown in Figure M.2.10.9 as an example and demonstrate how to use *prolong*, *autoSurface* and *keepCant* flags when an edge highlighted in Figure M.2.10.9 is rounded-off.

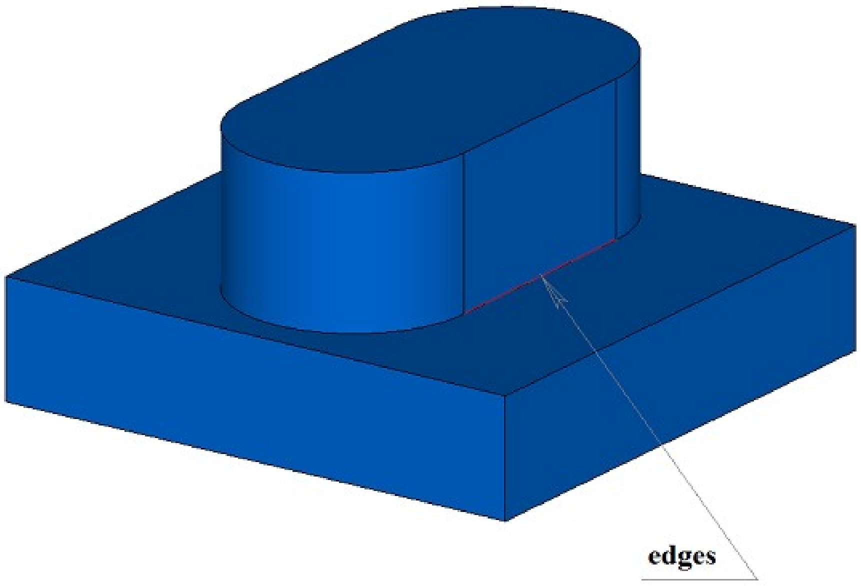


Fig. M.2.10.9.

prolong flag determines what edges should be processed. If *prolong*=false, then only edges from **edges** container should be processed (Figure M.2.10.10).

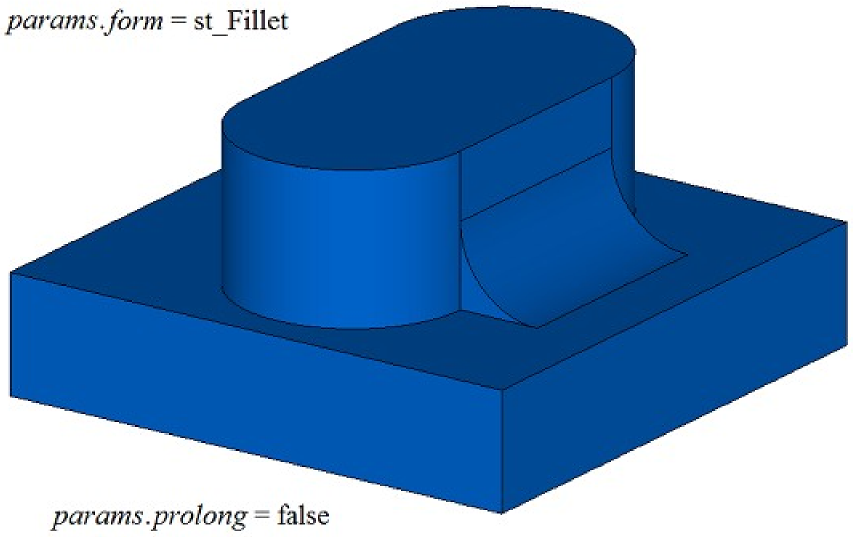


Fig. M.2.10.10.

if *prolong*=true, then edges from **edges** container should be processed, as well as the edges smoothly joined with them (Figure M.2.10.11).

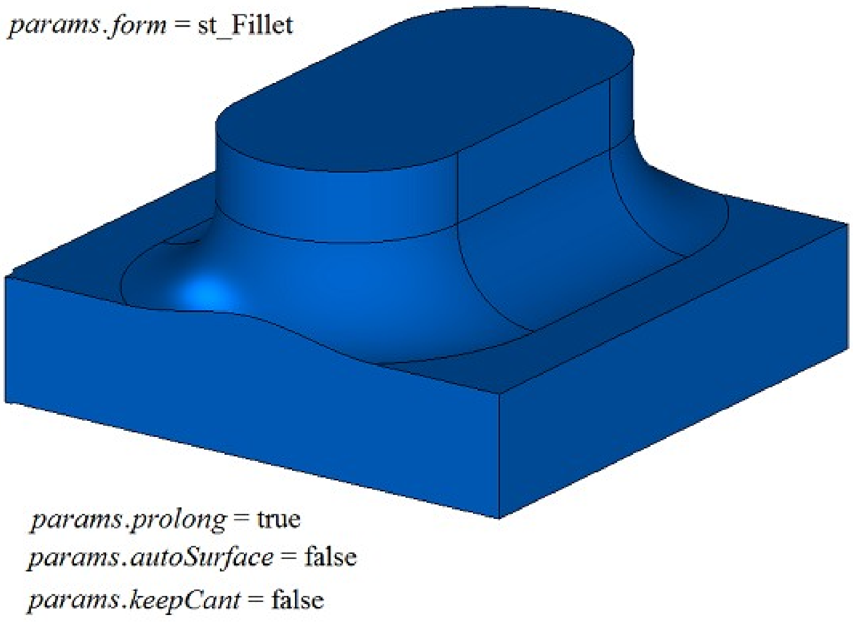

params.form = st_Fillet
params.prolong = true
params.autoSurface = false
params.keepCant = false

Fig. M.2.10.11.

autoSurface and *keepCant* flags are used to handle situations when reference edges of the face fall beyond the adjacent face. If *autoSurface*=false and *keepCant*=false, then in situations when reference edges of the face go beyond an adjacent face with an acute edge, the mating face keeps its original shape and it is cut off by the adjacent face, see Figure M.2.10.11. If *autoSurface*=true or *keepCant*=true, then in situations when face reference edge goes beyond the adjacent face with an acute edge, the mating face changes its shape and goes by its reference edge along the boundary, keeping it unchanged as shown in Figure M.2.10.12.

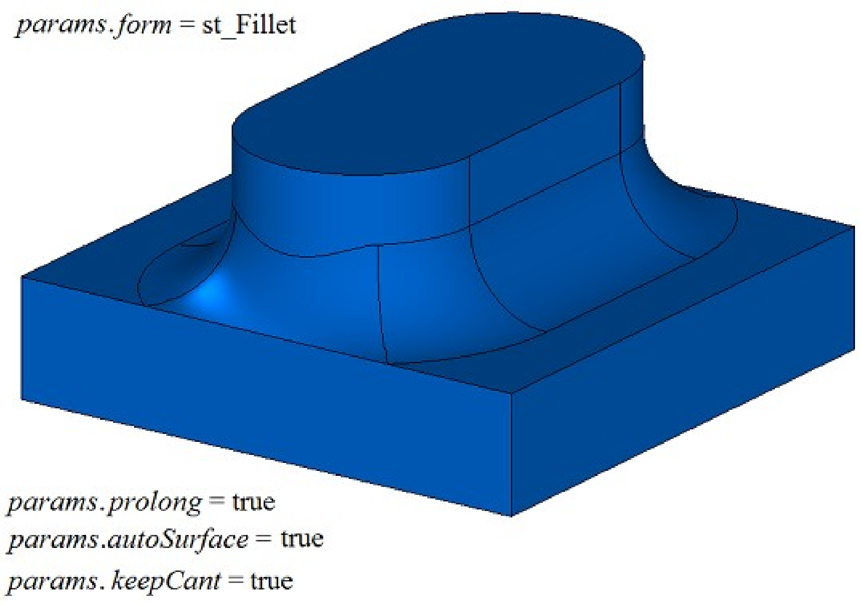


Fig. M.2.10.12.

If *autoSurface*=true and *keepCant*=false, then if reference edges of the mating face go beyond the adjacent face via a smooth edge, then the mating face replaces the adjacent face with its neighbor and changes its shape in this section as shown in Figure M.2.10.13.

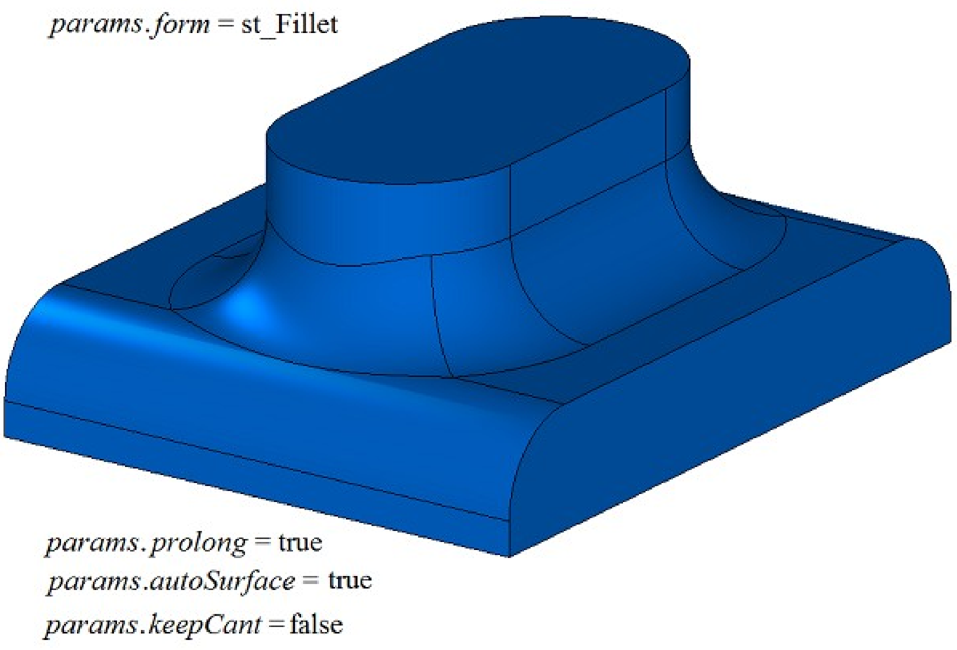


Fig. M.2.10.13.

In Fig. M.2.10.14, you can see rounding-off of four edges created by a single call of this method with *equable*=false flag.

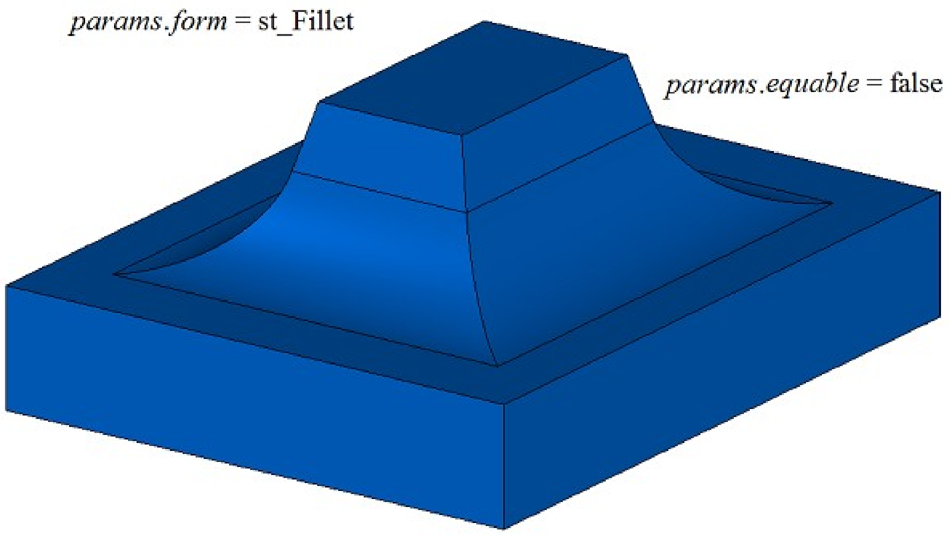


Fig. M.2.10.14.

In Fig. M.2.10.15, you can see rounding-off of four edges created by a single call of this method with *equable*=true; this flag indicates the need to insert toroidal surfaces at the joining corners of mating surfaces.

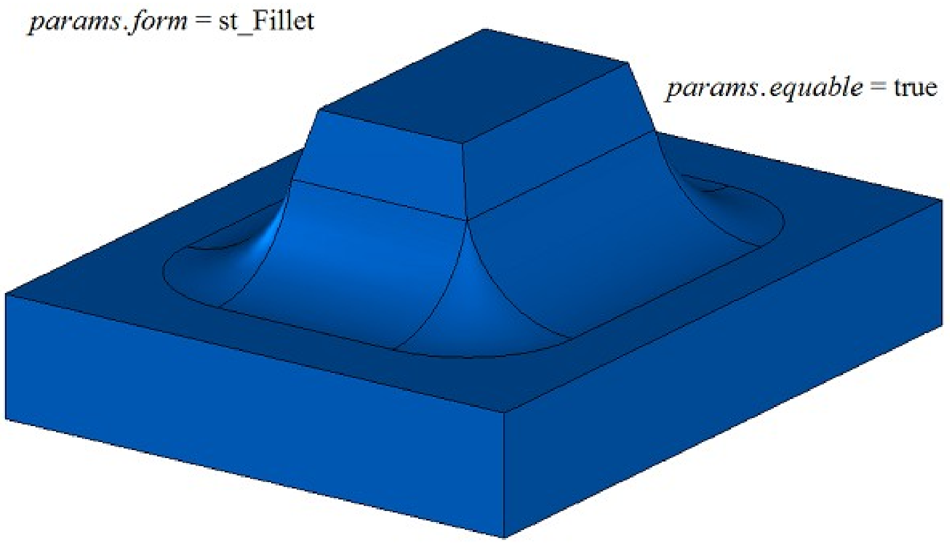


Fig. M.2.10.15.

If three edges mating in a single vertex are rounded-off, then *smoothCorner* determines suitcase corners rounding-off processing method. If *smoothCorner*=ec_pointed, then the corners where three edges with the same convexity are mating are not processed, see Figure M.2.10.16.

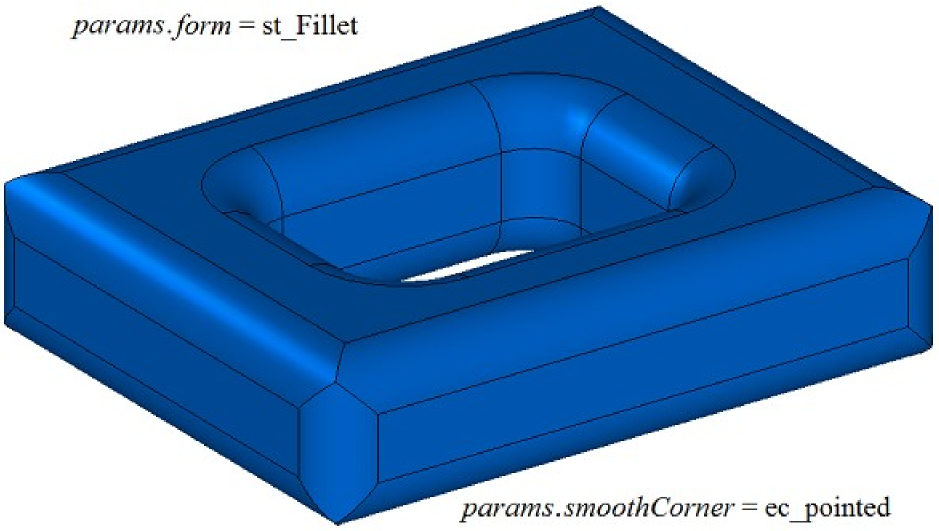


Fig. M.2.10.16.

If *smoothCorner*=ec_uniform, then corners that join three edges with different convexities are processed using the same method as shown in Figure M.2.10.17.

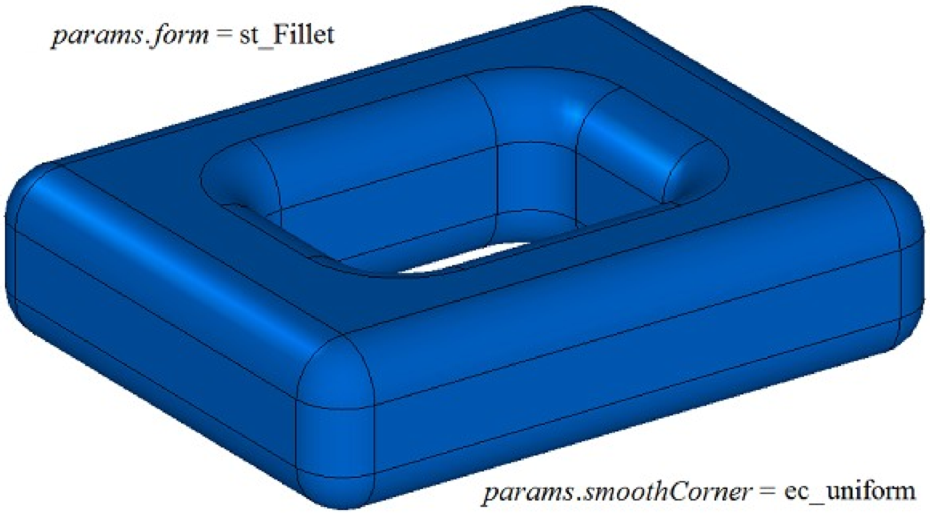


Fig. M.2.10.17.

If *smoothCorner*=ec_sharp, then corners that join three edges with different convexity are processed using the same method as shown in Figure M.2.10.18.

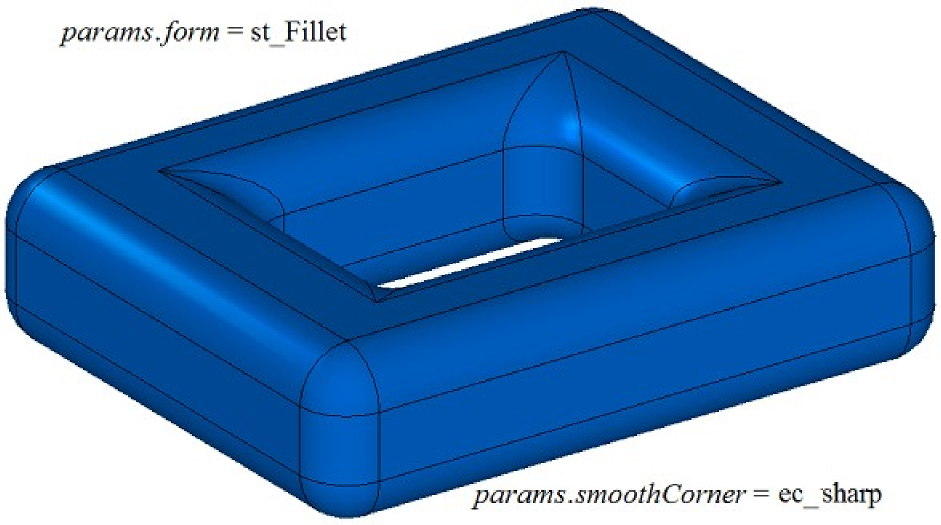


Fig. M.2.10.18

If *smoothCorner*=ec_either, then the corners that join three edges with different convexities may be processed using different methods.

In an ambiguous situation **bounds** parameter contains faces of **solid** solid that should be used to trim rounding-off faces. An example of using **bounds** parameter is given in Figures M.2.10.19 and M.2.10.20.

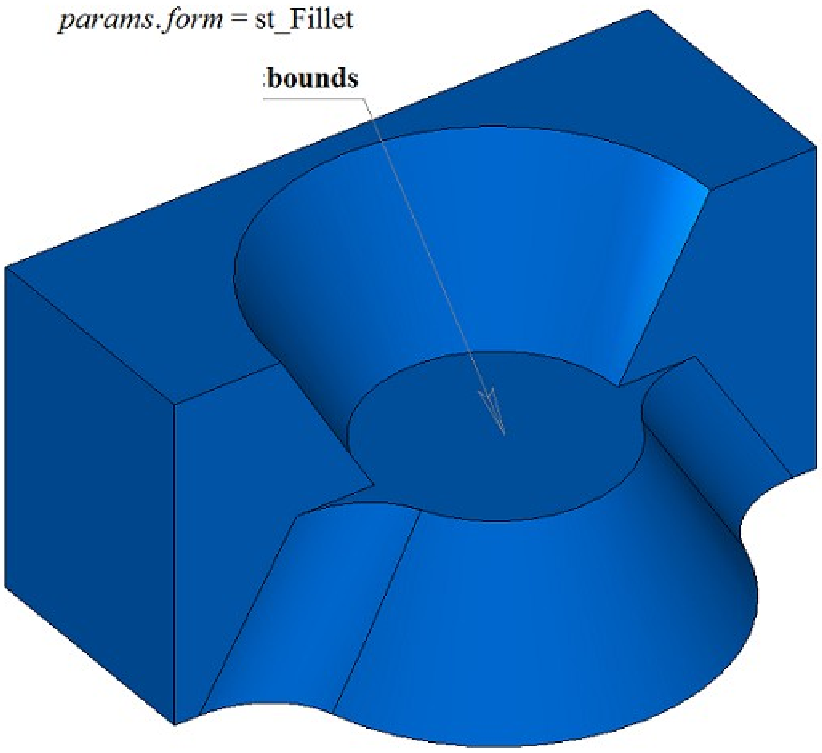


Fig. M.2.10.19

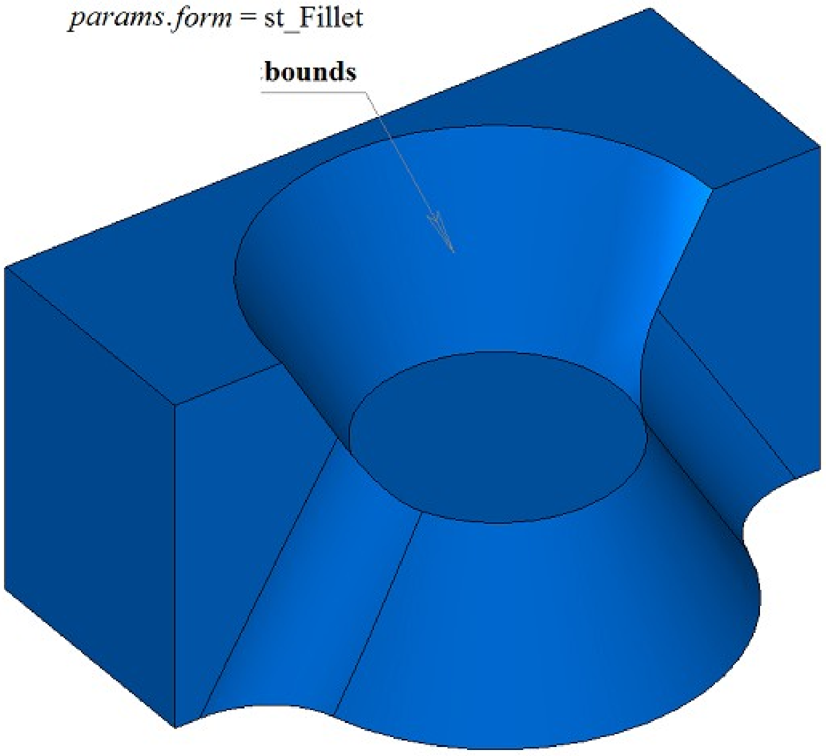


Fig. M.2.10.20

bounds parameter may be used to stop mating faces in the beginning and in the end. In this case, the edges defined by **bounds** parameter should belong to **solid** original solid.

In Fig. M.2.10.21, you can see a model, for which edge rounding-offs that completely cover the hole and the protrusion should be constructed.

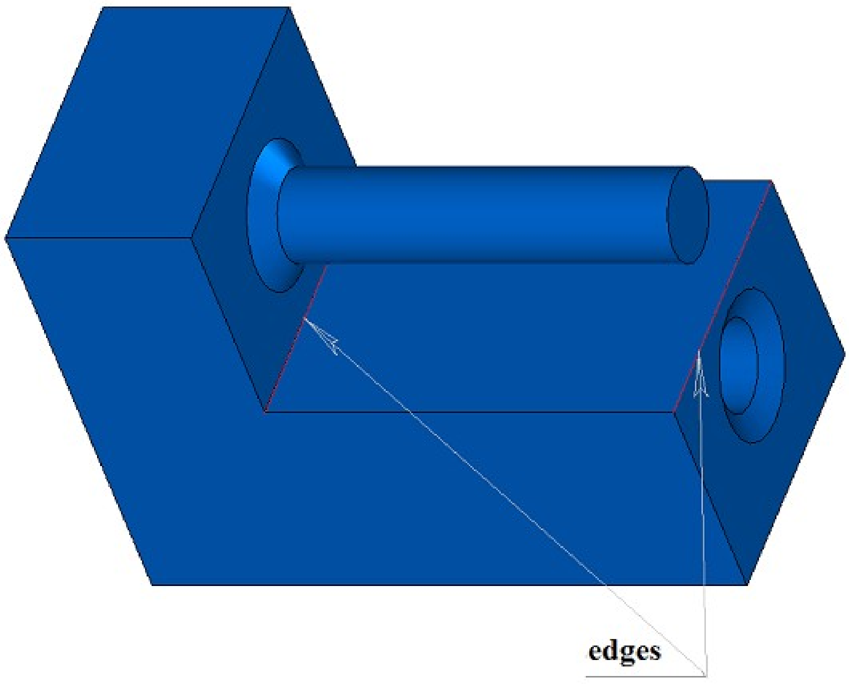


Fig. M.2.10.21

A rounding-off with avoidance of obstacles is shown in Figure M.2.10.22.

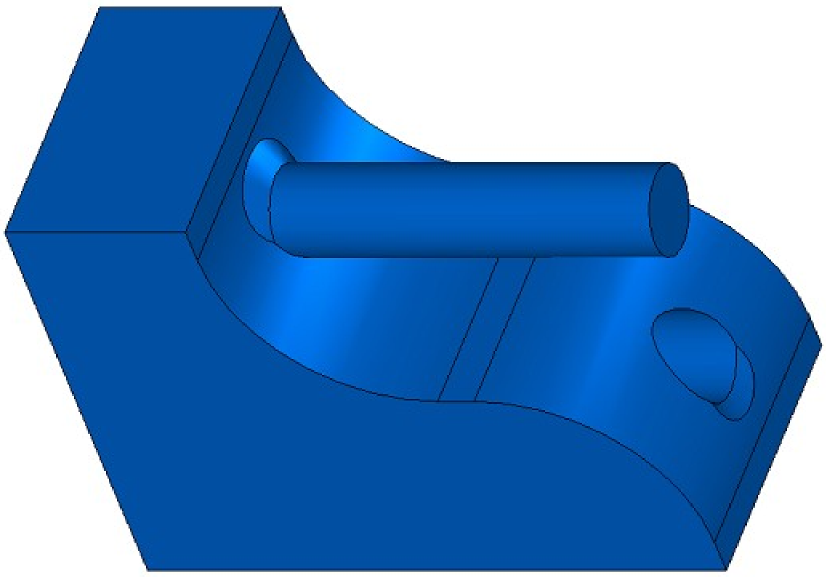


Fig. M.2.10.22

In Fig. M.2.10.23, you can see simultaneous rounding-off of six edges having a common vertex.

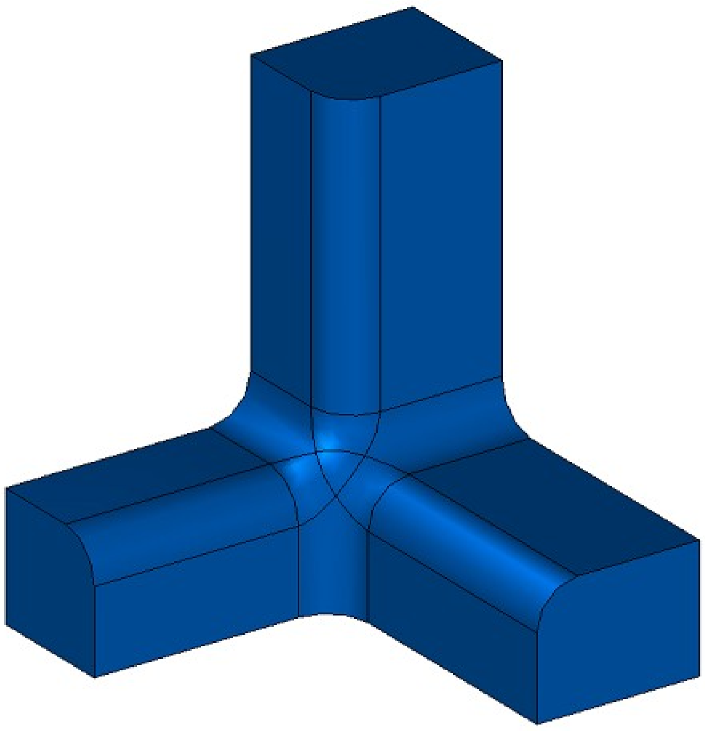


Fig. M.2.10.23

In Fig. M.2.10.24 you can see simultaneous rounding-off of several groups of four edges with common vertices. Rounding-off feature is that the groups are linked with each other and can be processed simultaneously only. Original solid for the solid shown in Figure M.2.10.24 was constructed by subtracting four cylinders with axes coinciding with cube diagonals from a cube.

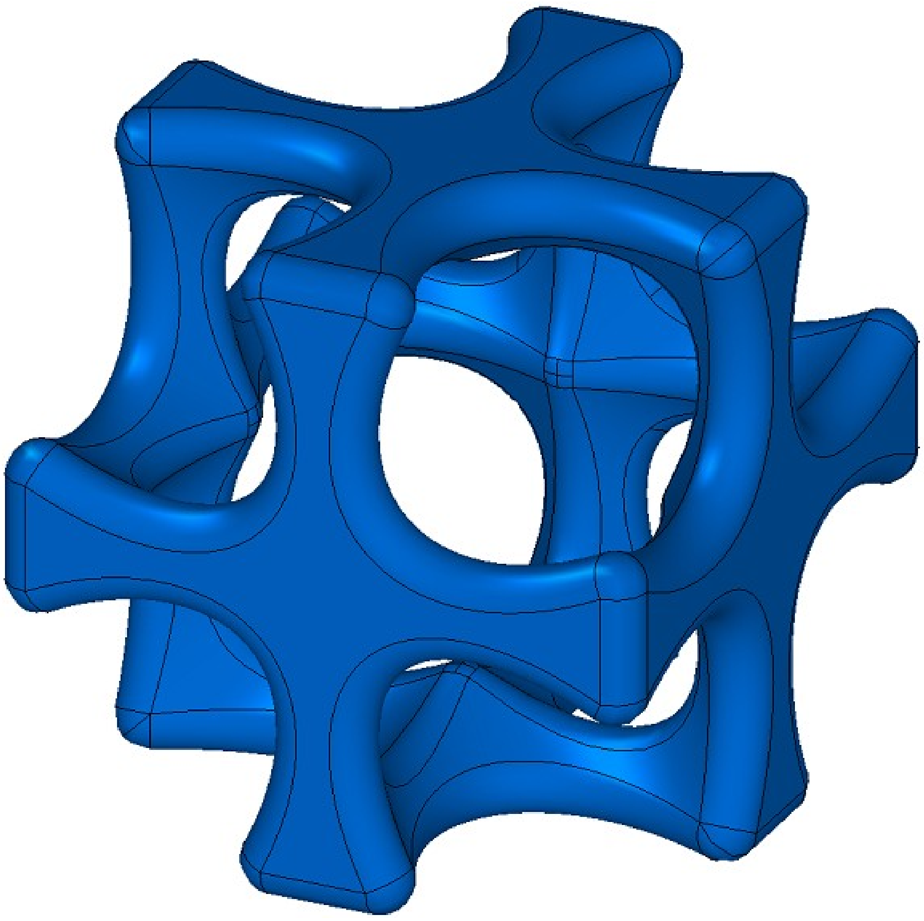


Fig. M.2.10.24

If rounding-off based on edges is constructed, then the method adds MbFilletSolid constructor in the log of newly constructed solid. The constructor is declared in cr_fillet_solid.h file.

test.exe test application processes the edges of the solid using New ->Solid -> By Processing Edges -> Round-off by Radius and New ->Solid -> By Processing Edges -> Round-off by Chord menu commands.

M.2.11. Rounding-off Edges of the Solid Using Variable Radius

Method
MbResultType
FilletSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
SArray<MbEdgeFunction> & **edges**,
RPArray<MbFace> & **bounds**,
const SmoothValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
rounds-off specified edges in a copy of original solid using a variable radius.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,

- **edges** is the set of rounding-off edges with specified radius change methods.
- **bounds** is a set of faces used to cut rounded-off edges (the set may be empty),
- *params* are construction parameters,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method replaces specified edges of the original solid with rounded-off faces in order to ensure smooth mating of adjacent faces of specified edges. When the edges are rounded-off, the mating faces may have the shape of a circular arc with variable radius. The method is similar to the method described in the preceding item, the difference is the third parameter: **edges**.

solid parameter contains the original solid, its edges should be processed. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** resulting solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

bounds parameter contains faces of **solid** solid that should be used to trim rounding-off in an ambiguous situation. names parameter provides naming of mating faces.

params rounding-off parameters contain data on the form and mating method of adjacent faces of processed edges, please see Figure M.2.10.1. SmoothValues class is described in shell_parameter.h file.
params input parameter contains the following data:

- *distance*1 is the first rounding-off radius,
- *distance*2 is the second rounding-off radius,
- *conic* is a shape coefficient of the mating surface,
- *begLength* is the distance from the starting vertex to mating end point (a negative value means that end point is missing),
- *endLength* is the distance from end vertex to mating end point (a negative value means that the end point is missing),
- *form* is mating type from an enumeration MbeSmoothForm,
- *smoothCorner* is rounding-off method for suitcase corners,
- *prolong* is a flag indicating that rounding-off is continued at tangent edges,
- *autoSurface* is a flag for automatic edge-keeping determination,
- *keepCant* is a flag for edge keeping,
- *strict* is construction "strictness" flag: if it is equal to false, then try to round-off everything that is possible,
- *equable* is a flag for insertion of a toroidal surface at joining corners of a mating surface,
- **vector**1 is a vector of normal to the mating end plane in the beginning,
- created **vector**2 is a vector of normal to the mating end plane in the end.

edges parameter contains processed edges of **solid** solid and the function of radius change along the edge. Each element of **edges** set consists of a pointer to an edge and a pointer to a scalar function, its values should be multiplied by the first and second rounding-off radii (*distance*1 and *distance*2), see shown Figure M.2.11.1.

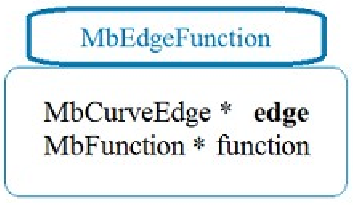


Fig. M.2.11.1.

form parameter defines rounding-off type. If *form* parameter is equal to st_Fillet or st_Span, then the edges of variable radius are rounded-off; any other values of *form* are not used by the method. For each

point of processed **edges**[i].**edge**->Point(*t*,**point**), curvature radii of rounding-off surface are equal to *distance*1 and *distance*2 parameters multiplied by **edges**[i].function->Value(*t*) function value. In Fig. M.2.11.2, you can see a rounding-off with variable radii of rectangular prism edge. If *distance*1=*distance*2 and *conic*=0, then rounding-off surface is constructed by moving the sphere of variable radius that touches two adjacent faces of the rounded-off edge. Reference edges of mating faces are located at contact points of the sphere and corresponding adjacent face. A cross-section of mating face is a circular arc.

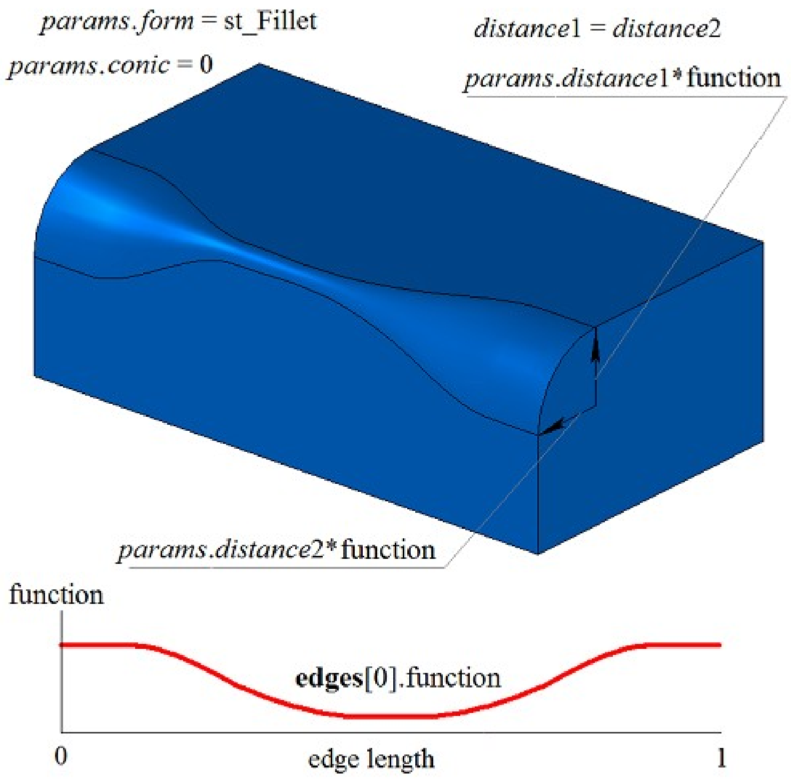

params.form = st_Fillet
params.conic = 0
*distance*1 = *distance*2
*params.distance*1 * function
*params.distance*2 * function
function
edges[0].function
0
edge length
1

Fig. M.2.11.2.

In Fig. M.2.11.3, you can see an elliptical rounding-off with variable radii for rectangular prism edge.

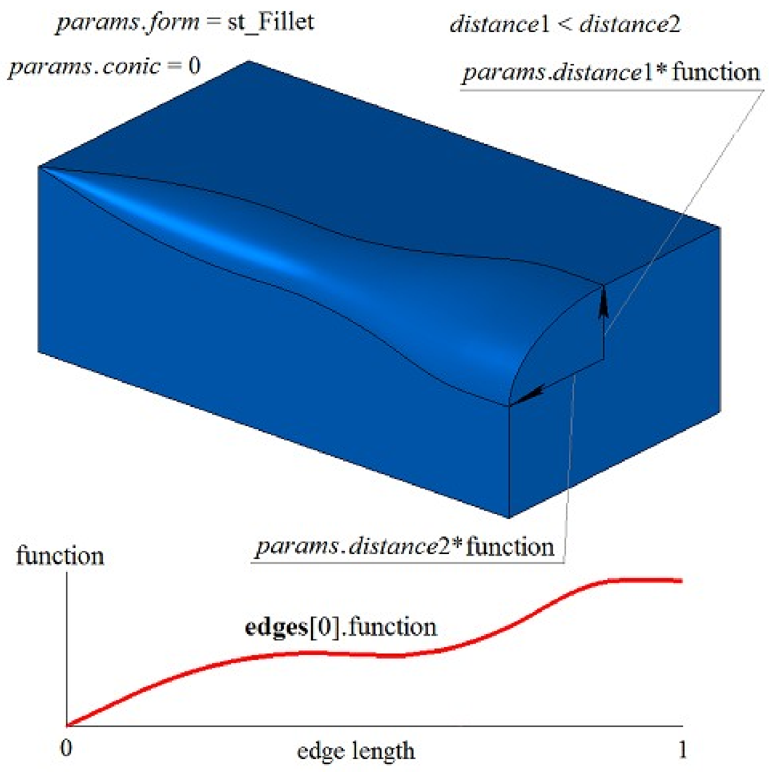


Fig. M.2.11.3.

conic coefficient defines the shape of rounding-off surface. If *conic*=0 (_ARC_ macros), then the section of the mating surface is a circular arc or an ellipse with predetermined radii. Shape coefficient can be equal to zero or it can range from 0.05 to 0.95. If *conic*=0.5, then rounding-off face cross-section is a parabolic arc. If *conic*>0.5, then rounding-off face cross-section is a hyperbolic arc. If *conic*<0.5, then rounding-off face cross-section is an elliptical arc. In Fig. M.2.11.4 and M.2.11.5, you can see rounding-off with variable radii for rectangular prism edge with various form factors.

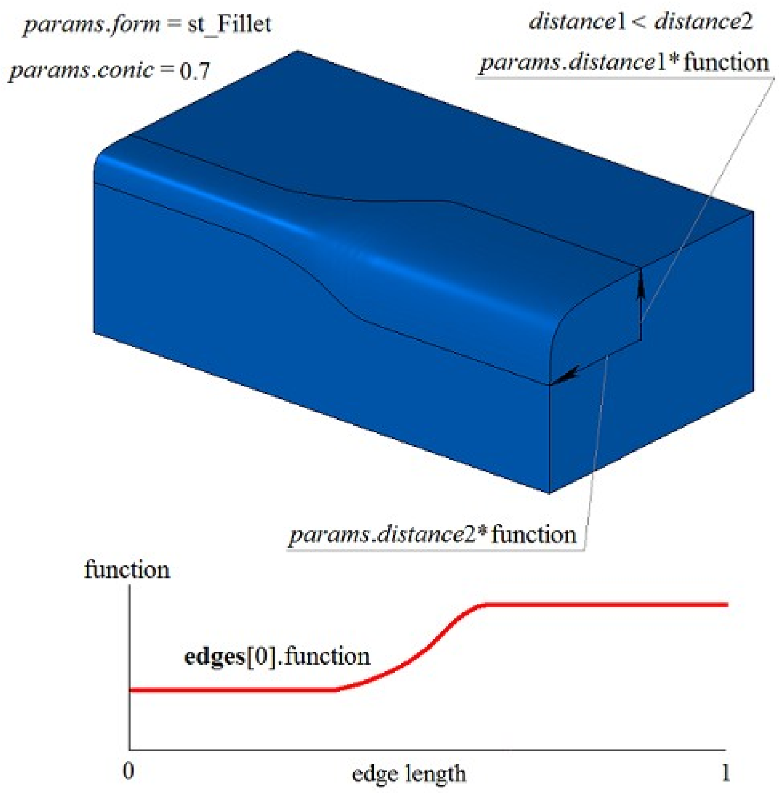


Fig. M.2.11.4.

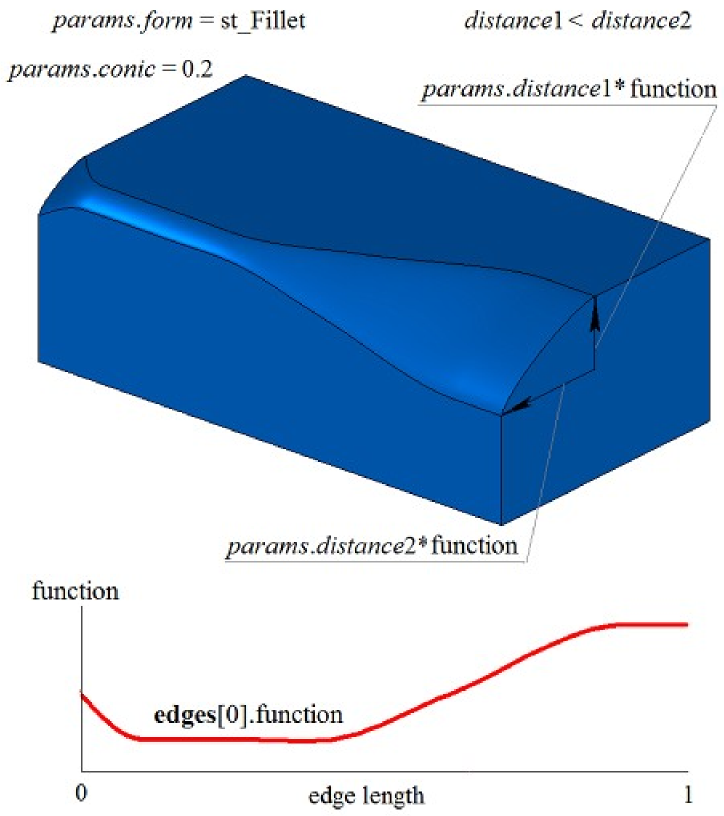


Fig. M.2.11.5.

In Fig. M.2.10.6, you can see rounding-off stopping points located *begLength* away from start vertex and *endLength* away from end vertex. When settings of stop rounding-off are configured, the method used to change curvature radii is set for the whole edge. If there is no need to stop mating, then *begLength* and *endLength* should take negative values.

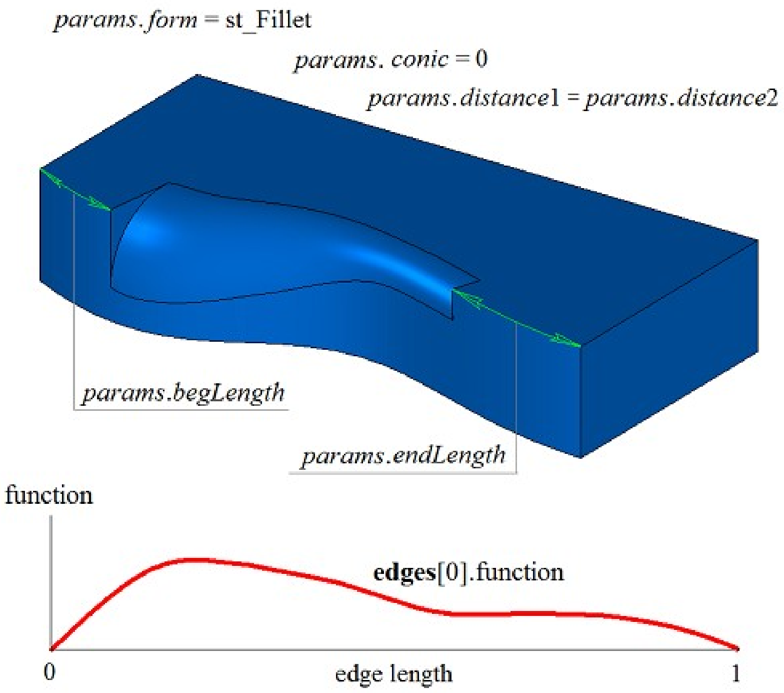


Fig. M.2.11.6.

prolong flag determines what edges should be processed. If *prolong*=false, then only edges from **edges** container should be processed. If *prolong*=true, then edges from edges container should be processed, as well as **edges** smoothly joined with them. In order to continue rounding-off edges, radius changing method takes a constant value equal to function value at the edge of the previous smoothly jointed edges. In Fig. M.2.11.7, you can see the original solid, edges that should be rounded-off, and radius change method for specified edges.

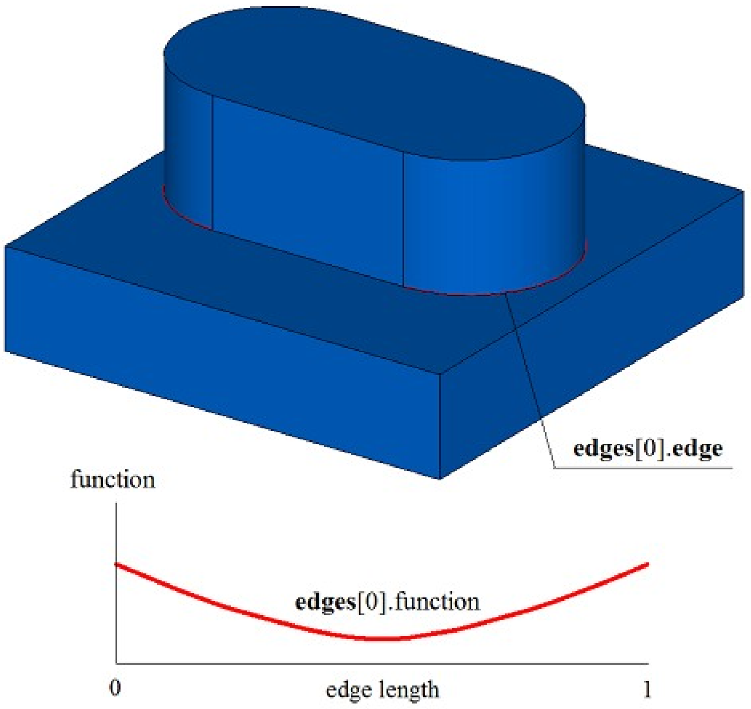


Fig. M.2.11.7.

In Fig. M.2.11.8, you can see the result of operation for the method for the original solid and the method used to change the radius of edges shown in Figure M.2.11.7. In Fig. M.2.11.8, you can see that at the edges, radius changing method was picked up to ensure smooth matching with adjacent rounding-off edge.

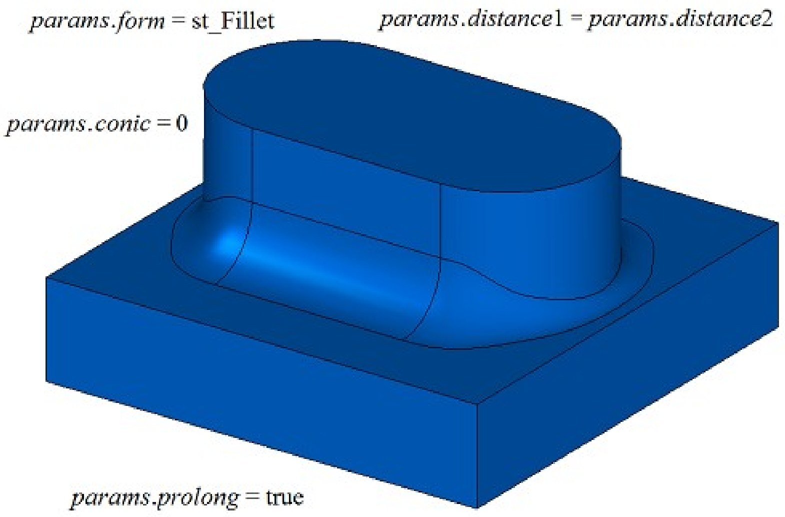


Fig. M.2.11.8.

autoSurface, *keepCant* and *equable* flags are not used in the method.

If three edges mating in a single vertex are rounded-off, then *smoothCorner* determines suitcase corners rounding-off processing method. If *smoothCorner*=ec_pointed, then the corners where three edges with the same convexity are mating are not processed, see Figure M.2.10.16. If *smoothCorner*=ec_uniform, then corners that join three edges with different convexities are processed using the same method as shown in Figure M.2.10.17. If *smoothCorner*=ec_sharp, then corners that join three edges with different convexity are processed using the same method as shown in Figure M.2.10.18. If *smoothCorner*=ec_either, then the corners that join three edges with different convexities may be processed using different methods. If the functions used to change curvature radius at the edges of rounding-off faces do not coincide, then these functions are modified to ensure smooth mating of rounding-off faces.

In an ambiguous situation **bounds** parameter contains faces of **solid** solid that should be used to trim rounding-off faces. An example of using **bounds** parameter is given in Figures M.1.20.19 and M.1.20.20.

bounds parameter may be used to stop mating faces in the beginning and in the end. In this case, the edges defined by **bounds** parameter should belong to **solid** original solid.

If rounding-off based on edges is constructed, then the method adds MbFilletSolid constructor in the log of newly constructed solid. The constructor is declared in cr_fillet_solid.h file.

test.exe test application processes the edges of the solid using New ->Solid -> By Processing Edges -> Variable Rounding-off menu command.

M.2.12. Constructing a Solid with Edge Chamfers

Method
MbResultType
ChamferSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
RPArray<MbCurveEdge> & **edges**,
const SmoothValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs chamfers at specified edges in a copy of the original solid.

Input parameters of the method are as follows:

- **solid** is the original solid,
- sameShell is original solid copying option,
- **edges** is a set of rounded-off edges,
- params are construction parameters,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

This method replaces the specified edges of the original solid with chamfer faces.

The method is declared in action_solid.h file.

The method replaces specified edges of the original solid with chamfer faces.

solid parameter contains the original solid, its edges should be processed. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** resulting solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

edges parameter contains processed edges of **solid** solid. names parameter provides naming of chamfer faces.

To construct edge chamfers and fillets the same SmoothValues and *params* parameters are used, see Figure M.1.20.1. SmoothValues class is described in shell_parameter.h file. *params* parameters used to create a chamfer contain data on the form and mating method for adjacent faces of processed edges. The following data from *params* parameter are used to construct chamfers:

- *distance*1 is the first chamfer side,
- *distance*2 is the second chamfer side,
- *begLength* is the distance from the starting vertex to mating end point (a negative value means that end point is missing),
- *endLength* is the distance from end vertex to mating end point (a negative value means that the end point is missing),
- *form* is mating type from an enumeration MbeSmoothForm,
- *smoothCorner* is rounding-off method for suitcase corners,
- *prolong* is a flag indicating that rounding-off is continued at tangent edges,

- **vector**1 is a vector of normal to the mating end plane in the beginning,
- **vector**2 is a vector of normal to the mating end plane at the end.

conic, *autoSurface*, *keepCant*, *strict*, and *equable* values are not used to construct a chamfer.

form parameter controls the method that is used to describe the chamfer. Values of the *form* parameter equal to st_Chamfer, st_Slant1 and st_Slant2 are used to construct edge chamfers. If *form*=st_Fillet, then the method constructs chamfer surface with predetermined sides that define *distance*1 and *distance*2 parameters. M.2.12.1.

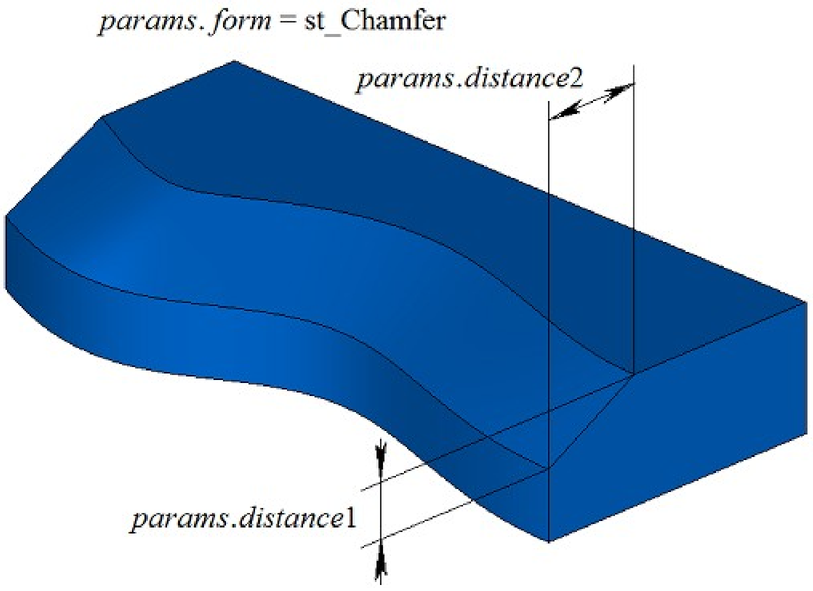


Fig. M.2.12.1.

If *form*=st_Slant1, then the method constructs a chamfer face for the specified leg and its adjacent angle. The leg defines *distance*1 parameter, and *distance*2 corresponds to the leg belonging to the adjacent angle; see Figure M.2.12.2.

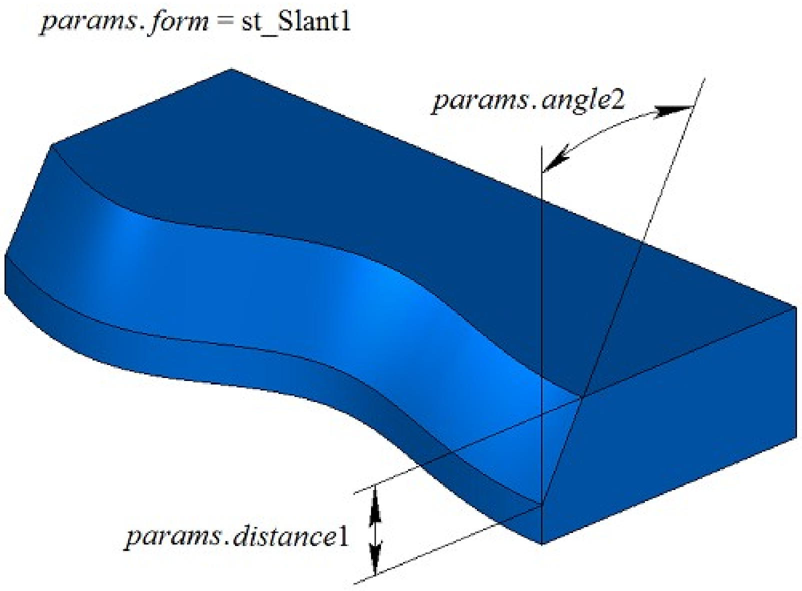


Fig. M.2.12.2.

If *form*=st_Slant2, then the method constructs a chamfer surface with specified angle and adjoining side.

*distance*1 corresponds to the side providing the specified angle, and *distance*2 parameter defines the adjoining side, see Figure M.2.12.3.

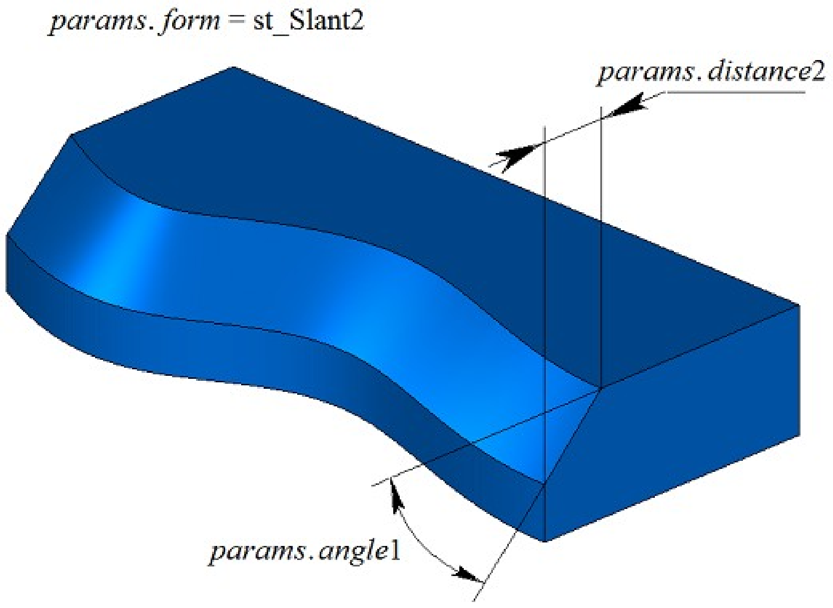


Fig. M.2.12.3.

In Fig. M.2.12.4, you can see chamfer stopping example with stopping points *begLength* away from start vertex and *endLength* away from the end vertex of the processed edge. If there is no need to stop mating, then *begLength* and *endLength* should take negative values.

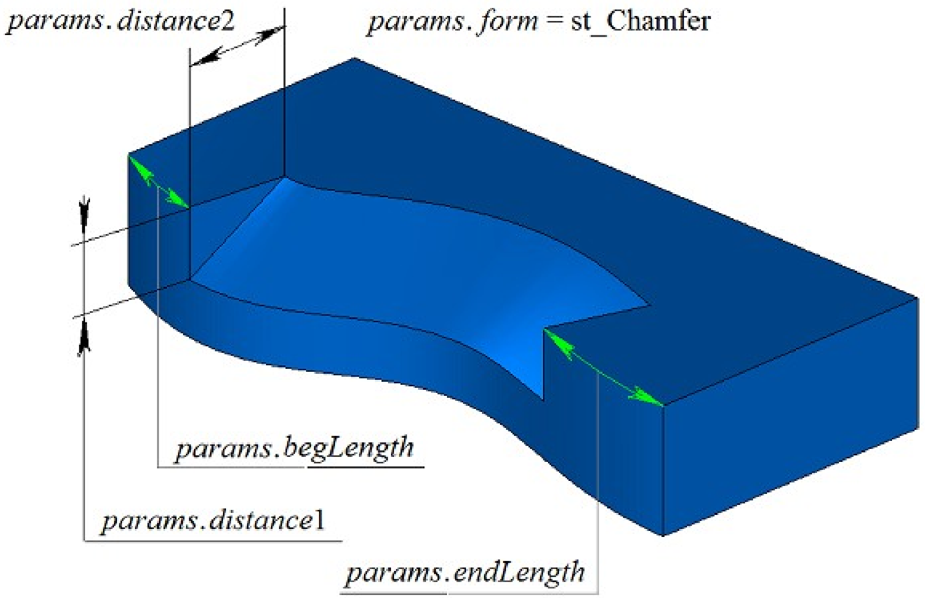


Fig. M.2.12.4.

Let's take the solid shown at Figure M.2.12.5 as an example to demonstrate how to use *prolong* flag to construct a chamfer for the edge highlighted in Figure M.2.12.5.

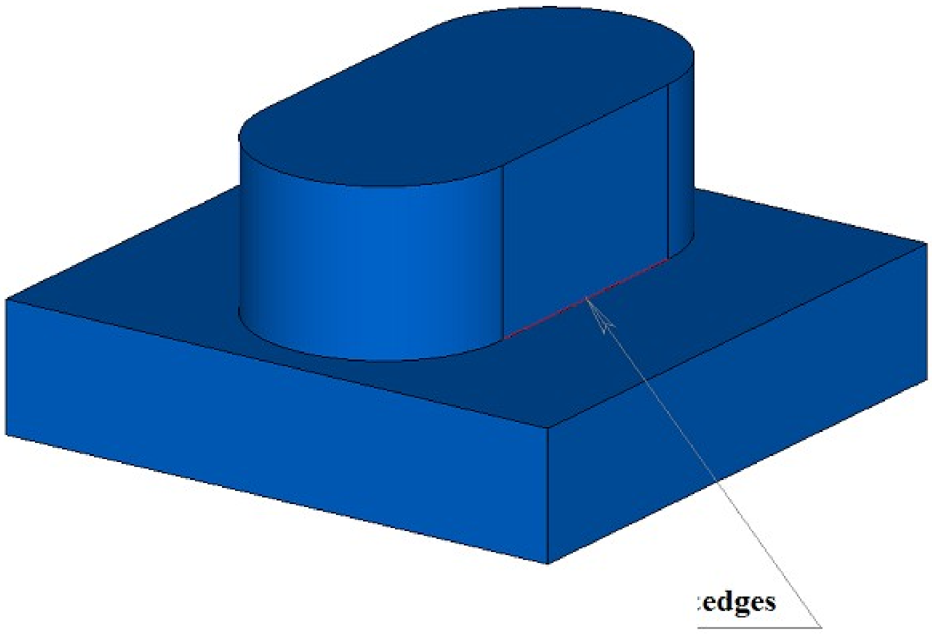


Fig. M.2.12.5.

prolong flag determines what edges should be processed. If *prolong*=false, then only the edges from **edges** container should be processed, see Figure M.2.12.6.

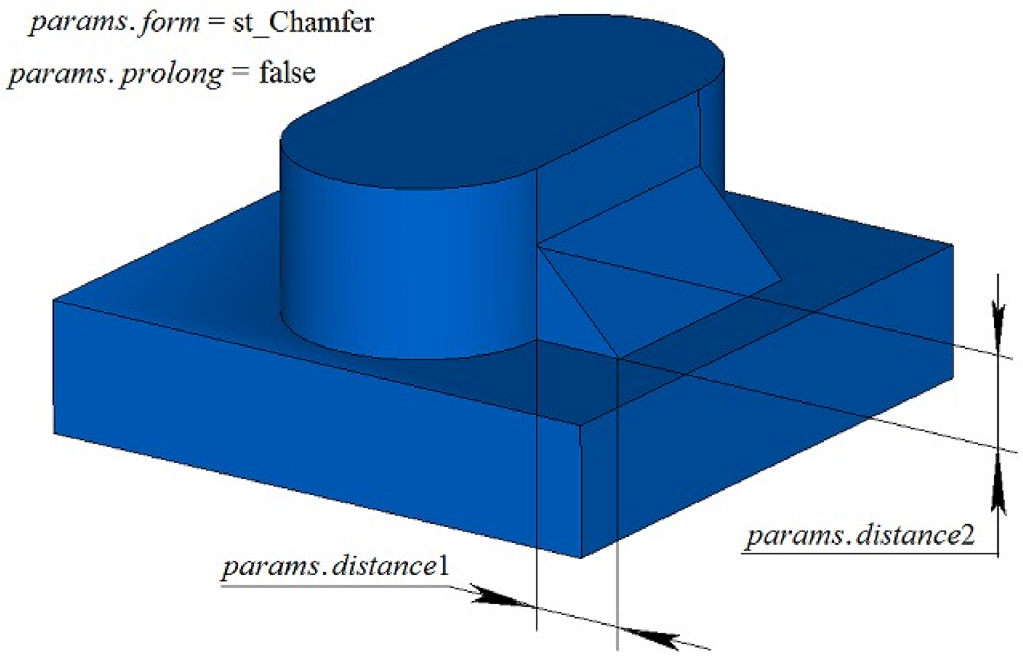


Fig. M.2.12.6.

If *prolong*=true, then edges specified in **edges** container should to be processed, as well as edges smoothly joined with them, see Figure M.2.12.7.

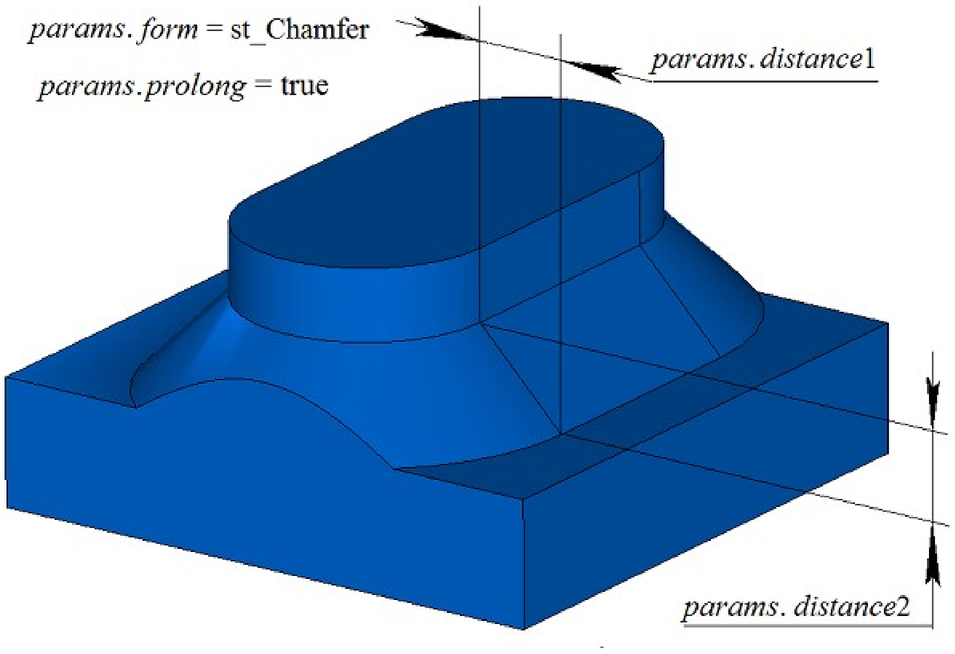


Fig. M.2.12.7.

When chamfers of three edges mating in a single vertex are constructed, *smoothCorner* parameter defines the method used to process suitcase corners. Let's take as an example the solid shown in Figure M.2.12.8 and show how to use *smoothCorner* parameter to construct chamfers at all edges of the solid.

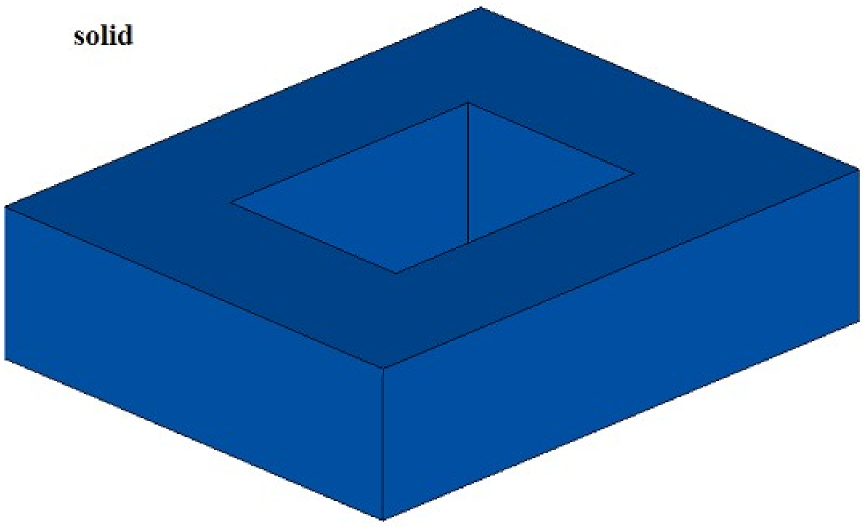


Fig. M.2.12.8.

If *smoothCorner*=ec_pointed, then corners that join three edges of the same convex are not processed, and constructed solid has a point where three faces with the same chamfer meet, see Figure M.2.12.9.

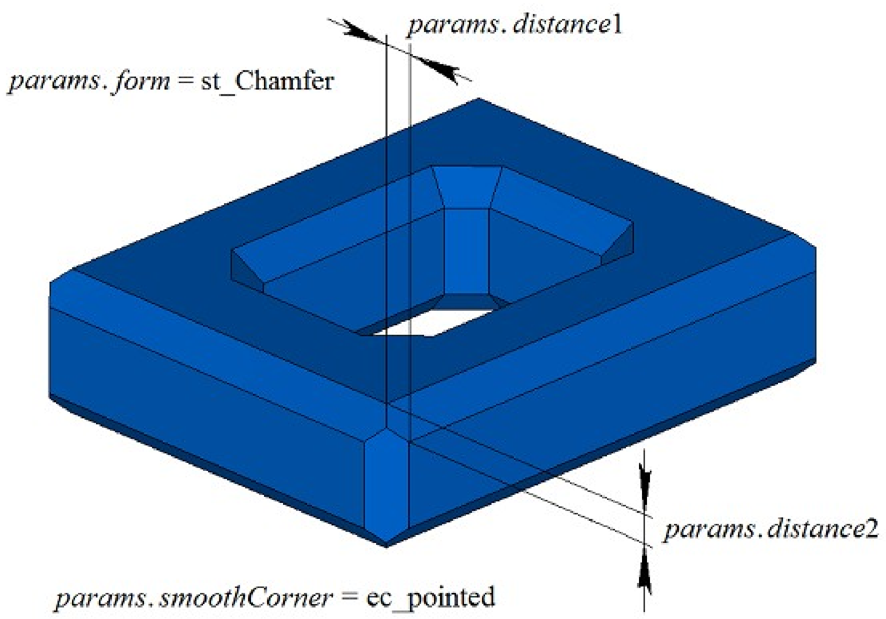


Fig. M.2.12.9.

If *smoothCorner* parameter has any other value, then corners where three edges meet are processed by constructing an additional face as shown in Figure M.2.12.10.

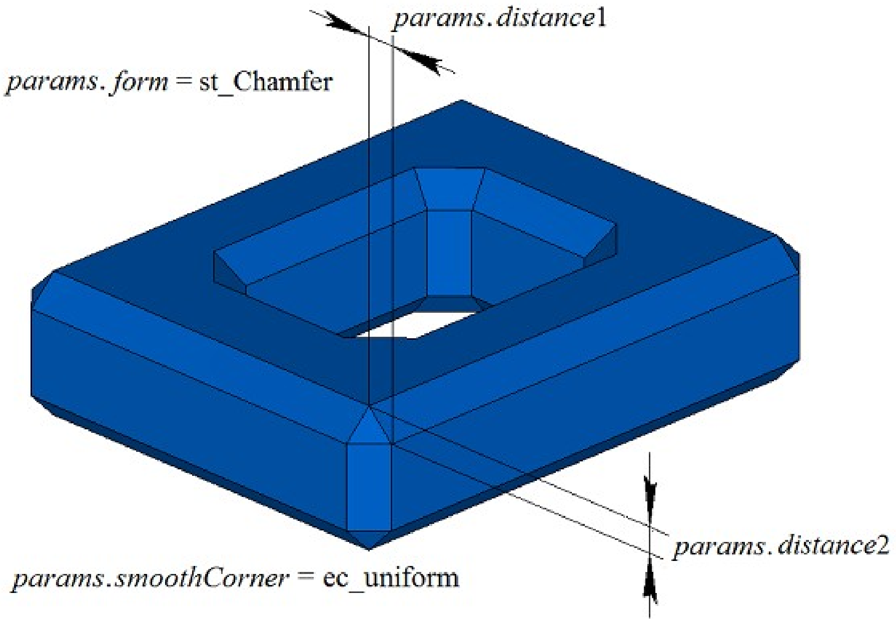


Fig. M.2.12.10.

Let's take a pyramidal solid shown in Figure M.2.12.11 as an example and show how to use this method to construct chamfers in particular cases. Result of solid construction for symmetrical configuration of edges and symmetrical chamfer is shown in Figure M.2.12.12.

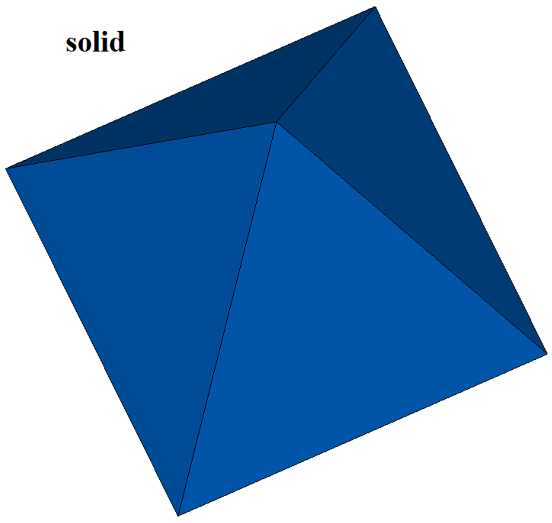


Fig. M.2.12.11.

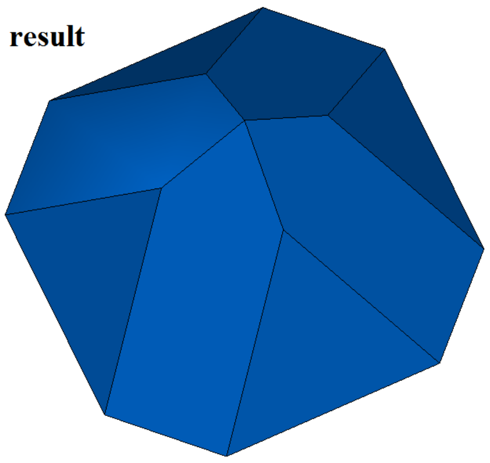


Fig. M.2.12.12.

It should be noted that if you construct a chamfer for four or more edges that meet in a single vertex, all edge surfaces should intersect in a single point. An example of symmetrical chamfer for seven edges that meet in a single vertex is shown in Figure M.2.12.13.

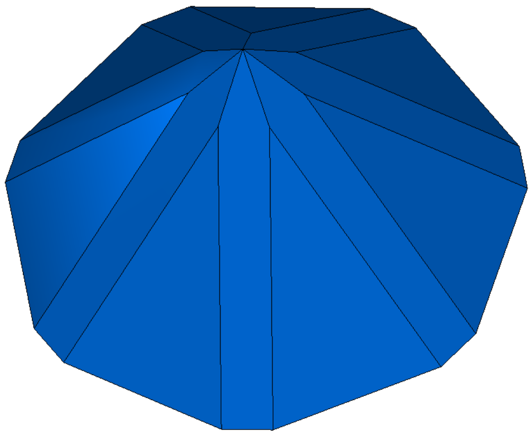


Fig. M.2.12.13.

When edge chamfer is constructed, the method adds MbFilletSolid constructor in the log of newly constructed solid. The constructor is declared in cr_chamfer_solid.h file.

test.exe test application processes solid edges using New ->Solid -> By Processing Edges -> Leg-Leg Chamfer, New ->Solid -> By Processing Edges -> Leg-Corner Chamber and New ->Solid -> By Processing Edges -> Corner-Leg Chamfer menu commands.

M.2.13. Constructing a Thin-Wall Solid

Method
MbResultType
ThinSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
RPArray<MbFace> & **outFaces**,
SweptValues & *params*,

const MbSNameMaker & names,
MbSolid *& **result**)
constructs a thin-wall solid by excluding specified faces from the original solid.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **outFaces** is a set of faces that should be excluded,
- *params* are construction parameters,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method excludes **outFaces** faces from **solid** original solid, it also "sets a predetermined thickness" for remaining faces. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. *params* parameter contains data on wall thickness in remaining faces, as well as data on closure of constructed **result** solid. The thickness of remaining faces may be equal to *params.thickness*1 in positive direction of normal to the face or *params.thickness*2 in the negative direction of normal to the face. If *params.shellClosed*=false, then a nonclosed solid will be constructed. names parameter is used to name the faces of the constructed solid. To execute the operation, **outFaces** to be deleted should not have smooth edges attached to remaining edges of the original solid at a shared perimeter.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

In Fig. M.2.13.1, you can see **solid** original solid and **outFaces** faces that are deleted.

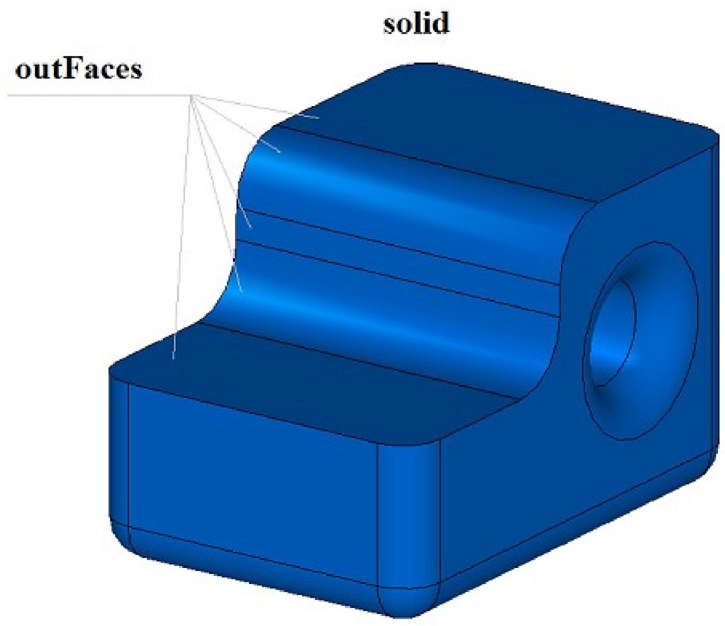


Fig. M.2.13.1.

In Fig. M.2.13.2, you can see **result** newly constructed thin-wall solid with the remaining faces thickened inside the original solid.

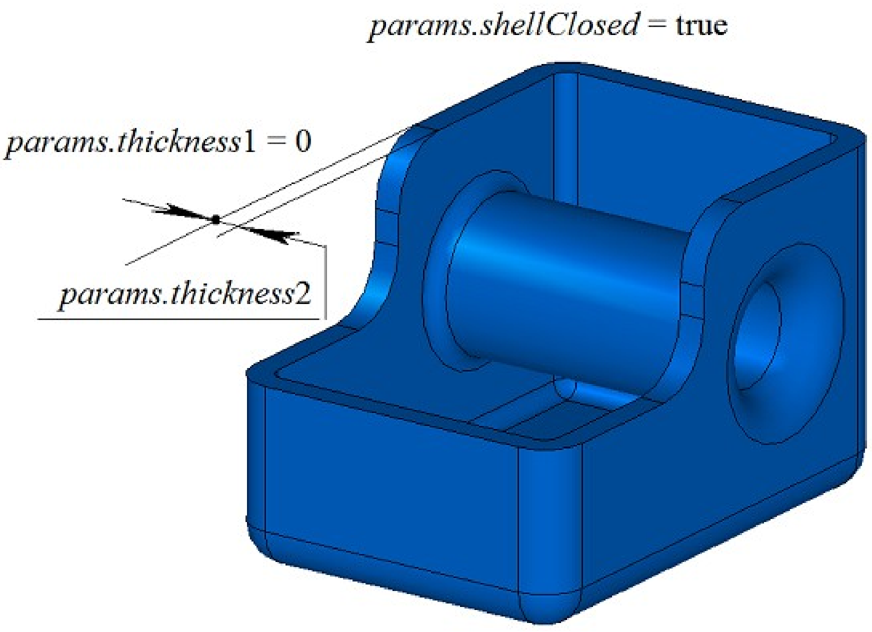


Fig. M.2.13.2.

In Fig. M.2.13.3, you can see newly constructed **result** thin-wall solid with remaining faces thickened outside the original solid.

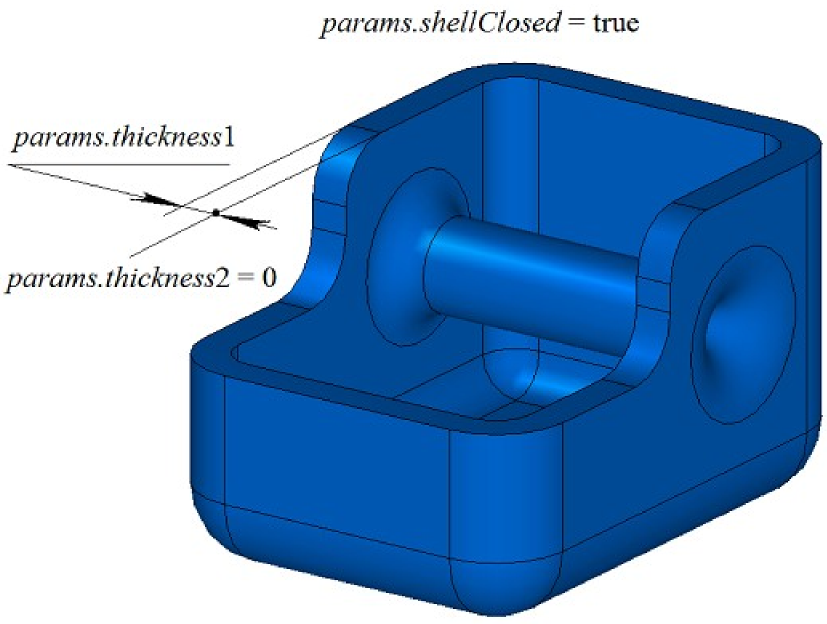


Fig. M.2.13.3.

In Fig. M.2.13.4, you can see **result** non-closed solid.

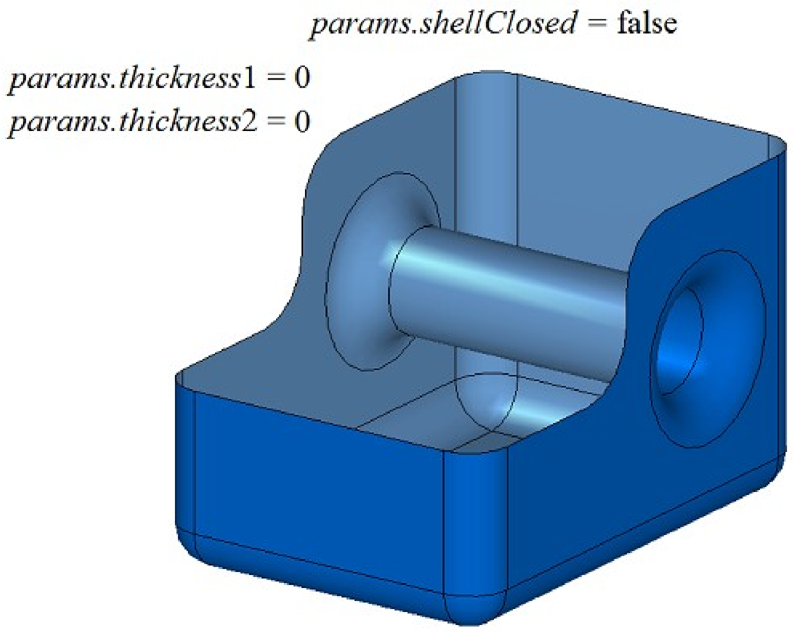


Fig. M.2.13.4.

In Fig. M.2.13.5, you can see a thin walled-solid that was constructed in case of empty set of faces that should be deleted.

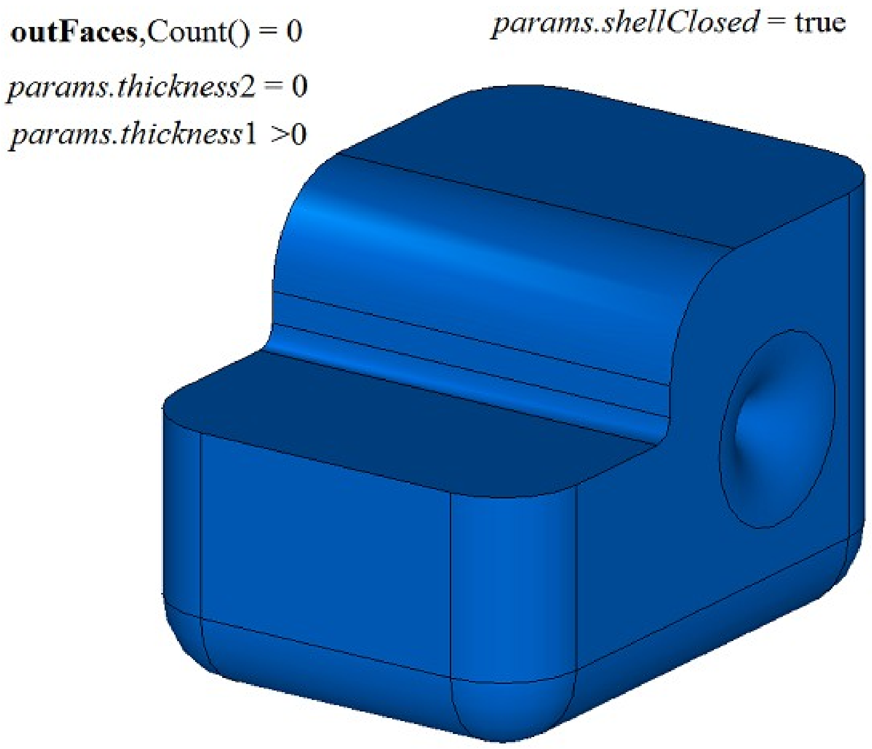


Fig. M.2.13.5.

ThinSolid method adds MbShellSolid constructor to the log of the newly constructed solid that contains all data required to execute the operation. MbShellSolid constructor is declared in cr_thin_shell_solid.h file.

test.exe test application constructs a thin-wall solid using New ->Solid -> By Processing Faces -> Uniform Thickening menu command.

M.2.14. Constructing a Thin-Wall Solid with Various Wall Thickness

Method
MbResultType
ThinSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
RPArray<MbFace> & **outFaces**,
RPArray<MbFace> & **offFaces**,
SArray<double> & *offDistances*,
SweptValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a thin-wall solid by excluding specified faces and setting various thickness of remaining faces in the original solid.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **outFaces** is a set of faces that should be excluded,
- **offFaces** is a set of faces for which individual thicknesses were set.
- *offDistances* is the set of individual thicknesses (it is synchronized with **offFaces**),
- *params* are construction parameters,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method deletes **outFaces** faces from **solid** original solid, it also "sets a predetermined thickness" for remaining faces. Face thickness may vary from face to face. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. **offFaces** parameter contains faces, for which individual values of *offDistances* thicknesses were set. **offFaces**[*i*] thickness will be set for *offDistances*[*i*] faces. Thickness of the remaining faces is defined by *params* parameter. *params* parameter contains data on closure of **result** solid, as well as data on wall thickness for the faces that should be kept and do not belong to **offFaces** set. The tickness of remaining faces may be equal to *params.thickness*1 in positive direction of normal to the face or *params.thickness*2 in the negative direction of normal to the face. names parameter is used to name the faces of the constructed solid. To execute the operation, **outFaces** to be deleted should not have smooth edges attached to remaining edges of the original solid at a shared perimeter.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces

In Fig. M.2.14.1, you can see **solid** original solid, **outFaces** faces that should be deleted and **offFaces** for which individual *offDistances* thickness values were set.

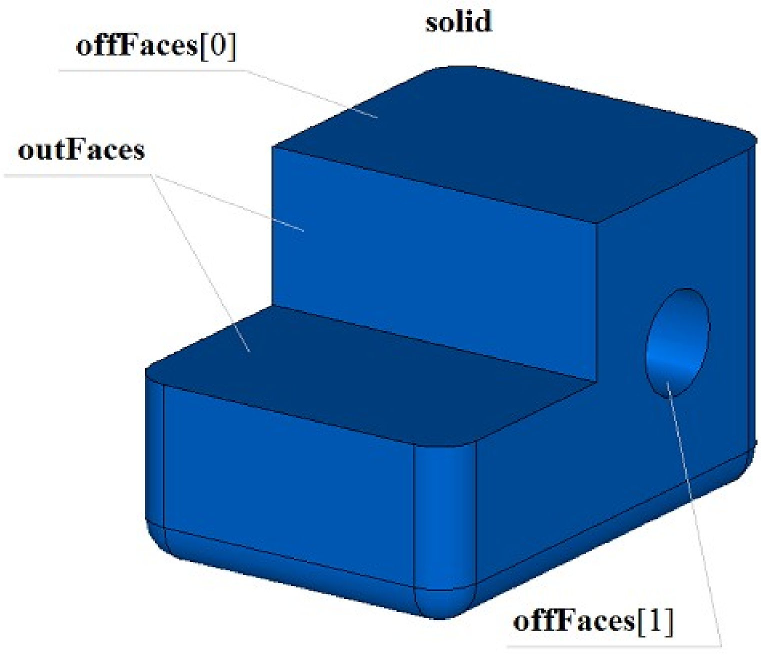


Fig. M.2.14.1.

In Fig. M.2.14.2, you can see **result** constructed solid with kept and newly constructed faces.

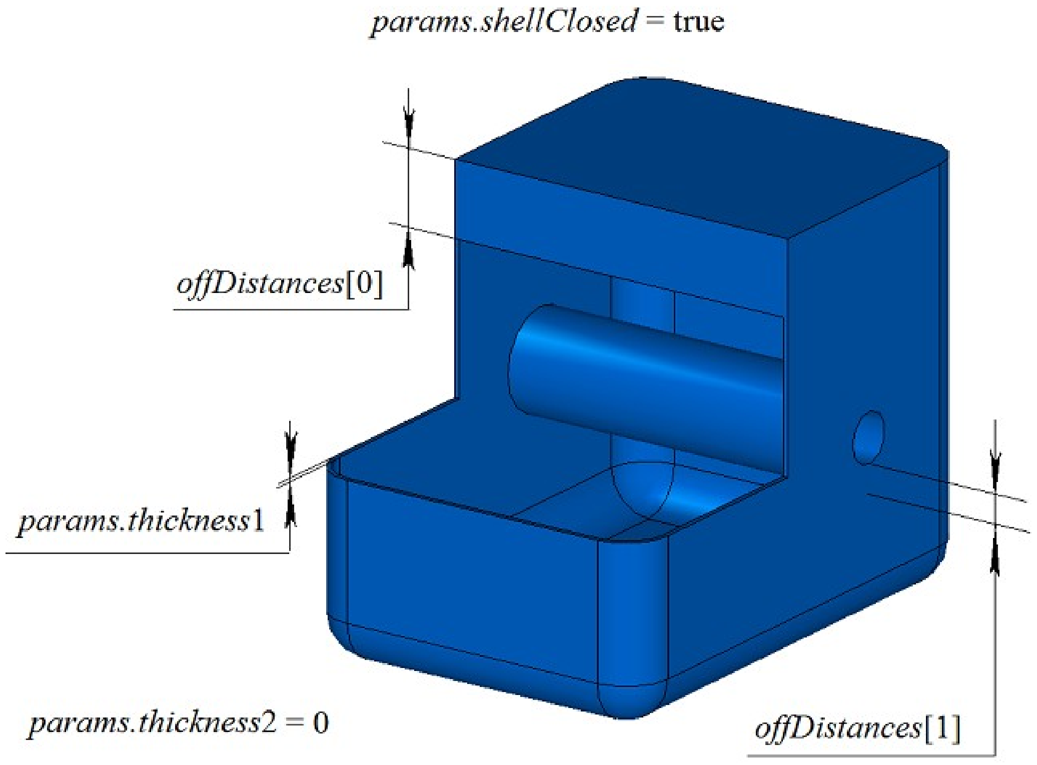


Fig. M.2.14.2.

To execute the operation, each of the **offFaces** faces should not have smooth edges attached to the remaining edges of the original solid along the shared perimeter if such faces have different thickness.

ThinSolid method adds MbShellSolid constructor to the log of the newly constructed solid that contains all data required to execute the operation. MbShellSolid constructor is declared in cr_thin_shell_solid.h file.

test.exe test application constructs a thin-wall solid using New ->Solid -> By Processing Faces -> With Uneven Thickness menu command.

M.2.15. Constructing Solids by Thickening the Surface

Method
MbResultType
ThinSolid (const MbSurface & **surface**,
bool *faceSense*,
SweptValues & *params*,
const MbSNameMaker & names,
SimpleName name,
MbSolid *& **result**)
constructs a solid by defining the thickness of the specified surface.

Input parameters of the method are as follows:

- **surface** is the specified surface,
- *faceSense* determines orientation of normal to the surface at the face of the constructed solid,
- *params* are construction parameters,
- names is face namer,
- name is operation name.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method constructs a face based on **surface** surface, and then a solid is constructed by "giving a thickness to this face". *faceSense* indicates whether the direction of normal to the **surface** coincides with the direction of normal to the face. New thickness of the face is determined by *params* parameter. *params* parameter contains wall thickness data for constructed **result** solid. Wall thickness may be equal to *params.thickness*1 (for positive direction of the normal to the face) or *params.thickness*2 (for negative direction of the normal to the face). names and name parameters provide naming of the faces of the newly constructed solid.

In Fig. M.2.15.1, you can see **surface** original surface.

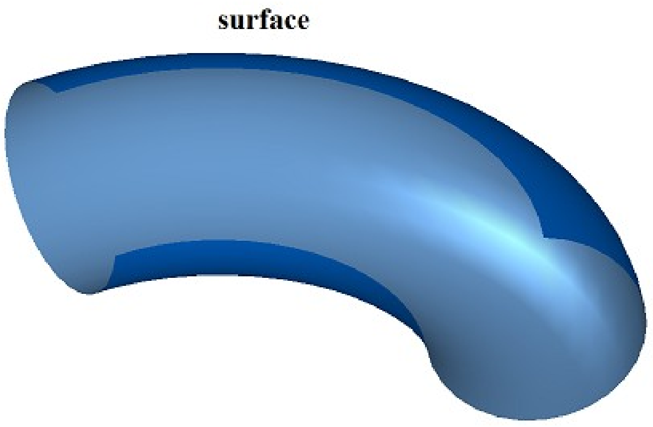


Fig. M.2.15.1.

In Fig. M.2.15.2, you can see constructed **result** solid.

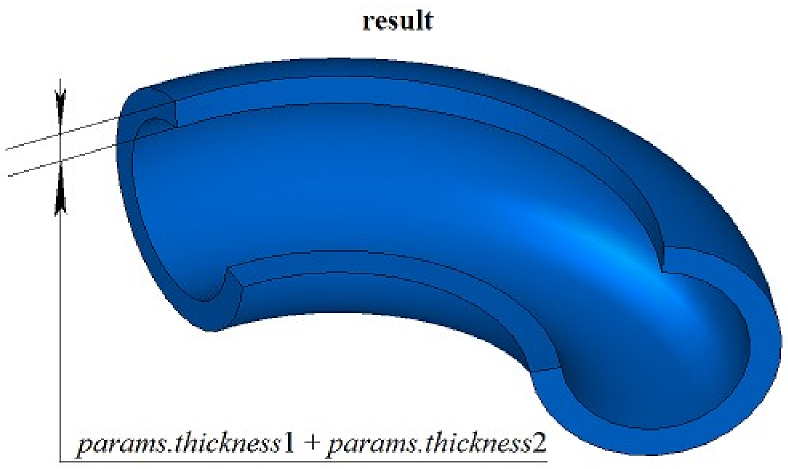


Fig. M.2.15.2.

ThinSolid method adds MbShellSolid constructor to the log of the newly constructed solid that contains all data required to execute the operation. MbShellSolid constructor is declared in cr_thin_shell_solid.h file.

test.exe test application constructs a thin-wall solid using New ->Solid -> Based on Surface -> Thickening menu command.

M.2.16. Constructing a Mirror Solid

Method
MbResultType
MirrorSolid (const MbSolid & **solid**,
const MbPlacement3D & **place**,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a mirror copy of the original solid in relation to the given plane.

Input parameters of the method are as follows:

- **solid** is the original solid,
- **place** is local coordinate system, its XY plane is a mirror plane,
- names is cut face namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method constructs a mirror copy of **solid** original solid in relation to XY plane of specified **place** local coordinate system. names parameter is used to name the faces of the constructed solid.

In Fig. M.2.16.1, you can see **solid** original solid and **place** symmetry plane.

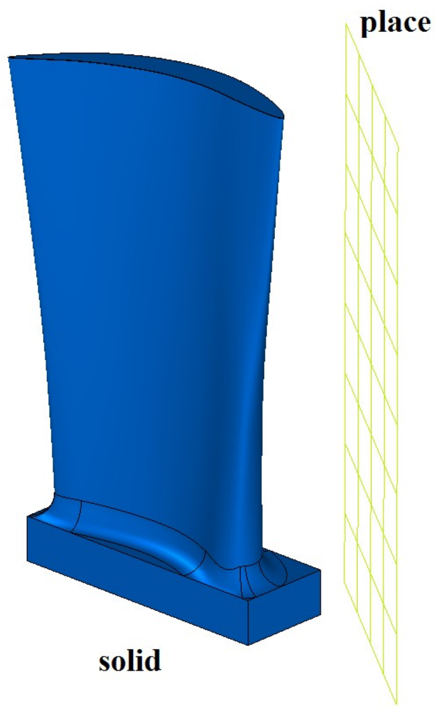


Fig. M.2.16.1.

In Fig. M.2.16.2, you can see **solid** original solid and **result** constructed solid.

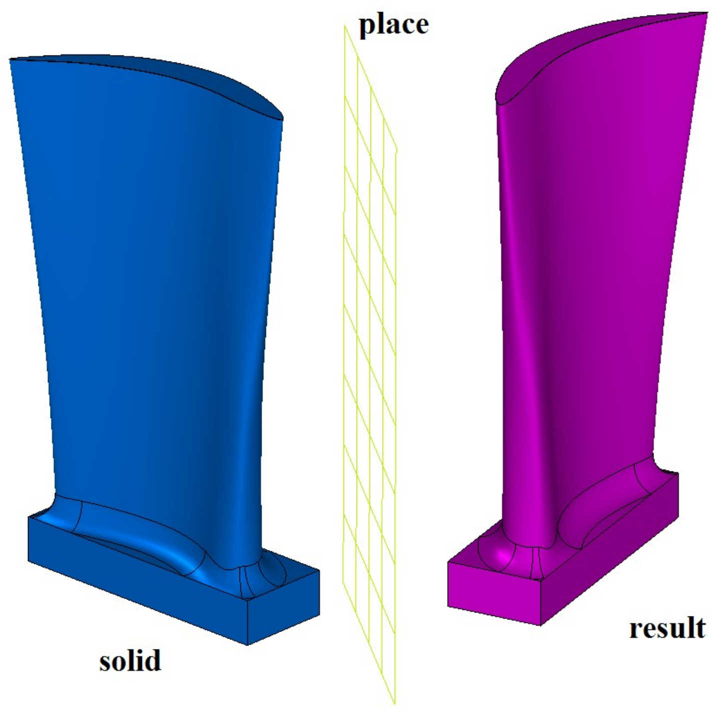


Fig. M.2.16.2.

MirrorSolid method adds MbSymmetrySolid constructor in a log of the newly constructed solid that contains all data required to execute the operation. MbSymmetrySolid constructor is declared in cr_symmetry_solid.h file.

test.exe test application constructs a symmetrical solid using New ->Solid -> Based on Solid ->

Symmetrical menu command.

M.2.17. Boolean Operation on Solids and Set of Solids

Method
MbResultType
UnionResult (MbSolid * **solid**,
MbeCopyMode *sameShell*,
RPArray<MbSolid> & **solids**,
MbeCopyMode *sameShells*,
OperationType *oType*,
bool *checkIntersect*,
bool *mergeFaces*,
const MbSNameMaker & names,
bool *isArray*,
MbSolid *& **result**,
RPArray<MbSolid> * **notGluedSolids** = NULL)

merges a given set of solids and executes a determined Boolean operation on the original solid if it was specified.

Input parameters of the method are as follows:

- **solid** is the original solid (it may be equal to zero),
- *sameShell* is a version of original solid copying method,
- **solids** is the set of solids,
- *sameShells* is copying method for the solids in the set,
- *oType* is Boolean operation type: bo_Union means merging of the solids,
  bo_Intersect means intersection of the solids,
  bo_Difference means subtraction of the solids,
- *checkIntersect* is a flag used to check intersection for a set of solids (false means "no check"),
- *mergeFaces* indicates whether similar faces should be merged,
- names is face namer,
- *isArray* is regularity flag for the set of solids.

Results of the method are **result** constructed solid and **notGluedSolids** set of solids that were not used in the operation (it may be equal to zero).

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method is a version of **BooleanSolid** Boolean operation that accelerates execution when the same Boolean operation with **solid** solid is applied to many other solids. First, this method merges the solids from **solids** set and creates a temporary solid, then it executes specified *oType* Boolean operation for **solid** solid on this temporary solid. **solids** solids may not overlap with each other. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. *sameShells* parameter controls transfer of faces, edges and vertices from **solids** set of solids to **result** resulting solid. *checkIntersect* and *isArray* parameters control construction of the temporary solid for **solids** set of solids. *mergeFaces* parameter controls merging of similar faces. names parameter is used to name the faces of the constructed solid.

sameShell (*sameShells*) parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

OperationType *oType* parameter defines Boolean operation type; it takes one of the following three values: bo_Union, bo_Intersect, bo_Difference. If *oType*=bo_Union, then the method merges **solid** solid and **solids** set of solids; if *oType*=bo_Intersect, then the method intersects **solid** solid and **solids** set of solids; if *oType*=bo_Difference, then the method subtracts **solids** set of solids from **solid** solid.

checkIntersect and *isArray* parameters are used to accelerate **UnionResult** method.

checkIntersect parameter gives a command to check intersection of solids included in **solids** set with each other. If *checkIntersect*==true, then all intersecting solids from **solids** set are merged during construction using a Boolean operation. Otherwise, all faces of solids from the original set are copied to the newly constructed solid. Despite the value of *checkIntersect* parameter, all nonintersecting **solids** solids transfer their faces to newly constructed temporary solid.

mergeFaces parameter permits to merge similar faces in **result** solid that was constructed or to keep them separated. Influence of *mergeFaces* parameter is shown in Figures M.2.17.2 and M.2.17.3. If *mergeFaces*==false, then similar faces are not merged.

isArray is used only if *checkIntersect*==true and it informs on regularity of **solids** set of solids. If *isArray*==true, then the solids of the set are located in nodes of rectangular or circular grid, and positions of the solids are specified in face names.

notGluedSolids parameter contains solids that were not used in the operation because it is impossible to merge them with the common temporary solid.

In Fig. M.2.17.1, you can see **solid** original solid and **solids** set of solids.

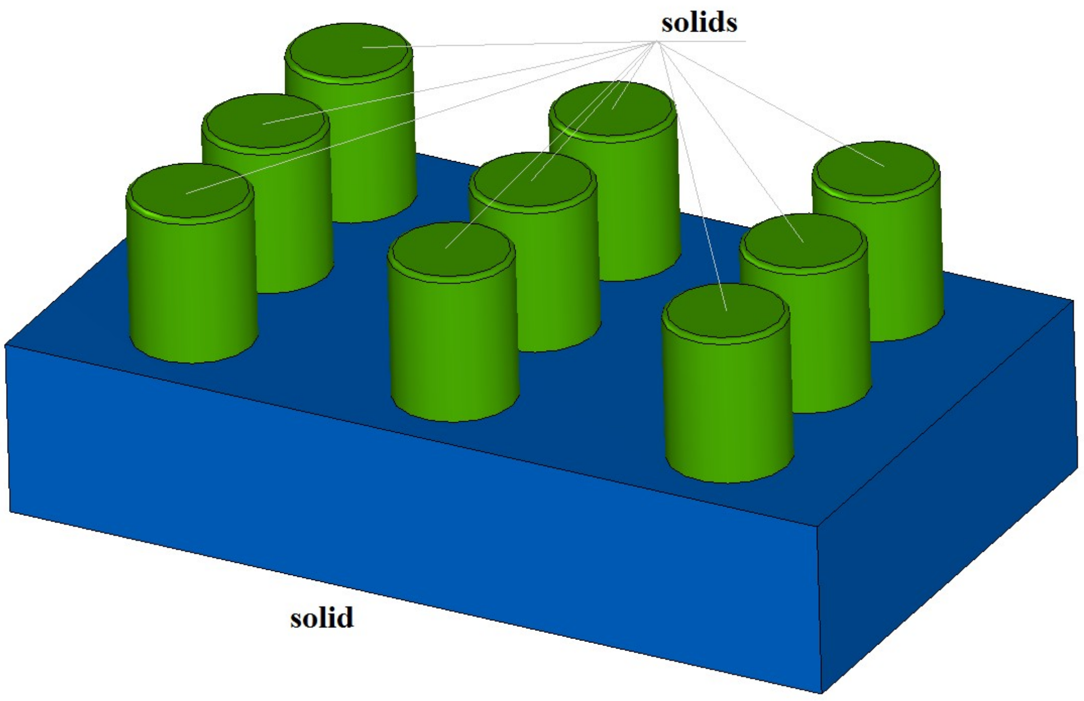


Fig. M.2.17.1.

In Fig. M.2.17.2, you can see **result** solid, which is the result of gluing **solids** solids to **solid** solid. In this case, *checkIntersect* parameter may be equal to false, as **solids** solids do not intersect with each other.

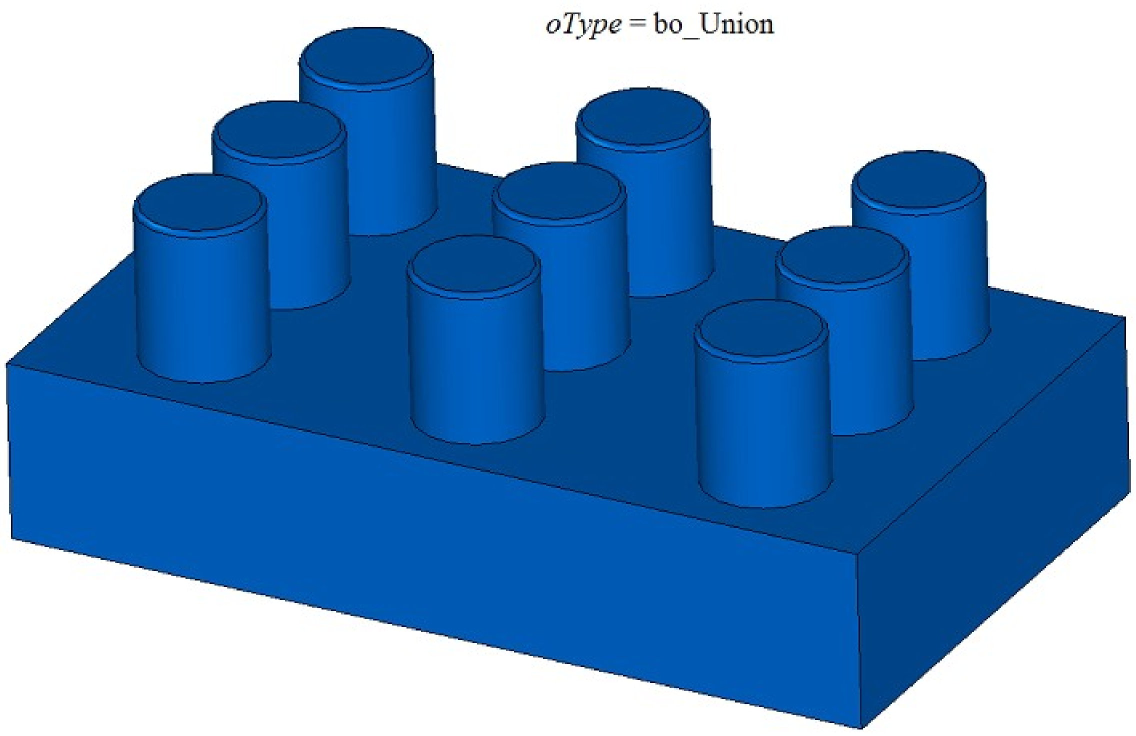


Fig. M.2.17.2.

In Fig. M.2.17.3, you can see **solid** original solid and **solids** set of solids.

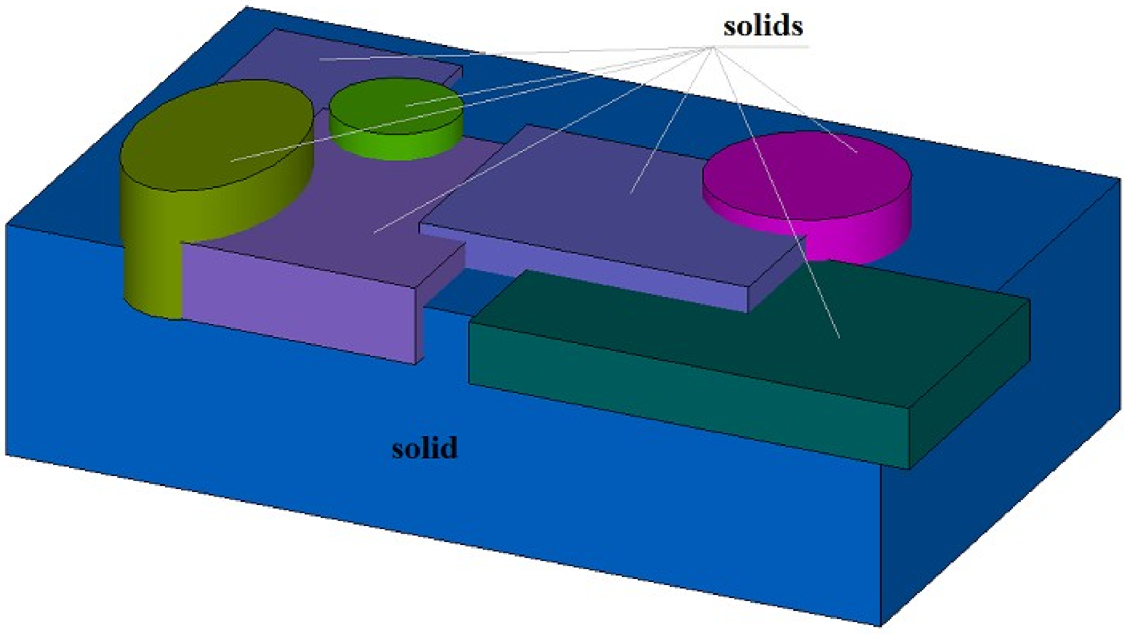


Fig. M.2.17.3.

In Fig. M.2.17.4, you can see **result** solid that was constructed by subtracting **solids** solids from **solid** solid, if the method was used with *mergeFaces*== true. In this case, *checkIntersect* parameter should be equal to true, as **solids** solids intersect with each other.

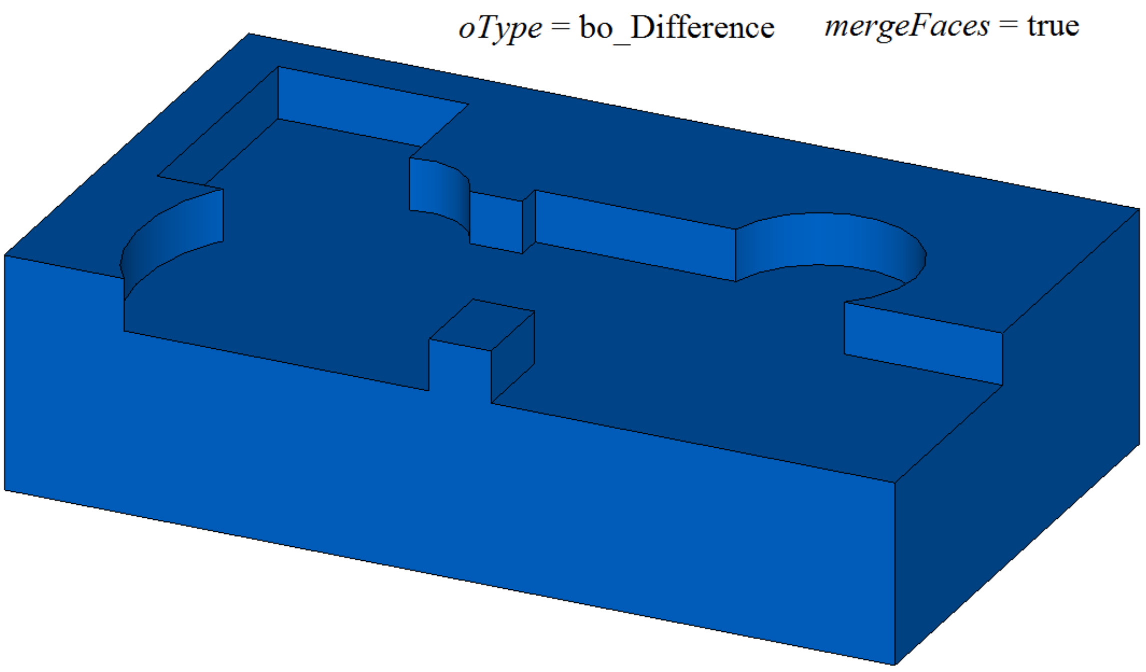


Fig. M.2.17.4.

In Fig. M.2.17.5, you can see **result** solid that was constructed by subtracting **solids** solids from **solid** if the method was used with *mergeFaces* ==false.

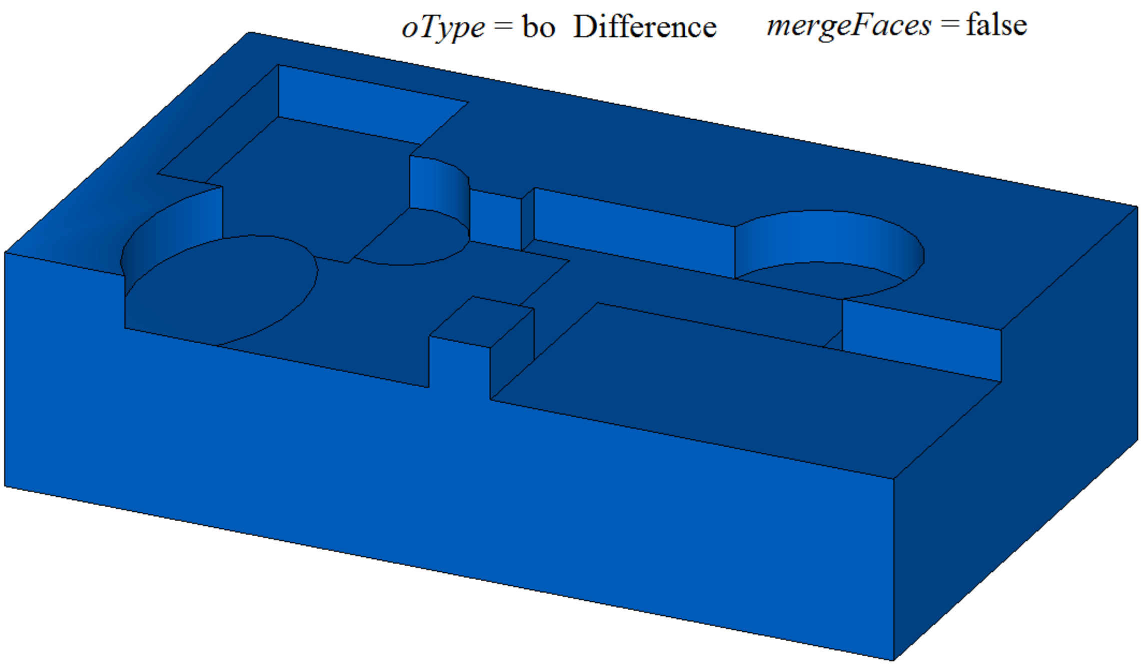


Fig. M.2.17.5.

In Fig. M.2.17.6, you can also see faces of the resulting solid (that was constructed by subtracting **solids** solids from **solid** solid (given in Figure M.2.17.5)) colored in the colors of the original solids. The shape of faces permits you to determine the sequence how **solids** solids were included into the temporary solid: a solid leaves a more complete impress if it was included in the temporary solid before other solids.

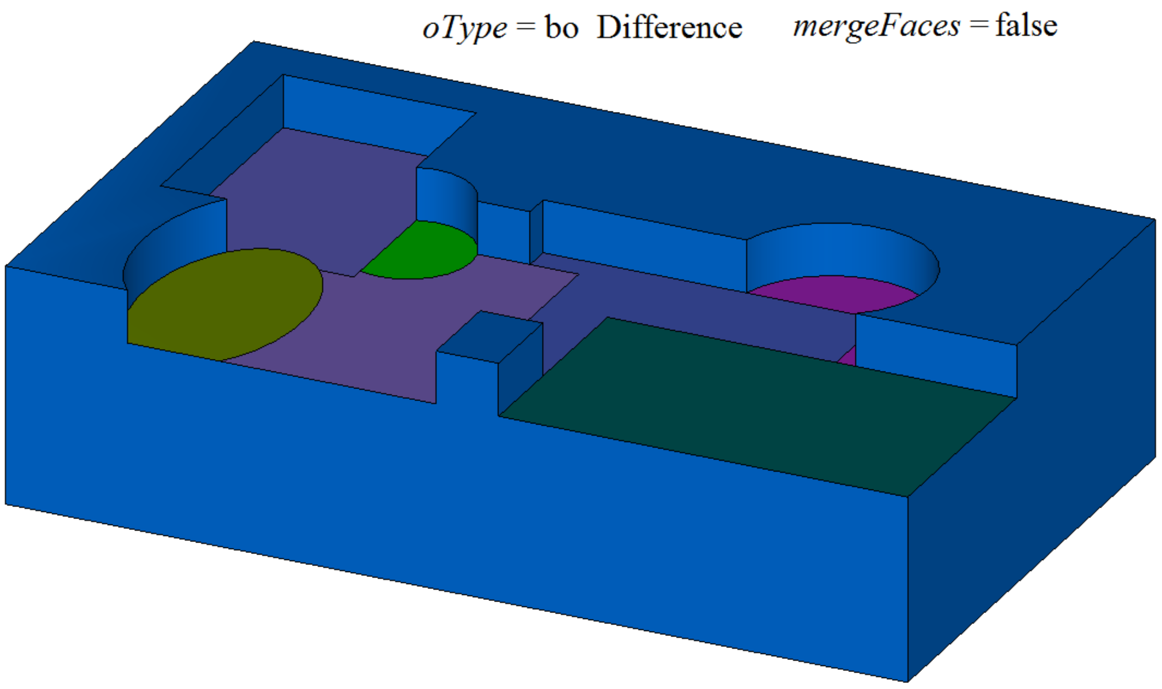


Fig. M.2.17.6.

UnionResult method adds MbUnionSolid constructor in the log of the newly constructed solid that contains all data required to execute the operation. MbUnionSolid constructor is declared in cr_union_solid.h file.

test.exe test application executes solid Boolean operations on a set of solids using New ->Solid -> Attach to Other Solid -> In a Set of Solids, New ->Solid -> Cut from Other Solid -> In a Set of Solids, New ->Solid -> Intersection with Other Solid -> In a Set of Solids menu commands.

M.2.18. Merging a Set of Solids

Method
MbResultType
UnionSolid (RPArray<MbSolid> & **solids**,
MbeCopyMode *sameShells*,
bool *checkIntersect*,
const MbSNameMaker & names,
bool *isArray*,
MbSolid*& **result**,
RPArray<MbSolid> * **notGluedSolids** = NULL)
merges solids of the specified set.

Input parameters of the method are as follows:

- **solids** is the set of solids,
- *sameShells* is copying method for the solids in the set,
- *checkIntersect* is a flag used to check intersection for a set of solids (false means "no check"),
- names is face namer,
- *isArray* is a flag defining whether the set of solids is regular.

Results of the method are **result** constructed solid and **notGluedSolids** set of solids that were not used in the operation (it may be equal to zero).

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

The method is similar to **UnionResult** method, if **solid**=0, *sameShell*=*cm_Same*, *oType*=bo_Base and *mergeFaces*=true. This method accelerates execution when it is required to merge many solids. The method merges **solids** solids and constructs **result** solid; the solids may not intersect with each other. *sameShells* parameter controls transfer of faces, edges and vertices from **solids** set of solids to **result** resulting solid. *checkIntersect* and *isArray* parameters control construction of the temporary solid for **solids** set of solids. names parameter is used to name the faces of the constructed solid.

sameShells parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces

checkIntersect and *isArray* parameters are used to accelerate **UnionResult** method.

checkIntersect parameter gives a command to check intersection of solids included in **solids** set with each other. If *checkIntersect*==true, then a Boolean operation is executed to merge all intersecting solids included in **solids** set. Otherwise, all faces of solids from the original set are copied to the newly constructed solid. Despite the value of *checkIntersect* parameter, all non-intersecting **solids** solids transfer their faces to newly constructed temporary solid.

isArray is used only if *checkIntersect*==true and it informs on regularity of **solids** set of solids. If *isArray*==true, then the solids of the set are located in nodes of rectangular or circular grid, and positions of the solids are specified in face names.

notGluedSolids parameter contains solids that were not used in the operation because it was impossible to merge them.

In Fig. M.2.18.1, you can see **solids** original solids.

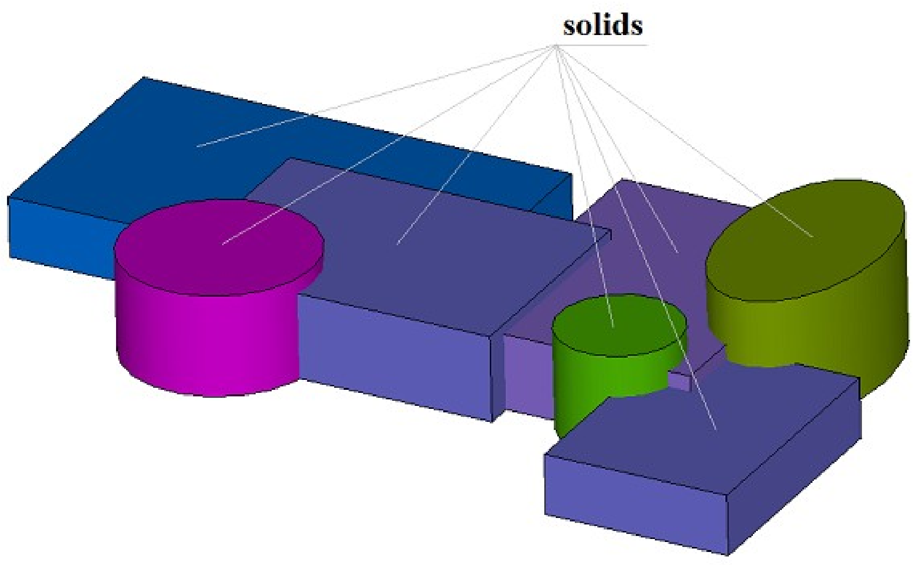


Fig. M.2.18.1.

In Fig. M.2.18.2, you can see **result** solid constructed by merging **solids** solids. In this case, *checkIntersect* parameter should be equal to true, as **solids** solids intersect with each other.

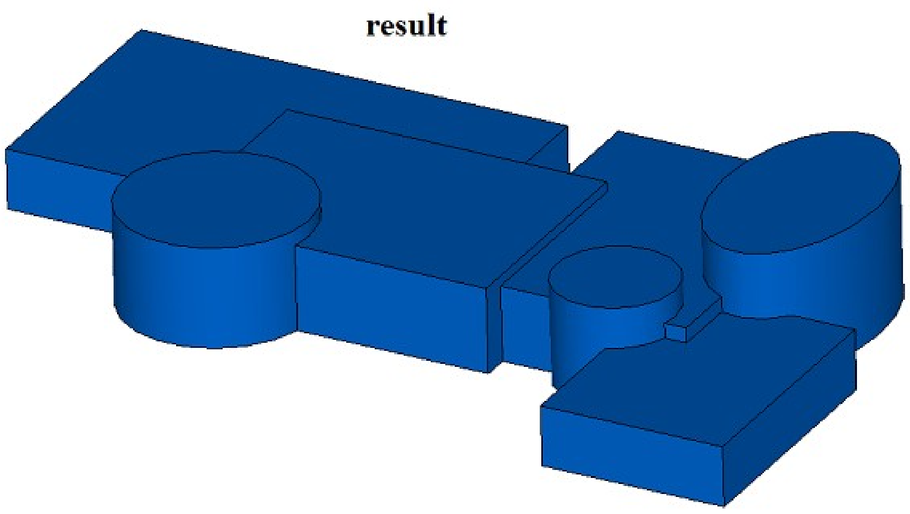


Fig. M.2.18.2.

Method
MbResultType
UnionSolid (const RPArray<MbSolid> & **solids**,
const MbSNameMaker & names,
MbSolid *& **result**)
is a simplified version of discussed method having the same name, the two methods coincide if *sameShells*=*cm_Same*, *checkIntersect*==false, *isArray*==false and **notGluedSolids**==NULL. The latter method does not check or construct anything, rather it simply composes **result** solid using all faces of **solids** solids. So original solids and a newly constructed solid have the same faces.

UnionSolid methods add MbUnionSolid constructor in the log of the newly constructed solid that contains all data required to execute the operation. MbUnionSolid constructor is declared in cr_union_solid.h file.

test.exe test application executes solid Boolean operations on a set of solids using New ->Solid -> Based on Solid -> Set of Solids menu command.

M.2.19. Divide a Solid to Disconnected Parts

Method
unsigned int
DetachParts (MbSolid & **solid**,
RPArray<MbSolid> & **parts**,
bool *sort*,
const MbSNameMaker & names)
divides a solid to disconnected parts.

Input parameters of the method are as follows:

- **solid** is the original solid,
- sort is a flag used to sort disconnected parts in descending order by the larges dimension,
- names is face namer.

Input parameters of this method are **solid** original solid and **parts** set of its disconnected parts.

This method returns the number of disconnected parts.

The method is declared in action_solid.h file.

After subtracting **solid2** solid from **solid2** solid shown in Figure M.2.19.1, the result of **solid** Boolean operation would consist of several topologically disconnected parts (Figure M.2.19.2), although they would behave as a single object. The method permits to divide **solid** solid that consists of several topologically disconnected parts to individual solids. One part stays in **solid** original solid, and all other parts are sent to received **parts** container.

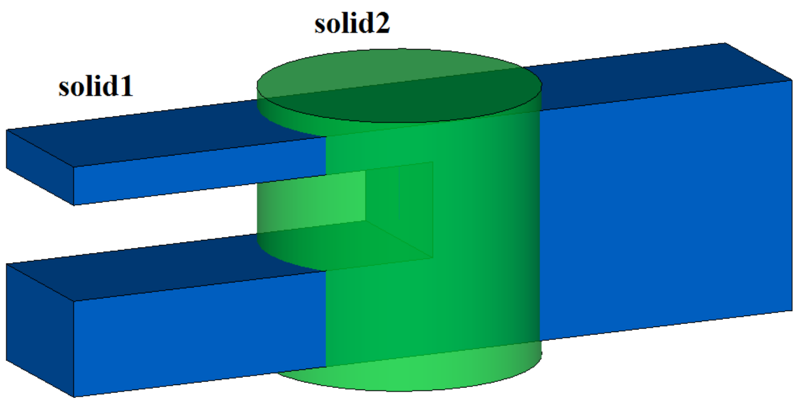


Fig. M.2.19.1.

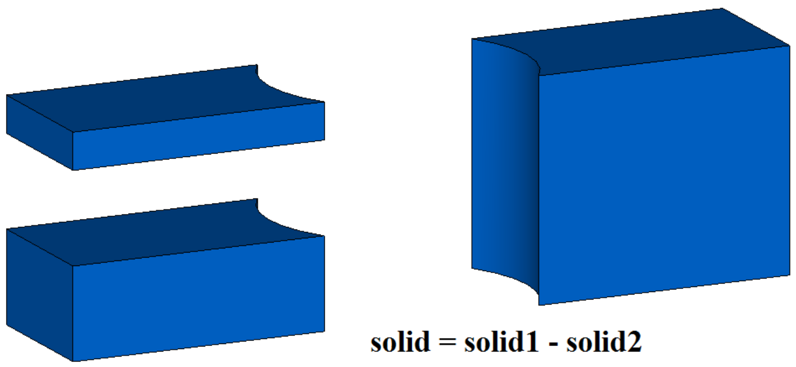


Fig. M.2.19.2.

If sorting flag sort==true, then the part with largest dimensions will remain in the original solid, and separated parts will be sorted by dimensions in descending order as shown in Figure M.2.19.3. Otherwise, the part topologically related to the first face will remain in the original solid, and separated parts will be sorted by the number of initial face in the original solid.

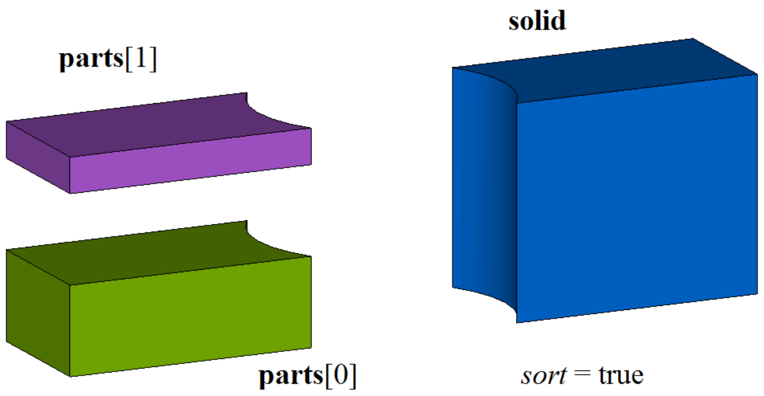


Fig. M.2.19.3.

names parameter provides naming of faces in the created solid and operation versioning.

Method

unsigned int
CreateParts (const MbSolid & **solid**,
RPArray<MbSolid> & **parts**,
const MbSNameMaker & names)
executes the same operations as the previous method, the difference is that it does not change **solid** original solid and adds all topologically disconnected parts of the original solid to **parts** solids as shown in Figure M.2.19.4.

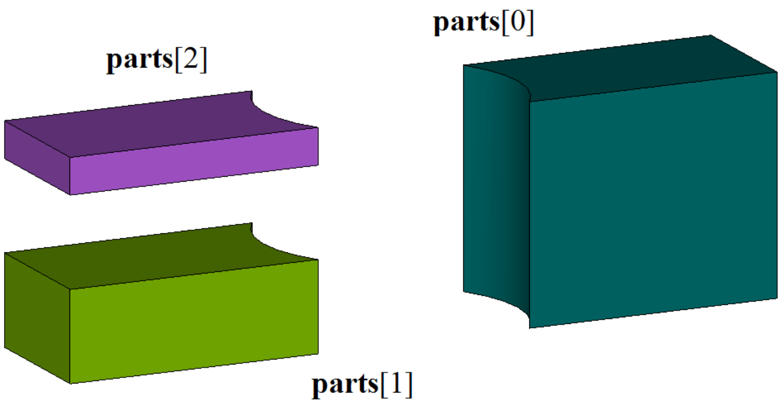


Fig. M.2.19.4.

parts solids will be constructed on the same faces as **solid** original solid.

DetachParts and **CreateParts** methods add MbDetachSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbDetachSolid constructor is declared in cr_detach_solid.h file.

test.exe test application executes solid Boolean operations on a set of solids using Modify ->Solid or Shell->Divide Parts menu command.

M.2.20. Separation of Disconnected Parts

Method
MbResultType
ShellPart (const MbSolid & **solid**,
size_t *id*,
const MbPath & path,
const MbSNameMaker & names,
MbPartSolidIndices & partIndices,
MbSolid * & **result**)
creates a separate solid from a specified part of the original solid that falls apart.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *id* is the number of the selected part of the original solid,
- path is the identifier of the selected part of the original solid in the model,
- names is face namer.
- partIndices are indices of solid parts.

Output parameters of this method are **result** constructed solid and indices of solid parts.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration. The method is declared in action_solid.h file.

The method constructs a solid from the specified part of original solid. The original solid should consist of separate parts. In Fig. M.2.20.1, you can see the result of Boolean operation that subtracts solids shown in Figure M.2.19.1. The resulting solid consists of several topologically separated parts. This method permits to create a solid keeping only one of topologically separated parts of the original solid.

id indicates part number of **solid** original solid. path parameter contains path to the solid part. In a simple case, the path to solid part contains part number in *id* original solid.

In Fig. M.2.20.1, you can see an original solid that consists of several topologically separated parts.

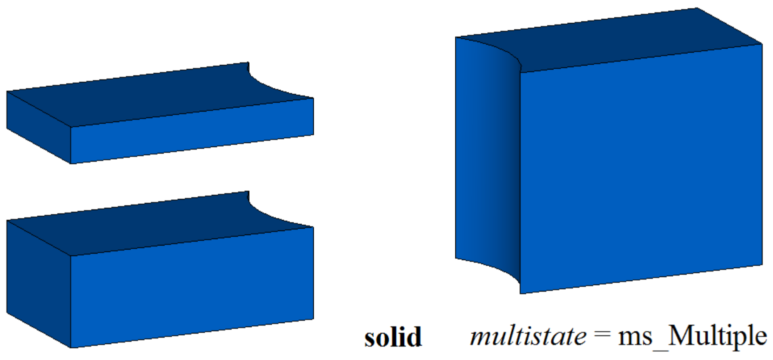


Fig. M.2.20.1.

In Fig. M.2.20.2, you can see a newly constructed solid consisting of one selected part of the original solid.

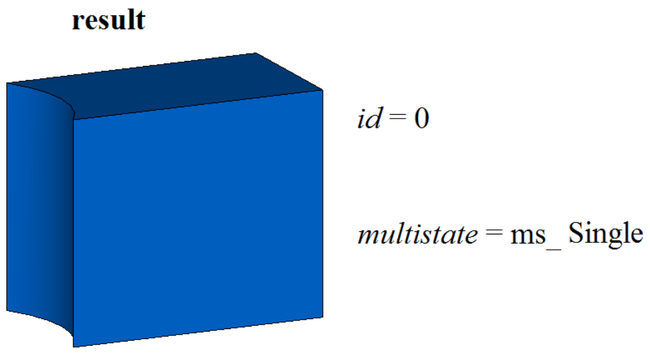


Fig. M.2.20.2.

result solid will be constructed on the same faces as **solid** original solid.

ShellPart method adds MbDetachSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbDetachSolid constructor is declared in cr_detach_solid.h file.

test.exe test application executes solid Boolean operations on a set of solids using New ->Solid -> Based on Solid -> Part of Solids Set menu command.

M.2.21. Splitting Solid Faces

Method
MbResultType
SplitSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const RPArray<MbSpaceItem> & **items**,
bool *same*,
RPArray<MbFace> & **faces**,
const MbSNameMaker & names,
MbSolid *& **result**)
splits specified solid faces with spatial curves, surfaces and shells.

Input parameters of the method are as follows:

- **solid** is the original solid,

- *sameShell* is a version of original solid copying method,
- **items** are spatial elements that split the faces,
- *same* indicates whether original spatial elements (*true*) or theirs copies (*false*) should be used,
- **faces** is a set of splitted faces,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.The method is declared in action_solid.h file.

The method splits specified **faces** of **solid** original solid using **items** 3D objects, if specified faces intersect with **items** objects. Curves, surfaces or a solid may be used as **items** objects. To execute the operation, the cutting objects should fully intersect with the specified faces of the original solid. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. *same* parameter controls copying of cutting objects. names parameter is used to name the faces of the constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

In Fig. M.2.21.1, you can see **solid** original solid, **faces** that should be split and **items**[0] cutting surface.

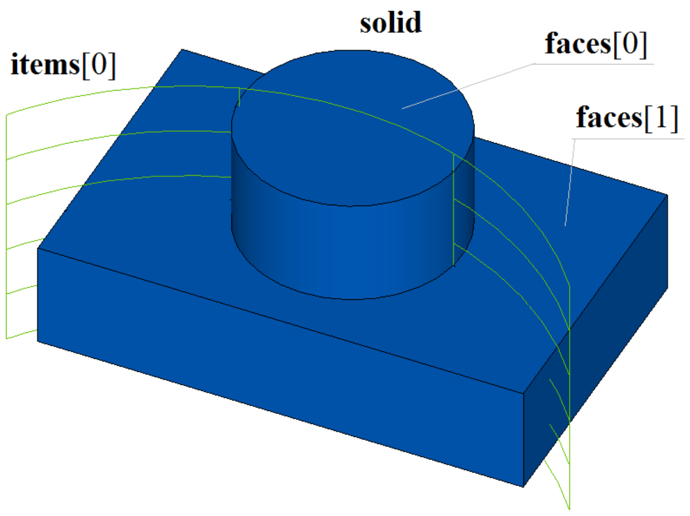
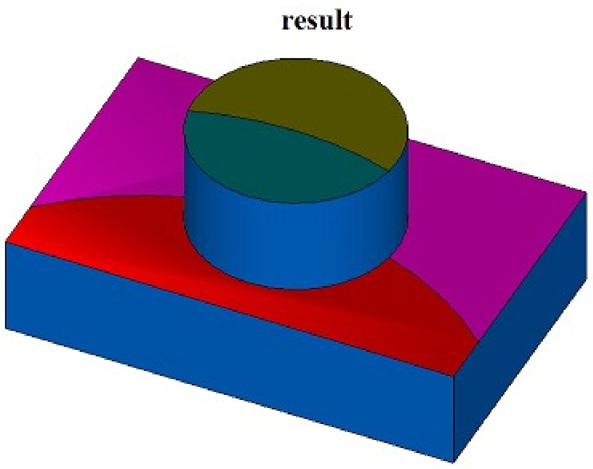


Fig. M.2.21.1.

In Fig. M.2.21.2, you can see newly constructed **result** solid with splitted specified faces. New edges return true to **IsSplit** query. In Fig. M.2.21.3, splitted faces of the constructed solid are painted in different colors.



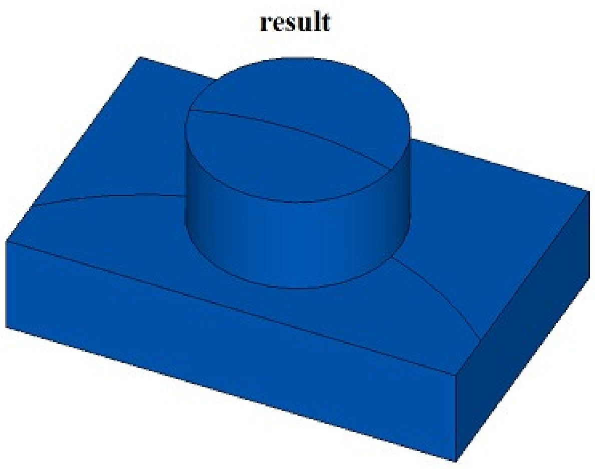


Fig. M.2.21.2. *Fig. M.2.21.3.*

Method
MbResultType
SplitSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **place**,
MbeSenseValue *type*,
const RPArray<MbContour> & **contours**,
bool *same*,
RPArray<MbFace> & **faces**,
const MbSNameMaker & names,
MbSolid *& **result**)
executes the same actions as the method considered above, the difference is that instead of using **items** splitting objects, **solid** solid faces are split by surfaces constructed by extruding two-dimensional **contours** located in XY plane of **place** local coordinate system. The contours are extruded in direction of **place.axis.Z** of the local coordinate system; extrusion length should provide a complete intersection with the original solid.

SplitSolid methods add MbSplitShell constructor in the log of the newly constructed solid that contains all data required to execute the operation. MbSplitShell constructor is declared in cr_split_shell.h file.

test.exe test application splits specified faces of the solid using New ->Solid -> By Processing Faces -> By Splitting Face menu command.

M.2.22. Constructing a Hole, Pocket or Slot in a Solid

Method
MbResultType
HoleSolid (MbSolid * **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **place**,
const HoleValues & *parameters*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a hole, a pocket or a cam slot in a solid.

Input parameters of the method are as follows:
- **solid** is the original solid (it may be equal to zero),
- *sameShell* is solid copying method,
- **place** is local coordinate system used to position a cutting tool,
- *parameters* are construction parameters,
- names is face namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

This method constructs an auxiliary solid in the form of deleted object for a hole, a pocket or a slot. If **solid** original solid is specified, then the method returns the difference between the original solid and the auxiliary solid. If the original solid is not specified (**solid**==0), then the method returns the auxiliary solid. To execute the operation, auxiliary solid should intersect with the original solid. *parameters* parameter defines shape of hole, pocket or slot. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. names parameter is used to name the faces of the constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode intersection is described in item O.7.9. Copying a Set of Faces

Construction is executed in **place** local coordinate system taking into account *parameters.placeAngle* and *parameters.azimuthAngle* rotation angles. *parameters.placeAngle* parameter defines the angle of rotation of

place local coordinate system with respect to **place.axisY** axis. *parameters.azimuthAngle* parameter defines the angles of rotation of **place** local coordinate system with respect to **place.axisZ** axis. *parameters*.**surface** surface may be not given. If *parameters*.**surface** is not equal to zero, then this surface is used to properly handle an inlet to a hole, pocket or slot.

In Fig. M.2.22.1, you can see data used for construction and parameters inheritance scheme from HoleValues abstract class.

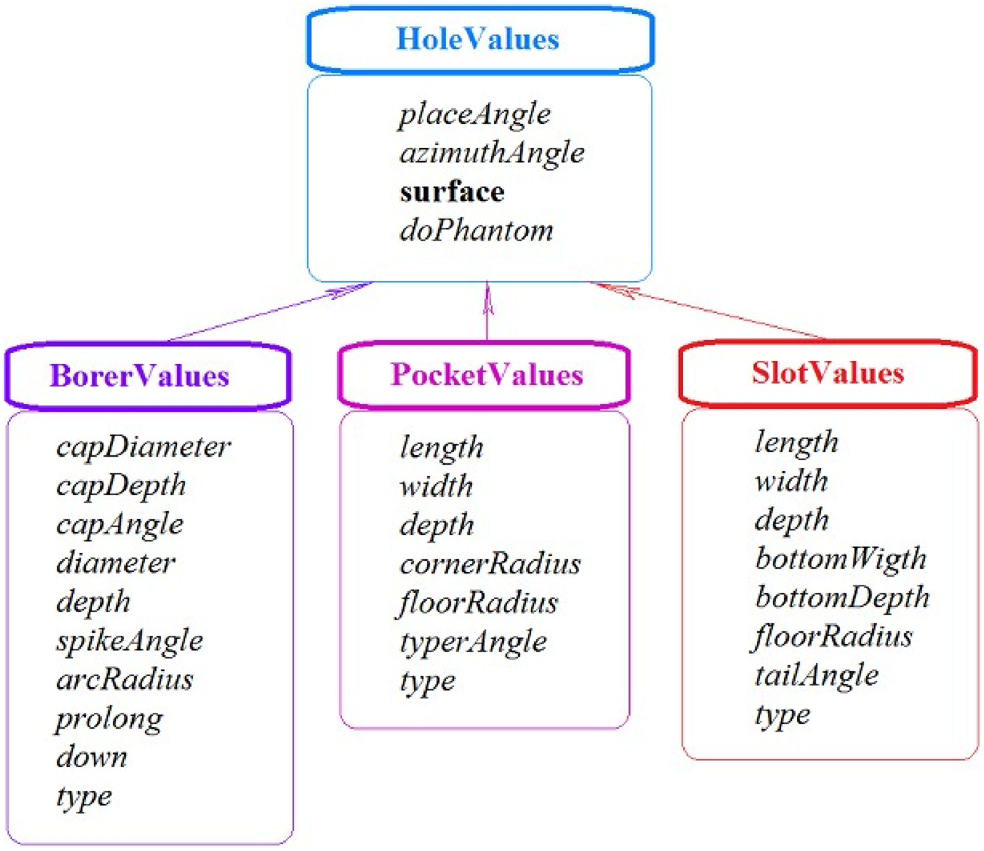
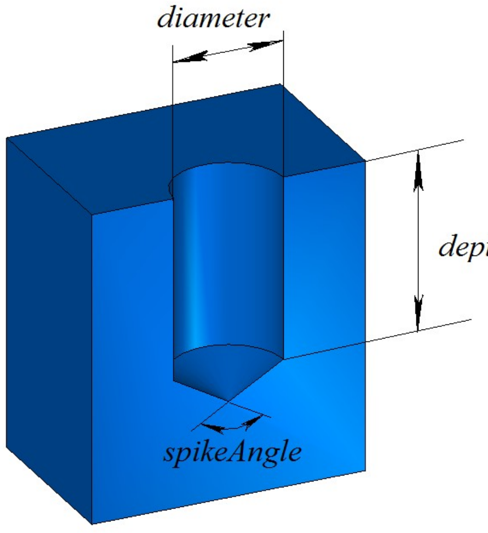
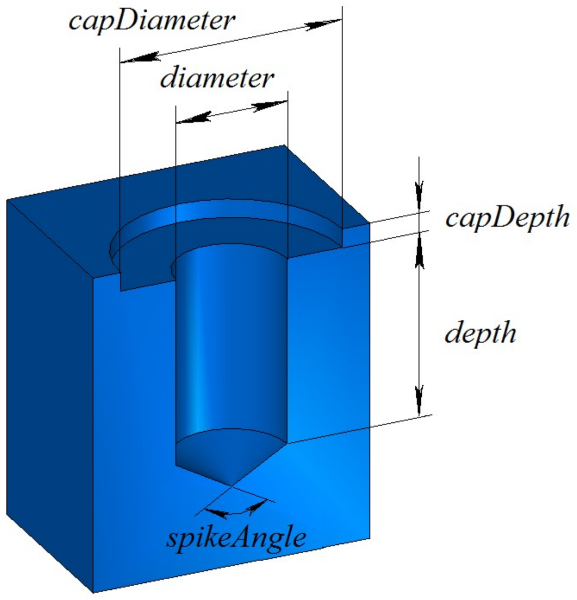


Fig. M.2.22.1.

BorerValues parameters should be used to construct a hole. There are six hole types defined by the BorerValues::*type* parameter that takes one of the following values: bt_SImpleCylinder, bt_TwofoldCylinder, bt_ChamferCylinder, bt_ComplexCylinder, bt_SImpleCone, bt_ArcCylinder. In Fig. M.2.22.2, M.2.22.3, M.2.22.4, M.2.22.5, M.2.22.6, M.2.22.7, you can see holes having different shapes.

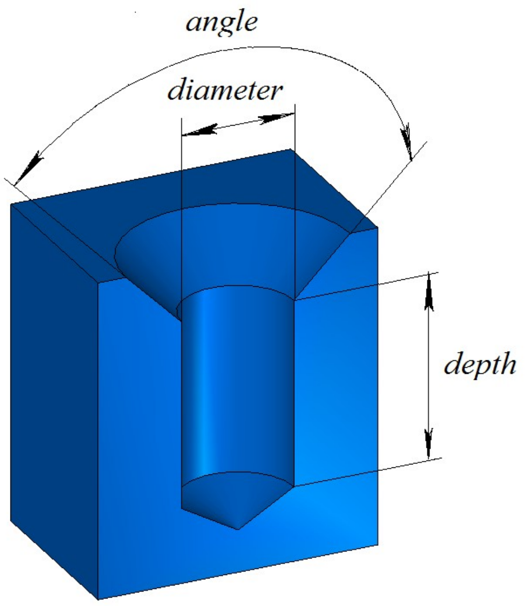


BorerValues::*type* = bt_SimpleCylinder

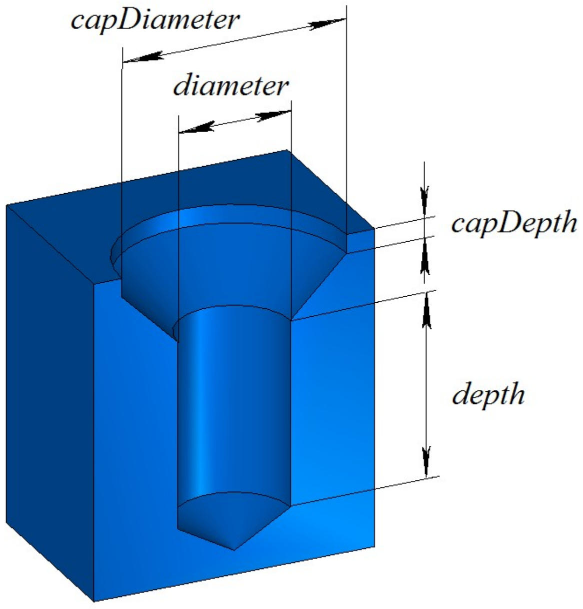


BorerValues::*type* = bt_TwofoldCylinder

Fig. M.2.22.2. *Fig. M.2.22.3.*

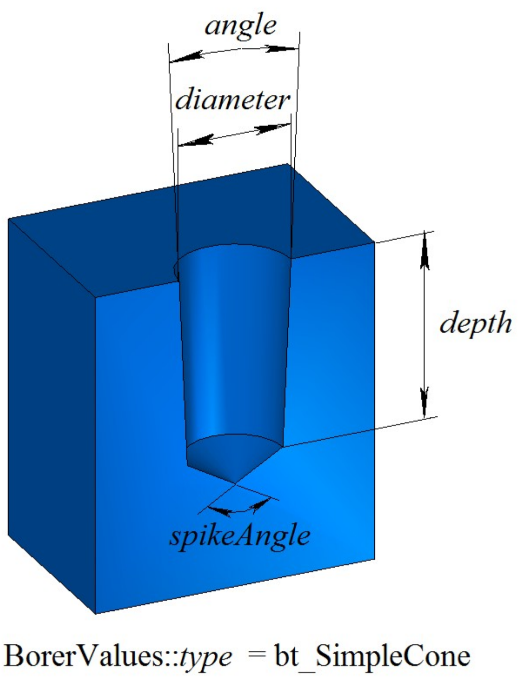


BorerValues::*type* = bt_ChamferCylinder



BorerValues::*type* = bt_ComplexCylinder

Fig. M.2.22.4. *Fig. M.2.22.5.*



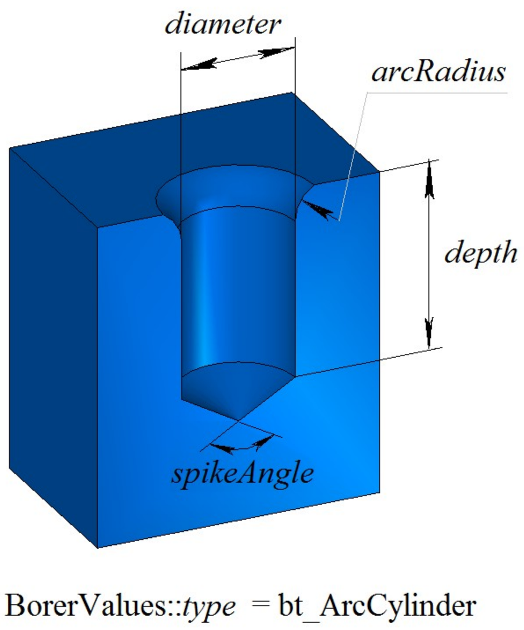


Fig. M.2.22.6. *Fig. M.2.22.7.*

PocketValues parameters should be used to construct a pocket or a protrusion. If PocketValues::*type*=false, then specified *parameters* are used to construct a pocket; if PocketValues::*type*=true, then specified *parameters* are used to construct a protrusion. In Fig. M.2.22.8, you can see a solid with a rectangular pocket without a slope of side faces.

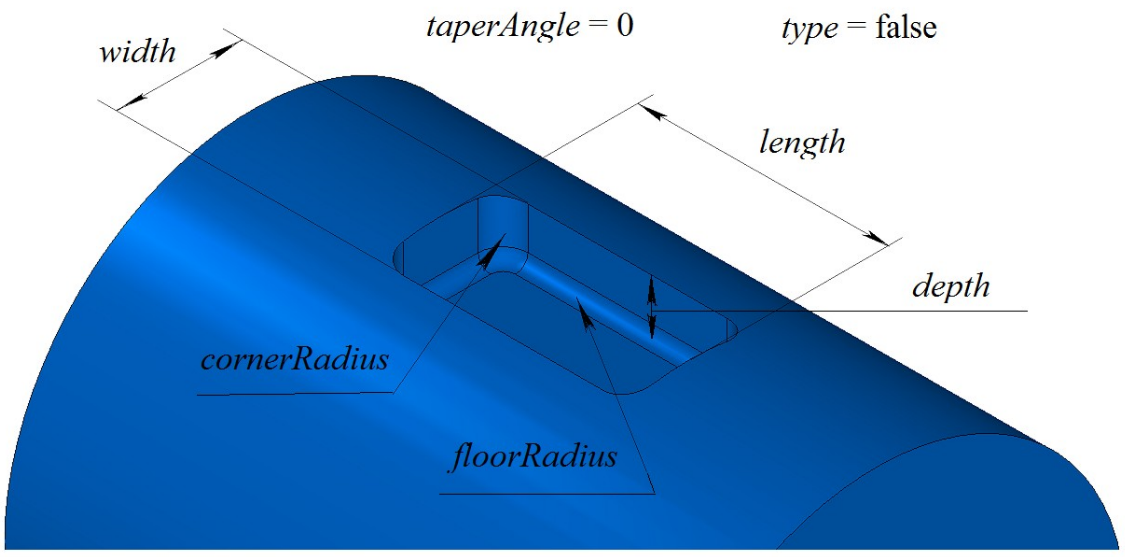


Fig. M.2.22.8.

SlotValues parameters should be used to construct a slot. There are four slot types defined by SlotValues::*type* parameter that takes one of the following values: st_BallEnd, st_Rectangular, st_TShaped, st_DoveTail.

HoleSolid method adds MbRibSolid constructor in the log of the newly constructed solid that contains data required to execute the operation. MbHoleSolid constructor is declared in cr_hole_solid.h file.

test.exe test application splits specified faces of the solid using New ->Solid -> Based on Solid -> With a Hole menu command.

M.2.23. Constructing a Solid with an Enforcement Rib

Method
MbResultType
RibSolid (MbSolid & **solid**,
  MbeCopyMode sameShell,
  const MbPlacement3D & **place**,
  const MbContour & **contour**,
  size_t *index*,
  RibValues & *params*,
  const MbSNameMaker & names,
  MbSolid *& **result**)
constructs a solid with an enforcement rib.

Input parameters of the method are as follows:

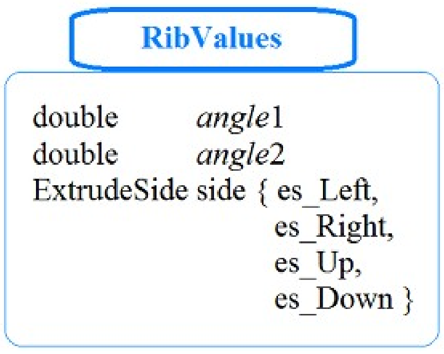
- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **place** is a local coordinate system, its XY plane is a symmetry plane,
- **contour** is shape-generating contour in XY plane of local coordinate system,
- *index* is segment number in the contour,
- *params* are parameters of the enforcement rib,
- names is the namer of rib faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration. The method is declared in action_solid.h file.

This method constructs an enforcement rib using specified **contour** contour and it merges the rib with **solid** original solid. Contour segment with specified number defines a slope vector.

params parameter defines data for building (Fig. M.2.23.1).



RibValues

```
double       angle1
double       angle2
ExtrudeSide side { es_Left,
                   es_Right,
                   es_Up,
                   es_Down }
```

Fig. M.2.23.1.

RibValues structure defined in the file swept_parameter.h

sameShell parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. names parameter is used to name the faces of the constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

In Fig. M.2.23.2, you can see **solid** original solid, **place** local coordinate system and **contour** in XY plane of the latter.

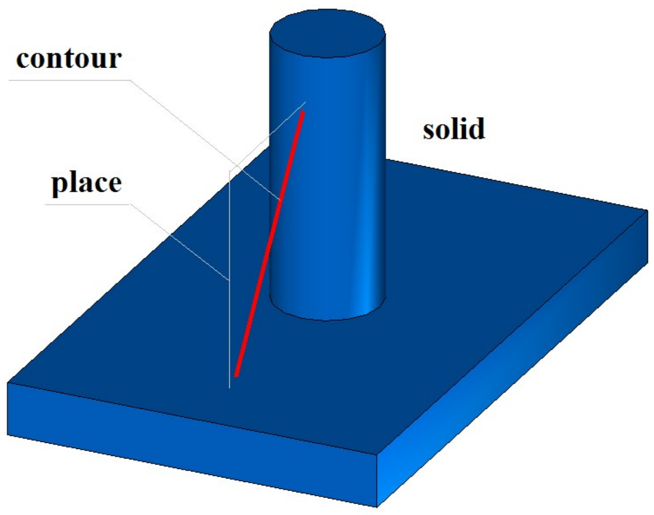


Fig. M.2.23.2.

In Fig. M.2.23.3, you can see newly constructed enforcement rib without a slope of side faces.

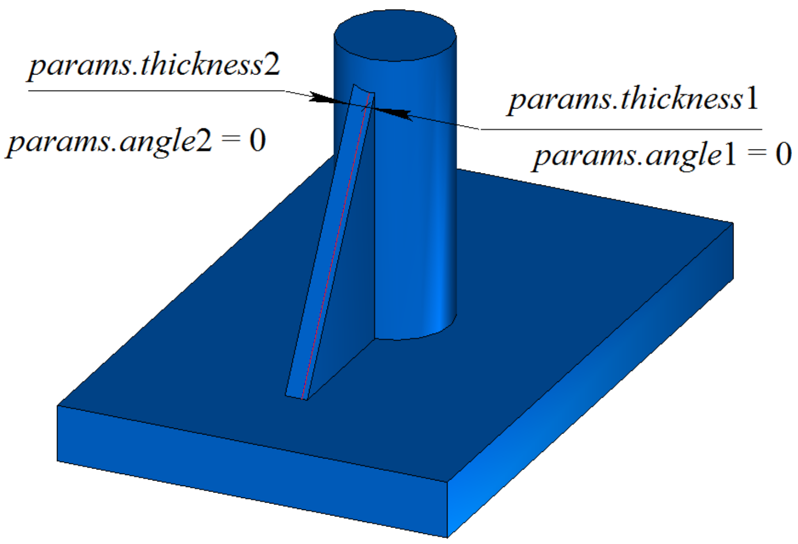


Fig. M.2.23.3.

In Fig. M.2.23.4, you can see newly constructed enforcement rib with a slope of side faces.

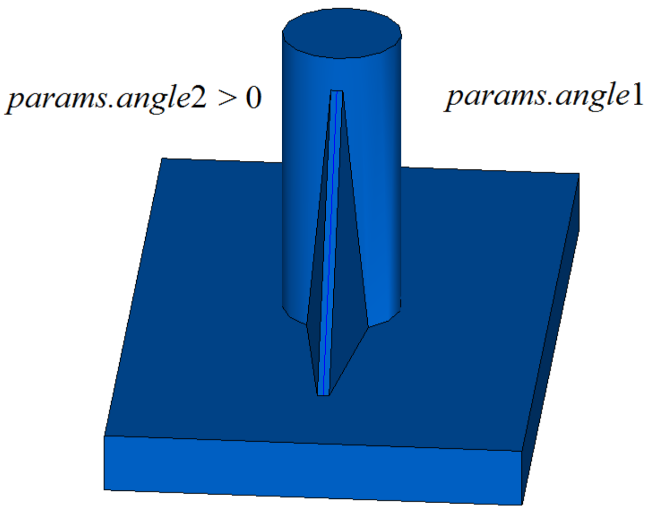


Fig. M.2.23.4.

RibSolid method adds MbRibSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbRibSolid constructor is declared in cr_rib_solid.h file.

test.exe test application splits specified faces of the solid using New ->Solid -> Based on Solid -> With Enforcement Rib menu command.

M.2.24. Sloping Solid Faces

Method
MbResultType
DraftSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **place**,
double *angle*,
const RPArray<MbFace> & **faces**,
MbeFacePropagation *propagation*,
bool *reverse*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a solid with specified faces of the solid sloped from neutral isometric plane at a predetermined angle.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *sameShell* is a version of original solid copying method,
- **place** is neutral plane,
- *angle* is slope angle,
- **faces** is a set of faces that should be sloped,
- *propagation* is a flag of capturing faces that are smoothly joined with sloping faces,
- *reverse* is a flag indicating a reverse slope,
- names is a namer of constructed faces.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.The method is declared in action_solid.h file.

This method constructs a solid with faces sloped relative to their position in **solid** original solid. *params* parameter defines construction parameters. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original solid to **result** constructed solid. XY plane in **place** local coordinate system defines the plane, in relation to which solid faces are sloped. *angle* parameter defines slope angle. **faces** set contains the faces that should be sloped. *propagation* parameter controls addition to the set of faces that should be sloped other solid **faces** that should be smoothly joined with sloped faces. *reverse* parameter defines slope direction. names parameter is used to name the faces of the constructed solid.

sameShell enumeration parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*. MbeCopyMode enumeration is described in item O.7.9. Copying a Set of Faces.

In Fig. M.2.24.1, you can see **solid** original solid, **place** local coordinate system, in relation to XY plane of the latter sloping is executed and **faces** that should be sloped.

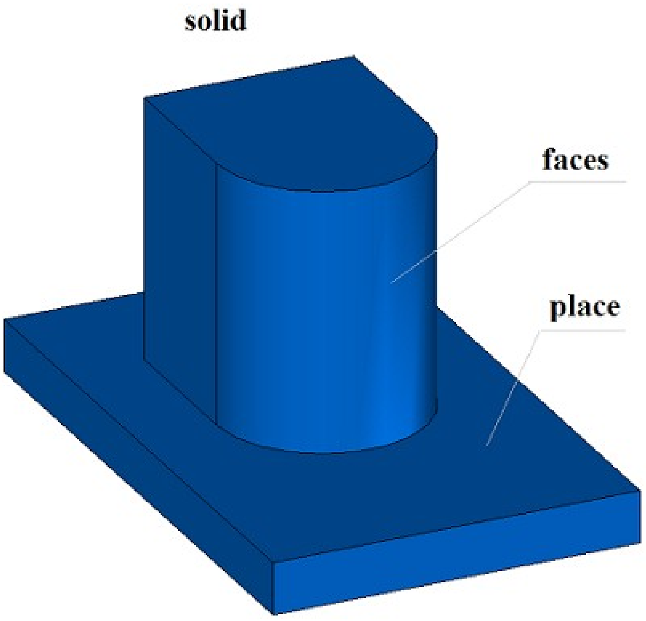


Fig. M.2.24.1.

In Fig. M.2.24.2, you can see the constructed solid; its specified faces are sloped.

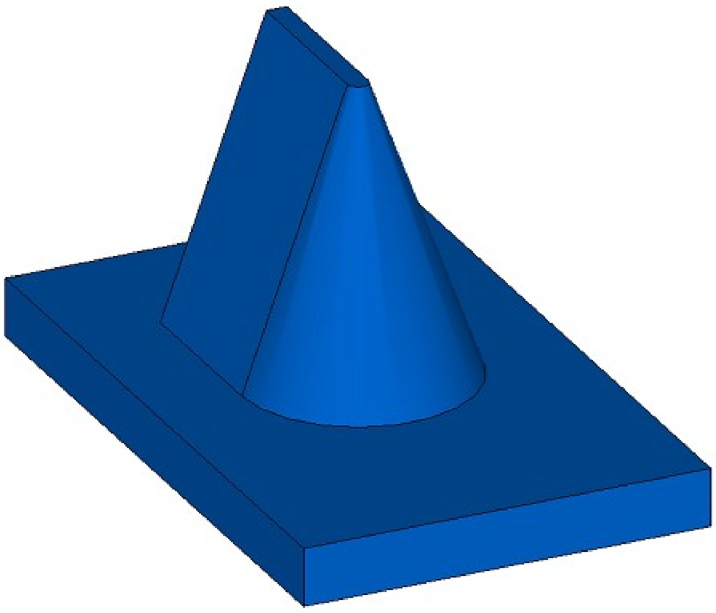


Fig. M.2.24.2.

DraftSolid method adds MbDraftSolid constructor in the log of the newly constructed solid that contains all data required to execute the operation. MbDraftSolid constructor is declared in cr_draft_solid.h file.

test.exe test application splits the specified faces of the solid using New ->Solid -> By Processing Faces -> By Sloping Faces menu command.

M.2.25. Multiplication of Solids

Method
MbResultType
DuplicationSolid (const MbSolid & **solid**,
const DuplicationValues & *parameters*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs copies of the original solid, transforms them according a specified rule and merges them into a single solid.

Input parameters of the method are as follows:

- **solid** is the original solid,
- *parameters* are construction parameters,
- names is face namer.

Method output parameter is **result** constructed solid.

If successful, the method returns rt_Success, otherwise it returns error code from MbResultType enumeration.

The method is declared in action_solid.h file.

names parameter is used to name the faces of the constructed solid. *parameters* parameter defines construction parameters. In Fig. M.2.25.1, you can see data used for construction and parameter inheritance scheme from DuplicationValues abstract class.

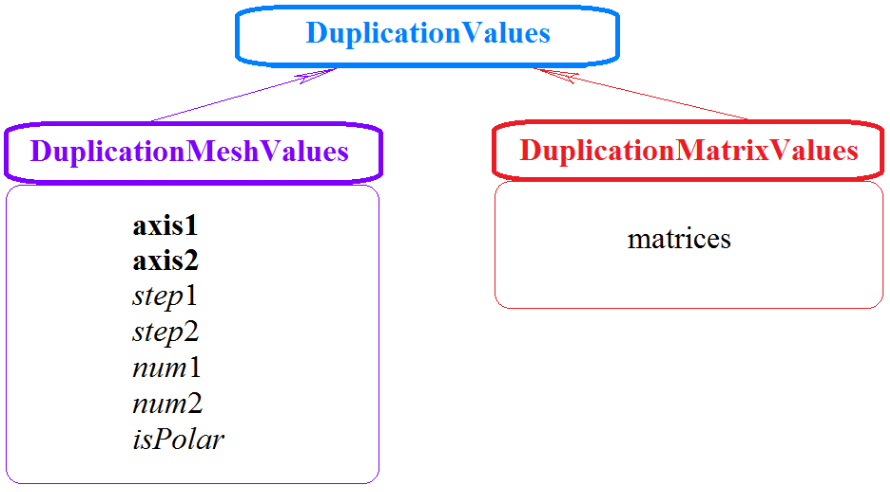


Fig. M.2.25.1.

DuplicationMeshValues parameters should be used to make copies of the solid and align them to 2D grid. The following two multiplication methods are supported: using two directions and using a polar grid. *parameters.isPolar* parameter defines grid type. If *parameters.isPolar*=false, then the original solid and its copies are located in the nodes of 2D grid having *parameters.***axis1** and *parameters.***axis2**. The original solid is the reference point. Along *parameters.***axis1**, *parameters.num*1 copies of the solid are located with *parameters.step*1; along *parameters.***axis2** axis, *parameters.num*2 copies of the solid are located with *parameters.step*2, including the original solid. If *parameters.isPolar*=true, than newly constructed copies of the solid are located in nodes of a polar grid. The original solid is the reference point. Radial direction of the grid is determined by *parameters.***axis1** vector, and rotation axis is determined by vectors product of *parameters.***axis1** and *parameters.***axis2**. *parameters.num*1 copies of the solid are located with *parameters.step*1 along radial directions; *parameters.num*2 copies of the solid are located on each circle with angular *parameters.step*2 .

You should use DuplicationMatrixValues parameters to multiply a solid and to transform its copies by a set of matrices. *parameters.*matrices parameter defines a set of transformation matrices.

If after construction the original solid or its copies intersect with each other, then a Boolean operation is executed to merge intersecting solids. In Fig. M.2.25.2, you can see a solid multiplied in a polar grid.

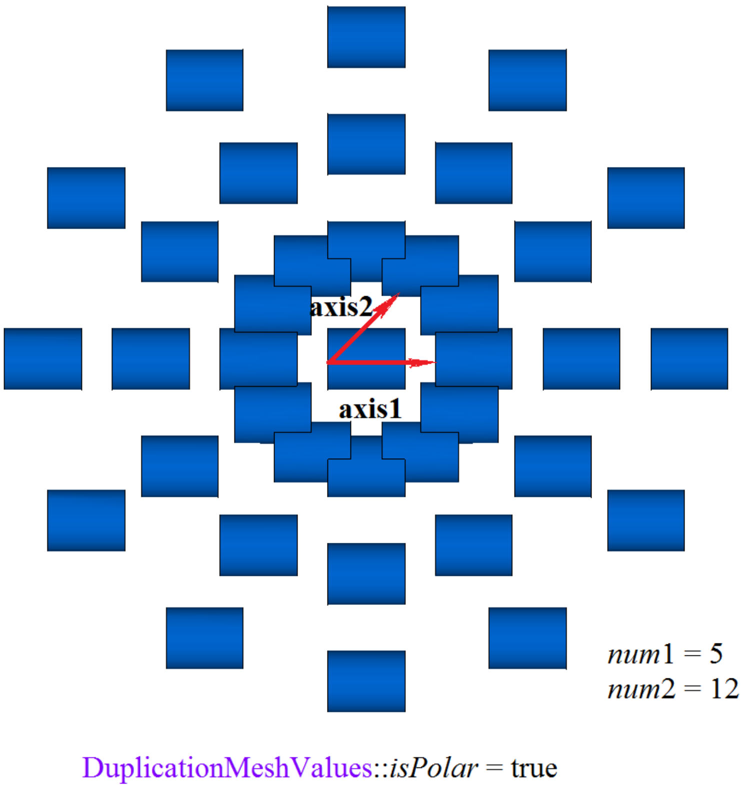


Fig. M.2.25.2.

DuplicationSolid method adds MbDuplicationSolid constructor in the log of newly constructed solid that contains all data required to execute the operation. MbDuplicationSolid constructor is declared in cr_duplication_solid.h file.

test.exe test application splits specified faces of the solid using New ->Solid -> Based on Solid -> By Grid Multiplication and New ->Solid -> Based on Solid -> By Matrix Multiplication menu commands.

O.1. ELEMENTARY OBJECTS

Elementary objects are C3D kernel geometric objects that describe the following mathematical entities: a vector, a point, an axis, local coordinate system, transformation matrix, bounding box and bounding rectangle. Elementary objects have simple data structures. Elementary object is a tool and building block for more complex geometric objects, so they are used by all modules of the geometric kernel.

O.1.1. MbVector3D Vector in Three-Dimensional Space

MbVector3D class is declared in mb_vector3d.h file.

MbVector3D vector describes movement or direction in three-dimensional space. It is determined by *x*, *y* and *z* components in Cartesian coordinate system.

We will use one or more bold lower-case Roman letters for 3D vectors; all vector components will be placed in square brackets, for example:

$$\mathbf{vector} = [x \;\; y \;\; z].$$

MbVector3D vector is not attached to any point in the space, so it does not have a method that moves in space.

O.1.2. MbCartPoint3D Radius Vector of Point in 3D Space

MbCartPoint3D class is declared in mb_cart_point3d.h file.

MbCartPoint3D radius vector (Cartesian point) describes location in 3D space; it is determined by *x*, *y* and *z* components in Cartesian coordinate system. Radius vector describes a transformation that moves the initial point of the Cartesian coordinate system to a point in space having specified coordinates in the Cartesian coordinate system.

We will use one or more bold lower-case Roman letters for points in 3D space; point coordinates will be placed in square brackets, for example:

$$\mathbf{point} = [x \;\; y \;\; z].$$

Unlike a vector, a radius vector is associated with the origin of coordinates. Coordinates of MbCartPoint3D radius vector and MbVector3Dundergo different changes in case of transition from one coordinate system to other, and also when their position in space is changed using the following methods:
MbCartPoint3D & **Transform**(constMbMatrix3D&),
MbCartPoint3D & **Rotate**(const MbAxis3D &, double angle),
MbCartPoint3D & **Move**(const MbVector3D&).
Mentioned methods return a reference to itself after transformation.

O.1.3. MbHomogenius3D Homogenius Vector in Three-Dimensional Space

MbHomogenius3D class is declared in mb_homogenius3d.h file.

MbHomogenius3D homogenius radius vector describes location of a point in 3D space; it is defined by four coordinates: *x*, *y*, *z* and *w*. The fourth coordinate is called weight. homogenius radius vector is used to calculate radius vector for *B*-curves and *B*-surfaces constructed based on *B*-splines. x_w, y_w, z_w, w coordinates of MbHomogenius3D homogenius radius vector are linked with *x*, *y*, *z* coordinates of the radius vector; these relationships are described by the following equations:

$$x = \frac{x_w}{w},\ y = \frac{y_w}{w},\ z = \frac{z_w}{w}.$$

For MbMatrix3D multiplication operations, we can assume that vectors and points also have the fourth coordinate; it is equal to zero for MbVector3Dand it is equal to one for MbCartPoint3D.

O.1.4. MbPlacement3D Local Coordinate System

MbPlacement3D class is declared in mb_placement3d.h file.

Local coordinate system in MbPlacement3D three-dimensional space is described by **origin** initial point and three non-coplanar vectors (**axisX**, **axisY** and **axisZ**). Please see Figure O.1.4.1.

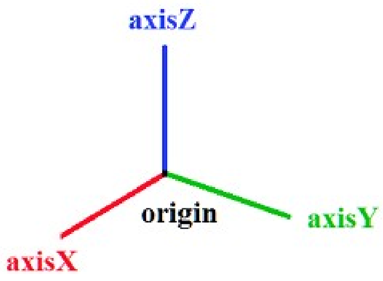


Fig. O.1.4.1.

In most cases, right-handed coordinate system is used and the vectors are orthonormal. The coordinate system can become left-handed and non-orthonormal after transformation. The following methods are used to request information on the state of coordinate system:
bool **IsLeft**() the method permits to find out whether the coordinate system is left-handed,
bool **IsRight**() the method permits to find out whether the coordinate system is right-handed,
bool **IsTranslation**() the method permits to find out whether **origin** coordinate system has an offset,
bool **IsRotation**() the method permits to find out whether the coordinate system is rotated,
bool **IsOrt**() the method permits to find out whether the coordinate system is orthogonal and not normalized,
bool **IsSingle**() the method permits to find out whether the coordinate system coincides with the coordinate system where it was defined,
bool **IsNormal**() the method permits to find out whether the coordinate system is orthonormal,
bool **IsOrtogonal**() the method permits to find out whether the coordinate system is orthogonal,
bool **IsCircular**() the method permits to find out whether the coordinate system is orthogonal and has **axisX** and **axisY** with equal length; a circle in this coordinate system remains a circle,
bool **IsIsotropic**() the method permits to find out whether the coordinate system is orthogonal and has **axisX**, **axisY** and **axisZ** axes with equal length; objects in this coordinate system are not distorted, they are rather scaled,
bool **IsAffine**() the method permits to find out whether the coordinate system is affine (otherwise, it is orthonormal).

Local coordinate system is Cartesian. A point in a Cartesian coordinate system is defined by three coordinates: x, y and z. Local system can be cylindrical or spherical coordinate system.

If you use MbPlacement3D local coordinate system as a cylindrical system, then Z axis of the cylindrical coordinate system coincides with Z axis of the Cartesian one, polar axis of the cylindrical system coincides with X axis of the Cartesian system, and polar angle of the cylindrical system is measured from X axis towards Y axis. x coordinate plays the role of projection of radius vector to XY plane; y coordinate plays the role of polar angle.

If you use MbPlacement3D local coordinate system as a spherical system, then the plane of spherical coordinate system coincides with XY plane of the Cartesian system, and longitude of spherical system is measured from X axis towards Y axis. x coordinate plays the role of the length of radius vector; y coordinate plays the role of longitude.

O.1.5. MbMatrix3D Extended Matrix in Three-Dimensional Space

MbMatrix3D class is declared in mb_matrix3d.h file.

In a three-dimensional space, Matrix3D matrix describes transformation from one coordinate system to other one. It is a 4-by-4 matrix. Let the specified coordinate system have a local affine coordinate system with origin in **r** point with r_1, r_2, r_3 coordinates and **a**=[a_1 a_2 a_3], **b**=[b_1 b_2 b_3] and **c**=[c_1 c_2 c_3] basis vectors. **a**, **b** and **c** vectors should be linearly independent, but they may be non-orthogonal and may have arbitrary length. Matrix3D matrix for transformation from the local coordinate system to the specified one looks as follows

$$\mathbf{M} = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

We will use bold capital Roman letters to denote extended matrices in 3D space, for example: **M**. Please note that each of **a**, **b** and **c** basis vectors and **r** initial point of the local coordinate system has a corresponding row in the matrix that executes transformation from the local coordinate system to the specified one.

MbMatrix3D is an extended matrix that works with uniform radius vectors and MbHomogenius3D in three-dimensional space.

If MbCartPoint3D radius vector is transformed using MbMatrix3D matrix, then the point should be assigned the forth coordinate equal to one. Let the point with x_1, x_2, x_3 coordinates in local coordinate system have p_1, p_2, p_3 coordinates in specified coordinate system. If MbMatrix3D extended matrix is used, then these coordinates will be related as follows:

$$\begin{bmatrix} p_1 & p_2 & p_3 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

Please note that 3D radius vector is multiplied by MbMatrix3D extended matrix on the right.

If MbVector3D vector is transformed using MbMatrix3D matrix, then the vector should have the forth coordinate equal to zero. Let a vector with components y_1, y_2, y_3 in local coordinate system have r_1, r_2, r_3 components in the specified coordinate system. If MbMatrix3D extended matrix is used, then these components will be related as follows:

$$\begin{bmatrix} r_1 & r_2 & r_3 & 0 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & y_3 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

Please note that 3D vector is multiplied by MbMatrix3D extended matrix on the right.

O.1.6. MbCube Bounding Box in Three-Dimensional Space

MbCube class is declared in mb_cube.h file.

MbCube bounding box describes the dimensions of extended object (curve, surface, solid or several solids) in 3D space and it is defined by two points: **pmin** and **pmax**. The faces of bounding box are parallel to the planes of the coordinate system where the cube is described. **pmin** and **pmax** points describe two opposite vertexes of the bounding box with minimal and maximal coordinates respectively. Please see Figure O.1.6.1.

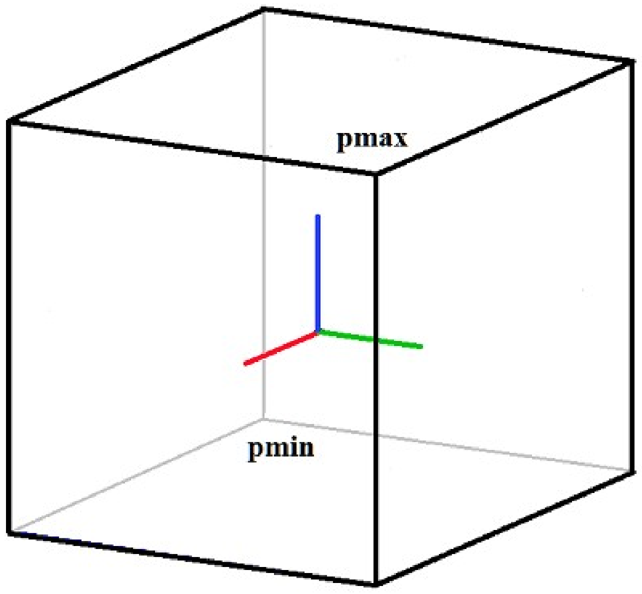


Fig. O.1.6.1.

If the dimensions of extended object are not set, then the bounding box is considered empty and **pmin**=[10^{-300} 10^{-300} 10^{-300}], **pmax**=[-10^{-300} -10^{-300} -10^{-300}]. The following condition holds for an empty bounding box: **pmin**>**pmax**; IsEmpty() method returns true.

O.1.7. MbRect1D Univariate Dimension

MbRect1D class is declared in mb_rect1d.h file.

MbRect1D univariate dimension describes one-dimensional area (for example, curve parameter definition area); it is defined by two values: *zmin* and *zmax*. Please see Figure O.1.7.1.

Fig. O.1.7.1.

zmin and *zmax* values describe leading and trailing edges of the area. If one-dimensional area is not defined, then univariate dimension is considered empty and *zmin*>*zmax*. If univariate dimension is empty, then **IsEmpty**() method returns true.

O.1.8. MbVector Vector in Two-Dimensional Space

MbVector class is declared in mb_vector.h file.

MbVector vector describes movement or direction in two-dimensional space. The vector is determined by *x* and *y* components in a Cartesian coordinate system.

We will use one or more bold and italic lower-case Roman letters for vectors in two-dimensional space; and vector components will be placed in square brackets, for example:

$$\boldsymbol{vector} = [x \;\; y].$$

MbVector vector is not attached to any points in space, so it does not have a method used to move it in the space.

O.1.9. MbDirection Normalized Vector in Two-Dimensional Space

MbDirection class is declared in mb_vector.h file.

MbDirection normalized vector describes direction or rotation angle in 2D space; it is defined by two components (*ax* and *ay*) in Cartesian coordinate system. The length of normalized vector is equal to one, and its components are sine and cosine of the angle between OX axis and the normalized vector. Therefore, ax=cos(α), ay=sin(α), where α is the angle between the normalized vector and x-axis of the coordinate system.

O.1.10. MbCartPoint Point Radius Vector in Two-Dimensional Space

MbCartPoint class is declared in mb_cart_point.h file.

MbCartPoint3D (Cartesian point) radius vector describes a location in 2D space. This vector is determined by *x* and *y* components in Cartesian coordinate system. Radius vector describes a transformation that moves the initial point of the Cartesian coordinate system to a point in space having specified coordinates in the Cartesian coordinate system.

We will use one or more bold and italic lower-case Roman letters for points in 3D space; point coordinates will be placed in square brackets, for example:

$$\boldsymbol{point} = [x \;\; y].$$

Unlike a vector, a radius vector is associated with the origin of coordinates. Coordinates of MbCartPoint radius vector and MbVector vector components undergo different changes during transition from one coordinate system to other, as well as when their position in space is changed using the following methods:
void **Transform**(const MbMatrix&),
void **Rotate**(const MbCartPoint &, double angle),
void **Move**(const MbVector&).

O.1.11. MbHomogenius Homogenios Vector in Two-Dimensional Space

MbHomogenius class is declared in mb_homogenius.h file.

MbHomogenius extended radius vector describes location of a point in 2D space; it is defined by three coordinates: *x*, *y* and *w*. The third coordinate indicates weight. extended radius vector is used to calculate a radius vector of *B*-curves constructed on the basis of *B*-splines. x_w, y_w, w coordinates of MbHomogenius extended radius vector are linked to *x* and *y* coordinates of the MbCartPoint radius vector MbCartPoint by the following equations:

$$x = \frac{x_w}{w},\ y = \frac{y_w}{w}.$$

As for multiplication by an extended matrix MbMatrix, we can assume that 2D vectors and points also have the third coordinate, which is zero for MbVector vector and one for MbCartPoint.

O.1.12. MbPlacement Local Coordinate System

MbPlacement class is declared in mb_placement.h file.

Local coordinate system in MbPlacement 2D space is described by ***origin*** initial point and two non-parallel vectors (***axisX*** and ***axisY***). Please see Figure O.1.12.1.

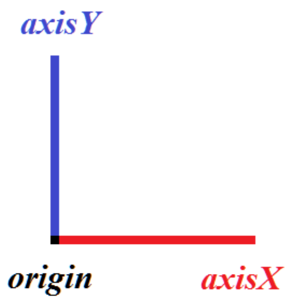


Fig. O.1.12.1.

In most cases, right-handed coordinate system and orthonormal vectors are used. The coordinate system can become left-handed and non-orthonormal after transformation. The following methods are used to request information on the state of coordinate system:
bool **IsLeft**() the method permits to find out whether the coordinate system is left-handed,
bool **IsSingle**() the method permits to find out whether the coordinate system coincides with the coordinate system where it was defined,
bool **IsNormal**() the method permits to find out whether the coordinate system is orthonormal,
bool **IsCircular**() the method permits to find out whether the coordinate system is orthogonal and has ***axisX*** and ***axisY*** with equal length; a circle in this coordinate system remains a circle,
bool **IsIsotropic**() the method permits to find out whether the coordinate system is orthogonal and has ***axisX*** and ***axisY*** of equal length; objects in this coordinate system are not distorted, they are rather scaled,
bool **IsAffine**() the method permits to find out whether the coordinate system is affine (otherwise, it is orthonormal).

Local coordinate system is Cartesian.

O.1.13. MbMatrix Extended Matrix in Two-Dimensional Space

MbMatrix3D class is declared in mb_matrix3d.h file.

In 2D space, MbMatrix matrix describes transformation from one coordinate system to other one. It is 3-by-3 matrix. Let specified coordinate system have a local affine coordinate system with origin at ***r*** point with r_1 and r_2 coordinates and $\boldsymbol{a}=[a_1\ a_2]$ and $\boldsymbol{b}=[b_1\ b_2]$ basis vectors. ***a*** and ***b*** vectors shouldn't be collinear, but they may be non-orthogonal and they may have arbitrary length. MbMatrix matrix used for transformation from the local coordinate system to the specified one looks as follows

$$\boldsymbol{M} = \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}.$$

We will use bold and italic capital Roman letters to denote an extended matrix in 3D space, for example: ***M***. Please note that each of ***a*** and ***b*** basis vectors and ***r*** initial point of the local coordinate system has a corresponding row in the matrix that executes transformation from the local coordinate system to the specified coordinate system.

MbMatrix is an extended matrix that works with uniform radius vectors and homogeneous vectors MbHomogenius in 2D space.

When a radius vector MbCartPoint is transformed using MbMatrix matrix, the point should be assigned the third coordinate that should be equal to one. Let the point with x_1 and x_2 coordinates in the local coordinate system have p_1 and p_2 coordinates in the specified coordinate system. If MbMatrix extended matrix is used, then these coordinates will be related as follows:

$$\begin{bmatrix} p_1 & p_2 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}.$$

Please note that 2D radius vector is multiplied by MbMatrix extended matrix on the right.

If vector MbVector is transformed using MbMatrix matrix, then the vector should have the forth coordinate that should be equal to zero. Let a vector with y_1 and y_2 components in local coordinate system have r_1 and r_2 components in the specified coordinate system. If MbMatrix extended matrix is used, then these components will be related as follows:

$$\begin{bmatrix} r_1 & r_2 & 0 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}.$$

Please note that 2D vector is multiplied by MB Matrix extended matrix on the right.

O.1.14. MbRect Bounding Rectangle in Two-Dimensional Space

MbRect class is declared in mb_rect.h file.

MbRect bounding rectangle describes the dimensions of extended object (one or several curves) in 2D space, it is defined by four points: *left*, *right*, *bottom* and *top*. The sides of the bounding rectangle are parallel to the axes of the coordinate system where the rectangle is described. *left* and *right* values describe minimum and maximum abscissas of the bounding rectangle; *bottom* and *top* values describe minimum and maximum ordinates of the bounding rectangle. Please see Figure O.1.14.1.

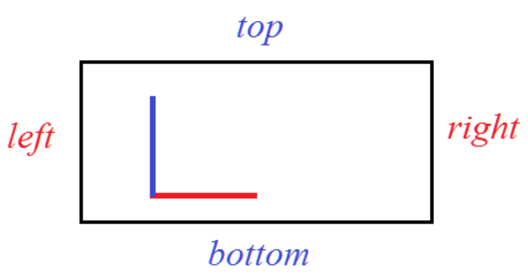


Fig. O.1.14.1.

If the dimensions of an extended object are not determined, then the bounding rectangle is considered empty and *left*>*right*, *bottom*>*top*. If the bounding rectangle is empty, then **IsEmpty**() method returns true.

O.2. GEOMETRICAL OBJECTS

A geometrical object describes the form of the modeled object. Geometric objects include curves, surfaces, solids as well as topological objects that describe geometric properties that don't depend on quantitative features and describe permanently interconnected points in 3D space. There are two-dimensional and three-dimensional geometric objects. Two-dimensional objects are used to work with definition areas of surface parameters, as well for work with planes of local 3D coordinate systems. Parent classes of geometrical objects are described in this part.

O.2.1. MbRefItem Reference Counter

MbRefItem class is declared in reference_item.h file.

MbRefItem class is described by the number of its *useCount* owners; it is a counter of objects that own this object.

All geometrical objects of C3D kernel are divided into three groups: two-dimensional geometrical objects, three-dimensional geometrical objects, and topological objects. All geometrical objects are inheritors of MbRefItem and TapeBase classes. Please see Figure O.2.1.1.

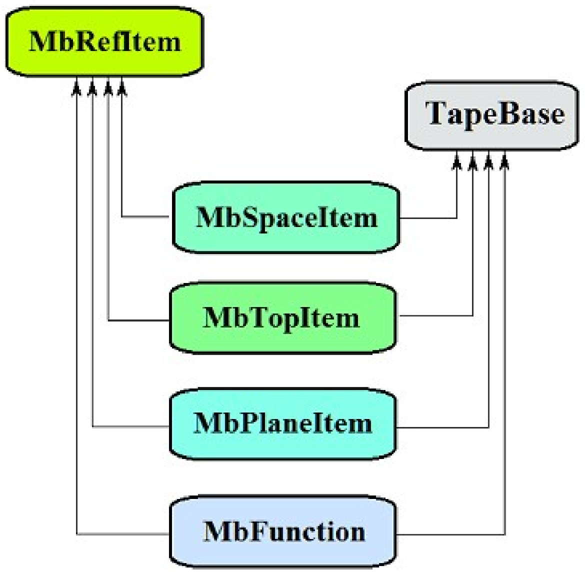


Figure O.2.1.1.

TapeBase class opens a stream for its inheritors for both reading and writing.

The following geometrical objects are inheritors of MbRefItem and TapeBase classes:
MbSpaceItem – base abstract class of three-dimensional geometrical objects,
MbTopItem – base abstract class of topological objects,
MbPlaneItem – base abstract class of two-dimensional geometrical objects,
MbFunction – base abstract class of scalar functions.

Reference counter provides correct operation of classes and methods that contain pointers to geometrical objects. If a certain class contains a pointer to a geometrical object, then it should increase reference counter of the geometrical object by one in the constructor using **AddRef**() method; and it should call **Release**() method for the geometrical object that reduces geometrical object reference counter by one in destructor. If a reference counter becomes zero then the geometrical object is deleted. **DecRef**() method decreases reference counter of geometrical object by one. MbRefItem class is processed by MbRegDuplicate duplication registrar and MbRegTransform transformation registrar.

MbeRefType **RefType**()

method returns registration type of the object that uses the reference counter.

O.2.2. MbSpaceItem Three-Dimensional Geometrical Object

MbSpaceItem class is declared in space_item.h file.

MbSpaceItem is an inheritor of MbRefItem and TapeBase classes and it is a parent class for three-dimensional geometrical objects.

Three-dimensional geometrical objects of C3D kernel include: a point, curves, surfaces, auxiliary objects and objects of geometrical model. Please see Figure O.2.2.1.

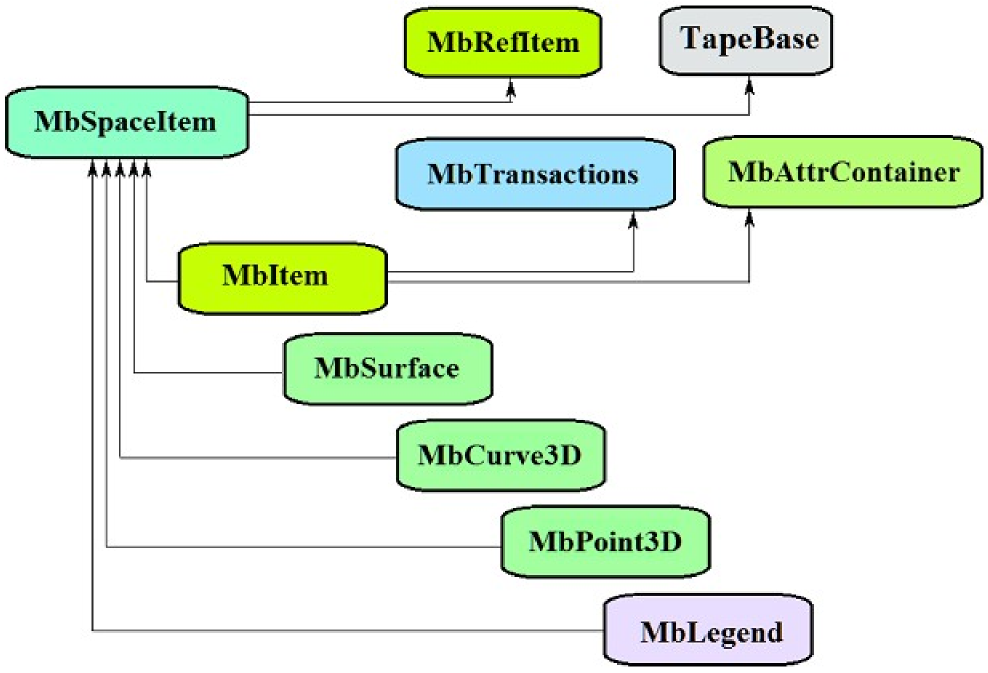


Figure O.2.2.1.

The following families of 3D geometrical objects are inheritors of MbSpaceItem class:
MbPoint3D — a point or a curve,
MbSurface — a surface,
MbLegend — an auxiliary geometrical object,
MbItem — an object of geometrical model.

The main methods of 3D geometrical objects are:
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL), MbMatrix3D
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const & **m**, MbRegTransform * iReg = NULL).

These methods are used to transform a geometrical object. MbRegTransform registrar is used to prevent multiple transformations of embedded objects. If an object contains pointers or references to other objects, then all embedded objects are also transformed. The registrar should be used for serial transformations of several interrelated objects if relationships between them are due to pointers or references to shared objects present in them. If the registar is not used during transformation, then multiple transformations of common embedded objects are possible.

In addition, all geometrical objects have methods that permit to copy, check for coincidence, check whether it's possible to make objects coinciding and to make them coinciding:
MbSpaceItem & **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbSpaceItem & **item**),
bool **IsSimilar**(const MbSpaceItem & **item**),
bool **SetEqual**(const MbSpaceItem & **item**).

MbRegDuplicate registrar is used to prevent multiple copying of embedded objects. If an object contains pointers or references to other objects, all embedded objects are also copied. The registrar should be used to copy several interrelated objects in serial manner if the objects have pointers or references to shared objects. If the registar is not used for copying, then you can get a set of copies of the same embedded object instead of its single copy.

The following methods are used to identify the type of geometrical object:
MbeSpaceType **IsA**(),
MbeSpaceType **Type**(),
MbeSpaceType **Family**().
These methods return a type from the enumeration of three-dimensional geometric objects.

Methods
MbProperty & **CreateProperty**(MbePrompt name),
void **GetProperties**(MbProperties & *properties*),
void **SetProperties**(MbProperties & *properties*)
ensure that internal data of geometrical objects is accessible and editable. **GetProperties** method adds object data to *properties* set as inheritors of MbProperty class.

CalculateWire(double sag, MbMesh & **mesh**) method constructs a polygonal copy of a geometrical object that is used for visualization.

O.2.3. MbTopItem Topological Object

MbTopItem class is declared in topology_item.h file.

MbTopItem class is an inheritor of MbRefItem and TapeBase classes and it is a parent class for topological objects. Topological objects contain a class of named topological objects MbTopologyItem that inherits MbTopItem and MbAttributeContainer classes.MbTopologyItem class is also declared in topology_item.h file.

C3D geometric kernel works with topological objects shown in Figure O.2.3.1.

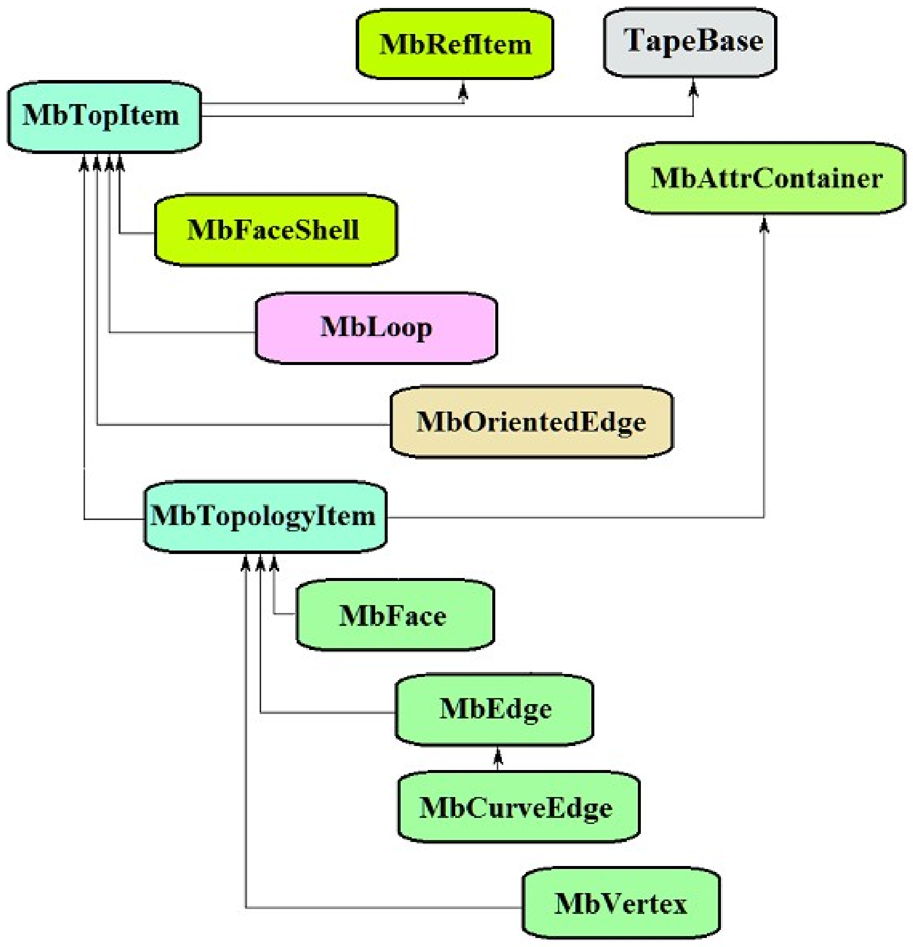


Figure O.2.3.1.

The following topological objects are the inheritors of MbTopItem class:
MbFaceShell — a set of faces
MbLoop — an edge cycle at face border
MbOrientedEdge — an oriented cycle edge
MbTopologyItem — a named topological object.
The following objects inherit MbTopologyItem named topological object:
MbVertex — a vertex
MbEdge — an edge
MbFace — a face.
An edge describing a smooth section that either joins two faces or is a face edge has MbCurveEdge inheritor.
MbAttrContainer attribute container provides work of named topological objects with attributes.
The main methods of named topological objects are listed below:
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL).

These methods are used when a topological object is transformed; they are also used to work with the name and attributes. MbRegTransform registrar is used to prevent multiple transformations of embedded objects. If an object contains pointers or references to other objects, then all embedded objects are also transformed. The registrar should be used for serial transformations of several interrelated objects if relationships between them are due to pointers or references to shared objects present in them. If the registar is not used during transformation, then multiple transformations of common embedded objects are possible.

A topological object has **IsA**() method used to identify its type. The method returns a type from MbeTopologyType enumeration of topological objects.

O.2.4. MbPlaneItem Two-Dimensional Geometrical Object

MbPlaneItem class is declared in plane_item.h file.

MbPlaneItem is an inheritor of MbRefItem and TapeBase classes and it is a parent class for all 2D geometrical objects.

C3D kernel includes the following 2D geometrical objects: curves, a multiline and region. Please see Figure O.2.4.1.

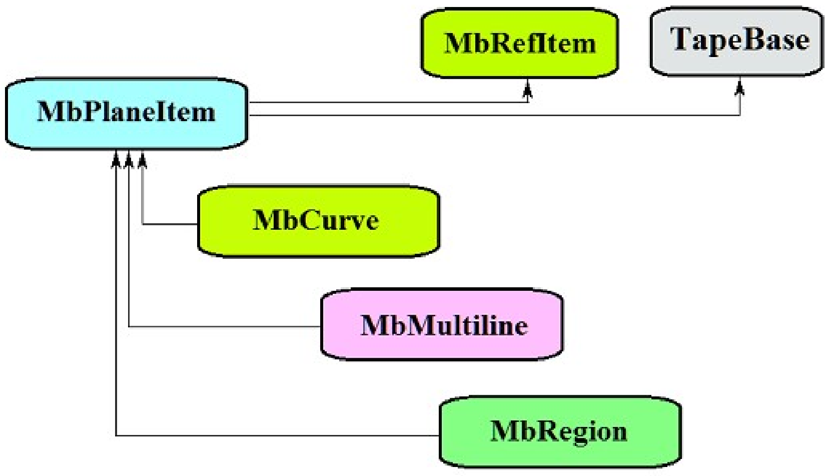


Figure O.2.4.1.

The following families of 2D geometrical objects are inheritors of MbPlaneItem class:
MbCurve — a two-dimensional curve,
MbMultiline — a multiline,
MbRegion — a region.

The main methods of 2D geometrical objects are:
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL, …),
void **Rotate**(const MbCartPoint & **p**, const MbDirection & **angle**, MbRegTransform * iReg = NULL, …),
void **Transform**(const MbMatrix & **m**, MbRegTransform * iReg = NULL, ...).

These methods are used to transform a 2D geometrical object. MbRegTransform registrar is used to prevent multiple transformations of embedded objects. If an object contains pointers or references to other objects, then all embedded objects are also transformed. The registrar should be used for serial transformations of several interrelated objects if relationships between them are due to pointers or references to shared objects present in them. If the registar is not used during transformation, then multiple transformations of common embedded objects are possible.

In addition, all geometrical objects have methods that permit to duplicate, check for coincidence, check whether it's possible to make objects coinciding and to make them coinciding:
MbPlaneItem & **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbPlaneItem & **item**),
bool **IsSimilar**(const MbPlaneItem & **item**),
bool **SetEqual**(const MbPlaneItem & **item**).

MbRegDuplicate registrar is used to prevent multiple copying of embedded objects. If an object contains pointers or references to other objects, all embedded objects are also copied. The registrar should be used to copy several interrelated objects in serial manner if the objects have pointers or references to shared objects. If the registar is not used for copying, then you can get a set of copies of the same embedded object instead of its single copy.

The following methods are used to identify the type of geometrical object:
MbePlaneType **IsA**(),
MbePlaneType **Type**(),
MbePlaneType **Family**(),
These methods return a type from the enumeration of 2D geometric objects.

Methods
MbProperty & **CreateProperty**(MbePrompt name),
void **GetProperties**(MbProperties & *properties*),
void **SetProperties**(MbProperties & *properties*)
ensure that internal data of geometrical objects are accessible and editable. **GetProperties** method adds object data to *properties* set as inheritors of MbProperty class.

O.3. TWO-DIMENSIONAL CURVES

Two-dimensional curves are used to describe definition area of surface parameters, to construct flat sketches, to construct 3D curves on surfaces, curves of surface intersections, and projections of 3D curves on surfaces and planes of local coordinate systems. Many 2D curves are similar to 3D ones, the difference is that 2D curves use 2D rather than 3D points and vectors. We will use ***bold and italic*** Roman letters to designate vectors, radius vectors of points, and matrices in 2D space.

O.3.1. MbCurve Two-Dimensional Curve

MbCurve abstract class is declared in curve.h file.

MbCurve 2D curve is an inheritor of MbPlaneItem class. Please see Figure O.3.1.1.

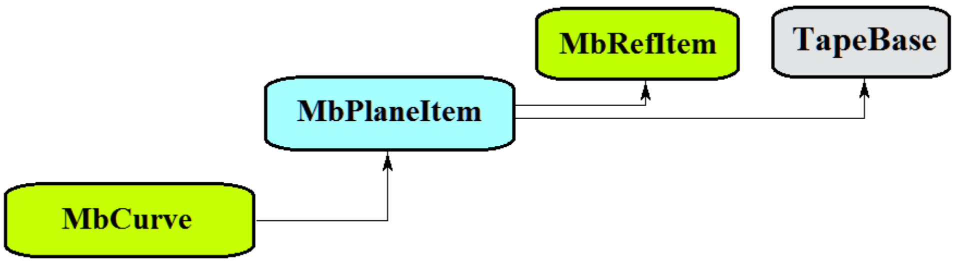


Fig. O.3.1.1.

Two-dimensional curve is an abstract class. The following 2D curves are inheritors of MbCurve class realized in C3D geometric kernel:
MbLine – 2D straight line,
MbLineSegment – 2D straight line segment,
MbArc – 2D elliptical arc,
MbPolyline – 2D polyline,
MbNurbs – 2D B-curve (NonUniform Rational B-Spline),
MbBezier – 2D Bezier composite curve,
MbHermit – 2D Hermite curve,
MbCubicSpline – 2D cubic spline,
MbOffsetCurve – 2D equidistant curve,
MbTrimmedCurve – 2D trimmed curve,
MbReparam – 2D reparameterized curve,
MbCharCurve – 2D curve with symbolical coordinate functions,
MbCosinusoid – 2D cosine wave,
MbPointCurve – point curve,
MbProjCurve – projection curve,
MbContour – 2D contour (composite curve)
MbContourWithBreaks – two-dimensional contour with breaks.

MbCurve two-dimensional curve is a vector function

$$\text{curve}(t) = [u(t) \quad v(t)]$$

of t scalar parameter with values belonging to $[t_{\min}, t_{\max}]$ segment. The curve is a continuous projection of some part of the number axis to 2D space. Two-dimensional space is XY plane of the local 3D coordinate system and the definition area of surface parameters. Curve parameter variation area is $[t_{\min}, t_{\max}]$ segment in one-dimensional space. $u(t)$, $v(t)$ coordinates of a point at ***curve***(t) are single-valued continuous functions of t parameter.

$t_{\min}$ and $t_{\max}$ limit values of parameter definition area are received using double **GetTMin**() and double

GetTMax() curve methods, respectively.

A curve shall be called periodic if there is p>0 such that for ***curve***(t±kp)=***curve***(t), where k is an integer. bool **IsClosed**() method returns true for a periodic curve. double **GetPeriod**() method for a periodic curve (or a curve that can be extended to become periodic) returns p period. Periodic curve parameter definition area is always limited by one period.

The main method for a curve is:
void **PointOn**(double & t, MbCartPoint & $\boldsymbol{r}$).
It returns $\boldsymbol{r}$ radius vector of the curve point for specified t parameter. Methods
void **FirstDer**(double & t, MbVector & $\boldsymbol{r}_t$),
void **SecondDer**(double & t, MbVector & $\boldsymbol{r}_{tt}$),
void **ThirdDer**(double & t, MbVector & $\boldsymbol{r}_{ttt}$)
respectively return the first ($\boldsymbol{r}_t$), the second ($\boldsymbol{r}_{tt}$) and the third ($\boldsymbol{r}_{ttt}$) derivatives of the curve radius vector for specified t parameter. These methods adjust the curve parameter if it goes beyond the definition area (an exception is a straight line MbLine). If t curve parameter goes beyond [$t_{\min}$, $t_{\max}$] segment, then non-periodic curves move t parameter to the nearest limit $t_{\min}$ or $t_{\max}$, and periodic curves add or subtract the required number of periods.

Method
void **_PointOn**(double t, MbCartPoint & $\boldsymbol{r}$)
returns $\boldsymbol{r}$ radius vector of the curve point for specified t parameter, both inside and outside the definition area of t curve parameter. In general case, a non-periodic curve is extended outside of the parameter definition area by tangent in its end point. Periodic curves, arc (MbArc), cosine wave (MbCosinusoid), character curve (MbCharacterCurve) and truncated curve (MbTrimmedCurve) within the basic curve are th exceptions. Periodic curves are extended cyclically outside of the parameter definition area.
Methods
void **_FirstDer**(double t, MbVector & $\boldsymbol{r}_t$),
void **_SecondDer**(double t, MbVector & $\boldsymbol{r}_{tt}$),
void **_ThirdDer**(double t, MbVector & $\boldsymbol{r}_{ttt}$)
respectively return the first ($\boldsymbol{r}_t$), the second ($\boldsymbol{r}_{tt}$) and the third ($\boldsymbol{r}_{ttt}$) derivatives of curve radius vector for specified t parameter both inside and outside of the curve definition area.

The curves reload such methods for 2D geometrical object as follows:
the methods that serve transformation of a geometrical object,
void **Move**(const MbVector & **v**, MbRegTransform * iReg = NULL, …),
void **Rotate**(const MbCartPoint & **p**, const MbDirection& **angle**, MbRegTransform * iReg = NULL, …),
void **Transform**(const MbMatrix & **m**, MbRegTransform * iReg = NULL, ...),
methods that permit to copy, check for coinciding objects, or check whether it's possible to make objects coinciding and that make them coinciding,
MbPlaneItem & **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbPlaneItem & **item**),
bool **IsSimilar**(const MbPlaneItem & **item**),
bool **SetEqual**(const MbPlaneItem & **item**),
methods that return a type from an enumeration of geometric objects,
MbePlaneType **IsA**(),
MbePlaneType **Type**(),
MbePlaneType **Family**(),
methods that ensure access and editing of internal data of the object,
MbProperty & **CreateProperty**(MbePrompt name),
void **GetProperties**(MbProperties & *properties*),
void **SetProperties**(MbProperties & *properties*).

All curves other than MbContour and MbContourWithBreaks usually do not have bends. MbContour and MbContourWithBreaks are composite curves that may have bends at the points where the segments join.

O.3.2. MbLine Two-Dimensional Straight Line

MbLine class is declared in cur_line.h file.

MbLine two-dimensional straight line is described by MbCartPoint ***origin*** initial point and MbVector ***direction*** directional vector. Please see Figure O.3.2.1.

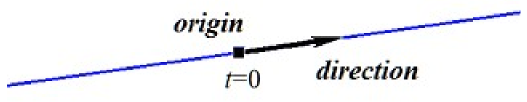


Fig. O.3.2.1.

In **PointOn**(double & t, MbCartPoint & $\boldsymbol{r}$) method, radius vector of $\boldsymbol{r}$ straight line is described by vector function

$$\boldsymbol{r}(t) = \boldsymbol{origin} + t\,\boldsymbol{direction}.$$

Straight line behaves as an infinite object, despite the fact that it has *tmin* and *tmax* parameter limits. Note that unlike all other curves, a straight line does not adjust t parameter when it goes beyond *tmin* and *tmax* limits in radius vector and its derivatives calculation methods.

O.3.3. MbLineSegment Two-Dimensional Straight Line Segment

MbLineSegment class is declared in cur_line_segment.h file.

Two-dimensional MbLineSegment straight line segment is described by MbCartPoint ***point*1** initial point and MbCartPoint ***point*2** end point. Please see Figure O.3.3.1.

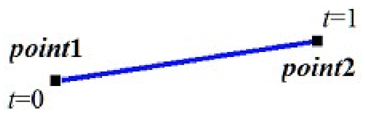


Fig. O.3.3.1.

In **PointOn**(double & t, MbCartPoint & $\boldsymbol{r}$) method, radius vector of $\boldsymbol{r}$ segment is described by

$$\boldsymbol{r}(t) = (1 - t)\,\boldsymbol{point1} + t\,\boldsymbol{point2}$$ vector function.

Segment parameter definition area ranges from zero to one. **point1** initial point of the segment corresponds to $t_{\min}$=0 parameter, **point2** end point of the segment corresponds to $t_{\max}$=1 parameter.

O.3.4. MbArc Two-Dimensional Elliptical Arc

MbArc class is declared in cur_arc.h file.

Two-dimensional elliptical arc is an inheritor of MbCurve curve. MbArc elliptical arc is described by a and b radii, *trim*1 and *trim*2 angles and *sense* direction in MbPlacement ***position*** local coordinate system.

*trim*1 and *trim*2 angles are measured along the arc from ***position.axisX*** vector towards ***position.axisY*** vector. *trim*1 and *trim*2 angles shall be designated as "trimming parameters." Trimming parameters equal to zero and 2π correspond to a point on ***position.axisX*** axis. t curve parameter takes values in $0 \le t \le |trim2 - trim1|$. The curve may be periodic. $|trim2 - trim1| = 2\pi$ holds for a periodic curve. *sense* parameter takes values +1 or -1 and indicates arc construction direction. If *sense*=+1, then *trim*1<*trim*2 and the arc is constructed from *trimm*1 parameter in angle increase direction. If *sense*=–1, then *trim*1>*trim*2 and the arc is constructed from *trimm*1 in angle decrease direction.

In **PointOn**(double & t, MbCartPoint & $\boldsymbol{r}$) method, the radius vector of $\boldsymbol{r}$ curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{position.origin} + a\cos(trim1+(sense)t)\,\boldsymbol{position.axisX} + b\sin(trim1+(sense)t)\,\boldsymbol{position.axisY} \text{ vector function.}$$

Elliptical arc is shown in Figure O.3.4.1.

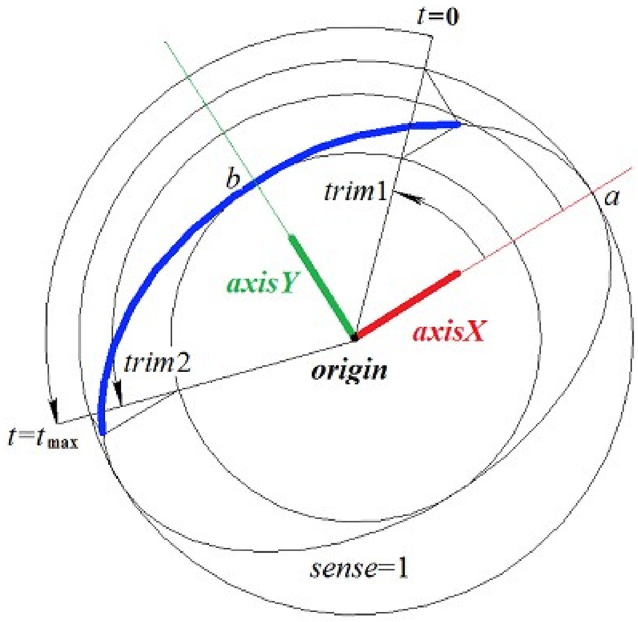


Fig. O.3.4.1.

Curve radii should be positive: $a>0$, $b>0$. The following inequalities should hold for trimming parameters: $trim1<trim2$ if $sense=1$ and $trim\ 1> trim\ 2$ if $sense=-1$.

position local coordinate system may be either left- or right-handed. If local coordinate system is right-handed and $sense=+1$, or if local coordinate system is right-handed and $sense=-1$, then the arc is directed counter-clockwise.

O.3.5. MbPolyline Two-Dimensional Polyline

MbPolyline class is declared in cur_polyline.h file.

Polyline is an inheritor of PolyCurve curve. MbPolyline two-dimensional polyline is described by the number of segments (*segmentsCount*), SArray<MbCartPoint>***pointList*** set of control points and *closed* curve periodicity sign.

The curve goes through ***pointList***[*i*], *i*=0,...,*segmentsCount.* set of points when *t*=0,...,*segmentsCount.* If *closed=true,* then the curve contains a segment that connects the last point of ***pointList***[*segmentsCount*-1] set with the initial point of ***pointList***[0] set. *t* curve parameter takes values in $0 \le t \le segmentsCount$.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{pointList}[i]\ (1-w) + \boldsymbol{pointList}[i+1]\ w \text{ vector function,}$$

where $w = \dfrac{t - t_i}{t_{i+1} - t_i}$, and $t_i \le t \le t_{i+1}$. Polyline is the simplest curve constructed based on a set of points. It consists of segments that consequently connect control points. The curve may be periodic. *segmentsCount* is a period of the periodic curve. Periodic polyline is shown in Figure O.3.5.1.

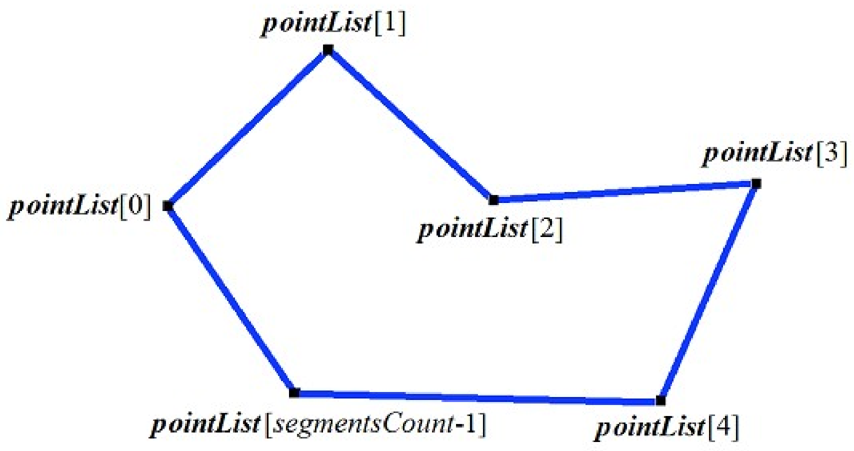


Fig. O.3.5.1.

Derivatives of the curve at control points (when parameter values are integers) lose the continuity by length and direction. Derivatives of the curve in control points have special length and direction.

O.3.6. MbNurbs Two-Dimensional NURBS-Curve

MbNurbs class is declared in cur_nurbs.h file.

B-curve or NURBS-curve is an abbreviation of NonUniform Rational B-Spline. The curve is an inheritor of MbPolyCurve curve. The curve is described by SArray<MbCartPoint>***pointList*** set of two-dimensional control points, *weights* set of weights of two-dimensional control points, *knots* nodal vector, *degree* spline order, *form* curve form parameter and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

The curve is constructed based on B-splines. *knots* nodal vector is a non-decreasing sequence of real numbers that defines curve parameter definition area and the form of the curve. In general, *form* curve form parameter is equal to ncf_Unspecified; and in particular cases it stores data on the original curve that was used to make a NURBS-copy. *degree* of a NURBS-curve is equal to the degree of divided differences used to calculate B-splines. Let node vector have *knotsCount* elements, and the set of control points contain *pointsCount* elements. For non-periodic NURBS-curve, the following equation holds for the number of elements in the sets: *knotsCount*=*pointsCount*+*degree*. For periodic NURBS-curve, the following equation holds for the number of elements in the sets: *knotsCount*=*pointsCount*+2*degree*–1.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$r(t)=\frac{\sum\limits_{j=0}^{pointsCount-1} N_j^{degree}(t)\,weight[j]\,\mathrm{pointList}[j]}{\sum\limits_{j=0}^{pointsCount-1} N_j^{degree}(t)\,weight[j]}$$ vector function,

where $N_j^{order}(t)$ are B-splains of *degree* for *j*th control point from ***pointList***[*j*] list. NURBS-curve of the fourth order is shown in Figure O.3.6.1.

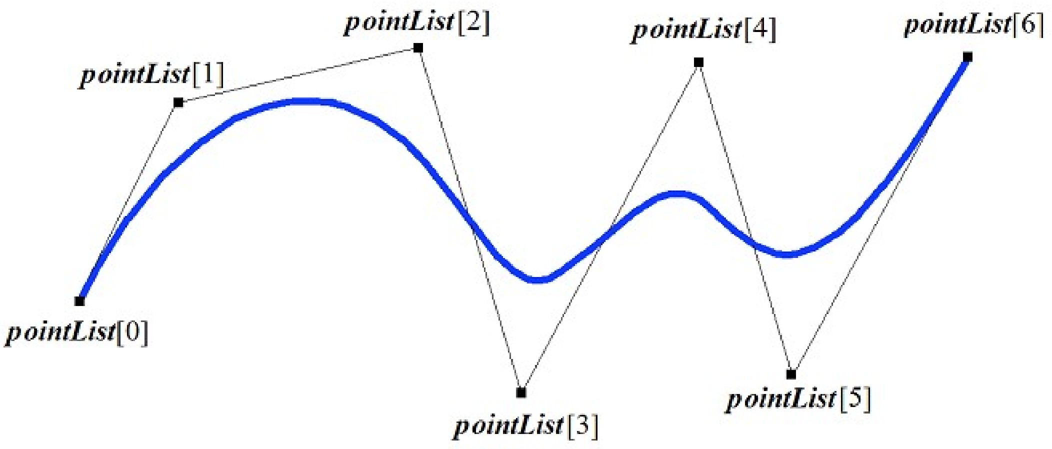


Fig. O.3.6.1.

The curve may be periodic. Periodic NURBS-curve is shown in Figure O.3.6.2.

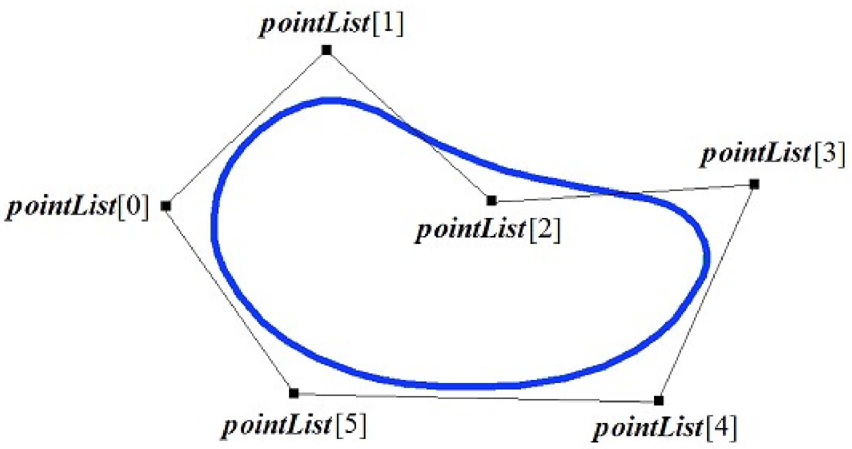


Fig. O.3.6.2.

t curve parameter takes values in *tmin*≤*t*≤*tmax* range, where *tmin*=*knots*[*degree*–1], *tmax*=*knots*[*knotsCount*–*degree*].

Form of NURBS-curve depends on location and weight of control points, as well as values of the nodal vector. In general, NURBS-curve does not go through ***pointList***[i], i=0,...,*pointsCount*–1 set of points. In order that non-closed NURBS-curve goes through extreme control points, it is required that the first *degree* elements and the last *degree* elements of *knots* node vector should coincide. Other things equal, the distance between the curve and the control point depends on the weight of the control point.

Any curve can construct its NURBS-copy using **NurbsCurve**(const MbNurbsParameters & *tParameters*) virtual method.

O.3.7. MbHermit Two-Dimensional Hermite Curve

MbHermit class is declared in cur_hermit.h file.

Hermite two-dimensional curve is an inheritor of MbPolyCurve curve. A curve is described by SArray<MbCartPoint>***pointList*** set of control points, SArray<MbVector>***vectorList*** set of curve derivatives in control points, *tList* set of parameter values in curve control points, *splinesCount* Hermite cubic splines and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

If *tList*[*i*], *i*=0,1,...,*splinesCount*, then the Hermite curve goes through ***pointList***[*i*] control point and has

vectorList[*i*] derivative in it. A curve is constructed on the basis of *splinesCount* smoothly joined 2D third-order Hermite splines. Each Hermite cubic spline describes a segment of the curve between two neighboring control points. Each Hermite cubic spline is defined by two extreme points and two derivatives of the curve in these points.

When a radius vector of the Hermite curve point is calculated, we first use the value of *t* parameter to find out the *i* number of the working segment (Hermite cubic spline number) from *tList*[*i*]≤*t*≤*tList*[*i*+1] condition. The radius vector of the curve is calculated as the radius vector of the found segment for its local parameter *w* that is defined by *tList*[*i*] and *tList*[*i*+1].

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, ***r*** radius vector of the curve is described by vector function of the found segment for its local parameter *w*:

$$r(t) = (1 - 3w^2 + 2w^3)\mathrm{pointList}[i] + (3w^2 + 2w^3)\mathrm{pointList}[i+1] +$$
$$+ ((w - 2w^2 + w^3)\mathrm{vectorList}[i] + (-w^2 + w^3)\mathrm{vectorList}[i+1])(\mathrm{tList}[i+1] - \mathrm{tList}[i])\text{vector function,}$$

where $w = \dfrac{t - \mathrm{tList}[i]}{\mathrm{tList}[i+1] - \mathrm{tList}[i]}$, and *tList*[*i*]≤*t*≤*tList*[*i*+1]. A Hermite curve is shown in Figure O.3.7.1.

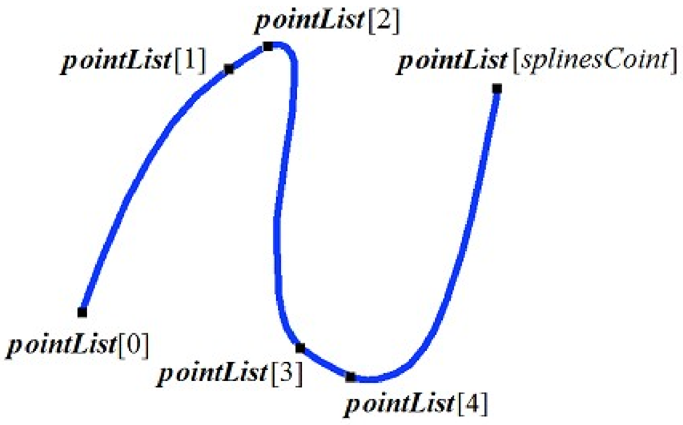


Fig. O.3.7.1.

t curve parameter takes values in *tmin*≤*t*≤*tmax* section, where *tmin*=*tList*[0], *tmax*=*tList*[*splinesCount*]. The curve may be periodic.

Curve form depends on location of control points, curve derivatives in control points, and on *tList* set of parameter values in control points. If a curve is constructed using only control points, then the values of curve parameter in *tList*[*i*], *i*=0,1,...,*splinesCount* control points are directly proportional to distance between points, and ***vectorList***[*i*], *i*=1,2,...,*splinesCount*–1 derivatives are calculated by constructing a parabola that goes through three neighboring points (***pointList***[*i*–1], ***pointList***[*i*], ***pointList***[*i*+1]) in corresponding parameter values (*tList*[*i*–1], *tList*[*i*], *tList*[*i*+1]), then parabola derivative is calculated in the middle point.

O.3.8. MbBezier Two-Dimensional Bezier Composite Curve

MbBezier class is declared in cur_bezier.h file.

Bezier 2D composite curve is an inheritor of MbPolyCurve curve. A curve is described by SArray<MbCartPoint>***pointList*** set of control points, *splinesCount* number of Bezier curves and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Curve is constructed on the basis of *splinesCount* third-order smoothly meeting Bezier curves. Each Bezier curve is defined by four control points and it goes through only two extreme points. A composite curve is used to construct a spline that goes through specified points. Specified points are joining points of third-order Bezier curves. A pair of internal control points for each third-order Bezier curve should be defined taking intp account the fact that this curve should smoothly meet with neighboring curves. For a composite curve, the number of control points is equal to 3(*splinesCount*+1). For a non-periodic composite curves, the first ***pointList***[0] control point and the last one are not used.

Every third-order Bezier curve increases composite curve parameter by one. When the radius vector is calculated, we first use the value of *t* parameter to find the number of the working segment (number of third-order Bezier curve) that is equal to the maximum integer not exceeding *t*. Let the number of third-order Bezier curve be equal to *n*. Then the fractional part of *w*=*t*–*n* parameter is defined. Radius vector of the composite curve is calculated as the radius vector of the found segment for its local parameter *w*.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, ***r*** radius vector of the curve is described by vector function of the found segment for its local parameter *w*:

$$r(t)=\frac{\sum_{j=0}^{pointsCount-1} N_{j_{degree}}(t)\, weight[j]\, \mathrm{pointList}[j]}{\sum_{j=0}^{pointsCount-1} N_{j_{degree}}(t)\, weight[j]}$$ vector function,

where *w*=*t*–*n*, *n*≤*t*≤*n*+1, 0≤*w*≤1, $B_j^3(w)=\frac{3!}{j!(3-j)!}w^j(1-w)^{3-j}$ are third-order Bernstein functions for *j*th, *j*=0,1,2,3, ***pointList***[3*n*+*j*] control point of the found segment number *n*. Bezier composite curve is shown in Figure O.3.8.1.

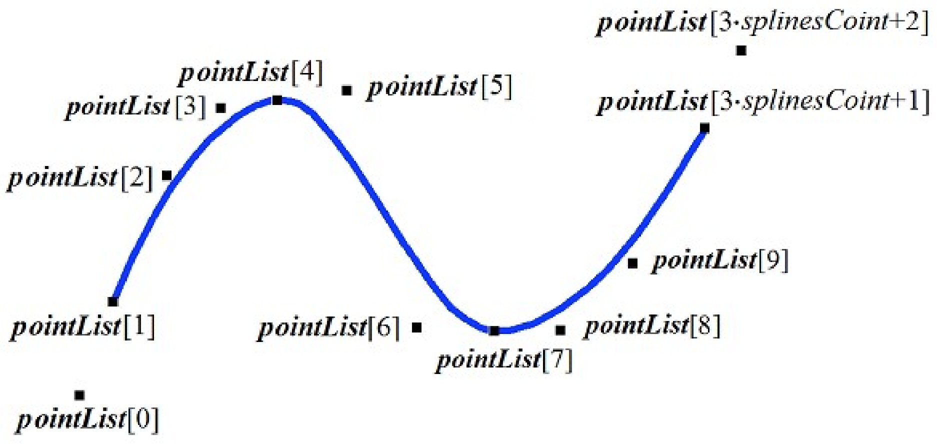


Fig. O.3.8.1.

t curve parameter takes values in 0≤*t*≤*splinesCount* segment. The curve may be periodic. The period of the periodic curve is equal to *splinesCount*.

If the parameter takes integer values, then the curve goes through control points. For example, if *t*=*n*, then the curve goes through ***pointList***[3*n*], *n*=0,1,...,*splinesCount* control point. Derivatives of the curve in joining points of third-order Bezier curves (at integer parameter values) lose the continuity by length.

O.3.9. MbCubicSpline Two-Dimensional Cubic Spline

MbCubicSpline class is declared in cur_cubic_spline.h file.

Two-dimensional cubic spline is an inheritor of MbPolyCurvecurve. A curve is described by SArray<MbCartPoint>***pointList*** set of 2D control points, SArray<MbVector>***vectorList*** set of second derivatives of the curve in control points, *tList* set of parameter values in curve control points, maximum index value of *splinesCount* set of parameters, and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

If *tList*[*i*], *i*=0,1,...,*splinesCount*, then the cubic spline goes through ***pointList***[*i*] control point and has ***vectorList***[*i*] second derivative in it. The curve is constructed so that at transition from ***pointList***[*i*] point to

pointList[i+1], the second derivative of curve radius vector varies linearly and takes values from ***vectorList***[i] to ***vectorList***[i+1].

When radius vector of composite curve is calculated, we first use the value of t parameter to find i number of the working segment from $tList[i] \le t \le tList[i+1]$ condition. Curve radius vector is calculated using ***pointList***[i], ***pointList***[i+1], ***vectorList***[i], ***vectorList***[i+1] values of the found segment for w local parameter, that is defined based on *tList*[i] and *tList*[i+1].

In **PointOn**(double & t, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$r(t) = (1 - w)\text{pointList}[i] + w\text{pointList}[i+1] +$$
$$+ ((-2w + 3w^2 - w^3)\text{vectorList}[i] + (-w + w^3)\text{vectorList}[i+1])\frac{(\text{tList}[i+1] - \text{tList}[i])^2}{6}$$

vector function,

where $w = \dfrac{t - \text{tList}[i]}{\text{tList}[i+1] - \text{tList}[i]}$, and $tList[i] \le t \le tList[i+1]$. A cubic spline that was constructed based on the same control points as Hermite composite curve is shown in Figure O.3.9.1.

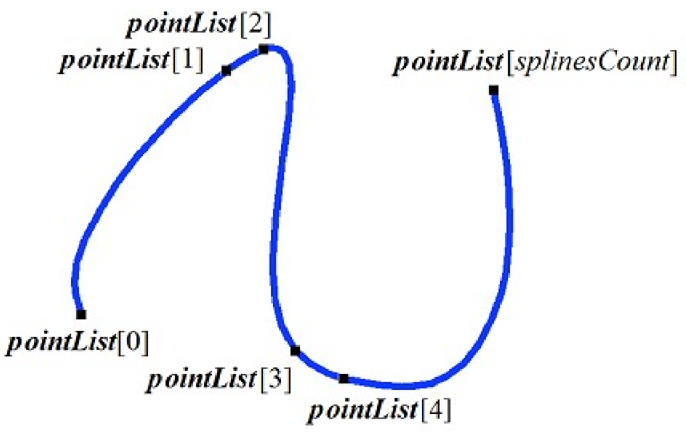


Fig. O.3.9.1.

t curve parameter takes values in $tmin \le t \le tmax$ section, where $tmin=tList[0]$, $tmax=tList[splinesCount]$. The curve may be periodic.

Curve form depends on the location of control points and *tList* set of parameter values in control points. If a curve is constructed using only control points, then the values of curve parameter in *tList*[i], i=0,1,...,*splinesCount* control points are directly proportional to distance between points, and ***vectorList***[i], i=1,2,...,*splinesCount*–1 second derivatives are calculated by solving a system of equations.

O.3.10. MbTrimmedCurve Two-Dimensional Truncated Curve

MbTrimmedCurve class is declared in cur_trimmed_curve.h file.

Two-dimensional truncated curve is described by MbCurve* ***basisCurve*** base curve, *trim*1 initial truncating parameter of the base curve, *trim*2 end truncating parameter of the base curve, and the *sense* sign of coincidence of directions of the base curve and the truncated curve.

Truncated curve coincides with the base curve within a segment defined by *trim*1 and *trim*2 parameters; at the same time, it can have a direction opposite to that of the segment. If *sense*=1, then *trim*1<*trim*2, then the directions of truncated curve and the base curve are the same. If *sense*=–1, then *trim*2<*trim*1 then the direction of the trimmed curve is opposite to that of the base curve.

In **PointOn**(double & t, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{basisCurve}(trim1 + sense\,t).$$

A truncated curve is shown in Figure O.3.10.1.

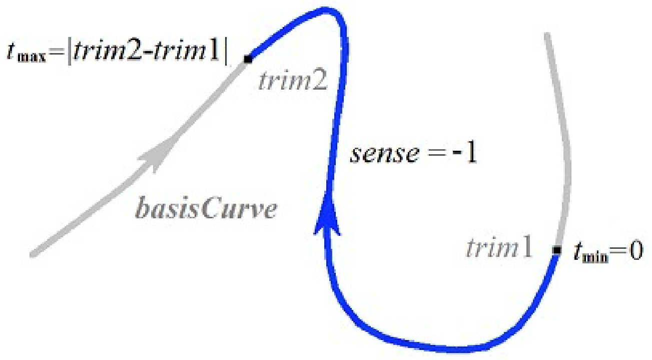


Fig. O.3.10.1.

t curve parameter takes values in 0≤*t*≤*sense*(*trim*2–*trim*1) range.

Theoretically, a truncated curve can be used to change the direction of the curve, but it is recommended to use **Inverse**() method.

A truncated curve permits you to change location of the initial point of periodic curve. In this case, the base curve should be periodic and *trim*2=*trim*1+*period*. In this case, a truncated curve will also be periodic.

A trimmed curve can't use other trimmed curve as a base curve; a base curve of other truncated curve should be used subject to corresponding recalculation of truncation parameters.

Every curve can construct its truncated copy using **Trimmed**(double *t*1, double *t*2, int *sense*) virtual method.

O.3.11. MbReparamCurve Two-Dimensional Reparameterized Curve

MbReparamCurve class is declared in cur_reparam_curve.h file.

Two-dimensional reparameterized curve is described by MbCurve* ***basisCurve*** base curve, *tmin* initial parameter, *tmin* end parameter, and *dt* derivative of the base curve parameter with respect to the parameter of reparameterized curve.

Reparameterized curve almost completely coincides with the base curve, but it has other parameter variation area.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{basisCurve}(v(t)) \text{ function,}$$

where $v(t) = b\mathrm{min}\dfrac{\mathrm{trim}2 - t}{\mathrm{trim}2 - \mathrm{trim}1} + b\mathrm{max}\dfrac{t - \mathrm{trim}1}{\mathrm{trim}2 - \mathrm{trim}1}$, *bmin*, *bmax* are the limit values of the base curve parameter definition area.

t curve parameter takes values in *tmin*≤*t*≤*tmax range.*

Reparameterized curve almost completely coincides with the base curve, but it has other parameter definition area. A curve with modified length parameter is used to align parameter variation areas of two curves. For example, if you want a segment and an arc to have the same parameter variation area, then it is required to create a reparameterized curve on the basis of another curve from the list using parameter variation area of the another curve.

A reparameterized curve should't use another reparameterized curve as the base curve; the base curve of another reparameterized curve should be used.

O.3.12. MbOffsetCurve Two-Dimensional Equidistant Curve

MbOffsetCurve class is declared in cur_offset_curve.h file.

Two-dimensional equidistant curve is described by MbCurve* ***basisCurve*** base curve, MbVector *distance* offset, *dmin* modified minimum parameter of base curve, *dmax* modified maximum parameter of base curve, *tmin* minimum parameter of base curve, *tmax* maximum parameter of base curve, MbMatrix ***transform*** transformation matrix and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Two-dimensional equidistant curve is a curve having corresponding parameter points are set off at *distance* from the corresponding point of ***basisCurve*** base curve. Parameter variation area of 2D equidistant curve differs from parameter variation area of base curve by *dmin* for the minimum value and by *dmax* for the maximum value.

Radius vector of a point of equidistant curve is calculated as follows. Point and normal are calculated for the specified parameter of the base curve. Then the point is set off by *distance* along the normal to the curve.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{basisCurve}(t) + \boldsymbol{normal}(t) \cdot distance \text{ vector function,}$$

where ***normal***(*t*) is the normal to the base curve that was obtained by rotating the tangent of the base curve at the given point by 90 degrees counterclockwise. Equidistant curve and base curve are shown in Figure O.3.12.1.

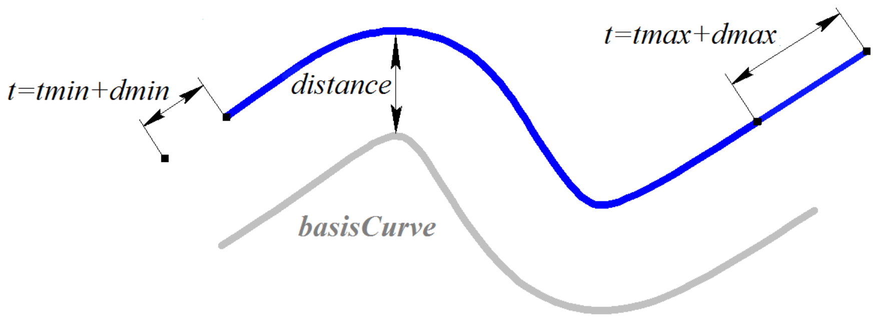


Fig. O.3.12.1.

t curve parameter takes values in $tmin+dmin \le t \le tmax+dmax$ range. If the parameter goes beyond the definition area, then the radius vector of a point in the base curve is calculated using **_PointOn**(double *t*, MbCartPoint & ***r***) method. If *distance*=0, *dmin*=0, *dmax*=0, then the equidistant curve coincides with the base curve.

An equidistant curve may not use other equidistant curve as the base curve; the base curve of other equidistant curve should be used subject to corresponding recalculation of the offset.

Every curve can construct an equidistant curve using **Offset**(double *distance*) virtual method.

O.3.13. MbCharCurve Two-Dimensional Character Curve

MbCharacterCurve class is declared in cur_character_curve.h file.

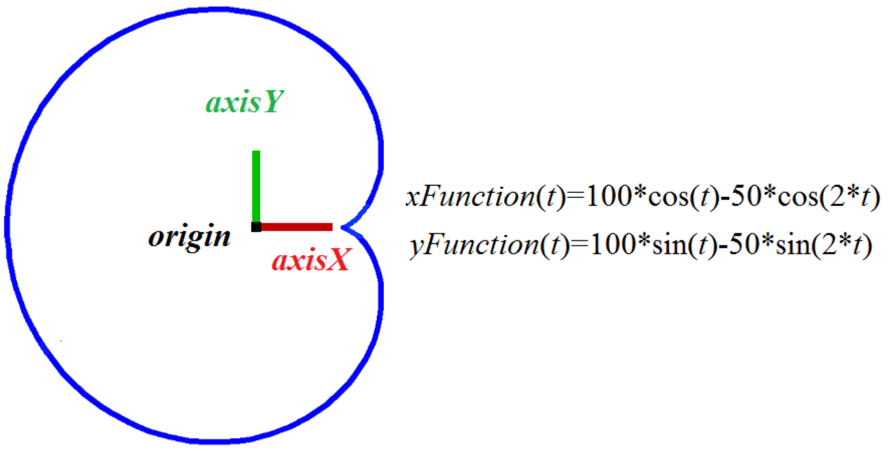
Character curve is described by *xFunction*, *yFunction* coordinate functions, MbPlacement ***position*** local coordinate system, ***transform*** transformation matrix, *tmin* and *tmax* limit values of curve parameter definition area, *closed* curve periodicity sign and *coordinateType* type of coordinate system (Cartesian, polar), in which coordinate functions are defined. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

xFunction(*t*), *yFunction*(*t*) coordinate functions of the character curve are scalar functions of *t* common parameter that are defined as character expressions. Lexical analysis was conducted for each character expression. In addition, a tree was constructed to calculate values of character expression for specified parameters, as well as derivatives of character expressions with respect to the parameter. *t* curve parameter has values in: *tmin*≤*t*≤*tmax* range.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = [xFunction(t) \quad yFunction(t)] \text{ vector function.}$$

A character curve is shown in Figure O.3.13.1.



axisY
origin
axisX
xFunction(t)=100*cos(t)-50*cos(2*t)
yFunction(t)=100*sin(t)-50*sin(2*t)

Fig. O.3.13.1.

The curve may be periodic. Character expressions in curve definition area should describe continuous and single-valued functions.

O.3.14. MbCosinusoid Two-Dimensional Cosine Wave

MbCosinusoid class is declared in cur_cosinusoid.h file.

Two-dimensional cosine wave is described by MbPlacement ***position*** local coordinate system, *frequency* cyclic frequency, *phase* initial phase, *amplitude* amplitude, *tmin* minimum curve parameter and *tmax* maximum curve parameter. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Two-dimensional cosine wave is a cosine function, its argument is given along ***position.axisX*** vector; the value of the function is plotted along ***position.axisY*** vector. The function has *amplitude* amplitude, *frequency* frequency and *phase* initial phase.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{position.origin} + \\ (((tmin+t-phase) \,/\, frequency)\, \boldsymbol{position.axisX}) + (amplitude \cos(tmin+t)\, \boldsymbol{position.axisY}) \text{ vector function.}$$

Cosine wave is shown in figure O.3.14.1.

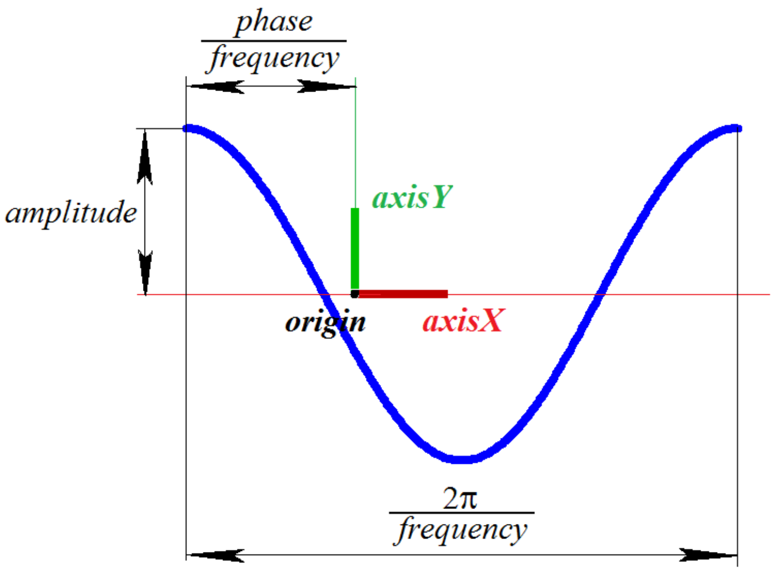


Fig. O.3.14.1.

t curve parameter takes values in *tmin*≤*t*≤*tmax* range. The following inequality should hold for parameters: *tmin*<*tmax*. The curve can't be periodic. Amplitude and frequency of the curve should be greater than zero: *amplitude*>0, *frequency*>0.

position local coordinate system may be either left- or right-handed. A cosine wave is used to describe intersection of a cylindrical surface and a plane.

O.3.15. MbPointCurve Two-Dimensional Curve-Point

MbPointCurve class is declared in cur_point_curve.h file.

Two-dimensional curve-point is described by MbCartPoint ***point***, *tmin* minimum curve parameter, *tmax* maximum curve parameter and *closed* curve periodicity sign.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{point} \text{ function.}$$

t curve parameter takes values in *tmin*≤*t*≤*tmax* range. The curve may be periodic. The following inequality should hold for parameters: *tmin*<*tmax*.

Two-dimensional curves-points are used in pair with other two-dimensional curve that describes the intersection of surfaces, one of which has a special point such as a pole. *tmin*, *tmax*, *closed* parameters of the curve-point coincide with parameters of the two-dimensional curve that is used in pair with the curve-point.

O.3.16. MbProjCurve Two-Dimensional Projection Curve

MbProjCurve class is declared in cur_projection_curve.h file.

Two-dimensional projection curve is described by MbCurve3D* **spaceCurve** 3D curve, MbSurface* **surface** surface and MbCurve* ***curve*** 2D curve. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Two-dimensional projection curve is a projection of **spaceCurve** 3D curve to **surface**, that is approximately described by ***curve*** 2D curve in surface parameter definition area. Parameter definition areas of **spaceCurve** and ***curve*** curves are the same. ***curve*** 2D curve is usually a spline, its control points are received by projecting points of **spaceCurve** 3D curve to **surface**. Parameterization of ***curve*** is aligned with parameterization of the 3D curve in control points. ***curve*** 2D curve can be located outside of the surface parameter definition area.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = [u \;\; v] \text{ vector function,}$$

where *u*, *v* are parameters of projection of **spaceCurve**(*t*) point to **surface**. Initial approximation of *u* and *v* parameters are calculated using the following method: ***curve***–>**PointOn**(*t*,***point***), *u*=***point***.*x*, *v*=***point***.*y*. Then *u* and *v* parameters are improved by iterative method based on the following equations

$$\textbf{deriveU} \cdot (\textbf{spaceCurve}(t) - \textbf{surface}(\, u, v \,)) = 0,$$
$$\textbf{deriveV} \cdot (\textbf{spaceCurve}(t) - \textbf{surface}(\, u, v \,)) = 0,$$

where **deriveU** and **deriveV** are partial derivatives of surface radius vector, they are calculated using the **surface**–>**_DeriveU**(*u,v*,**deriveU**) and **surface**–>**_DeriveV**(*u,v*,**deriveV**) methods, respectively. A projection curve is shown in Figure O.3.16.1.

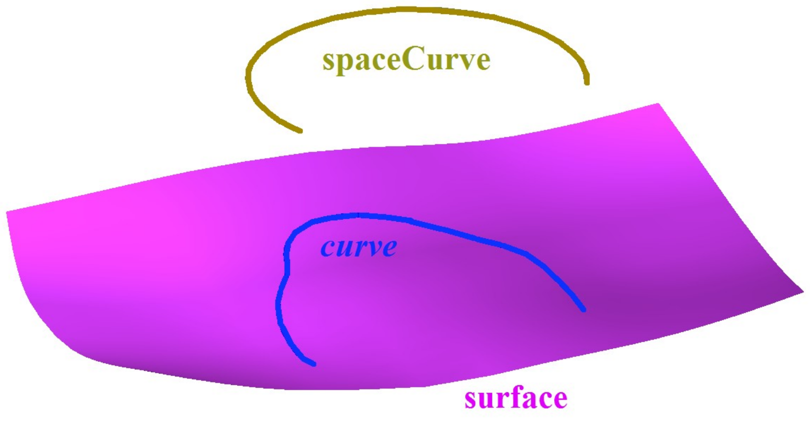


Fig. O.3.16.1.

A projection curve is used to accurately describe a projection of 3D curve on a surface.

O.3.17. MbContour Two-Dimensional Contour

MbContour class is declared in cur_contour.h file.

MbContour 2D contour is described by RPArray<MbCurve>***segments*** set of sequentially joined curves and *closed* curve periodicity sign.

Two-dimensional contour is a composite curve. Unlike other curves, a contour may have kinks. A curve that creates a contour will be called a segment. The following conditions are met for contour segments: initial point of each successive segment coincides with the end point of the previous one. For periodic contour, initial point of the first segment coincides with the end point of the last one. In general, contour derivatives have discontinuities by length and direction at joining points of segments.

Initial value of contour parameter is zero: $t_{\min}$=0. Parametric length of a contour is equal to the sum of the lengths of parametric lengths of its segments: $t_{\max} = \sum(w_{i\max} - w_{i\min})$, where $w_{i\min}$ and $w_{i\max}$ are minimum and maximum values of the *i*th segment parameter. When radius vector of a point of the contour is calculated, we first use parameter value to determine the working segment and the value of its local parameter, and then we calculate a radius vector of the working segment, which is a radius vector of the contour.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, the radius vector of ***r*** curve is described by

$$\boldsymbol{r}(t) = \boldsymbol{segments}[k](w_k) \text{ vector function}$$

where ***segments***[*k*](w_k) is the working segment of the *k*th contour, w_k is the parameter of the working segment that is equal to: $w_k = w_{k\min} + t - \sum_{i=0}^{k-1}(w_{i\max} - w_{i\min})$. The *k*th segment is defined by the value of *t* parameter of the contour according to condition $\sum_{i=0}^{k-1}(w_{i\max} - w_{i\min}) \le t < \sum_{i=0}^{k}(w_{i\max} - w_{i\min})$, where $w_{i\min}$ and $w_{i\max}$ are minimum and maximum values of the *i*th segment parameter. A contour is shown in Figure O.3.17.1.

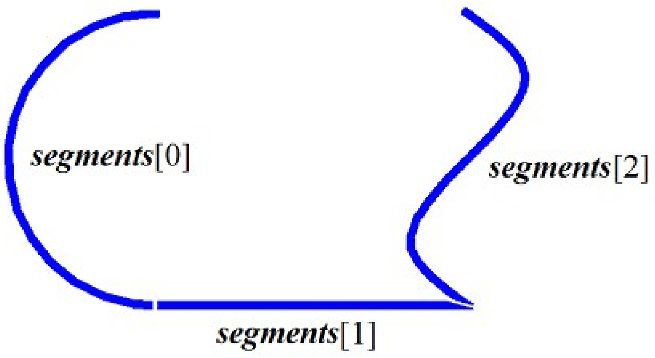


Fig. O.3.17.1.

A 2D contour can't be used as a segment of other 2D contours. If other contours should be used to construct a contour, then such initial contours should be considered as a set of curves rather than a single curve.

O.4. CURVES

Curves belong to MbSpaceItem family of three-dimensional geometric objects. All curves have the same parent class MbCurve3D. C3D geometric kernel uses curves that are constructed using analytic functions, by set of points, based on curves and based on surfaces. Curves are used to construct surfaces and auxiliary elements in a geometric model. We'll use **bold** Latin letters to designate vectors, radius vectors of points, and matrices in three-dimensional space.

O.4.1. MbCurve3D Curve

MbCurve3D class is declared in curve3d.h file.
MbCurve3D curve is an inheritor of MbSpaceItem class, see Figure O.4.1.1.

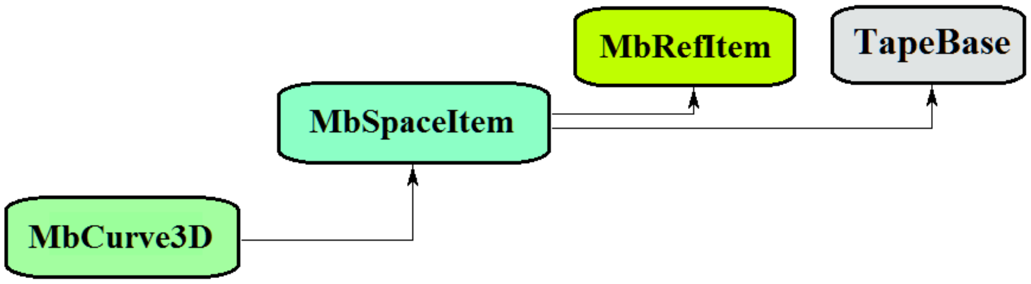


Fig. O.4.1.1.

The curve is an abstract class. The following curves are inheritors of MbCurve class in C3D geometric kernel:
MbLine3D – a straight line
MbLineSegment3D – a straight line segment
MbArc3D – an elliptical arc
MbPolyline3D – a polyline
MbNurbs3D – a B-curve (NonUniform Rational B-Spline)
MbBezier3D – a Bezier composite curve
MbHermit3D – a Hermite curve
MbCubicSpline3D – a cubic spline
MbOffsetCurve3D – a equidistant curve
MbTrimmedCurve3D – a trimmed curve
MbReparamCurve3D – a reparametrized curve
MbCharacterCurve3D – a curve with symbolical coordinate functions
MbConeSpiral – a conical spiral
MbCurveSpiral – a spiral with a rectilinear axis and variable radius
MbCrookedSpiral – a spiral with axis in the form of a flat curve
MbBridge – a Hermite spline connecting two curves
MbContour3D – a contour (composite curve)
MbPlaneCurve – a flat curve in 3D space
MbSurfaceCurve – a curve on a surface
MbSilhouetteCurve – a silhouette curve of a surface
MbContourOnSurface – a contour on a surface
MbContourOnPlane – a contour on a plane
MbSurfaceIntersectionCurve – an intersectional curve of surfaces.

MbCurve3D is a vector function

$$\text{curve}(t) = [x(t) \quad y(t) \quad z(t)]$$

of t scalar parameter taking values in [$t_{\min}$, $t_{\max}$] range. The curve is a continuous projection of a part of number axis into three-dimensional space. Curve parameter range is [$t_{\min}$, $t_{\max}$] range in one-dimensional space. $x(t)$, $y(t)$, $z(t)$ coordinates of a point in **curve**(t) curve are single-valued continuous functions of t parameter.

$t_{\min}$ and $t_{\max}$ limit values of parameter range are received using double **GetTMin**() and double **GetTMax**() curve methods, respectively.

A curve is referred as periodic if there is p>0 such that **curve**(t±kp)=**curve**(t) holds, where k is an integer. bool **IsClosed**() method returns true for a periodic curve. double **GetPeriod**() method returns p period of periodic curve (or a curve that can be extended and made periodic). Periodic curve parameter range is always limited to one period.

The main method for the curve is
void **PointOn**(double & t, MbCartPoint3D & **r**).
It returns **r** radius vector of curve point for specified t parameter.
void **FirstDer**(double & t, MbVector3D & $\mathbf{r}_t$),
void **SecondDer**(double & t, MbVector3D & $\mathbf{r}_{tt}$),
void **ThirdDer**(double & t, MbVector3D & $\mathbf{r}_{ttt}$) methods
return respectively the first ($\mathbf{r}_t$), the second ($\mathbf{r}_{tt}$) and the third ($\mathbf{r}_{ttt}$) derivatives of curve radius vector for specified parameter (t). These methods adjust curve parameter if it goes beyond the range (except for a straight line MbLine3D). If curve parameter (t) goes beyond [$t_{\min}$, $t_{\max}$] range, then a non-periodic curve moves the parameter (t) to the nearest limit $t_{\min}$ or $t_{\max}$, and a periodic curve adds or subtracts required number of periods.
void **_PointOn**(double t, MbCartPoint3D & **r**) method
returns radius vector (**r**) of curve point for specified parameter (t) both inside and outside curve parameter range. In general, a non-periodic curve as extended outside its parameter range along the tangent [to the curve] in the end point. Periodic curve, an arc (MbArc3D), a spiral (MbSpiral), a character curve (MbCharacterCurve3D) and a trimmed curve (MbTrimmedCurve3D) within base curve limits are the exceptions. A periodic curve is extended cyclically beyond the limits of its parameter range.
void **_FirstDer**(double t, MbVector3D & $\mathbf{r}_t$),
void **_SecondDer**(double t, MbVector3D & $\mathbf{r}_{tt}$),
void **_ThirdDer**(double t, MbVector3D & $\mathbf{r}_{ttt}$) methods
respectively return the first ($\mathbf{r}_t$), the second ($\mathbf{r}_{tt}$) and the third $\mathbf{r}_{ttt}$ derivatives of curve radius vector for specified t parameter both inside and outside curve range.

The curves reload the following 3D geometrical object methods:
the methods involved in transformation of a geometrical object,
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),
the methods that permit to copy, check for coinciding objects, check whether it's possible to make objects coinciding and make them coinciding:
MbSpaceItem & **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbSpaceItem & **item**),
bool **IsSimilar**(const MbSpaceItem & **item**),
bool **SetEqual**(const MbSpaceItem & **item**),
the methods that return a type from enumeration of geometric objects,
MbeSpaceType **IsA**(),
MbeSpaceType **Type**(),
MbeSpaceType **Family**(),
the methods that ensure access and editing of object internal data,
MbProperty & **CreateProperty**(MbePrompt name),
void **GetProperties**(MbProperties & *properties*),
void **SetProperties**(MbProperties & *properties*),
the method that fills up a polygonal copy of a geometrical object,
CalculateWire(double sag, MbMesh & **mesh**).

All curves besides MbContour3D, MbContourOnSurface, MbContourOnPlane usually don't have bends. MbContour3D, MbContourOnSurface, MbContourOnPlane are composite curves that may have bends in the points where their constituting segments join.

O.4.2. MbLine3D Straight Line

MbLinet3D class is declared in cur_line_3d.h file.

MbLine3D straight line is described by MbCartPoint3D **origin** initial point and MbVector3D **direction** directional vector, see Figure O.4.2.1.

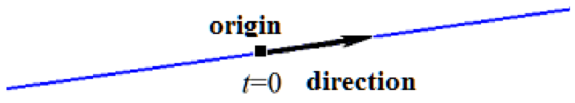


Fig. O.4.2.1.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, radius vector of **r** straight line is described by

$$\mathbf{r}(t) = \mathbf{origin} + t\ \mathbf{direction} \text{ vector function.}$$

The straight line behaves as an infinite object, but it has *tmin* and *tmax* parameter limits. Note that unlike all other curves, a straight line doesn't adjust *t* parameter when it goes beyond *tmin* and *tmax* limits when radius vector and its derivatives are calculated using corresponding methods.

O.4.3. MbLineSegment3D Straight Line Segment

MbLineSegment3D class is declared in cur_line_segment_3d.h file.

MbLineSegment3D straight line segment is described by MbCartPoint3D **point1** initial point and MbCartPoint3D **point2** end point, see Figure O.4.3.1.

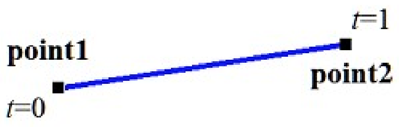


Fig. O.4.3.1.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, radius vector of **r** segment is described by

$$\mathbf{r}(t) = (1 - t)\ \mathbf{point1} + t\ \mathbf{point2} \text{ vector function.}$$

Segment parameter definition area ranges from zero to one. **point1** segment initial point corresponds to $t_{\min}=0$ parameter, and **point2** segment end point corresponds to $t_{\max}=1$ parameter.

O.4.4. MbArc3D Elliptical Arc

MbArc3D class is declared in cur_arc_3d.h file.

An elliptical arc is an inheritor of MbCurve3D curve. MbArc3D elliptical arc is described by *a* and *b* radii, as well as *trim*1 and *trim*2 angles defined in MbPlacement3D **position** local coordinate system.

*trim*1 and *trim*2 angles are measured along the arc from **position.axisX** vector towards **position.axisY** vector. *trim*1 and *trim*2 angles shall be designated as "trimming parameters". Trimming parameters equal to zero and 2π correspond to a point on **position.axisX** axis. *t* curve parameter takes values in $0 \le t \le trim2 - trim1$ range. The curve may be periodic. $trim2 - trim1 = 2\pi$ holds for a periodic curve.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{position.origin} + a\cos(trim1+t)\ \mathbf{position.axisX} + b\sin(trim1+t)\ \mathbf{position.axisY} \text{ vector function.}$$

An elliptical arc is shown in Figure O.4.4.1.

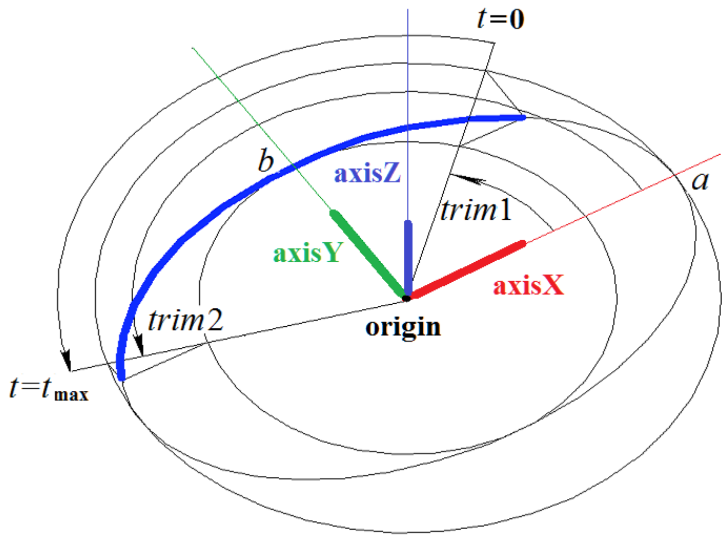


Fig. O.4.4.1.

Curve radii should be greater than zero: $a>0$, $b>0$. The following inequalities should hold for trimming parameters: $trim1 < trim2$.

position local coordinate system may be either right- or left-handed.

O.4.5. MbPolyline3D Polyline

MbPolyline3D class is declared in cur_polyline3d.h file.

A polyline is an inheritor of MbPolyCurve3D curve. MbPolyline3D polyline is described by *segmentsCount* number of segments, SArray<MbCartPoint3D>**pointList** set of control points and *closed* curve periodicity sign.

The curve goes through **pointList**[*i*], $i=0,\dots,segmentsCount.$ set of points at $t=0,\dots,segmentsCount$ parameter values. If *closed=true,* then the curve contains a segment connecting the last point of **pointList**[*segmentsCount*-1] set with the initial point of **pointList**[0] set. *t* curve parameter takes values in $0 \le t \le segmentsCount$ range.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{pointList}[i]\,(1-w) + \mathbf{pointList}[i+1]\, w \text{ vector function,}$$

where $w = \dfrac{t - t_i}{t_{i+1} - t_i}$, and $t_i \le t \le t_{i+1}$. A polyline is the simplest curve constructed based on a set of points. It consists of segments that consequently connect the control points. A polyline is shown in Figure O.4.5.1.

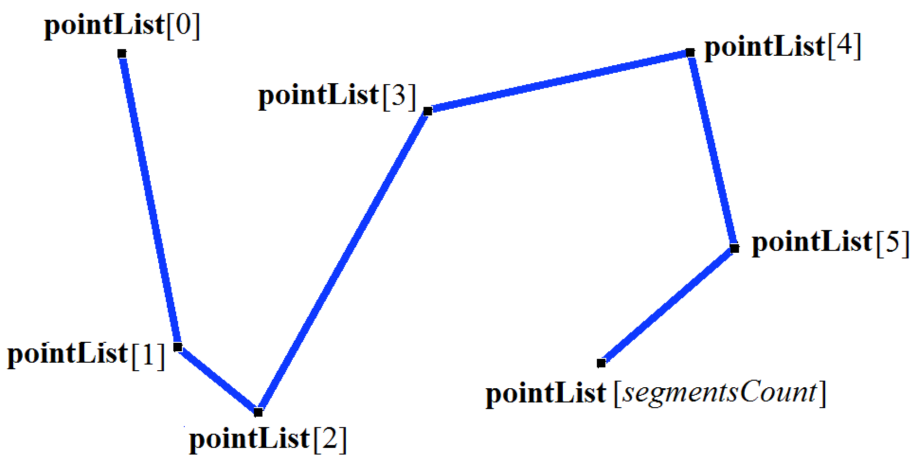


Fig. O.4.5.1.

The curve may be periodic. *segmentsCount* is a period of periodic curve. Derivatives of the curve in control points (when parameter values are integers) lose continuity by length and direction. A derivative of the curve in a control point has special length and direction. A polyline has a number of useful features: minimum amount of computation is required to work with it; projection of polyline will also be a polyline.

O.4.6. MbNurbs3D NURBS-Curve

MbNurbs3D class is declared in cur_nurbs3d.h file.

NURBS-curve is an acronym of NonUniform Rational B-Spline. The curve is an inheritor of MbPolyCurve3D. The curve is described by SArray<MbCartPoint3D>**pointList** set of control points, *weights* set of control point weights, *knots* node vector, *degree* spline order, *form* curve parameter and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

The curve is constructed based on B-splines. *knots* node vector is a non-decreasing sequence of real numbers that defines curve parameter range and curve shape. In general, *form* curve shape parameter is equal to ncf_Unspecified; in particular cases, it stores original curve data that were used to construct a NURBS-copy. *degree* order of NURBS-curve is equal to the order of divided differences used to calculate B-splines. Let node vector have *knotsCount* elements, and let the set of control points contain *pointsCount* elements. For non-periodic NURBS-curve, the following equation holds for the number of elements in sets: *knotsCount*=*pointsCount*+*degree*. For periodic NURBS-curve, the following equation holds for the number of elements in sets: *knotsCount*=*pointsCount*+2*degree*–1.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$r(t) = \frac{\sum_{j=0}^{\text{pointsCount}-1} N_j^{\text{degree}}(t)\,\text{weight}[j]\,\text{pointList}[j]}{\sum_{j=0}^{\text{pointsCount}-1} N_j^{\text{degree}}(t)\,\text{weight}[j]}$$

vector function, where $N_j^{degree}(t)$ is *degree* order B-splines for the *j*th control point from **pointList**[*j*]. NURBS-curve is shown in Figure O.4.6.1.

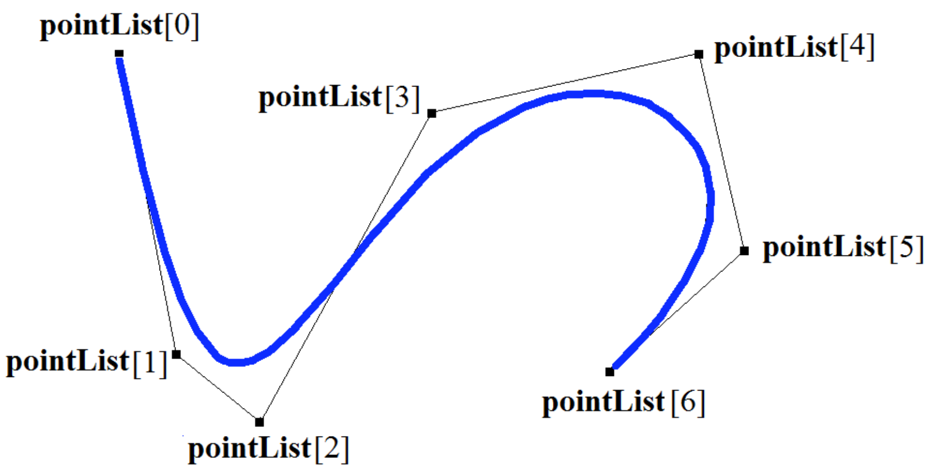


Fig. O.4.6.1.

t curve parameter takes values in *tmin*≤*t*≤*tmax* range, where *tmin*=*knots*[*degree*–1], *tmax*=*knots*[*knotsCount*–*degree*]. The curve may be periodic. A periodic NURBS-curve is shown in Figure O.4.6.2.

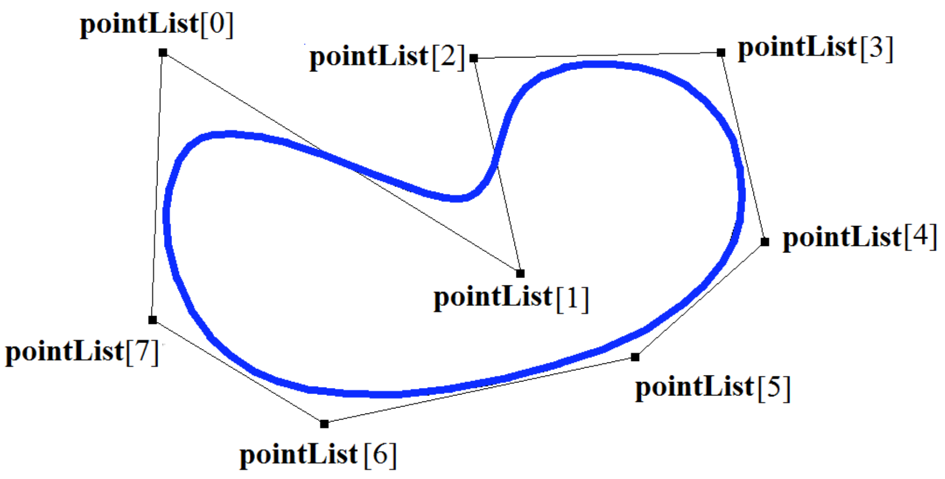


Fig. O.4.6.2.

The shape of NURBS-curve depends on location and weight of control points, as well as node vector values. In general, NURBS-curve does not go through **pointList**[i], i=0,...,*pointsCount*–1 set of points. For non-closed NURBS-curve to go through extreme control points it is required that the first *degree* elements and the last *degree* elements of *knots* node vector coincide. Other things equal, the distance between the curve and the control point depends on the weight of the control point.

NURBS-curve can be constructed by any curve using **NurbsCurve**(const MbNurbsParameters & *tParameters*) virtual method.

O.4.7. MbHermit3D Hermite Curve

MbHermit3D class is declared in cur_hermit3d.h file.

Hermite curve is an inheritor of MbPolyCurve3D. The curve is described by SArray<MbCartPoint3D>**pointList** set of control points, SArray<MbVector3D>**vectorList** set of curve derivatives in control points, *tList* set of parameter values in curve control points, *splinesCount* Hermite cubic splines and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

At *tList*[*i*], *i*=0,1,...,*splinesCount* value, a Hermite curve goes through **pointList**[*i*] control point and it has **vectorList**[*i*] derivative in it. The curve is constructed on the basis of *splinesCount* smoothly adjoined third-order Hermite splines. Each Hermite cubic spline describes a section of the curve between two neighboring control points. Each Hermite cubic spline is defined by two extreme points and two derivatives of the curve in these points.

When radius vector of Hermite curve point is calculated, we first use the value of *t* parameter to find *i*, working segment number (Hermite cubic spline number) from *tList*[*i*]≤*t*≤*tList*[*i*+1]. Curve radius vector is calculated as a radius vector of the found segment for its local parameter (*w*) that is defined from *tList*[*i*] and *tList*[*i*+1].

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by a vector function of the found segment for its local parameter *w*

$$r(t) = (1-3w^2+2w^3)\text{pointList}[i] + (3w^2+2w^3)\text{pointList}[i+1] +$$
$$+ ((w-2w^2+w^3)\text{vectorList}[i] + (-w^2+w^3)\text{vectorList}[i+1])(\text{tList}[i+1]-\text{tList}[i]),$$

where $w = \dfrac{t-\text{tList}[i]}{\text{tList}[i+1]-\text{tList}[i]}$, and *tList*[*i*]≤*t*≤*tList*[*i*+1]. A Hermite curve is shown in Figure O.4.7.1.

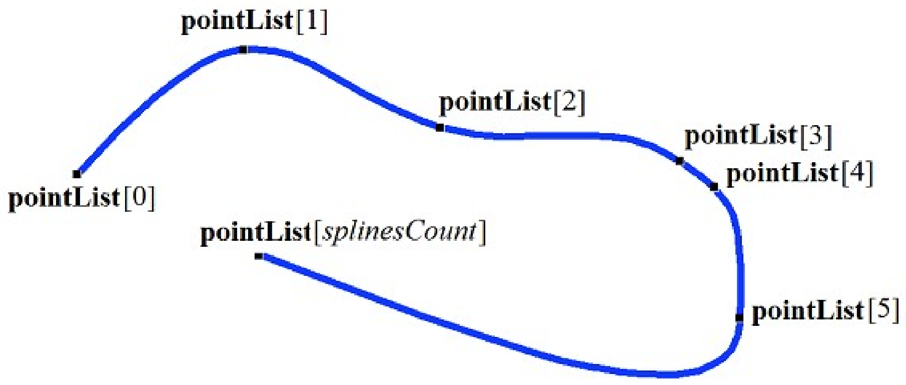


Fig. O.4.7.1.

t curve parameter takes values in *tmin*≤*t*≤*tmax* range, where *tmin*=*tList*[0], *tmax*=*tList*[*splinesCount*]. The curve may be periodic.

Curve shape depends on location of control points, derivatives of the curve in control points, as well as on *tList* set of parameter values in control points. If a curve is constructed using control points, only then values of curve parameter in *tList*[*i*], *i*=0,1,...,*splinesCount* control points are directly proportional to distance between the points, and **vectorList**[*i*], *i*=1,2,...,*splinesCount*–1 derivatives are calculated by constructing a parabola through three neighboring points (**pointList**[*i*–1], **pointList**[*i*], **pointList**[*i*+1]) taking into account corresponding values of parameter (*tList*[*i*–1], *tList*[*i*], *tList*[*i*+1]), and parabola derivative is calculated in the midpoint.

O.4.8. MbBezier3D Bezier Composite Curve

MbBezier3D class is declared in cur_bezier3d.h file.

Bezier two-dimensional composite curve is an inheritor of MbPolyCurve3D. A curve is described by SArray<MbCartPoint3D>**pointList** set of control points, *splinesCount* number of Bezier curves and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

The curve is constructed on the basis of *splinesCount* smoothly adjoined third-order Bezier curves. Each Bezier curve is defined by four control points and it goes through two extreme points only. A composite curve is used to construct a spline that goes through specified points. Specified points are used as joining points of third-order Bezier curves. A pair of internal control points for each third-order Bezier curve should be defined taking into account that the curve should to be smoothly adjoined to neighbor curves. For a composite curve the number of control points is equal to 3(*splinesCount*+1). For non-periodic composite curve, the first control point **pointList**[0] and the last control point are not used.

Every third-order Bezier curve increases composite curve parameter by one. When radius vector is calculated, we first use the value of *t* parameter to find working segment number (the number of third-order Bezier curve) that is equal to the maximum integer not exceeding *t*. Let the number of third-order Bezier curve be equal to *n*. Then a fractional part of *w*=*t*–*n* parameter is defined. Radius vector of composite curve is calculated as a radius vector of the found segment for its local parameter (*w*).

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by a vector function of the found segment for its local parameter *w*

$$r(t) = \sum_{j=0}^{3} \frac{3!}{j!(3-j)!} w^j (1-w)^{3-j} \text{pointList}[3n+j],$$

where $w=t-n$, $n \le t \le n+1$, $0 \le w \le 1$, $B_j^3(w) = \frac{3!}{j!(3-j)!} w^j (1-w)^{3-j}$ are third-order Bernstein functions for the *j*th, *j*=0,1,2,3, **pointList**[3*n*+*j*] control point of found section number *n*. A Bezier composite curve is shown in Figure O.4.8.1.

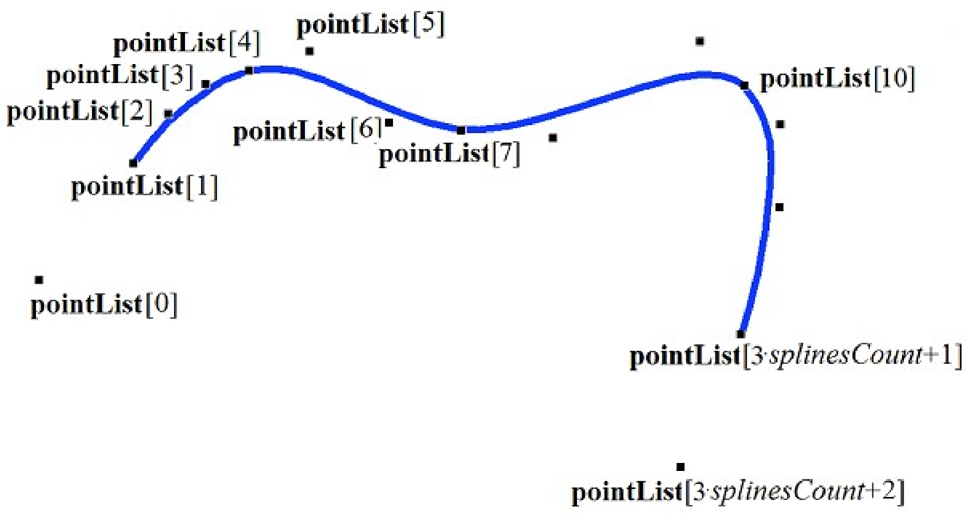


Fig. O.4.8.1.

t curve parameter takes values in 0≤*t*≤*splinesCount* range. The curve may be periodic. *splinesCount* is a period of periodic curve.

If the parameter takes integer values, then the curve goes through control points. For example, if *t*=*n*, then the curve goes through **pointList**[3*n*], *n*=0,1,...,*splinesCount* control point. Derivatives of the curve in joining points of third-order Bezier curves (when parameter values are integers) lose continuity by length.

O.4.9. MbCubicSpline3D Cubic Spline

MbCubicSpline3D class is declared in cur_cubic_spline3d.h file.

Cubic spline is an inheritor of MbPolyCurve3D. The curve is described by SArray<MbCartPoint3D>**pointList** set of control points, SArray<MbVector3D>**vectorList** set of second derivatives of the curve in control points, *tList* set of parameter values in curve control points, maximum index value of *splinesCount* set of parameters, and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

At *tList*[*i*], *i*=0,1,...,*splinesCount* parameter values, a cubic spline goes through **pointList**[*i*] control point and has **vectorList**[*i*] second derivative in it. The curve is constructed so that at transition from **pointList**[*i*] point to **pointList**[*i*+1] point the second derivative of curve radius vector changes linearly from **vectorList**[*i*] to **vectorList**[*i*+1].

When radius vector of composite curve is calculated, we first use *t* parameter value to find the *i* working segment number from *tList*[*i*]≤*t*≤*tList*[*i*+1]. Curve radius vector is calculated using **pointList**[*i*], **pointList**[*i*+1], **vectorList**[*i*], **vectorList**[*i*+1] values of the found segment for its local parameter *w,* that is defined from *tList*[*i*] and *tList*[*i*+1].

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$r(t) = (1-w)\text{pointList}[i] + w\text{pointList}[i+1] +$$
$$+ ((-2w+3w^2-w^3)\text{vectorList}[i] + (-w+w^3)\text{vectorList}[i+1])\frac{(\text{tList}[i+1]-\text{tList}[i])^2}{6}$$

vector function, where $w = \dfrac{t-\text{tList}[i]}{\text{tList}[i+1]-\text{tList}[i]}$, and *tList*[*i*]≤*t*≤*tList*[*i*+1]. A cubic spline that was constructed by the same control points as composite Hermite curve is shown in Figure O.4.9.1.

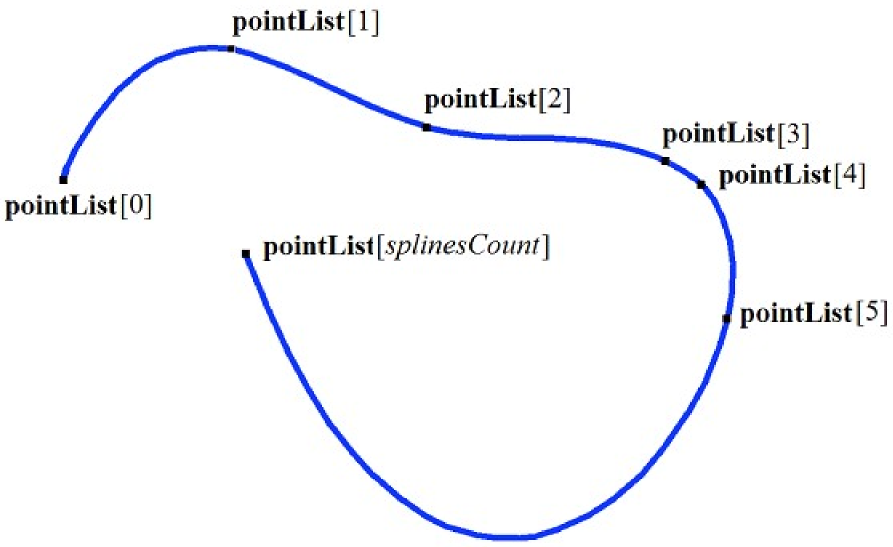


Fig. O.4.9.1.

t curve parameter takes values in *tmin*≤*t*≤*tmax* range, where *tmin*=*tList*[0], *tmax*=*tList*[*splinesCount*]. The curve may be periodic.

Curve shape depends on location of control points and *tList* set of parameter values in control points. If a curve is constructed by control points only, then the values of curve parameter in *tList*[*i*], *i*=0,1,...,*splinesCount* control points are directly proportional to distance between the points, and **vectorList**[*i*], *i*=1,2,...,*splinesCount*–1 second derivatives are calculated by solving a system of equations.

O.4.10. MbTrimmedCurve3D Trimmed Curve

MbTrimmedCurve3D class is declared in cur_trimmed_curve3d.h file.

A trimmed curve is described by MbCurve3D * **basisCurve** base curve, *trim*1 initial trimming parameter of the base curve, *trim*2 end trimming parameter of the base curve, and *sense* direction coincidence sign of the base curve and the trimmed curve.

The trimmed curve coincides with the base curve within a section defined by *trim*1 and *trim*2 parameters, but it can have an opposite direction. If *sense*=1, then *trim*1<*trim*2, and the directions of the trimmed curve and the base curves coinside. If *sense*=–1, then *trim*2<*trim*1, and direction of the trimmed curve is opposite to that of the base curve.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{basisCurve}(trim1+sense\cdot t) \text{ vector function.}$$

A trimmed curve is shown in Figure O.4.10.1.

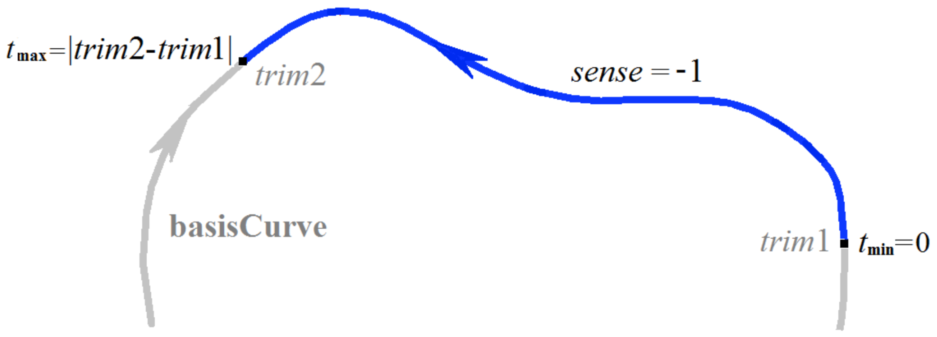


Fig. O.4.10.1.

t curve parameter takes values in $0 \le t \le sense(trim2-trim1)$ *range.*

Conceptually, a trimmed curve can be used to change the direction of the curve, but it is recommended to use **Inverse**() method.

A trimmed curve permits you to change location of the initial point of periodic curve. In this case, the base curve should be periodic and *trim*2=*trim*1+*period* should hold. In this case, a trimmed curve will also be periodic.

A trimmed curve can't use other trimmed curve as a base curve; base curve of other trimmed curve should be used subject to corresponding recalculation of trimming parameters.

Each curve can construct its trimmed copy using **Trimmed**(double *t*1, double *t*2, int *sense*) virtual method.

O.4.11. MbReparamCurve3D Reparametrized Curve

MbReparamCurve3D class is declared in cur_reparam_curve3d.h file.

A reparametrized curve is described by MbCurve3D * **basisCurve** base curve, *tmin* initial parameter, *tmin* end parameter, and *dt* derivative of base curve parameter to reparametrized curve parameter.

Reparametrized curve almost completely coincides with the base curve, but it has other parameter range.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{basisCurve}(v(t)) \text{ vector function,}$$

where $v(t) = bmin\dfrac{\mathrm{trim}2 - t}{\mathrm{trim}2 - \mathrm{trim}1} + bmax\dfrac{t - \mathrm{trim}1}{\mathrm{trim}2 - \mathrm{trim}1}$, *bmin*, *bmax* are the limiting values of base curve parameter range.

t curve parameter takes values in $tmin \le t \le tmax$ *range.*

Reparametrized curve almost completely coincides with the base curve, but it has other parameter definition range. A curve with modified parameter length is used to align parameter variation ranges of the two curves. For example, if you want a segment and an arc to have the same parameter range, then it is required to create a reparametrized curve with parameter range taken from the other curve.

A reparametrized curve can't use other reparametrized curve as a base curve; rather the base curve of other reparametrized curve should be used.

O.4.12. MbOffsetCurve3D Equidistant Curve

MbOffsetCurve3D class is declared in cur_offset_curve3d.h file.

Equidistant curve is described by MbCurve3D* **basisCurve** base curve and MbVector3D **offset** offset vector. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

basisCurve base curve is MbSpine object that constructs a local coordinate system moving along a specified curve, the first coordinate axis of its coordinate system is tangential to the curve. **offset** vector determines movement of the initial point of the base curve to the initial point of equidistant curve. **offset** vector is orthogonal to the tangent vector of the base curve in the initial point. In a moving local coordinate system, movement of any point of the base curve to the corresponding point of the equidistant curve is equal to the offset vector and orthogonal to the tangent vector of the base curve in the current point.

Radius vector of a point at equidistant curve is calculated as follows. The following elements are calculated for the specified parameter of base curve: the point on the guiding curve and local coordinate system with origin in this point and the first coordinate axis tangential to the curve in this point. Then, a matrix is calculated for rotating the local coordinate system when it is moved from the initial point of the base curve to the specified point. The rotation matrix is used to transform a copy of the offset vector; the calculated point of the base curve is moved using the calculated vector.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{basisCurve}(t) + \mathbf{offset} \cdot \mathbf{A}(t) \text{ vector function,}$$

where **A**(*t*) is a rotation matrix for rotating the local coordinate system when it is moved from the initial point of the base curve to the specified point. Curve equidistant to a conical spiral is shown in Figure O.4.12.1.

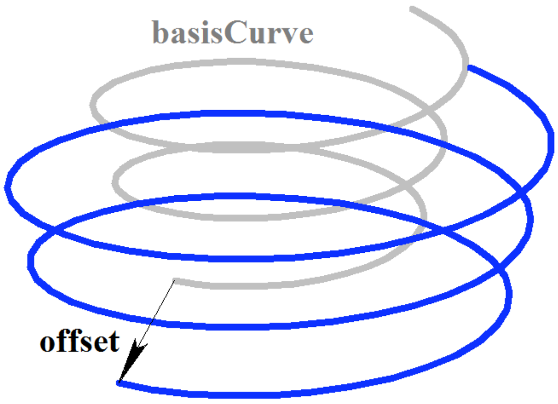


Fig. O.4.12.1.

Equidistant curve parameter range coincides with that of the base curve.

An equidistant curve can't use other equidistant curve as a base curve; guiding curve of other equidistant curve should be used subject to corresponding recalculation of the offset vector.

O.4.13. MbCharacterCurve3D Character Curve

MbCharacterCurve3D class is declared in cur_character_curve3d.h file.

Character curve is described by *xFunction*, *yFunction*, *zFunction* coordinate functions, MbPlacement3D **position** local coordinate system, **transform** transformation matrix, *tmin* and *tmax* limit values of curve parameter range, *closed* curve periodicity sign and *coordinateType* coordinate system type (Cartesian, cylindrical, spherical), where coordinate functions are defined. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

xFunction(*t*)**,** *yFunction*(*t*)**,** *zFunction*(*t*) coordinate functions of character curve are scalar functions of *t* common parameter, they are defined as character expressions. Lexical analysis was made for each character expression and a tree was constructed that calculates the value of character expression for a specified parameter, as well as derivatives of the character expression to the parameter. *t* curve parameter takes values in *tmin≤t≤tmax range.*

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

r(*t*) = [*xFunction*(*t*)   *yFunction*(*t*)   *zFunction*(*t*)] vector function.

A character curve is shown in Figure O.4.13.1.

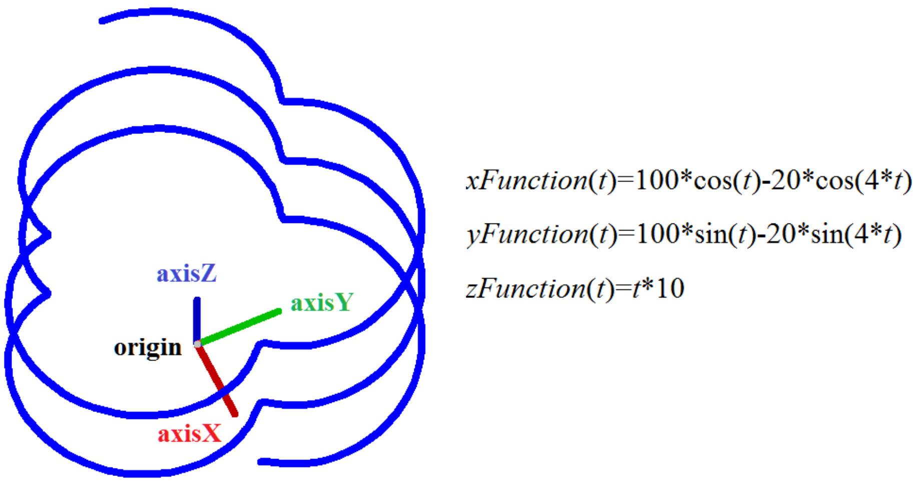

axisZ
axisY
origin
axisX
xFunction(t)=100*cos(t)-20*cos(4*t)
yFunction(t)=100*sin(t)-20*sin(4*t)
zFunction(t)=t*10

Fig. O.4.13.1.

The curve may be periodic. Character expressions should describe continuous finite single-valued functions in curve definition range.

O.4.14. MbConeSpiral Conical Spiral

MbConeSpiral class is declared in cur_cone_spiral.h file.

A conical spiral is an inheritor of MbSpiral curve. The spiral is described by MbPlacement3D **position** local coordinate system, *radius*, *tgAlpha* cone angle tangent, *step_2pi* spiral pitch divided by 2π, *tmin* and *tmax* spiral limits. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Spiral axis coincides with **position.axisZ** axis of the local coordinate system. *tgAlpha* parameter is equal to tangent of the angle between spiral axis and spiral cone generator. *tmin* and *tmax* parameters are angles, they are measured from **position.axisX** vector towards **position.axisY** vector. The angles that are 2π multiples correspond to curve point in XZ plane of the local coordinate system. *t* curve parameter takes values in *tmin≤t≤tmax* range.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

r(*t*) = **position.origin** +
(*radius* + *t tgAlpha step_2pi*) (cos(*t*) **position.axisX** + sin(*t*) **position.axisY**) +
((*t step_2pi*) **position.axisZ**) vector function.

A conical spiral is shown in Figure O.4.14.1.

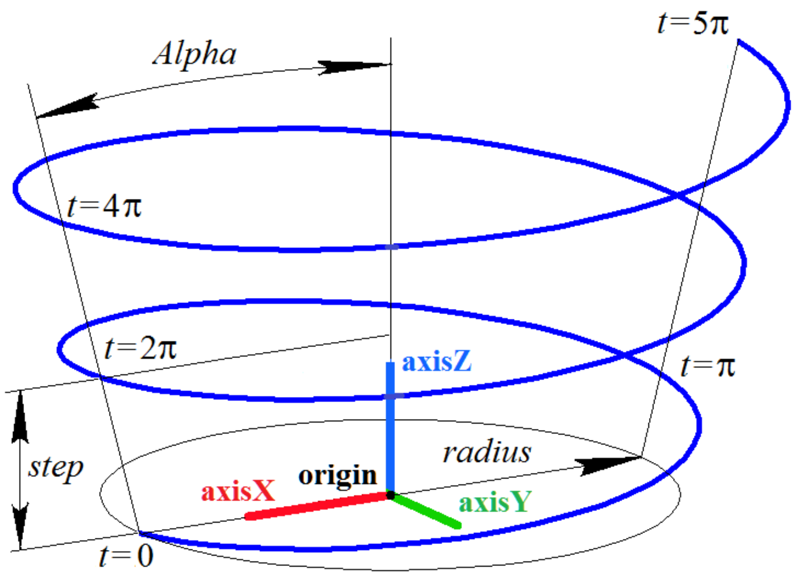


Fig. O.4.14.1.

Curve radius should be greater than zero: *radius*>0. The following inequalities should hold for limiting parameters: *tmin*<*tmax*. The curve can't be periodic.

position local coordinate system may be either right- or left-handed. *tgAlpha*=0 for a cylindrical spiral.

O.4.15. MbCurveSpiral Variable Radius Spiral

MbCurveSpiral class is declared in cur_curve_spiral.h file.

Variable radius spiral is an inheritor of MbSpiral curve. A spiral is described by MbPlacement3D **position** local coordinate system, ***curve*** 2D curve that defines radius variation law, *step spiral pitch, tmin* and *tmax* spiral limits. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Spiral axis coincides with **position.axisZ** axis of the local coordinate system. ***curve*** 2D curve lies in XZ plane of the local coordinate system, this curve defines spiral radius variation law. **position.axisZ** is abscissa axis, and **position.axisX** is ordinate axis in ***curve*** two-dimensional space. The origin of ***curve*** 2D coordinate system coincides with the origin of **position** local coordinate system. Spiral radius is equal to the ordinate of points in ***curve*** two-dimensional curve. *tmin* and *tmax* parameters are angles, they are measured from **position.axisX** vector towards **position.axisY** vector. The angles that are 2π multiples correspond to curve point in XZ plane of the local coordinate system.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\begin{gathered}\mathbf{r}(t) = \mathbf{position.origin} + \\ radius(t)\,(\cos(t)\,\mathbf{position.axisX} + \sin(t)\,\mathbf{position.axisY}) + \\ ((t \cdot step\,/\,2\pi)\,\mathbf{position.axisZ})\text{ vector function,}\end{gathered}$$

where *radius*(*t*) is local radius. *radius*(*t*) local radius is calculated as follows. We use defined *t* parameter to calculate *t·step/2π* abscissa of required 2D curve point. Then we define the point of intersection of ***curve*** and vertical straight line; this line intersects with abscissa axis in *t·step/2π* point. The ordinate of two-dimensional intersection point of ***curve*** and the vertical straight line is equal to the required spiral *radius*(*t*). Local radius is the distance between local abscissa axis and the point of intersection of the vertical straight line and ***curve*** in two-dimensional space in XZ plane in **position** local coordinate system. Variable radius spiral is shown in Figure O.4.15.1.

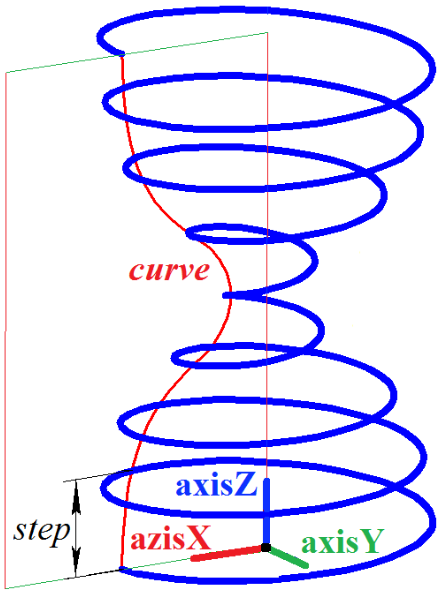


Fig. O.4.15.1.

curve should be located above the abscissa axis and it shouldn't cross abscissa axis of its two-dimensional coordinate system. ***curve*** shouldn't have vertical tangent lines. The following inequalities should hold for limiting parameters: *tmin*<*tmax*. The curve can't be periodic.

position local coordinate system may be either right- or left-handed.

O.4.16. MbCrookedSpiral Spiral with Curved Planar Axis

MbCrookedSpiral class is declared in cur_crooked_spiral.h file.

Spiral with a curvilinear planar axis is an inheritor of MbSpiral curve. A spiral is described by MbPlacement3D **position** local coordinate system, MbCurve* ***curve*** two-dimensional curve that defines spiral axis, *radius* spiral radius, *step* spiral pitch, and two spiral limits (*tmin* and *tmax*). There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

curve two-dimensional curve lies in XZ plane of **position** local coordinate system, it defines the spiral axis. **position.axisZ** is abscissa axis, and **position.axisX** is ordinate axis in ***curve*** two-dimensional space. The origin of ***curve*** 2D coordinate system coincides with the origin of **position** local coordinate system. Spiral radius is constant. *tmin* and *tmax* parameters are angles, they are measured from **position.axisX** vector towards **position.axisY** vector. The angles that are 2π multiples correspond to curve point in XZ plane of the local coordinate system.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\begin{gathered}\mathbf{r}(t) = \mathbf{position.origin} + \\ ((\boldsymbol{point}.y + radius\cos(t)\,\boldsymbol{normal}.y)\,\mathbf{position.axisX}) + \\ (radius\sin(t)\,\mathbf{position.axisY}) + \\ ((\boldsymbol{point}.x + radius\cos(t)\,\boldsymbol{normal}.x)\,\mathbf{position.axisZ})\text{ vector function,}\end{gathered}$$

where ***point*** is a point in 2D curve that is calculated using ***curve***–>**PointOn**(*t*,***point***) method and ***normal*** is a normal to 2D curve that is calculated using ***curve***–>**Normal**(*t*,***normal***) method. Variable radius spiral is shown in Figure O.4.16.1.

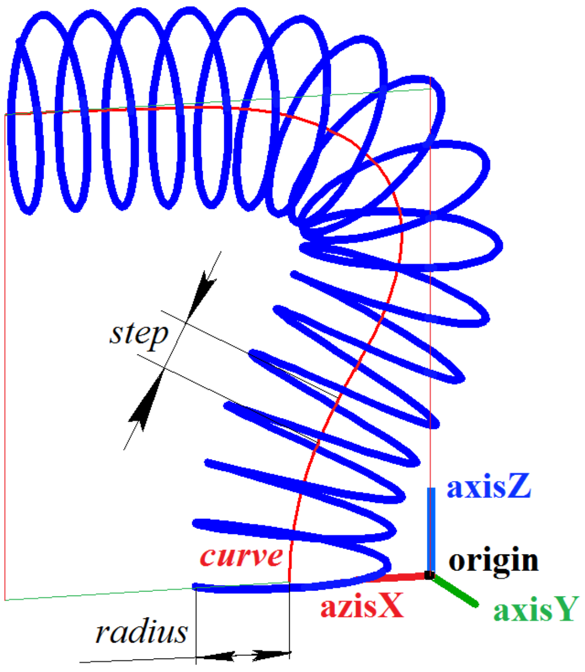


Fig. O.4.16.1.

The minimum curvature radius ***curve*** shouldn't be less than spiral radius. The following inequalities should hold for limiting parameters: *tmin<tmax*. The curve can't be periodic.

position local coordinate system may be either right- or left-handed.

O.4.17. MbBridgeCurve3D Joining Curve

MbBridgeCurve3D class is declared in cur_bridge3d.h file.

Joining curve is an inheritor of MbCurve3D curve. The curve is described by two curves (MbCurve3D* **curve1** and MbCurve3D* **curve2**), *param*1 and *param*2 point parameters of these curves, *sense*1 and *sense*2 direction coincidence signs for derivatives of the joining curve and the curves to be joined, and two joining curve limits (*tmin* and *tmax*). There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

A joining curve is used to smoothly join two specified points of **curve1** and **curve2**. **curve1** and **curve2** curve points are defined by *param*1 and *param*2 parameters. *sense*1 and *sense*2 parameters define joining curve direction in these points. A joining curve is a cubic Hermite spline constructed based on two extreme points and curve derivatives in these points. *t* curve parameter takes values in *tmin*≤*t*≤*tmax* range.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$r(t) = (1 - 3w^2 + 2w^3)\text{point1} + (3w^2 + 2w^3)\text{point2} +$$
$$+ ((w - 2w^2 + w^3)\text{derive1} + (-w^2 + w^3)\text{derive2})(\text{tmax} - \text{tmin})$$

vector function, where $w = \dfrac{t - \text{tmin}}{\text{tmax} - \text{tmin}}$ is a relative parameter value, **point1** is a point of **curve1** that is calculated using **curve1**–>**PointOn**(*param*1,**point1**) method, **point2** is a point of **curve2** that is calculated using **curve2**–>**PointOn**(*param*2,**point2**) method, **derive1** and **derive2** are joining curve derivatives in extreme points. **derive1** and **derive2** vectors are parallel to the derivatives of the curves being joined. The length of **derive1** and **derive2** vectors is equal to distance between two extreme points divided by *tmax*–*tmin*. A joining curve is shown in Figure O.4.17.1.

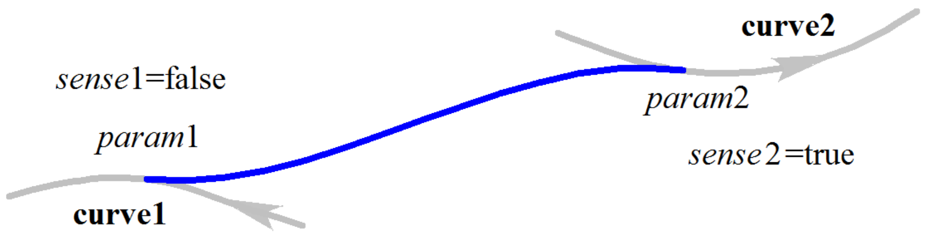


Fig. O.4.17.1.

Curve shape depends on location of extreme points and directions of curves being joined in these points. The following inequalities should hold for limiting parameters: *tmin*<*tmax*. The curve can't be periodic.

O.4.18. MbContour3D Contour

MbContour3D class is declared in cur_contour3d.h file.

MbContour3D contour is described by RPArray<MbCurve3D>**segments** set of sequentially joined curves and *closed* curve periodicity sign.

A contour is a composite curve. Unlike other curves, a contour may have bends. We'll call segments the curves that form a contour. The following conditions keep for contour segments: the initial point of each successive segment coincides with the end point of the previous one. For a periodic contour, the initial point of the first segment coincides with the end point of the last one. In general, contour derivatives have discontinuities by length and direction in the points where the segments join.

The initial value of contour parameter is zero: $t_{\min}$=0. Contour parametric length is equal to the sum of the lengths of parametric components of its segments: $t_{\max} = \sum(w_{i\max} - w_{i\min})$, where $w_{i\min}$ and $w_{i\max}$ are minimum and maximum values of the *i*th segment parameter. When radius vector of contour point is calculated, we first use parameter value to determine the working segments and the value of its local parameter, and then we calculate radius vector of the working segment, which is used as contour radius vector.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{segments}[k](w_k), \text{ vector function,}$$

where **segments**[*k*](w_k) is contour working segment with index value *k*, w_k is working segment parameter that is equal to: $w_k = w_{k\min} + t - \sum_{i=0}^{k-1}(w_{i\max} - w_{i\min})$.

The *k*th segment is defined by the value of *t* contour parameter from condition $\sum_{i=0}^{k-1}(w_{i\max} - w_{i\min}) \le t < \sum_{i=0}^{k}(w_{i\max} - w_{i\min})$, where $w_{i\min}$ and $w_{i\max}$ are minimum and maximum values of the *i*th segment parameter. A contour is shown in Figure O.4.18.1.

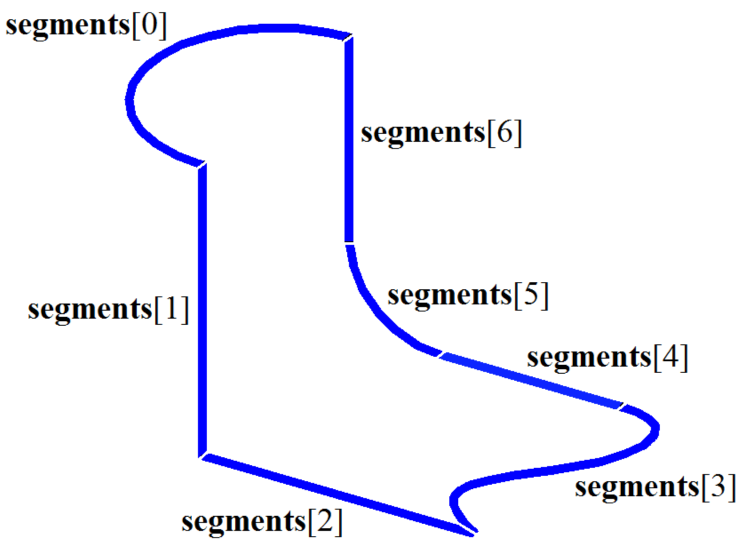


Fig. O.4.18.1.

Other contours shouldn't be used as contour segments. If other contours should be used to construct a contour, then such initial contours should be considered as a set of curves rather than as a single curve.

MbContour3D contour is the most common curve type.

O.4.19. MbPlaneCurve Plane Curve

MbPlaneCurve class is declared in cur_plane_curve.h file.

MbPlaneCurve plane curve is described by MbPlacement3D **position** local coordinate system and MbCurve* ***curve*** two-dimensional curve in XY plane of the local coordinate system.

A polar curve is a projection of the curve from 2D space of XY plane of local coordinate system into 3D space.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

r(*t*) = **position.origin** + (***point****.x* **position.axisX**) + (***point****.y* **position.axisY**) vector function,

where ***point*** is a point of 2D curve that is calculated using ***curve***–>**PointOn**(*t*,***point***) method. A plane curve that is part of a two-dimensional ellipse and a local coordinate system is shown in Figure O.4.19.1.

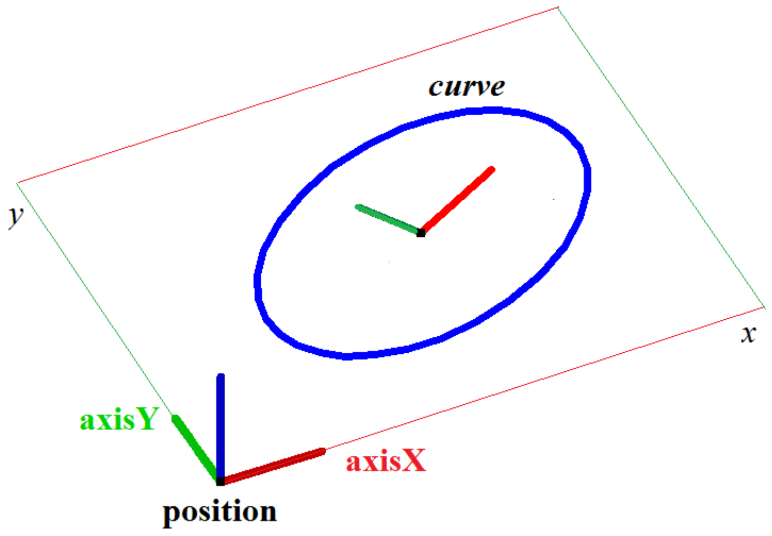


Fig. O.4.19.1.

Parameter range and plane curve periodicity coincide with those of two-dimensional ***curve***.

O.4.20. MbSurfaceCurve Curve on Surface

MbSurfaceCurve class is declared in cur_surface_curve.h file.

MbSurfaceCurve curve on surface is described by MbSurface* **surface** surface, MbCurve* ***curve*** two-dimensional curve in surface parameter space, and *closed* curve periodicity sign. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Curve on surface is a projection of 2D curve in surface parameter space into 3D space. ***curve*** two-dimensional curve can be located outside of surface parameter definition area. Parameter definition area of the curve on the surface coincides with that of ***curve*** *two-dimensional curve.*

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

r(*t*) = **surface**(***point***.*x*, ***point***.*y*) vector function,

where ***point*** is a point of 2D curve that is calculated using ***curve***–>**PointOn**(*t*,***point***) method. *x* and *y* coordinates of 2D ***point*** are *u* and *v* parameters of **surface**(*u*,*v*) surface. A curve on surface is constructed by introducing dependences of *u* and *v* parameters from some common parameter (*t*): *u*=*u*(*t*), *v*=*v*(*t*). This interdependence is described by ***curve*** two-dimensional curve. A curve on surface is shown in Figure O.4.20.1.

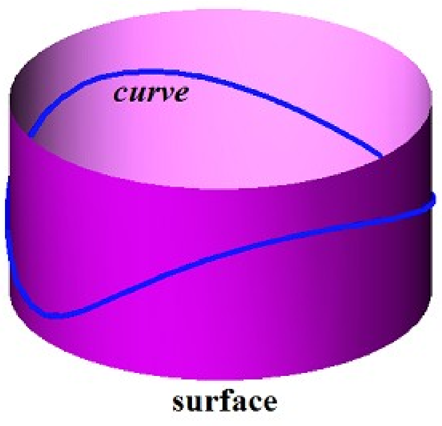


Fig. O.4.20.1.

Two-dimensional curve in surface parameter definition area is shown in Figure O.4.20.2.

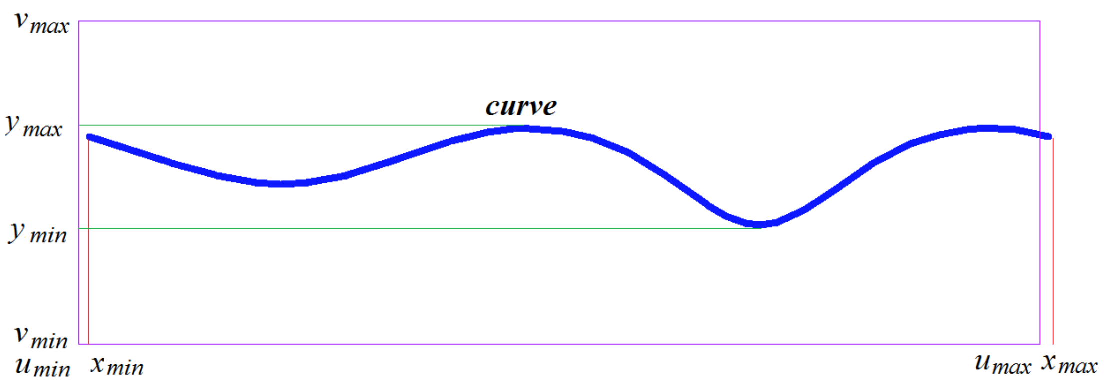


Fig. O.4.20.2.

Derivative of a curve on surface is calculated as a complex function

$$\frac{dr(t)}{\mathrm{dt}} = \frac{\partial \mathrm{surface}(u,v)}{\partial u}\mathrm{derive}.x + \frac{\partial \mathrm{surface}(u,v)}{\partial v}\mathrm{derive}.y,$$

where ***derive*** is a derivative of two-dimensional curve that is calculated using ***curve***–>**FirstDer**(*t*,***derive***) method. Derivative of a curve on surface is located in a tangent plane of a surface constructed in the specified point.

A curve on surface may be periodic if ***curve*** two-dimensional curve is periodic or **surface** is periodic, and ***curve*** has coincident derivatives in the ends, and the extreme points of the curve are set off by a corresponding period of periodic curve by **surface** first or second parameter.

surface of a curve on surface can be any surface, except for a surface limited by MbCurveBoundedSurface curves. If required, MbCurveBoundedSurface base surface will be used to construct a curve on surface limited by curves.

O.4.21. MbSilhouetteCurve Silhouette Curve

MbSilhouetteCurve class is declared in cur_silhouette_curve.h file.

MbSilhouetteCurve silhouette curve is an inheritor of MbSurfaceCurve curve on surface. A silhouette curve is described by MbSurface* **surface** surface, MbCurve* ***curve*** two-dimensional curve in surface parameter space, *closed* curve periodicity sign, *perspective* perspective sign, **eye** gaze vector, and *species* curve type. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

Silhouette curve is a curve on **surface**; the curve divides the surface into parts that are visible or invisible from the observation point. If *perspective=true*, then the observation point is described by **eye** *gaze vector.* If *perspective=false*, then the observation point is at an infinite distance, and **eye** gaze vector describes the direction from the observation point to the surface. Normal to **surface** at silhouette curve is orthogonal to a straight line that connects this point of surface and the observation point.

In particular case, when an exact silhouette curve of the surface can be constructed, *species* curve type is equal to *cbt_Ordinary*. For example, silhouette curve of a sphere is a circle. In particular case, **exactCurve** exact 3D curve is constructed; this curve accurately describes the silhouette of the surface and it is used to calculate radius vector of silhouette curve and its derivatives.

In general, *species* curve type has *cbt_Specific* value, and ***curve*** two-dimensional curve is a spline that approximates surface silhouette. In general, a point in silhouette curve is calculated by iterative method that uses ***curve*** two-dimensional curve as an initial approximation.

Parameter definition area of silhouette curve surface coincides with that of ***curve*** two-dimensional curve.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{surface}(\ u,\ v\ )\ \text{vector function},$$

where *u* and *v* are coordinates of two-dimensional point, its initial approximation is calculated using ***curve***–>**PointOn**(*t*,***point***), *u*=***point***.*x*, *v*=***point***.*y* method. Then *u* and *v* parameters are improved by iterative method that uses the following equation

$$\mathbf{vector}\ \ \mathbf{n}(u,v) = 0,$$

where **n**(*u*,*v*) is a normal to the surface that is calculated using **surface**–>**Normal**(*u*,*v*,**n**) method, **vector** is gaze vector (**vector**=**eye** for an observation point that is at an infinite distance). A silhouette curve of torus surface is shown in Figure O.4.21.1.

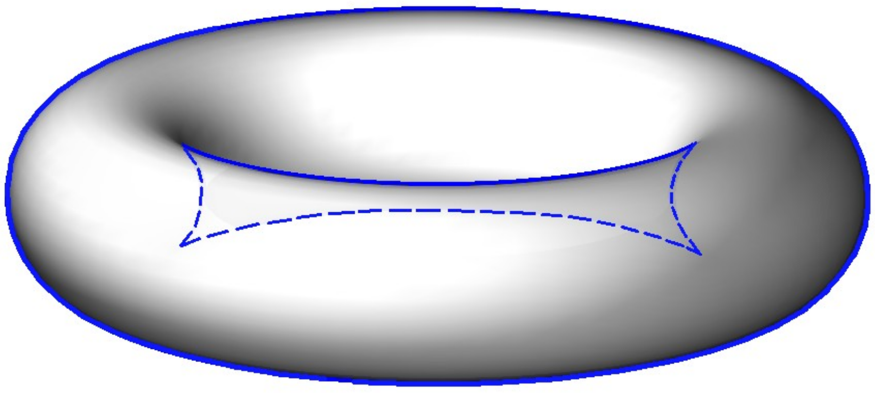


Fig. O.4.21.1.

The silhouette curve of torus surface from other observation point is shown in Figure O.4.21.2.

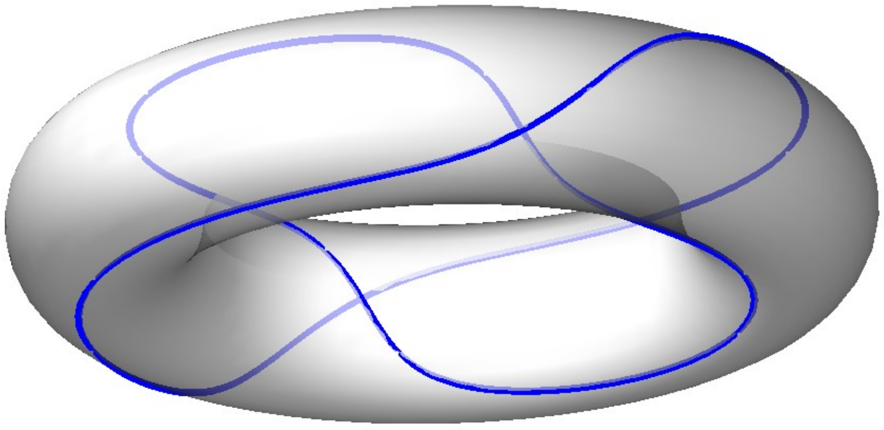


Fig. O.4.21.2.

When a silhouette curve is crossed, a scalar product of surface normal vector and gaze vector always changes its sign. Silhouette curve is always closed or it starts and ends at surface edges. A silhouette curve is used to construct projections of the curved surface silhouette on a plane.

O.4.22. MbContourOnSurface Contour on Surface

MbContourOnSurface class is declared in cur_contour_on_surface.h file.

MbContourOnSurface contour on surface is described by MbSurface* **surface** surface and MbContour* ***contour*** two-dimensional contour in surface parameter space. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

A contour on surface is a composite curve, so it can have bends in joining points of segments of 2D contour. A contour on surface is a projection of surface parameter contour 2D space into 3D space. ***contour*** 2D contour can be located outside of surface parameter definition area. Contour parameter definition area on a surface coincides with that of 2D ***contour***.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

r(*t*) = **surface**(***point***.*x*, ***point***.*y*) vector function, where ***point*** is a point of 2D contour that is calculated using ***contour***–>**PointOn**(*t*,***point***) method. *x* and *y* coordinates of 2D ***point*** are *u* and *v* parameters of **surface**(*u*,*v*) surface. A contour on surface is constructed by introducing interdependence of *u* and *v* parameters and some their common parameter (*t*): *u*=*u*(*t*), *v*=*v*(*t*). This interdependence describes ***contour*** 2D contour. A derivative of contour on surface is calculated as a complex function

$$\frac{dr(t)}{\mathrm{dt}} = \frac{\partial \mathrm{surface}(u,v)}{\partial u}\mathrm{derive}.x + \frac{\partial \mathrm{surface}(u,v)}{\partial v}\mathrm{derive}.y,$$

where ***derive*** is a derivative of 2D contour that is calculated using ***contour***–>**FirstDer**(*t*,***derive***) method. A derivative of the contour on surface lies in tangent plane of a surface constructed in the specified point. A contour on surface is shown in Figure O.4.22.1.

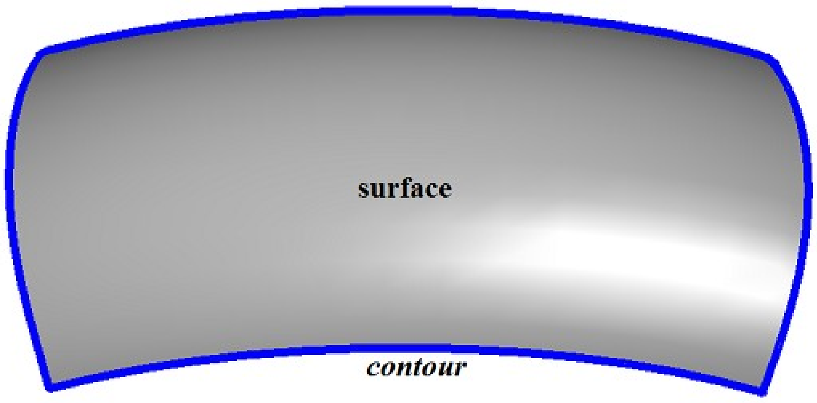


Fig. O.4.22.1.

A contour on surface may be periodic if ***contour*** 2D contour is periodic or **surface** surface is periodic, and ***contour*** end points are set off for a corresponding period of the periodic curve using **surface** first or second parameter.

A periodic contour on surface is usually used to describe a boundary of this surface.

Contour **surface** may be any surface, besides the surface limited by MbCurveBoundedSurface curves. MbCurveBoundedSurface base surface may be used to construct a contour on a surface limited by curves.

O.4.23. MbContourOnPlane Contour on Plane

MbContourOnPlane class is declared in cur_contour_on_plane.h file.

MbContourOnPlane contour on plane is an inheritor of MbContourOnSurface class. A contour on plane is described by MbSurface* **surface** surface and MbContour* ***contour*** 2D contour in surface parameter space. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

A contour on plane is a composite curve, so it can have bends in joining points of 2D contour segments. A contour on plane is a projection of plane parameter 2D contour into 3D space. Two-dimensional ***contour*** can be located outside of plane parameter definition area. Parameter definition area of a contour on plane coincides with that of 2D ***contour***.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

r(*t*) = **position.origin** + (***point***.*x* **position.axisX**) + (***point***.*y* **position.axisY**) vector function,

where **position** is a local coordinate system of **surface** plane, ***point*** is a point in 2D contour that is calculated using ***contour***–>**PointOn**(*t*,***point***) method. A contour on plane is shown in Figure O.4.23.1.

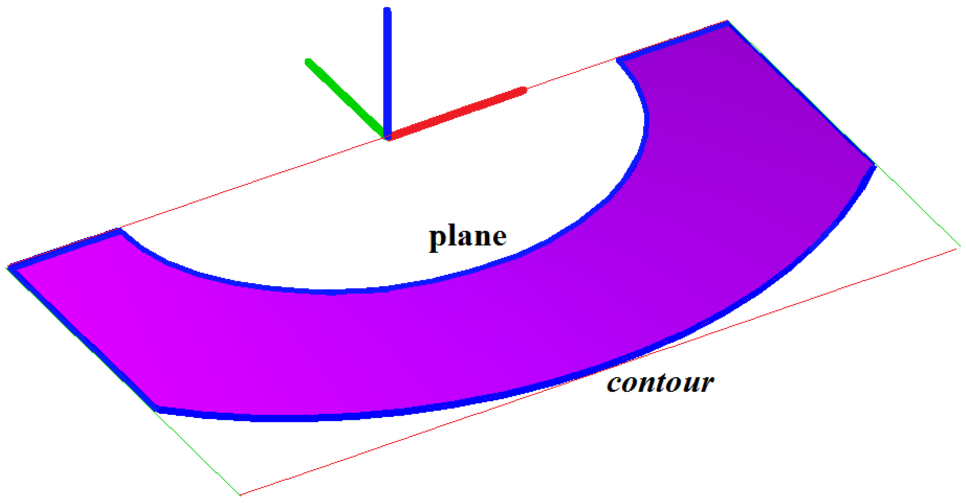


Fig. O.4.23.1.

A contour on plane may be periodic if 2D ***contour*** is periodic or **surface** is periodic, and ***contour*** end points are set off for a corresponding period of the periodic curve using the first or the second **surface** parameter.

A periodic contour on plane is usually used to describe the boundaries of this surface.

A contour on plane is similar to a contour on surface, but it provides higher computation speed.

O.4.24. MbSurfaceIntersectionCurve Surface Intersection Curve

MbSurfaceIntersectionCurve class is declared in cur_surface_intersection.h file.

MbSurfaceIntersectionCurve surface intersection curve is described by two curves (MbSurfaceCurve **curveOne** and MbSurfaceCurve **curveTwo**) that lie on intersecting surfaces, *buildType* construction parameter and *tolerance* accuracy. There are some other parameters of the curve that are not mandatory, they are used to speed up curve methods.

curveOne(*t*) and **curveTwo**(*t*) curves have the same *t* parameter ranges and they coincide in space within some accuracy. *buildType* parameter of intersection curve describes curve type and stores data on curve radius vector calculation method. *buildType* parameter takes the following values: cbt_Specific, cbt_Ordinary, cbt_Boundary, cbt_Tolerant. In Fig. O.4.24.1, you can see two surfaces and their intersection curve.

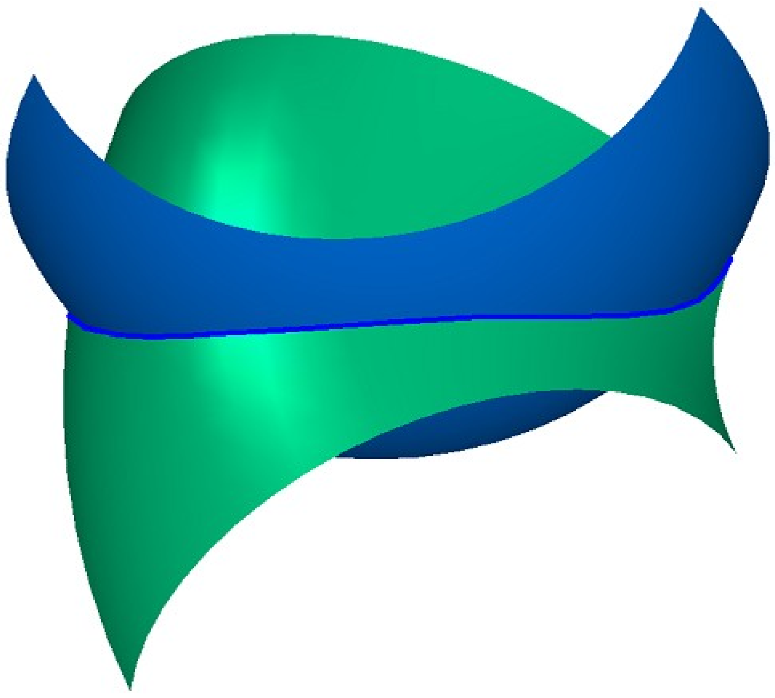


Fig. O.4.24.1.

In Fig. O.4.24.2 and O.4.24.3, you can see curves on surfaces that are used to construct an intersection curve.

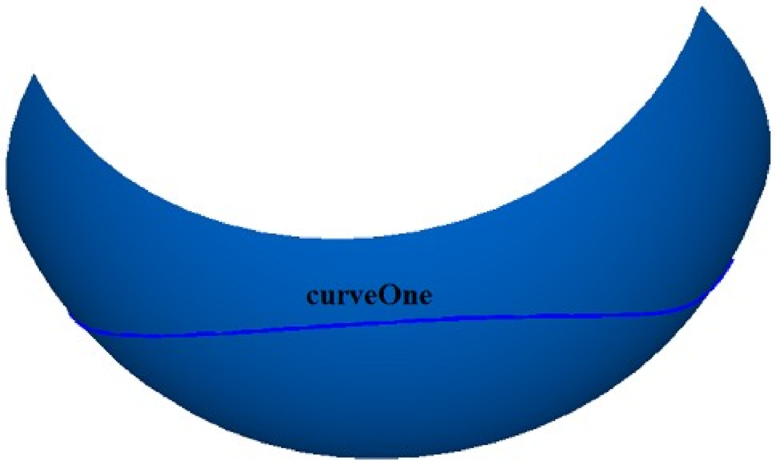


Fig. O.4.24.2.

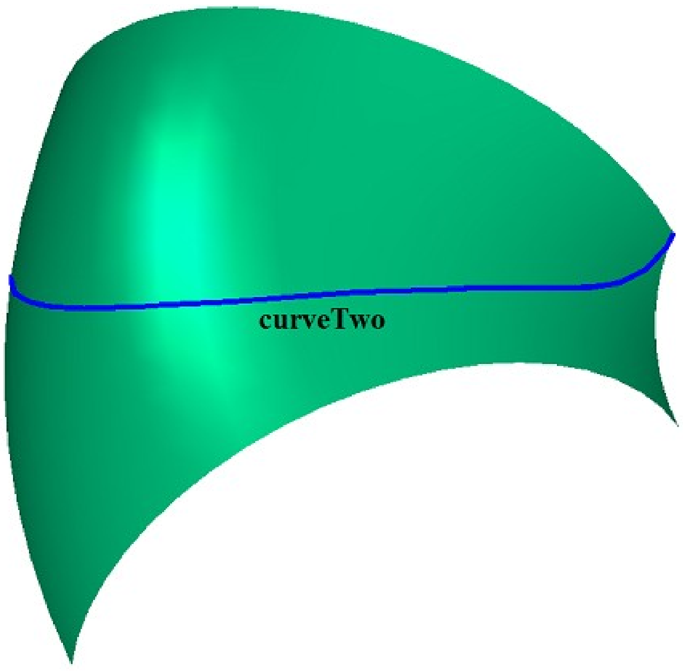


Fig. O.4.24.3.

In general, intersection curve has cbt_Specific type, **curveOne**.***curve*** and **curveTwo.*curve*** 2D curves are splines that approximate the intersection of **curveOne.surface** and **curveTwo.surface** surfaces. The splines on surfaces have aligned control points. **curveOne** and **curveTwo** splines on surfaces coincide and have the same parameters in control points. MbSurfaceIntersectionCurve curve also returns the exact value of point radius vector at the sections between control points of the splines. In general, a point on the intersection curve is calculated by iterative method that uses **curveOne**.***curve*** and **curveTwo.*curve*** two-dimensional curves as an initial approximation.

In special cases, intersection curve has cbt_Ordinary, cbt_Boundary, cbt_Tolerant types, and a point of intersection curve is calculated as the arithmetic average of radii vectors of **curveOne**(*t*) and **curveTwo**(*t*) curves.

If *buildType*=cbt_Ordinary, then MbSurfaceIntersectionCurve curve exactly describes the intersection of the curves, and **curveOne**(*t*) and **curveTwo**(*t*) curve coincide in space. An example of such curve is an intersection curve of a plane and a cylindrical surface having its axis orthogonal to the plane, see Figure O.4.24.4.

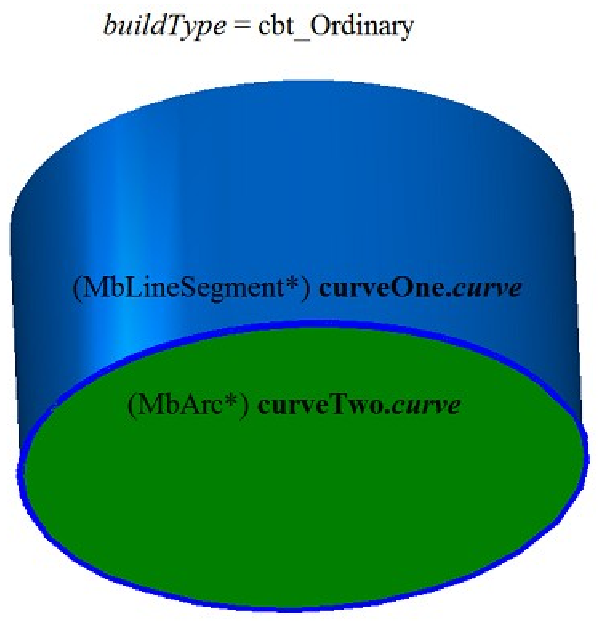

buildType = cbt_Ordinary
(MbLineSegment*) **curveOne.*curve***
(MbArc*) **curveTwo.*curve***

Fig. O.4.24.4.

On a plane, **curveOne.*curve*** is a circle; and on a cylindrical surface, **curveTwo.*curve*** is a segment having parametric length equal to the length of 2D parametric curve on the plane. In order to ensure equality of parametric lengths, a MbReparam reparametrized curve is constructed based on the segment.

If *buildType*=cbt_Boundary, then MbSurfaceIntersectionCurve curve describes surface edge, see Figure O.4.24.5. The following equations hold for such curve: **curveOne.*curve***=**curveTwo.*curve*** and **curveOne.surface**=**curveTwo.surface**.

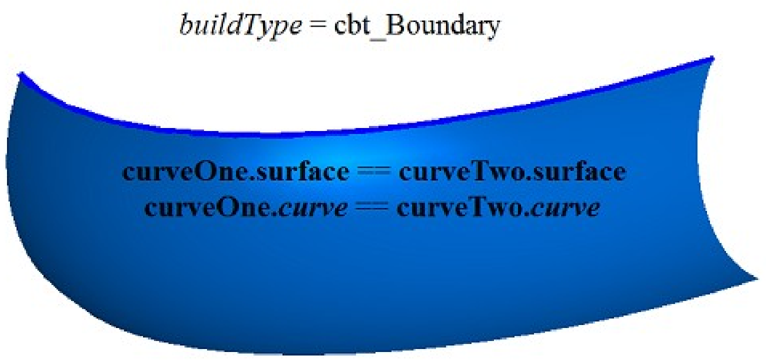

buildType = cbt_Boundary
curveOne.surface == curveTwo.surface
curveOne.*curve* == curveTwo.*curve*

Fig. O.4.24.5.

If *buildType*=cbt_Tolerant, then MbSurfaceIntersectionCurve curve describes the intersection of surfaces approximately. **curveOne**(t) and **curveTwo**(t) coincide in space with *tolerance* accuracy. Such curves are constructed in the cases when any other construction is impossible. For example, if it is required to cross two surfaces that touch each other not exactly, but rather with some "noise".

In **PointOn**(double & t, MbMatrix3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = 0.5\ (\ \mathbf{curveOne.surface}(u_1,v_1) + \mathbf{curveTwo.surface}(u_2,v_2)\ )\ \text{vector function,}$$

where u_1, v_1 are coordinates of 2D point, its initial approximation is calculated using **curveOne.*curve*–>PointOn**(*t*,***point***1), u_1=***point***1.*x*, v_1=***point***1.*y method,* u_2*,* v_2 *are* coordinates of 2D point, its initial approximation is calculated using **curveTwo.*curve*–>PointOn**(*t*,***point***2), u_2=***point***2.*x*, v_2=***point***2.*y* method. In general, (*buildType*=cbt_Specific), u_1, v_1, u_2, v_2 parameters are clarified by iterative method that uses the following equations:

curveOne.surface(u_1,v_1) = **plane**(x,y),
curveTwo.surface(u_2,v_2) = **plane**(x,y),

where **plane** is a plane perpendicular to the segment connecting two closest control points of the intersection curve. A general case of intersection curve and control points that were used to construct **curveOne** and **curveTwo** curves are shown in Figure O.4.24.6.

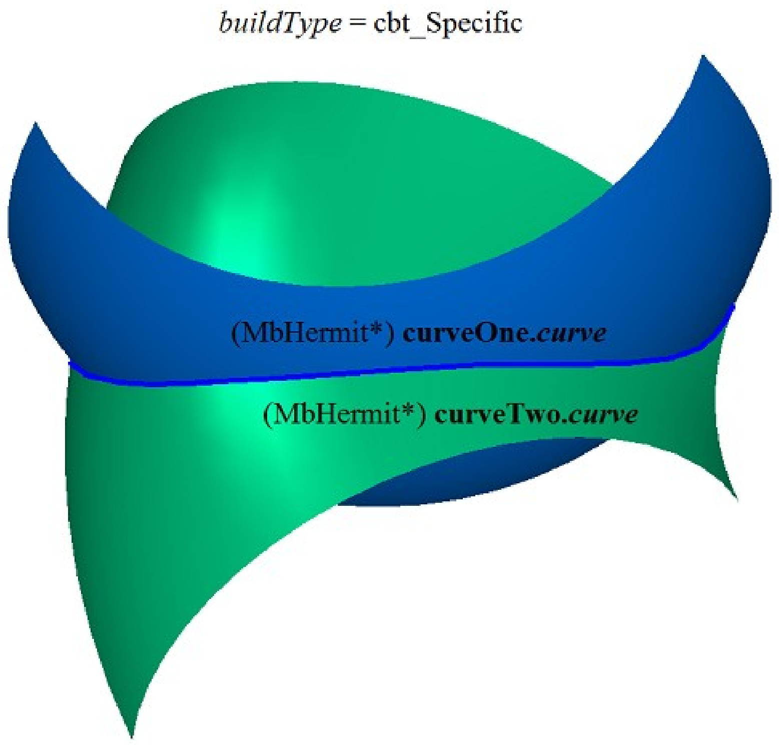

buildType = cbt_Specific
(MbHermit*) **curveOne.*curve***
(MbHermit*) **curveTwo.*curve***

Fig. O.4.24.6.

Parameter range of intersection curve coincides with that of shared parameter of **curveOne** and **curveTwo** curves. Surface intersection curve may be periodic.

Surface intersection curve contains data on **spaceCurve** 3D curve that coincides with the intersection curve within *tolerance* accuracy. **spaceCurve** curve is used to construct flat projections of edges. **spaceCurve** curve is an auxillary object; it is calculated only if it is required.

O.5. SURFACES

Surfaces belong to the family of MbSpaceItem three-dimensional geometric objects. Surfaces play a major role in construction of a geometric model. Surfaces are used to describe smooth sections of geometric form for simulated objects. Surfaces are constructed using analytical functions based on a set of points, as well as based on curves and based on surfaces. We'll use **bold** Roman font to designate vectors, radius vectors of points, and matrices in three-dimensional space.

O.5.1. MbSurface Surface

MbSurface class is declared in surface.h file.
MbSurface is an inheritor of MbSpaceItem class, see Figure O.5.1.1.

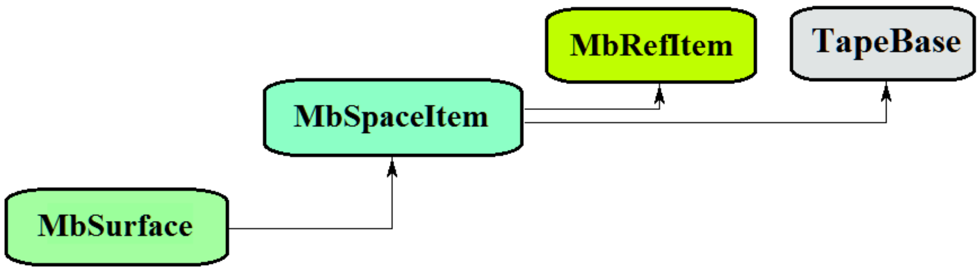


Figure O.5.1.1.

The surface is an abstract class. The following surfaces that are inheritors of MbSurface class are realized in C3D geometric kernel:
MbPlane – a plane,
MbCylinderSurface – a cylindrical surface,
MbConeSurface – a conical surface,
MbSphereSurface – a spherical surface,
MbTorusSurface – a toroidal surface,
MbExtrusionSurface – an extrusion surface,
MbRevolutionSurface – a rotation surface,
MbExpansionSurface – a plane-parallel kinematic surface,
MbSpiralSurface – a spiral surface,
MbEvolutionSurface – a kinematic surface,
MbExactionSurface – a kinematic surface with adaptation,
MbSectorSurface – a sectorial surface,
MbRuledSurface – a ruled surface,
MbLoftedSurface – a surface based on a family of curves,
MbElevationSurface – a surface based on a family of curves and a guiding curve,
MbCornerSurface – a surface based on three curves,
MbCoverSurface – a surface based on four curves,
MbCoonsPatchSurface – a bicubic Coons surface,
MbMeshSurface – a surface based on a network of curves,
MbJoinSurface – a joint surface,
MbSplineSurface – NURBS surface (NonUniform Rational B-Spline surface),
MbOffsetSurface – an equidistant surface,
MbChamferSurface – a chamfer surface,
MbFilletSurface – a fillet surface,
MbChannelSurface – a fillet surface with variable radius,
MbCurveBoundedSurface – a surface with arbitrary borders.

MbSurface is a vector function

$$\mathbf{surface}(u,v)=[x(u,v) \quad y(u,v) \quad z(u,v)]$$

of two scalar parameters (u and v) that take values in Ω connected two-dimensional area. The surface is a continuous projection of Ω connected 2D area in 3D space. Ω area will be described in 2D Cartesian coordinate system. In a particular case, Ω area is a rectangle, and surface parameters take values in $u_{\min}\leq u\leq u_{\max}$, $v_{\min}\leq v\leq v_{\max}$ ranges. In general case, Ω area is described by 2D curves. $x(u,v)$, $y(u,v)$, $z(u,v)$ coordinates of **surface**(u,v) surface are single-valued continuous functions of u and v parameters.

$u_{\min}$, $u_{\max}$, $v_{\min}$, $v_{\max}$ limits of parameter definition limits are returned by double **GetUMin**(), double **GetUMax**(), double **GetVMin**(), and double **GetVMax**() surface methods, respectively.

We'll call the surface periodic by the first parameter if there is such p_u>0 that **surface**($u\pm kp_u$,v)=**surface**(u,v), where k *is an* integer. We'll call the surface periodic by the second parameter if there is such p_v>0 that **surface**(u,$v\pm kp_v$)=**surface**(u,v), where k *is an* integer. Definition area of periodic surface parameter ranges within one period for the corresponding parameter.
bool **IsUClosed**() method returns "true" for a surface periodic by the first parameter.
bool **IsVClosed**() method returns "true" for a surface periodic by the second parameter.
double **GetUPeriod**() method returns p_u period for a surface periodic by the first parameter or for a surface that can be extended and made periodic. double **GetVPeriod**() method returns p_v period for a surface that is periodic by the second parameter or for a surface that can be extended and made periodic. Definition area of periodic surface parameter is always limited by one period.

We'll use the following designations:

$$s_u=\frac{\partial \text{surface}(u,v)}{\partial u};\ s_v=\frac{\partial \text{surface}(u,v)}{\partial v};$$

$$s_{\text{uu}}=\frac{\partial^2 \text{surface}(u,v)}{\partial u^2};\ s_{\text{vv}}=\frac{\partial^2 \text{surface}(u,v)}{\partial v^2};\ s_{\text{uv}}=\frac{\partial^2 \text{surface}(u,v)}{\partial u\partial v}=\frac{\partial^2 \text{surface}(u,v)}{\partial v\partial u};$$

$$s_{\text{uuu}}=\frac{\partial^3 \text{surface}(u,v)}{\partial u^3};\ s_{\text{uuv}}=\frac{\partial^3 \text{surface}(u,v)}{\partial u\partial u\partial v};\ s_{\text{uvv}}=\frac{\partial^3 \text{surface}(u,v)}{\partial u\partial v\partial v};\ s_{\text{vvv}}=\frac{\partial^3 \text{surface}(u,v)}{\partial v^3}$$

for private derivatives of surface by its parameters.

The main surface method is
void **PointOn**(double & u, double & v, MbCartPoint3D & **s**).
It returns **s**(u,v) radius vector of surface point for given parameters (u and v).
void **DeriveU**(double & u, double & v, MbVector3D & $\mathbf{s}_u$),
void **DeriveV**(double & u, double & v, MbVector3D & $\mathbf{s}_v$),
void **DeriveUU**(double & u, double & v, MbVector3D & $\mathbf{s}_{uu}$),
void **DeriveUV**(double & u, double & v, MbVector3D & $\mathbf{s}_{uv}$),
void **DeriveVV**(double & u, double & v, MbVector3D & $\mathbf{s}_{vv}$),
void **DeriveUUU**(double & u, double & v, MbVector3D & $\mathbf{s}_{uuu}$),
void **DeriveUUV**(double & u, double & v, MbVector3D & $\mathbf{s}_{uuv}$),
void **DeriveUVV**(double & u, double & v, MbVector3D & $\mathbf{s}_{uvv}$),
void **DeriveVVV** (double & u, double & v, MbVector3D & $\mathbf{s}_{vvv}$)
methods respectively return $\mathbf{s}_u$, $\mathbf{s}_v$, $\mathbf{s}_{uu}$, $\mathbf{s}_{uv}$, $\mathbf{s}_{vv}$, $\mathbf{s}_{uuu}$, $\mathbf{s}_{uuv}$, $\mathbf{s}_{uvv}$, $\mathbf{s}_{vvv}$ derivatives of surface radius vector for given parameters (u and v). These methods adjust surface parameters if they go beyond the definition area (the exception is MbPlane plane). If u parameter goes beyond [$u_{\min}$, $u_{\max}$] range then: a) the surfaces that are non-periodic by the first parameter, move u parameter to the nearest limit $u_{\min}$ or $u_{\max}$; b) the surfaces that are periodic by the first parameter add or subtract the required number of periods. If v parameter goes beyond [$v_{\min}$, $v_{\max}$] range then: a) the surfaces that are non-periodic by the second parameter move v parameter to the nearest limit $v_{\min}$ or $v_{\max}$; b) the surfaces that are periodic by the first parameter add or subtract the required number of periods.

void **_PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method
returns **s**(*u*,*v*) radius vector of the surface point for specified parameters *u* and *v* both within surface parameter definition area and outside it. Each non-periodic surface is extended outside the parameter definition area using its own law. If there is no such law (in general case), then non-periodic surface is extended outside the parameter definition area, it is extended tangentially to the corresponding extreme point of the surface.
void **_DeriveU**(double *u*, double *v*, MbVector3D & $\mathbf{s}_u$),
void **_DeriveV**(double *u*, double *v*, MbVector3D & $\mathbf{s}_v$),
void **_DeriveUU**(double *u*, double *v*, MbVector3D & $\mathbf{s}_{uu}$),
void **_DeriveUV**(double *u*, double *v*, MbVector3D & $\mathbf{s}_{uv}$),
void **_DeriveVV**(double *u*, double *v*, MbVector3D & $\mathbf{s}_{vv}$),
void **_DeriveUUU**(double *u*, double *v*, MbVector3D & $\mathbf{s}_{uuu}$),
void **_DeriveUUV**(double *u*, double *v*, MbVector3D & $\mathbf{s}_{uuv}$),
void **_DeriveUVV**(double *u*, double *v*, MbVector3D& $\mathbf{s}_{uvv}$),
void **_DeriveVVV**(double *u*, double *v*, MbVector3D & $\mathbf{s}_{vvv}$)
methods respectively return $\mathbf{s}_u$, $\mathbf{s}_v$, $\mathbf{s}_{uu}$, $\mathbf{s}_{uv}$, $\mathbf{s}_{vv}$, $\mathbf{s}_{uuu}$, $\mathbf{s}_{uuv}$, $\mathbf{s}_{uvv}$, $\mathbf{s}_{vvv}$ derivatives of surface radius vector by *u* and *v* parameters both within surface parameter definition area and outside it.
Surfaces reload the following methods of 3D geometrical object:
the methods involved in transformation of geometrical object:
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),
the methods that permit to copy, check for coinciding objects, check whether it's possible to make objects coinciding and make them coinciding:
MbSpaceItem & **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbSpaceItem & **item**),
bool **IsSimilar**(const MbSpaceItem & **item**),
bool **SetEqual**(const MbSpaceItem& **item**),
the methods that return type from enumeration of geometric objects:
MbeSpaceType **IsA**(),
MbeSpaceType **Type**(),
MbeSpaceType **Family**(),
the methods that ensure access to object internal data and their editing:
MbProperty & **CreateProperty**(MbePrompt name),
void **GetProperties**(MbProperties & *properties*),
void **SetProperties**(MbProperties & *properties*),
the method that fills up a polygonal copy of a geometrical object,
void **CalculateWire**(double sag, MbMesh & **mesh**).
In most cases a surface has rectangular parameter definition area. We'll separate MbCurveBoundedSurface from all surfaces as it is a universal surface.MbCurveBoundedSurface has curved edges and may have arbitrary cutouts inside. MbCurveBoundedSurface is constructed based on arbitrary surface with rectangular parameter definition area.

O.5.2. MbPlane Plane

MbPlane class is declared in surf_plane.h file.
MbPlane plane belongs to MbElementarySurface group of elementary surfaces. A plane is described by XY plane in MbPlacement3D **position** local coordinate system. The first parameter is measured along **position.axisX** vector, the second parameter is measured along **position.axisY**. Surface parameter definition area describes *umin*, *umax* and *vmin*, *vmax* limits, see Figure O.5.2.1.

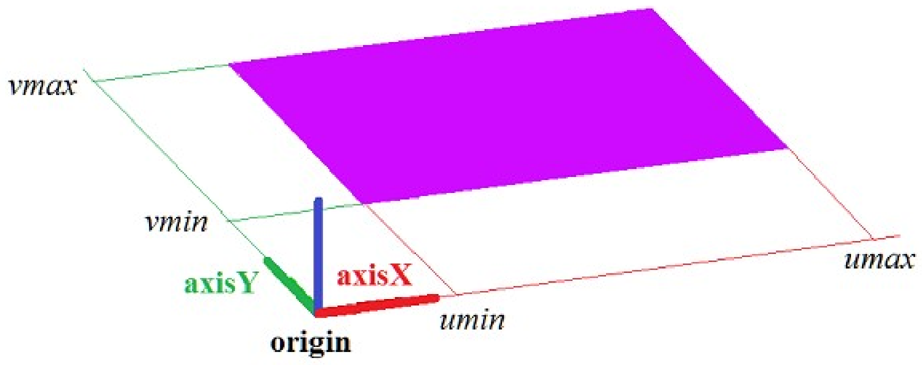


Figure O.5.2.1.

In **PointOn**(double u, double v, MbCartPoint3D & **s**) method, radius vector of **s** plane is described by

$$\mathbf{s}(u,v) = \mathbf{position.origin} + u\ \mathbf{position.axisX} + v\ \mathbf{position.axisY} \text{ vector function.}$$

A plane behaves like an infinite object, although it has extreme values of its parameters (*umin*, *umax* and *vmin*, *vmax*) in its data. Please note that unlike other surfaces, in radius vector and its derivatives calculation methods, the plane does not adjust u and v parameters if they go beyond the definition area defined by *umin*, *umax* and *vmin*, *vmax* values.

position local coordinate system may be either right- or left-handed. If local coordinate system is left-handed then direction of surface normal is opposite to direction of **position.axisZ** vector.

O.5.3. MbCylinderSurface Cylindrical Surface

MbCylinderSurface class is declared in surf_cylinder_surface.h file.

MbCylinderSurface cylindrical surface belongs to MbElementarySurface group of elementary surfaces. A cylindrical surface is described by *radius* and *height* defined in MbPlacement3D **position** local coordinate system.

The first surface parameter is measured along the arc from **position.axisX** vector towards **position.axisY** vector. The first surface parameter (u) takes values in $umin \le u \le umax$ range. u=0 and u=2π values correspond to a point in XZ plane. A surface may be periodic by the first parameter. $umax$-$umin$=2π holds for a periodic surface; $umax$-$umin$<2π holds for a non-periodic surface.

The second surface parameter is measured in a straight line that goes along **position.axisZ** vector. Surface second parameter (v) takes values in $vmin \le v \le vmax$ range. v=0 corresponds to the beginning of the local coordinate system, and v=1 corresponds to the pont located at distance *height* from XY plane of surface local coordinate system.

In **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{position.origin} + \\ radius\ (\cos(u)\ \mathbf{position.axisX} + \sin(u)\ \mathbf{position.axisY}) + \\ height\ v\ \mathbf{position.axisZ} \text{ vector function.}$$

A cylindrical surface is shown in Figure O.5.3.1.

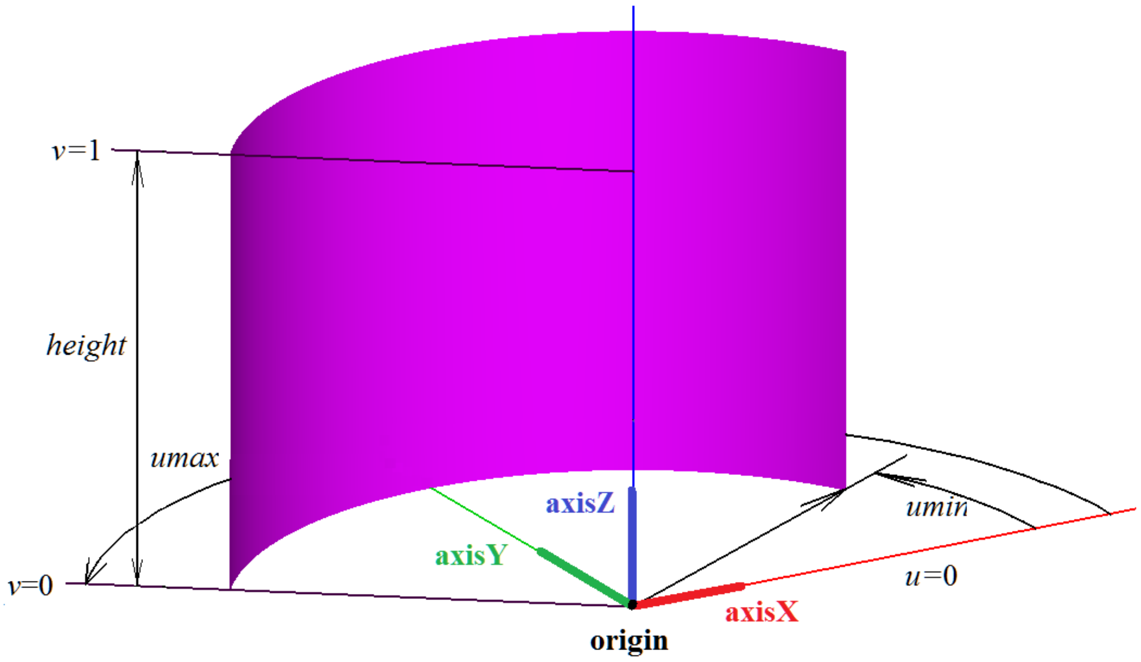


Figure O.5.3.1.

Radius and height should be popsitive: *radius*>0, *height*>0. The following inequalities should hold for surface limiting parameters: *umin*<*umax*, *vmin*<*vmax*.

position local coordinate system may be either right- or left-handed. If the local coordinate system is right-handed, then the normal is directed towards surface convexity (from the surface axis). If the local coordinate system is left-handed then the normal is directed towards surface concavity (to the surface axis).

O.5.4. MbConeSurface Conical Surface

MbConeSurface class is declared in surf_cone_surface.h file.

MbConeSurface conical surface belongs to MbElementarySurface group of elementary surfaces. A conical surface is described by *radius*, *height* and *angle* cone angle defined in MbPlacement3D **position** local coordinate system.

The first surface parameter is measured along the arc from **position.axisX** vector towards **position.axisY** vector. The first surface parameter (u) takes values in $umin \le u \le umax$ range. u=0 and u=2π values correspond to a point in XZ plane. A surface may be periodic by the first parameter. *umax*-*umin*=2π holds for a periodic surface; *umax*-*umin*<2π holds for a non-periodic surface.

The second surface parameter is measured in a straight line that goes along **position.axisZ** vector. Surface second parameter (v) takes values in $vmin \le v \le vmax$ range. v=0 corresponds to the beginning of the local coordinate system, and v=1 corresponds to the pont located at distance *height* from XY plane of surface local coordinate system.

In **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{position.origin} + \\ (radius + height\ v\ tg(angle))\ (\cos(u)\ \mathbf{position.axisX} + \sin(u)\ \mathbf{position.axisY}) + \\ height\ v\ \mathbf{position.axisZ} \text{ vector function.}$$

A conical surface is shown in Figure O.5.4.1.

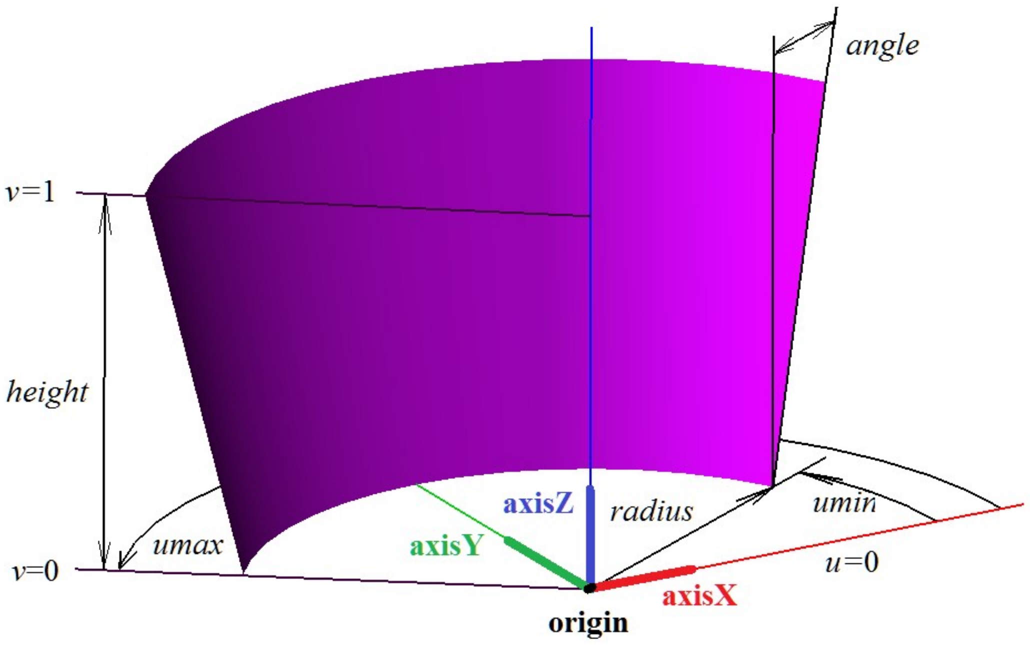


Figure O.5.4.1.

Radius and height should be positive, and angle modulo should not exceed π/2: *radius*>0, *height*>0, -π/2<*angle*<π/2. If *angle*=0, then the conical surface is equivalent to cylindrical surface. The following inequalities should hold for surface limiting parameters: *umin*<*umax*, *vmin*<*vmax*. The following value of the second parameter corresponds to surface pole: *v*=–*radius* / (*height tg*(*angle*)). *vmax* and *vmin* limits take the values at which the surface is located in one side from the pole.

position local coordinate system may be either right- or left-handed. If the local coordinate system is right-handed, then the normal is directed towards surface convexity (from the surface axis). If the local coordinate system is left-handed, then the normal is directed towards surface concavity (to the surface axis).

O.5.5. MbSphereSurface Spherical Surface

MbSphereSurface class is declared in surf_sphere_surface.h file.

MbSphereSurface sphere belongs to MbElementarySurface group of elementary surfaces. A sphere is described by *radius* determined in MbPlacement3D **position** local coordinate system.

The first surface parameter is measured along the arc from **position.axisX** vector towards **position.axisY** vector. The first surface parameter (*u*) takes values in *umin*≤*u*≤*umax* range. *u*=0 and *u*=2π values correspond to a point in XZ plane. A surface may be periodic by the first parameter. *umax*-*umin*=2π holds for a periodic surface; *umax*-*umin*<2π holds for a non-periodic surface.

The second surface parameter is measured along the arc from XY plane of local coordinate system towards **position.axisZ** vector. Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=0 corresponds to a point in XY plane of surface local coordinate system. A surface is non-periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{position.origin} + \\ radius\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + \\ radius\,\sin(v)\,\mathbf{position.axisZ} \text{ vector function.}$$

A sphere is shown in Figure O.5.5.1.

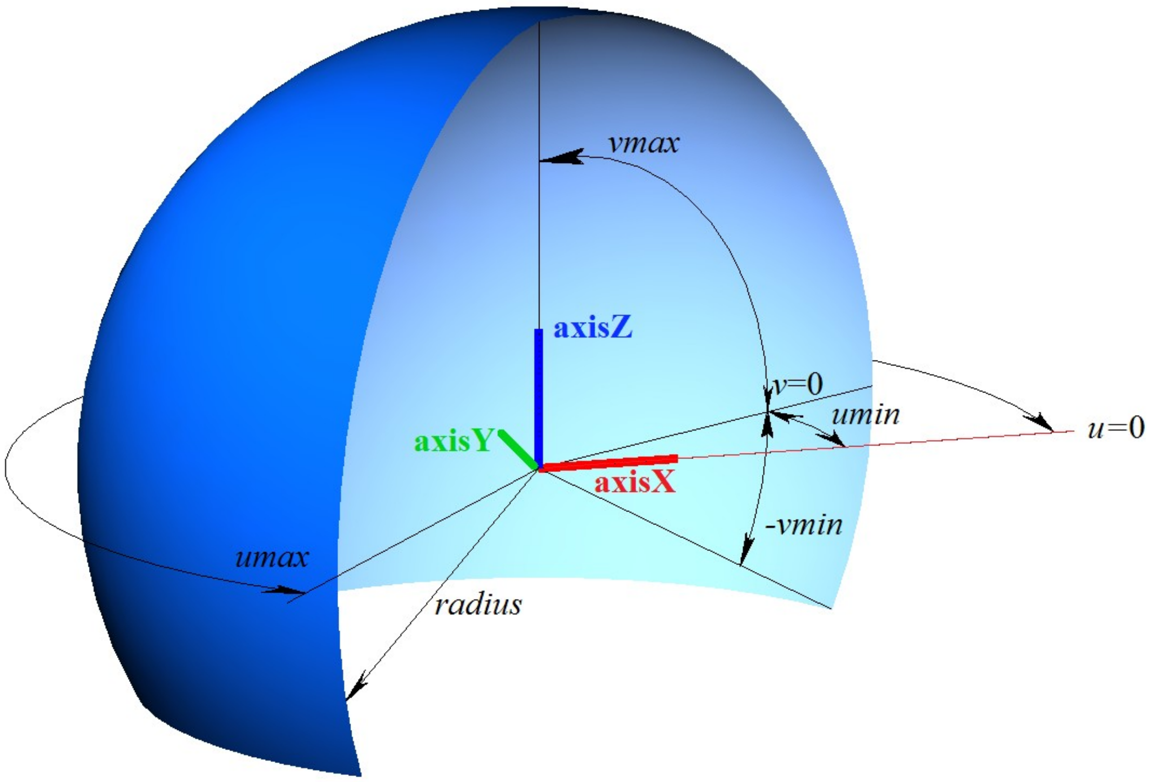


Figure O.5.5.1.

The radius of sphere should be greater than zero: *radius*>0. A sphere has poles for its parameter (*v*π/2 and *v*=–π/2). The following inequalities should hold for surface limiting parameters: *umin*<*umax*, *vmin*<*vmax*, *vmax*<=π/2, *vmin*>=–π/2.

position local coordinate system may be either right- or left-handed. If local coordinate system is right-handed, then the normal is directed from the sphere. If local coordinate system is left-handed, then the normal is directed inside the sphere.

O.5.6. MbTorusSurface Toroidal Surface

MbTorusSurface class is declared in surf_torus_surface.h file.

MbTorusSurface toroidal surface belongs to MbElementarySurface group of elementary surfaces. A toroidal surface is described by *majorRadius* radius of centers and *minorRadius* tube radius in MbPlacement3D **position** local coordinate system.

The first surface parameter is measured along the arc from **position.axisX** vector towards **position.axisY** vector. The first surface parameter (*u*) takes values in *umin*≤*u*≤*umax* range. *u*=0 and *u*=2π values correspond to a point in XZ plane. A surface may be periodic by the first parameter. *umax*-*umin*=2π holds for a periodic surface; *umax*-*umin*<2π holds for a non-periodic surface.

The second surface parameter is measured along the arc from XY plane of local coordinate system towards **position.axisZ** vector. Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=0 and *v*=2π values correspond to a point in XY plane of local coordinate system in the surface. If *majorRadius*>*minorRadius,* then a surface may be periodic by the second parameter. *vmax*-*vmin*=2π holds for a periodic surface; *vmax*-*vmin*<2π holds for a non-periodic surface.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

s(*u*,*v*) = **position.origin** +
(*majorRadius* + (*minorRadius* cos(*v*)) (cos(*u*) **position.axisX** + sin(*u*) **position.axisY**) +
minorRadius sin(*v*) **position.axisZ** vector function.

A toroidal surface is shown in Figure O.5.6.1.

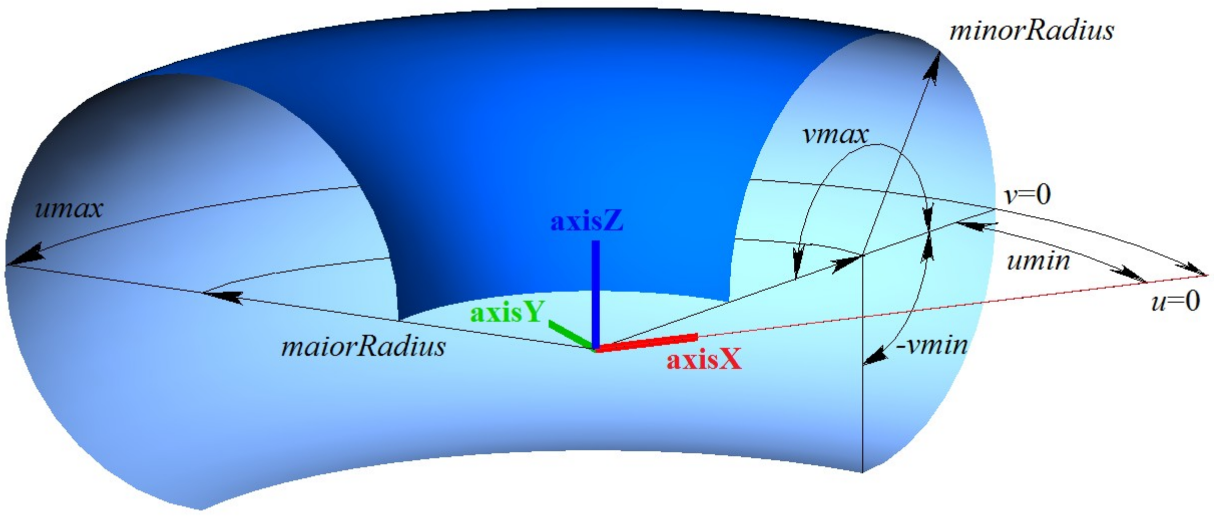


Figure O.5.6.1.

Tube radius should be positive: *minorRadius*>0. Radius of centers should not be smaller than radius of the tube taken with minus sign: *majorRadius*>–*minorRadius*. If *majorRadius*<*minorRadius* then surface has a pole for *v*=arccos(*majorRadius*/*minorRadius*)parameter and for *v*=2π–arccos(*majorRadius*/*minorRadius*) parameter. The following inequalities should hold for surface limiting parameters: *umin*<*umax*, *vmin*<*vmax*.

position local coordinate system may be either right- or left-handed. If local coordinate system is right-handed, then the normal is directed from the surface tube. If local coordinate system is left-handed, then normal is directed inside the surface tube.

O.5.7. MbExtrusionSurface Extrusion Surface

MbExtrusionSurface class is declared in surf_extrusion_surface.h file.

MbExtrusionSurface extrusion surface belongs to MbSweptSurface group of swept surfaces. Extrusion surface is a special case of sliding surfaces with rectilinear guide curve. Extrusion surface is described by MbCurve3D* **curve** curve generator, MbVector3D **direction** vector specifying extrusion direction and *distance* extrusion length.

The first surface parameter (*u*) coincides with curve generator parameter. The first surface parameter takes values in *umin*≤*u*≤*umax* range that corresponds to curve generator range. If the curve generator is periodic, then the surface is periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=0 value corresponds to a point on the curve generator; *v*=1 corresponds to a point on the curve generator displaced by **direction***distance* vector. The surface can't be periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{curve}(u) + (\mathbf{direction}\ distance\ v) \text{ vector function.}$$

Extrusion surface is shown in Figure O.5.7.1.

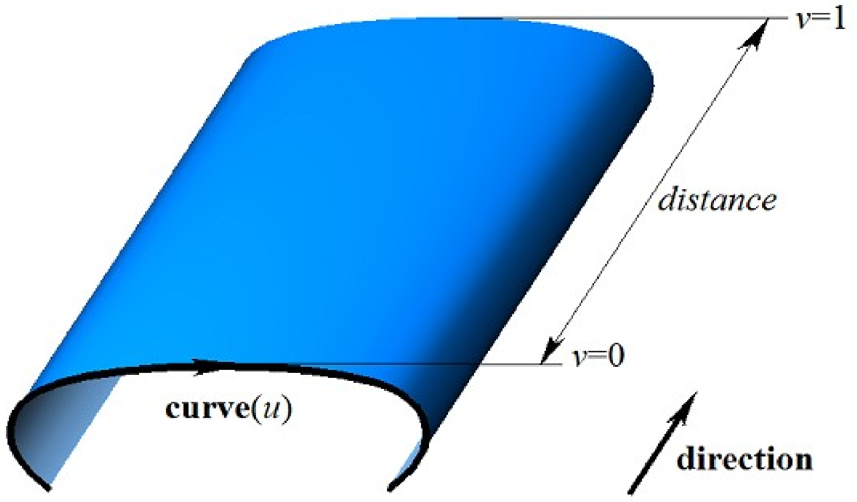


Figure O.5.7.1.

The following inequality should hold for the limits of the second parameter: *vmin*<*vmax*.

O.5.8. MbRevolutionSurface Revolution Surface

MbRevolutionSurface class is declared in surf_revolution_surface.h file.

MbRevolutionSurface revolution surface belongs to MbSweptSurface group of swept surfaces. Revolution surface is a special case of swept surface, its guiding curve is a circle or its arc. Revolution surface is described by MbCurve3D* **curve** curve generator, MbPlacement3D **position** local coordinate system, its **position.axizZ** vector is revolution axis, *planeData* sign indicating that the curve and the revolution axis are located in one plane, *poleMin* sign indicating the presence of a surface pole at the initial value of the first parameter, *poleMax* sign indicating the presence of a surface pole at the end value of the first parameter, *uPoleMin* and *uPoleMax* values of surface first parameter in surface poles (if the corresponding pole exists). There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The first surface parameter (u) coincides with curve generator parameter. The first surface parameter takes values in $umin \le u \le umax$ range that corresponds to curve generator range. If the curve generator is periodic, then the surface is periodic by the first parameter.

Surface second parameter (v) takes values in $vmin \le v \le vmax$ range. $v=0$ and $v=2\pi$ values correspond to a point in curve generator. The surface may be periodic by the second parameter. $vmax-vmin=2\pi$ holds for a periodic surface; $vmax-vmin<2\pi$ holds for a non-periodic surface.

In **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (\mathbf{curve}(u) - \mathbf{position.origin})\,\mathbf{M}(v) \text{ vector function,}$$

where $\mathbf{M}(v)$ is rotation matrix. Please note that multiplication of **(curve**(u)–**position.origin**) vector by $\mathbf{M}(v)$ matrix is a post-multiplication. Rotation matrix looks as follows

$$\mathbf{M}(v) = \mathbf{A}^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{A} =$$

$$= \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}.$$

A is a matrix used to transform coordinates of radius vector from **position** local coordinate system to the global coordinate system. Rows of **A** matrix consist of base vectors of the local coordinate system. **M**(v) matrix transforms **curve(*u*) position.origin** vector into the local coordinate system, rotates it by v angle around the rotation axis, and returns the rotated vector back into the global coordinate system. Revolution surface is shown in Figure O.5.8.1.

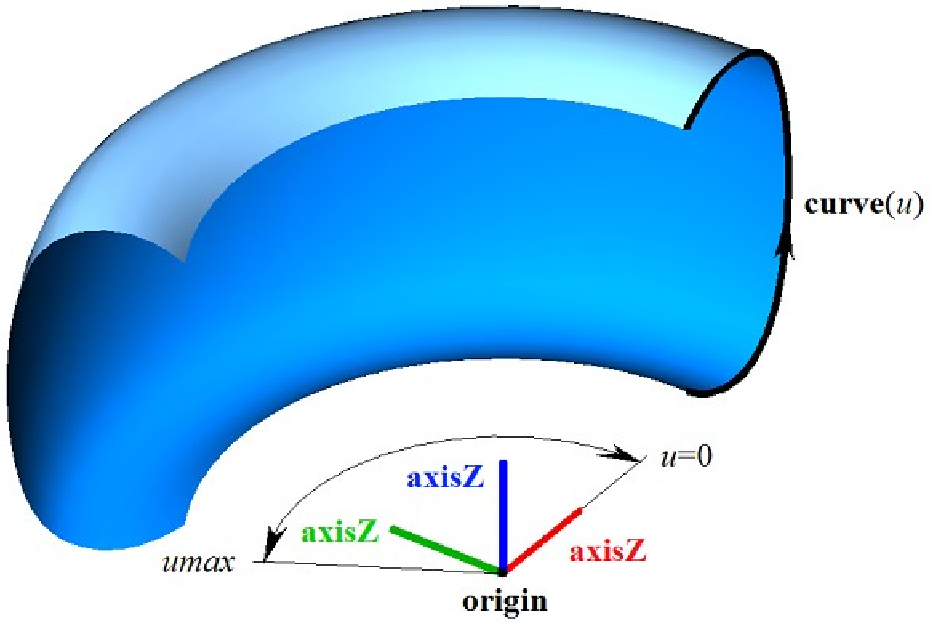


Figure O.5.8.1.

If initial or end edge of curve generator crosses the rotation axis then the surface has a pole for *umin* or *umax* parameter respectively. The following inequality should hold for the limits of the second parameter: $vmin<vmax$.

Points of the curve generator are rotated along an arc around **position.axizZ** vector from **position.axisX** vector towards **position.axisY** vector. **position** local coordinate system may be either right- or left-handed.

O.5.9. MbExpansionSurface Motion Surface

MbExpansionSurface class is declared in surf_expansion_surface.h file.

MbExpansionSurface motion surface belongs to MbSweptSurface group of swept surfaces. Motion surface is a special case of swept surface with a curvilinear guiding curve. A motion surface is described by MbCurve3D* **curve** curve generator, MbCurve3D* **spine** guiding curve, and MbCartPoint3D **origin** point that is the initial point of the guiding curve. The surface is constructed by moving the curve generator along the guiding curve. In a particular case, curve generator of motion surface may change its shape. In the latter case, the curve has the following components: **brink** second curve generator is available, **ending** is the end point of the guiding curve, *tmin* is the initial parameter of **brink** curve, and *dt* is the derivative of **brink** curve parameter by **curve** curve generator parameter. *dt* derivate is described by the following equation:

$$dt = \frac{tmax - tmin}{umax - umin},$$

where *tmax* is the ending parameter of **brink** curve. In general case, a pointer to the second curve generator (**brink**) may be zero, that means that second curve generator is missing.

The first surface parameter (u) coincides with **curve** curve generator parameter. The first surface parameter takes values in $umin \le u \le umax$ range that corresponds to curve generator range. If the curve generator is periodic, then the surface is periodic by the first parameter.

The first surface parameter (v) coincides with guiding curve parameter and it takes values in $vmin \le v \le vmax$ range. The surface may be periodic by the second parameter if the guiding curve is periodic and the second curve generator is missing.

In general case in **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** plane radius vector is described by

s(u,v) = **spine**(v) + **curve**(u) – **origin** vector function.

The general case for sliding surface is shown in Figure O.5.9.1.

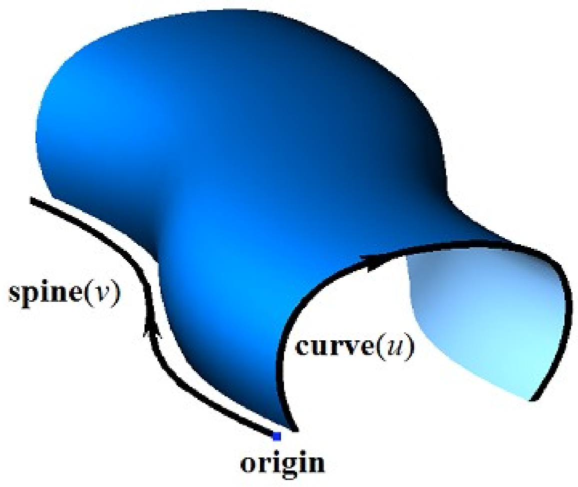


Figure O.5.9.1.

In a special case for **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** plane radius vector is described by

s(u,v) = **spine**(v) + (**curve**(u) – **origin**) (1–w) + (**brink**(t) – **ending**) w vector function,

where $w = \dfrac{v - vmin}{vmax - vmin}$, t=$tmin$+(u–$umin$)dt. A special case of sliding surface is shown in Figure O.5.9.2.

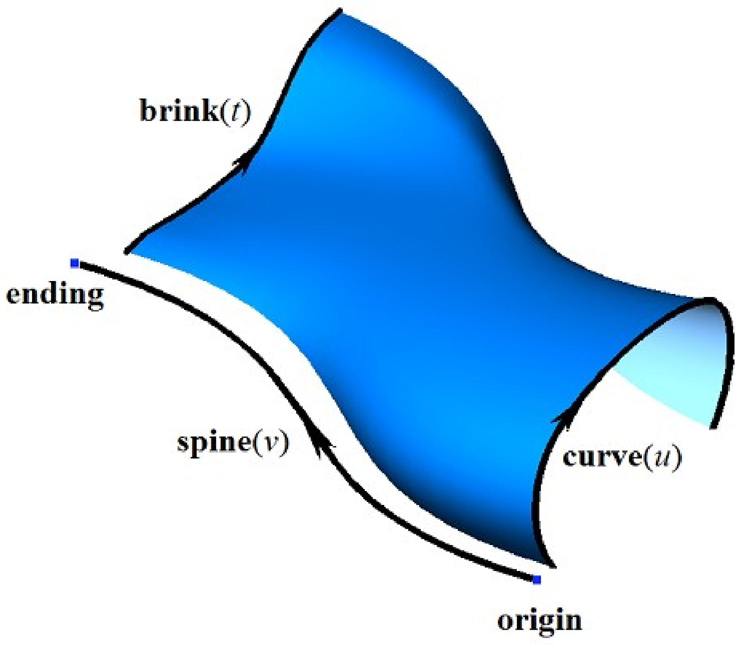


Figure O.5.9.2.

To ensure that there are no surface self-intersections, curve generating and guiding curve should not have sections parallel to each other. In certain cases, a sliding surface may have special points.

O.5.10. MbSpiralSurface Spiral Surface

MbSpiralSurface class is declared in surf_spiral_surface.h file.

MbSpiralSurface spiral surface belongs to MbSweptSurface group of swept surfaces. A spiral surface is a special case of swept surface with a guiding curve having cylindrical spiral form. A spiral surface is described by MbCurve3D* **curve** curve generator, MbPlacement3D **position** local coordinate system, **position.axizZ** vector that is spiral axis, *radius* spiral radius, *step* spiral pitch, **origin** spiral initial point position, as well as *vmin* and *vmax* spiral limiting parameters. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

Spiral axis coincides with **position.axisZ** axis in the local coordinate system. The first surface parameter (u) coincides with curve generator parameter. The first surface parameter takes values in $umin \le u \le umax$ range that corresponds to curve generator range. If the curve generator is periodic, then the surface is periodic by the first parameter.

Surface second parameter (v) takes values in $vmin \le v \le vmax$ range. v=0 corresponds to a point in the curve generator. *v= 2π values of the second parameter* correspond to the point in the curve generator with **position.axisZ** translational vector multiplied by *step*. The surface can't be periodic by the second parameter.

In **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{position.origin} +$$
$$radius\,(\cos(t)\,\mathbf{position.axisX} + \sin(t)\,\mathbf{position.axisY}) + ((t\,step/2\pi)\,\mathbf{position.axisZ}) +$$
$$(\mathbf{curve}(u) - \mathbf{origin})\,\mathbf{M}(v) \text{ vector function,}$$

where $\mathbf{M}(v)$ is rotation matrix. Please note that multiplication of (**curve**(u)–**origin**) vector by $\mathbf{M}(v)$ matrix is a post-multiplication. Rotation matrix looks as follows

$$\mathbf{M}(v) = \mathbf{A}^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{A} =$$
$$= \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}.$$

A is a matrix used to transform coordinates of radius vector from **position** local coordinate system to the global coordinate system. Rows of **A** matrix are the base vectors of the local coordinate system. $\mathbf{M}(v)$ matrix moves **curve(*u*)–origin** vector into the local coordinate system, rotates it by v angle around the rotation axis in it, and transforms the rotated vector back into the global coordinate system. A spiral surface is shown in Figure O.5.10.1.

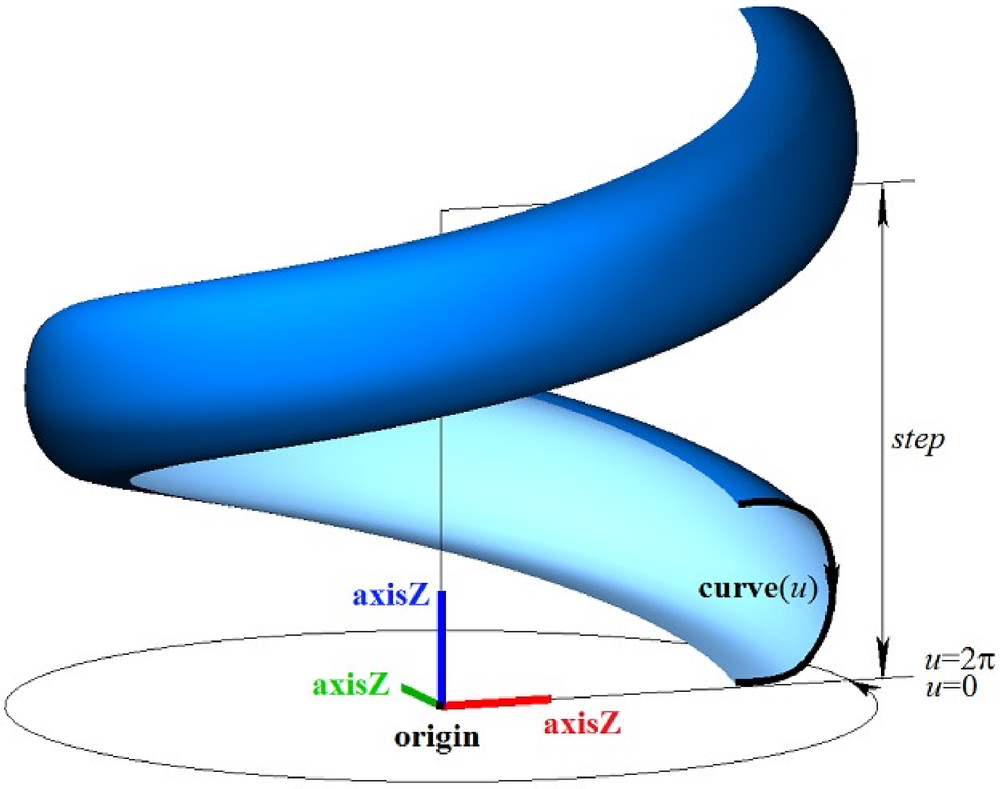


Figure O.5.10.1.

The following inequality should hold for the limits of the second parameter: *vmin*<*vmax*.

O.5.11. MbEvolutionSurface Swept Surface

MbEvolutionSurface class is declared in surf_evolution_surface.h file.

MbEvolutionSurface swept surface belongs to MbSweptSurface group of swept surfaces. A swept surface is the general case of sliding surface with an arbitrary guiding curve. A swept surface is described by MbCurve3D* **curve** curve generator, MbCurve3D* **spine** guiding object, and MbCartPoint3D **origin** location of guiding curve original point. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The first surface parameter (u) coincides with **curve** parameter of the curve generator. The first surface parameter takes values in $umin \le u \le umax$ range that corresponds to curve generator range. If the curve generator is periodic, then the surface is periodic by the first parameter.

spine guiding object replaces the guiding curve; it was constructed based on the curve and differs from the latter in that it can generate a local coordinate system associated with the curve. Surface second parameter (v) coincides with the parameter of the curve of **spine** guiding object. Surface second parameter takes values in $vmin \le v \le vmax$ range that corresponds to guiding curve parameter range. If the guiding curve is periodic, then the surface is periodic by the second parameter.

In **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + (\mathbf{curve}(u) - \mathbf{origin})\,\mathbf{M}(v) \text{ vector function,}$$

where $\mathbf{M}(v)$ is a matrix associated with the guiding curve. Please note that multiplication of $(\mathbf{curve}(u)-\mathbf{origin})$ vector by $\mathbf{M}(v)$ matrix is a post-multiplication. $\mathbf{M}(v)$ matrix looks as follows

$$\mathbf{M}(v) = \mathbf{A}^{-1}(vmin)\cdot\mathbf{A}(v),$$

where **A**(v) is the matrix used to transform coordinates of point radius vector in movable coordinate system associated with the guiding curve into the global coordinate system. **A**(v) matrix depends on second surface parameter. Rows of **A**(v) matrix are formed by base vectors of the movable coordinate system.

$$A(v) = \begin{bmatrix} i_1(v) \\ i_2(v) \\ i_3(v) \end{bmatrix},$$

where $\mathbf{i}_1(v)$ is a tangent vector of the guiding curve; $\mathbf{i}_2(v)$ is a vector orthogonal to $\mathbf{i}_1(v)$ and associated with **direction** vector of **spine** guiding object; $\mathbf{i}_3(v)$ is a vector orthogonal to $\mathbf{i}_1(v)$ and $\mathbf{i}_2(v)$. Revolution surface is shown in Figure O.5.11.1.

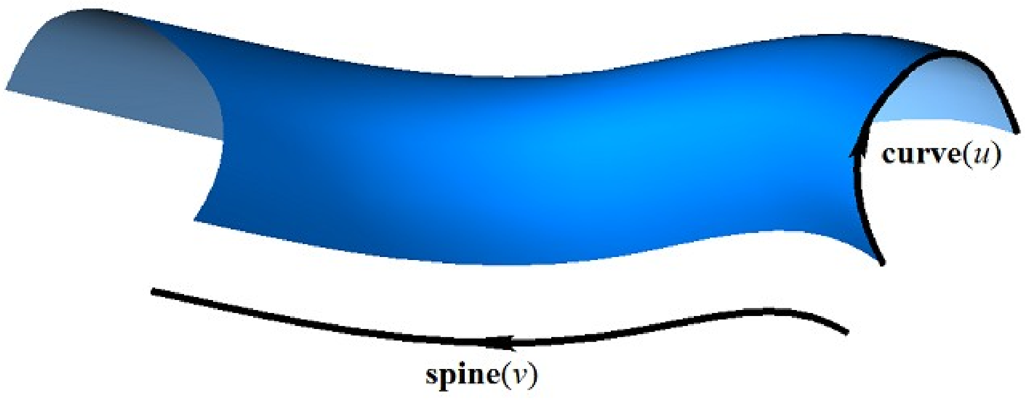


Figure O.5.11.1.

$\mathbf{i}_1(v)$ tangent vector is calculated based on the guiding curve. $\mathbf{i}_2(v)$ vector is calculated from the condition of smooth transition from a point to a point of the guiding curve, and orthogonality condition for $\mathbf{i}_1(v)$. $\mathbf{i}_3(v)$ vector is calculated as the vector product of $\mathbf{i}_1(v)$ vector and $\mathbf{i}_2(v)$ vector.

O.5.12. MbExactionSurface Swept Surface with Adaptation

MbExactionSurface class is declared in surf_exaction_surface.h file.

MbExactionSurface swept surface with adaptation is an inheritor of MbEvolutionSurface swept surface. A swept surface with adaptation is used to construct solids with the help of kinematic operations with kinked composite guiding curves, see Figure O.5.12.1.

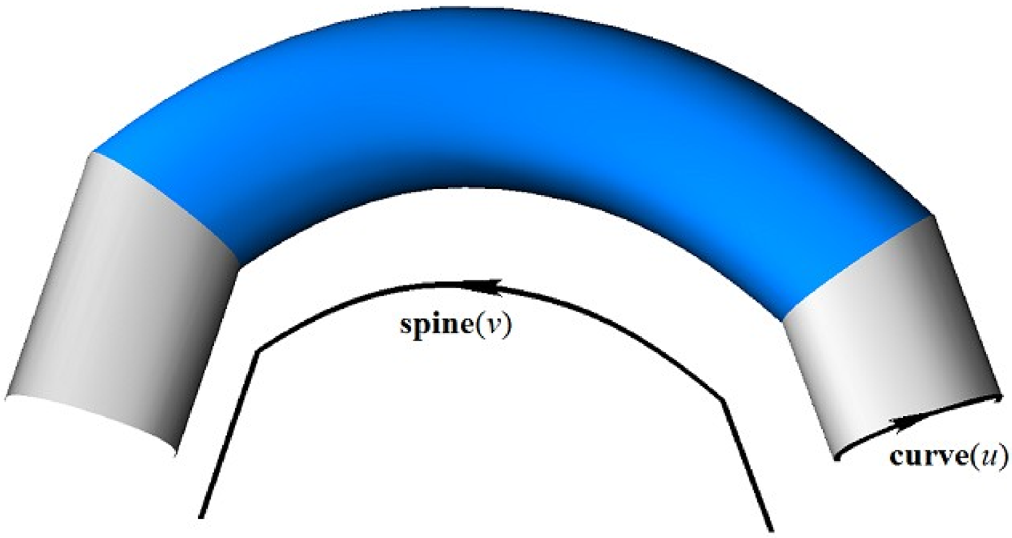


Figure O.5.12.1.

MbExactionSurface swept surface adjusts its ends in order to join it to other surface.

O.5.13. MbSectorSurface Sectorial Surface

MbSectorSurface class is declared in surf_sector_surface.h file.

MbSectorSurface sectorial surface belongs to MbSweptSurface group of swept surfaces. A sectorial surface is described by MbCurve3D* **curve** curve and MbCartPoint3D **origin** point.

The first surface parameter (*u*) coincides with **curve** curve parameter. The second surface parameter takes values belonging to *umin≤u≤umax* range that corresponds to **curve** range. If **curve** is periodic then surface is periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin≤v≤vmax* range. *v=vmin* value corresponds to a point in **curve**; *v=vmax* corresponds to **origin** point. The surface can't be periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{curve}(u)\,(1-w) + \mathbf{origin}\,w \text{ vector function,}$$

where $w = \dfrac{v - \text{vmin}}{\text{vmax} - \text{vmin}}$. A sectorial surface is shown in Figure O.5.13.1.

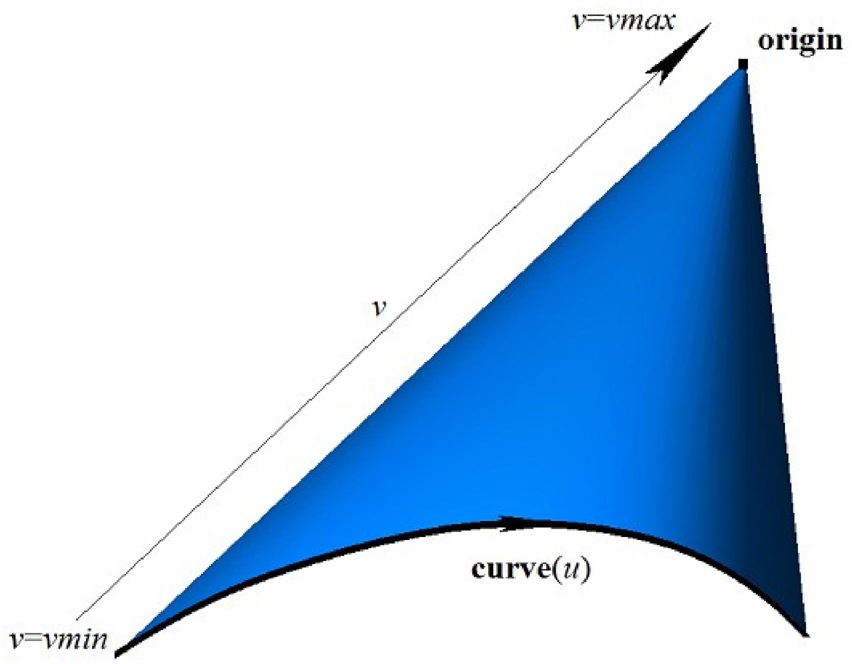


Figure O.5.13.1.

Curves of **s**(*const*,*v*) surface are line segments. The surface has a pole in **origin** point at *v*=*vmax*. A sectorial surface is a special case of ruled surface.

O.5.14. MbRuledSurface Ruled Surface

MbRuledSurface class is declared in surf_ruled_surface.h file.

MbRuledSurface ruled surface belongs to MbSweptSurface group of swept surfaces. A ruled surface is described by MbCurve3D* **curve** curve, MbCurve3D* **sline** curve, *poleMin* sign indicating availability of surface pole at the initial value of the first parameter, *poleMax* sign indicating availability of surface pole at the end value of the first parameter, *tmin* initial parameter of **sline** curve, *dt* derivative of **sline** curve parameter by **curve** curve parameter and *type* surface form. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The first surface parameter (*u*) coincides with **curve** curve parameter. The second surface parameter takes values belonging to *umin*≤*u*≤*umax* range that corresponds to **curve** range. *dt* derivate is described by the following equation:

$$dt = \frac{tmax - tmin}{umax - umin},$$

where *tmax* is the terminal parameter of **sline** curve. If **curve** curve and **sline** curve are periodic, then the surface is periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=*vmin* corresponds to a point in **curve** curve; *v*=*vmax* corresponds to a point in **sline** curve. The surface can't be periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{curve}(u)\,(1-w) + \mathbf{sline}(t)\,w \text{ vector function,}$$

where $w = \frac{v - \text{vmin}}{\text{vmax} - \text{vmin}}$, *t*=*tmin*+(*u*–*umin*)*dt*. A ruled surface is shown in Figure O.5.14.1.

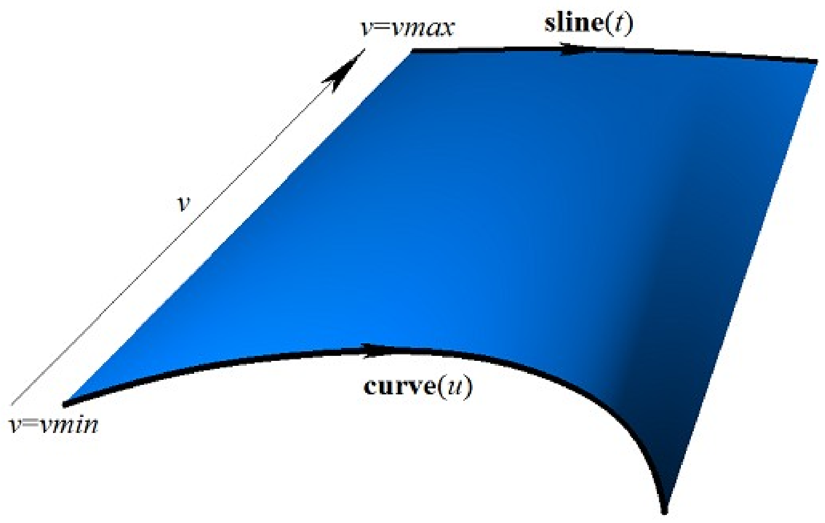


Figure O.5.14.1.

Curves at **s**(*const*,*v*) surface with *u*=const parameters are straight line segments. A surface may have a pole if **curve** curve or **sline** curve is reduced to a point, or if **curve** curve and **sline** curve coincide at one edge.

O.5.15. MbLoftedSurface Surface Based on a Family of Curves

MbLoftedSurface class is declared in surf_lofted_surface.h file.

MbLoftedSurface surface is described by RPArray<MbCurve3D>**uCurves** set of curves, *vParams* set of values of second surface parameter for curves, *vLabels* set of signs for identical curves, *umin*, *umax*, *vmin*, and *vmax* parameter limits, *uClosed*and *vClosed* surface closure signs by the first and the second parameters, directional vector for non-periodic curve at *vmin* MbVector3D **derive**1, directional vector for non-periodic curve at *vmax* MbVector3D **derive**2, *poleUMin*, *poleUMax*, *poleVMin* and *poleVMax* signs indicating availability of surface poles at the border of the definition area. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The first surface parameter (*u*) coincides with the parameters of **curves**. All **curves** should have the same parameter range. MbReperamCurve3D curves may be used for this purpose. The first surface parameter takes values in *umin*≤*u*≤*umax* range that corresponds to **curves** parameter range. If all curves in **curves** set are periodic then the surface may be periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=*vmin* value corresponds to the initial value of *vParams*[0] set; *v*=*vmax* corresponds to the terminal value of *vParams*[*vParams*.MaxIndex()] set. The surface may be periodic by the second parameter.

If all curves of **curves** set are different, then the values of *vLabels* set are equal to the index of curves in **curves** set. If there are the same adjacent curves in **curves** set that are displaced with respect to each other, then respective values of *vLabels* are equal to the minimum index of the cloned curve in **curves** set.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = (1-3w^2+2w^3)\,\mathbf{curves}[i](u) + (3w^2+2w^3)\,\mathbf{curves}[i+1](u) +$$
$$(w-2w^2+w^3)\,\mathbf{derive}[i](u) + (-w^2+w^3)\,\mathbf{derive}[i+1](u)\,(vParams[i+1]-vParams[i]) \text{ vector function,}$$

where $w = \dfrac{v - \text{vParams}[i]}{\text{vParams}[i+1] - \text{vParams}[i]}$, **derive**[*i*] and **derive**[*i*+1] are derivatives of **curves**[*i*] and **curves**[*i*+1], respectively. *i* index of working segment used to calculate radius vector of the point and its derivatives are calculated from *vParams*[*i*]≤*v*≤*vParams*[*i*+1]. If there are equal values among adjacent elements of *vLabels* set then the surface between corresponding curves is equal to the extrusion surface. A surface based on a family of curves is shown in Figure O.5.15.1.

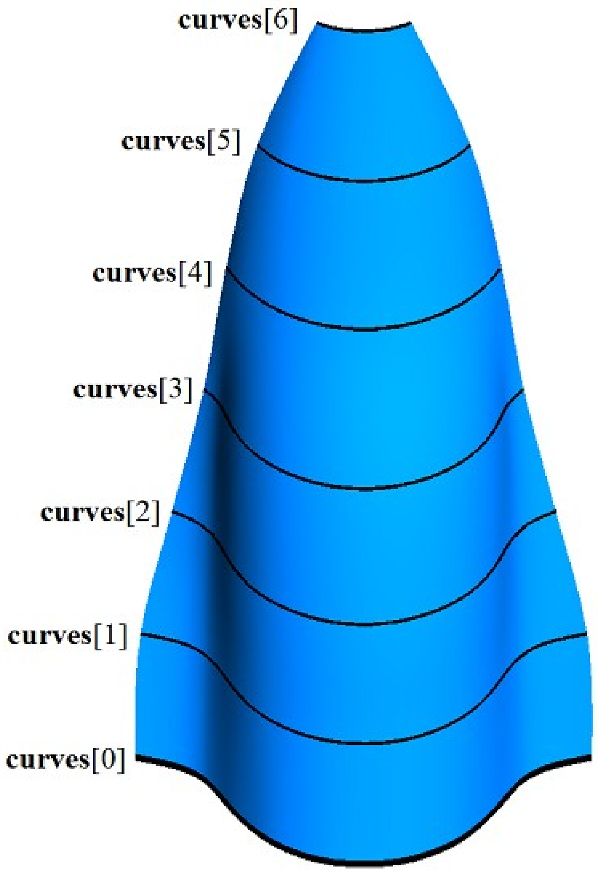


Figure O.5.15.1.

Surface form depends on location of the curves and *vParams* set of parameter values at which the surface goes by curves. In order to prevent self-intersections of the surface, the values of *vParams* set should vary in proportion to the average distance between the curves.

s(*const*,*v*) surface curves with *u*=const parameters are MbHermit3D Hermite curves. A surface may have poles, if the first and/or the last **curves** curve is reduced to a point, or if all **curves** curves coincide at one edge.

O.5.16. MbElevationSurface Surface Based on a Family of Curves and a Guiding Curve

MbElevationSurface class is declared in surf_elevation_surface.h file.

A surface based on a family of curves and a guiding curve is an inheritor of MbLoftedSurface class. Similar to MbLoftedSurface surface, MbElevationSurface surface is described by a set of generating curves (RPArray<MbCurve3D>**uCurves**), *vParams* set of values of second surface parameter for curves, *umin*, *umax*, *vmin*, *vmax* parameter limits, *uClosed* and *vClosed* surface closure signs for first and second parameters, and *poleUMin*, *poleUMax*, *poleVMin*, *poleVMax* signs indicating presence of surface poles at the border of the definition area. In addition to parameters listed above, MbElevationSurface surface is also described by MbCurve3D* **spine** guiding curve. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The first surface parameter (*u*) coincides with the parameters of **curves**. All **curves** should have the same parameter range. MbReperamCurve3D curves may be used for this purpose. The first surface parameter takes values in *umin*≤*u*≤*umax* range that corresponds to **curves** parameter range. If all curves in **curves** set are periodic, then the surface may be periodic by the first parameter.

The second parameter of *v* surface coincides with the parameter of **spine** guiding curve. Surface second parameter takes values in *vmin*≤*v*≤*vmax* range that corresponds to guiding curve parameter range. If the guiding curve is periodic, then the surface is periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is calculated similarly to MbLoftedSurface surface radius vector subject to adjustment of guiding curve offset. A surface based on a family of curves and a guiding curve is shown in Figure O.5.16.1.

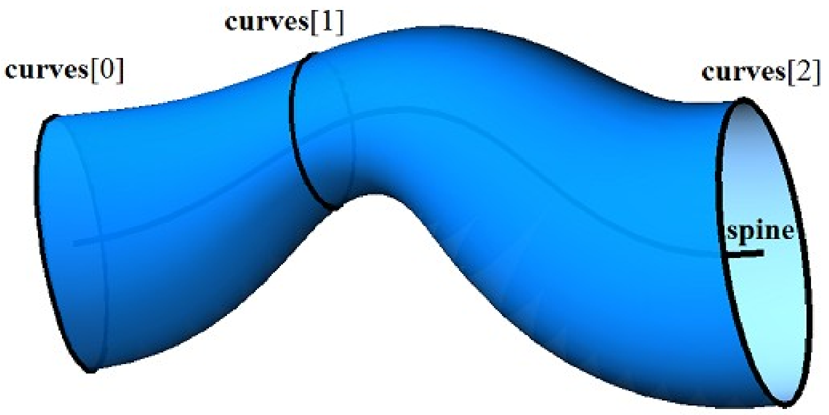


Figure O.5.16.1.

Surface form depends on location of generating curves, a guiding curve and *vParams* set of parameter values at which the surface crosses the curves. The values of *vParams* set are calculated by projecting mass centers of generating curves on the guiding curve.

O.5.17. MbCornerSurface Surface Based on Three Curves

MbCornerSurface class is declared in surf_corner_surface.h file.

MbCornerSurface surface based on three curves is described by MbCurve3D* **curve**0, **curve**1, **curve**2 curves, three MbCartPoint3D **vertex**[3] points and three pairs of limits for parameters of the corresponding curves: *t*0*min*, *t*0*max*, *t*1*min*, *t*1*max*, *t*2*min*, *t*2*max*. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

Surface first parameter (*u*) takes values in 0≤*u*≤1 range. The surface can't be periodic by the first parameter. The surface has a special point at the minimum value of the first parameter *u*=0. A derivative of the radius vector by the second parameter in the special point is zero.

Surface second parameter (*v*) takes values in 0≤*v*≤1 range. The surface can't be periodic by the second parameter.

curve0, **curve**1, **curve**2 curves should have intersection points or crossing points. *t*0*min*, *t*0*max*, *t*1*min*, *t*1*max*, *t*2*min*, *t*2*max* curve parameters are calculated using the intersection points or crossing points of the curves; these parameters define **vertex**[3] points and working segments of the curves. The directions of curves are of no importance for the surface.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\begin{aligned}\mathbf{s}(u,v) \;=\; & w0\,(\mathbf{curve}2(t2) + \mathbf{curve}1(s1) - \mathbf{vertex}[0]) + \\ & + w1\,(\mathbf{curve}0(t0) + \mathbf{curve}2(s2) - \mathbf{vertex}[1]) + \\ & + w2\,(\mathbf{curve}1(t1) + \mathbf{curve}0(s0) - \mathbf{vertex}[2]) \text{ vector function,}\end{aligned}$$

where *w*0=1–*u*, *w*1=0.5(*u*–*uv*), *w*2=0.5(*u*+*uv*) are barycentric coordinates of the surface, *t*0=*w*2·*t*0*min*+(1–*w*2)·*t*0*max*, *s*0=(1–*w*1)·*t*0*min*+*w*1·*t*0*max* are parameters of **curve**0 curve, *t*1=*w*0·*t*1*min*+(1–*w*0)·*t*1*max*, *s*1=(1–*w*2)·*t*1*min*+*w*2·*t*1*max* are parameters of **curve**1 curve, *t*2=*w*1·*t*2*min*+(1–*w*1)·*t*2*max*, *s*2=(1–*w*0)·*t*2*min*+*w*0·*t*2*max* are parameters of **curve**2 curve. A surface based on three crossing curves is shown in Figure O.5.17.1.

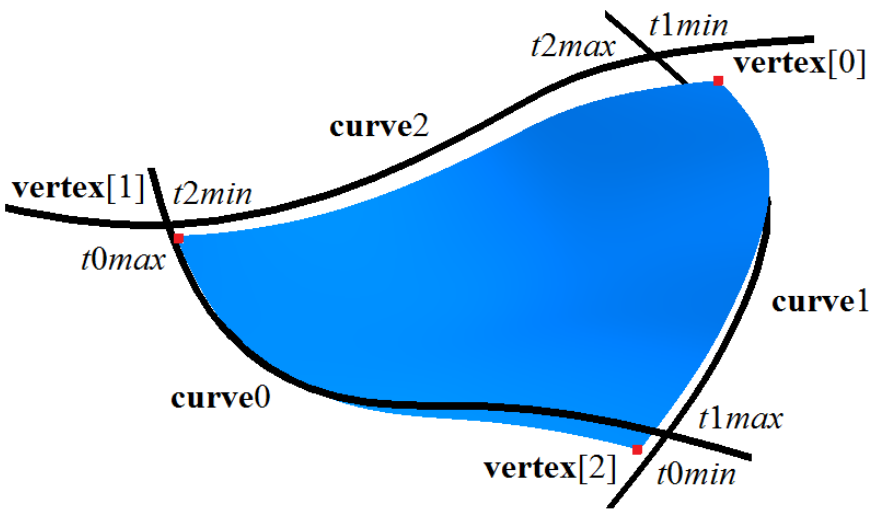


Figure O.5.17.1.

In Figure O.5.17.2, you can see a surface constructed on three identical circular arcs, the surface coincides with a part of sphere surface: planes of circular arcs are orthogonal to each other, arcs intersect in the endpoints, any arc makes a quarter of a circle.

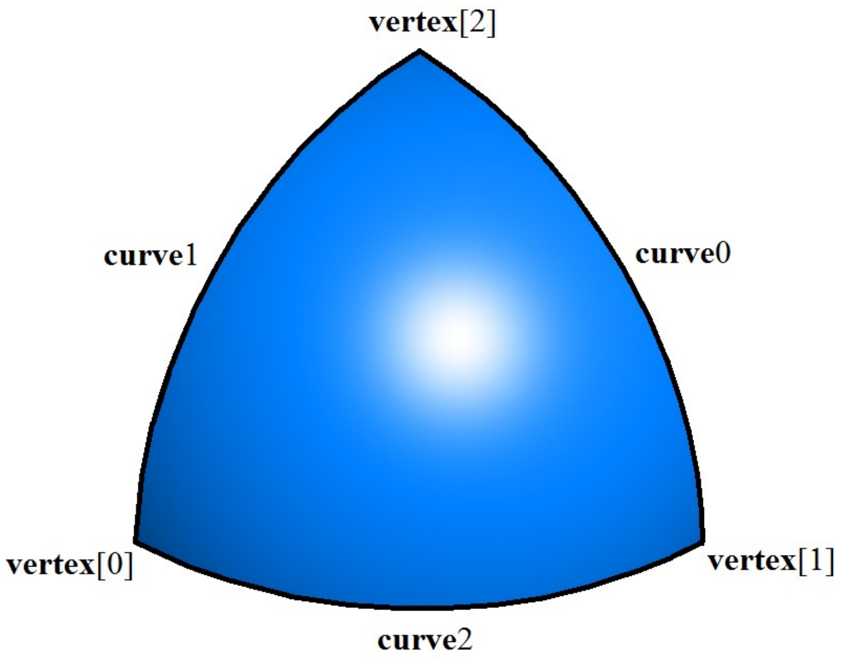


Figure O.5.17.2.

Surface form depends on the shape of the curves. If curves do not intersect, then surface does not contain them. If curves intersect then **vertex**[3] points are located in intersection points, and the surface contains segments of the curves: if *w*0=0 then the surface contains a segment of **curve**0 curve; if *w*1=0 then the surface contains a segment of **curve**1 curve; if *w*2=0, then the surface contains a segment of **curve**2 curve.

O.5.18. MbCoverSurface Coons Surface

MbCoverSurface class is declared in surf_cover_surface.h file.

MbCoverSurface Coons surface is described by MbCurve3D* **curve**0, **curve**1, **curve**2, **curve**3 curves, four MbCartPoint3D **vertex**[4] points, four pairs of parameter limits for corresponding curves (*t*0*min*, *t*0*max*, *t*1*min*, *t*1*max, t2min, t2max*, *t*3*min*, *t*3*max*), *uclosed* periodicity sign for surface first parameter, *vclosed* periodicity sign for surface second parameter, *poleUMin* sign indicating the presence of surface pole for the initial value of the first parameter, *poleUMax* sign indicating the presence of surface pole for the terminal value of the first parameter, *poleVMin* sign indicating the presence of surface pole for the initial value of the second parameter, *poleVMax* sign indicating the presence of surface pole for the terminal value of the second parameter. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

Surface first parameter (*u*) takes values in $0 \le u \le 1$ range. If **curve**0 and **curve**2 curves are periodic, then the surface is periodic by the first parameter.

Surface second parameter (*v*) takes values in $0 \le v \le 1$ range. If **curve**1 and **curve**3 curves are periodic, then the surface is periodic by the second parameter.

curve0, **curve**1, **curve**2, **curve**3 adjacent curves should have intersection points or crossing points. *t*0*min*, *t*0*max*, *t*1*min*, *t*1*max*, *t*2*min*, *t*2*max*, *t*3*min*, *t*3*max* curve parameters are calculated using intersection points or crossing points of curves; these parameters define working segments of curves and **vertex** points [4]. The directions of curves are of no importance for the surface.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\begin{aligned}\mathbf{s}(u,v) = {} & (1-v)\,(\mathbf{curve}0(t0) - (1-u)\,\mathbf{vertex}[0]) + \\ & + u\,(\mathbf{curve}1(t1) - (1-v)\,\mathbf{vertex}[1]) + \\ & + v\,(\mathbf{curve}2(t2) - u\,\mathbf{vertex}[2]) + \\ & + (1-u)\,(\mathbf{curve}3(t3) - v\,\mathbf{vertex}[3])\end{aligned}$$ vector function,

where $w0=1-u$, $w1=0.5(u-uv)$, $w2=0.5(u+uv)$ are barycentric coordinates of the surface, $t0=(1-u)\cdot t0min+u\cdot t0max$ is the parameter of **curve**0 curve, $t1=(1-v)\cdot t1min+v t1max$ is the parameter of **curve**1 curve, $t2=(1-u)\cdot t2min+u\cdot t2max$ is the parameter of **curve**2 curve, $t3=(1-v)\cdot t3min+v t3max$ is the parameter of **curve**3 curve. Coons surface based on four crossing curves is shown in Figure O.5.18.1.

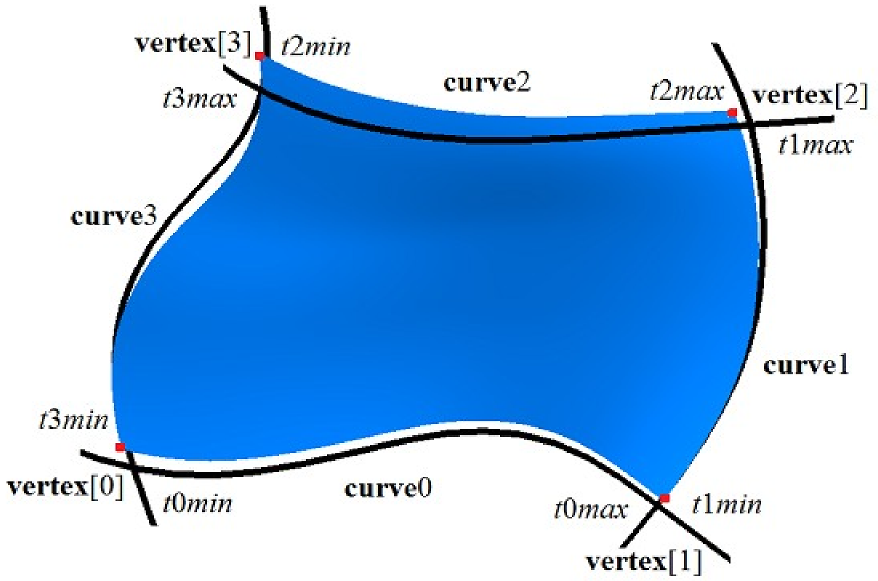


Figure O.5.18.1.

Surface form depends on the shape of the curves. If adjacent curves intersect in **vertex**[4] points, then the surface contains the following curve segments: if *u*=0, then the surface contains a segment of **curve**3 curve, if *v*=0, then the surface contains a segment of **curve**0 curve, if *u*=1, then the surface contains a segment of **curve**1, if *v*=1, then surface contains a segment of **curve**2 curve.

O.5.19. MbCoonsPatchSurface Coons Surface

MbCoonsPatchSurface class is declared in surf_coons_surface.h file.

MbCoonsPatchSurface bicubic Coons surface is constructed similar to MbCoverSurfaceCoons surface and it has additional conditions for radius vector at the edges. MbCoonsPatchSurface bicubic Coons surface is described by MbCurve3D* **curve**0, **curve**1, **curve**2, **curve**3 curves, **curve**0V derivative of surface radius vector along **curve**0 curve by surface second parameter, **curve**1U derivative of surface radius vector along **curve**1 curve by surface first parameter, **curve**2V derivative of surface radius vector along **curve**2 curve by surface second parameter, **curve**3U derivative of surface radius vector along **curve**3 curve by surface first parameter, four **vertex**[4] corner points, four **vertex**U[4] derivatives of the surface by surface first parameter in the corners, four **vertex**V[4] derivatives of the surface by surface second parameter in the corners, four **vertex**UV[4] mixed derivatives by surface first and second parameters in the corners, four pairs of parameter limits for the corresponding curves (*t*0*min*, *t*0*max*, *t*1*min*, *t*1*max, t2min, t2max*, *t3min*, *t3max*), *uclosed* periodicity sign for surface first parameter, *vclosed* periodicity sign for surface second parameter. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

Surface first parameter (u) takes values in $0 \le u \le 1$ range. The surface can be periodic by the first parameter if **curve**0, **curve**2, **curve**V0, **curve**V2 curves are periodic, and **curve**U1 and **curve**U3 curves coincide.

Surface second parameter (v) takes values in $0 \le v \le 1$ range. Surface can be periodic by the first parameter if **curve**1, **curve**3, **curve**U1 and **curve**U3 curves are periodic, and **curve**V0 and **curve and** V2 curves coincide.

curve0, **curve**1, **curve**2, **curve**3 adjacent curves should have intersection or crossing points. *t*0*min*, *t*0*max*, *t*1*min*, *t*1*max*, *t2min*, *t2max*, *t3min*, *t3max* curve parameters are calculated based on intersection points or crossing points of the curves; these parameters define working segments of curves: **vertex**[4], **vertex**U[4], **vertex**V[4], **vertex**UV[4] points. Directions of the curves are of no importance for the surface. However, parametrization of the following pairs of curves should coincide: **curve**0 and **curve**V0, **curve**2 and **curve**V2, **curve**1 and **curve**U1, **curve**3 and **curve**U3.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** radius vector of the surface is described by a vector function that is discussed in Geometric Modeling book authored by N. N. Golovanov. Bicubic Coons surface is constructed based on calculated data, it is used to construct mates and patches with mate conditions at the edges. Bicubic Coons surface is shown in Figure O.5.19.1.

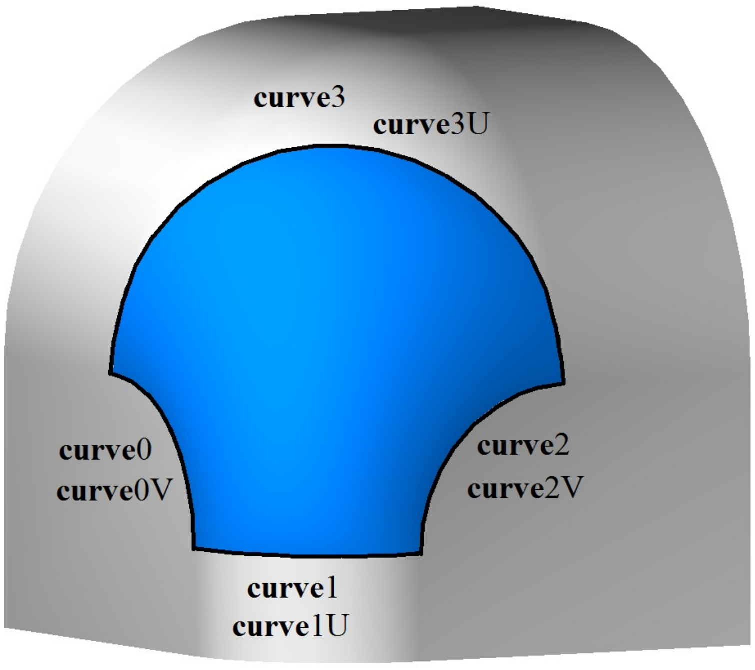


Figure O.5.19.1.

Surface form depends on the shape of curves and derivatives. If adjacent curves intersect in **vertex**[4] points then the surface contains the following curve segments: if *u*=0 then the surface contains a segment of **curve**3 curve, if *v*=0 then the surface contains a segment of **curve**0 curve, if *u*=1 then the surface contains a segment of **curve**1, if *v*=1 then surface contains a segment of **curve**2 curve.

O.5.20. MbMeshSurface Surface Based on a Network of Curves

MbMeshSurface class is declared in surf_mesh_surface.h file.

MbMeshSurface surface based on a network of curves is described by RPArray<MbCurve3D>**uCurves** set of curves, RPArray<MbCurve3D>**vCurves** set of curves, *uParams* set of values of surface first parameter, *vParams* set of values of surface second parameter, *umin* and *umax* limits of surface first parameter, *vmin* and *vmax* limits of surface second parameter, signs indicating the presence of surface poles in *poleUMin*, *poleUMax* limits of surface first parameter, signs indicating the presence of surface poles in *poleVMin*, *poleVMax* limits of surface second parameter, *uclosed* periodicity sign for surface first parameter, *vclosed* periodicity sign for surface second parameter, *type*0 surface mating type in the edge corresponding to second parameter *vmin* , *type*1 surface mating type in the edge corresponding to first parameter *umin*, *type*2 surface mating type in the edge corresponding to second parameter *vmax*,*type*3 surface mating type in the edge corresponding to first parameter *umax*. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The number of elements in **vCurves** curve set and *uParams* set of values of surface first parameter are aligned so that **vCurves**[*j*] curve points correspond to *uParams*[*j*] parameter. Surface first parameter (*u*) takes values in *uParams*[0]≤*u*≤*uParams*[*uParams*.MaxIndex()] range. If all **uCurves** curves are periodicб then the surface is periodic by the first parameter.

The number of elements in **uCurves** curve set and *vParams* set of values of surface second parameter are aligned so that **uCurves**[*i*] curve points match *vParams*[*i*] parameter. Surface second parameter (*v*) takes values in *vParams*[0]≤*v*≤*vParams*[*vParams*.MaxIndex()] range. If all **vCurves** curves are periodic then the surface is periodic by the second parameter.

Each **uCurves**[*i*] curve should have intersection points or crossing points with each **vCurves**[*j*] curve. Adjacent curves in **uCurves** set should not have opposite directions. Adjacent curves in **vCurves** set also should not have opposite directions.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by a vector function that is described in Geometric Modeling book, the author of the book is N.N. Golovanov. A surface based on a network of curves is shown in Figure O.5.20.1.

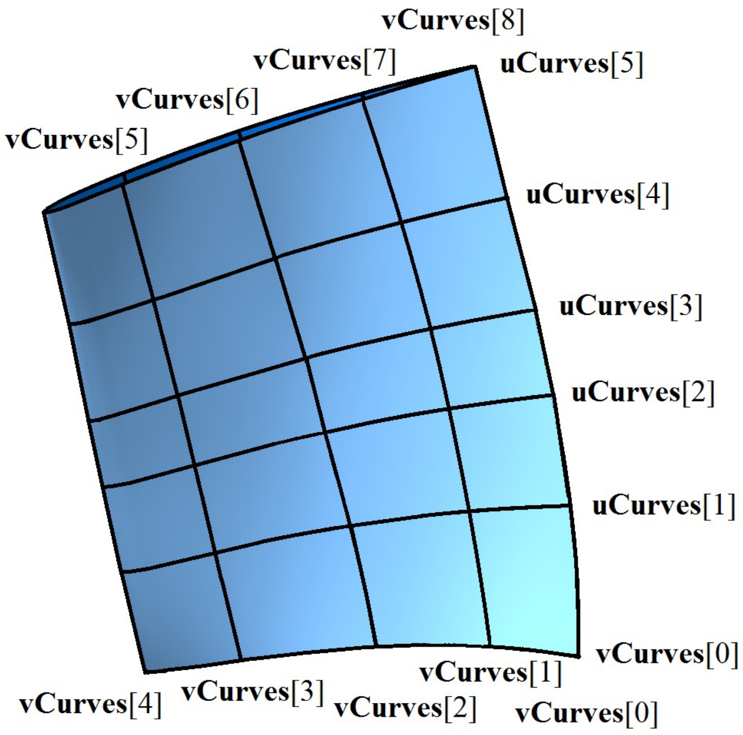


Figure O.5.20.1.

Surface form depends on the shape of curves, their relative position and the values of parameters in *uParams* and *vParams* sets. If each **uCurves**[*i*] curve intersects with each **vCurves**[*j*] curve, then the surface contains **vCurves**[*j*] curves if parameters *u*=*uParams*[*j*], and it contains **uCurves**[*i*] curves if parameters *v*=*vParams*[*i*].

O.5.21. MbJoinSurface Joint Surface

MbJoinSurface class is declared in surf_joint_surface.h file.

MbJoinSurface joint surface is described by RPArray<MbCurve3D>**curves** set of curves, *knots* nodal vector, *degree* spline order, *umin* and *umax* limits of surface first parameter, *closedU* periodicity sign of surface first parameter, *closedV* periodicity sign of the surface second parameter, a sign indicating the presence of surface poles at *isPoleUmin*, *isPoleUmax* limits of surface first parameter, a sign indicating the presence of surface poles at *isPoleUmin*, *isPoleUmax* limits of surface second parameter.

curves curves are aligned with each other: they have the same direction and parameter range. Surface first parameter (*u*) coincides with **curves** parameter of curves that is a common parameter for them. Surface first parameter takes values in *umin*≤*u*≤*umax* range that corresponds to **curves** curves parameter range. If all **curves** curves are periodic, then the surface is periodic by the first parameter. Let *knots* nodal vector contain *knotsCount* elements, and let **curves** set have *curvesCount* curves. The following equation holds for the number of elements in these sets: *curvesCount*+*degree*=*knotsCount*.

Surface second parameter (*v*takes values in *vmin*≤*v*≤*vmax* range, where *vmin*=*knots*[*degree*–1], *vmax*=*knots*[*knotsCount*–*degree*]. The surface can't be periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by vector function

$$s(u,v) = \frac{\sum_{j=0}^{\text{curvesCount}-1} N_j^{\text{degree}}(v)\text{curves}[j](u)}{\sum_{j=0}^{\text{curvesCount}-1} N_j^{\text{degree}}(v)},$$

where $N_j^{degree}(v)$ are B-splains of *degree* order for *j*th **curves**[*j*] curve. A joint surface is shown in Figure O.5.21.1.

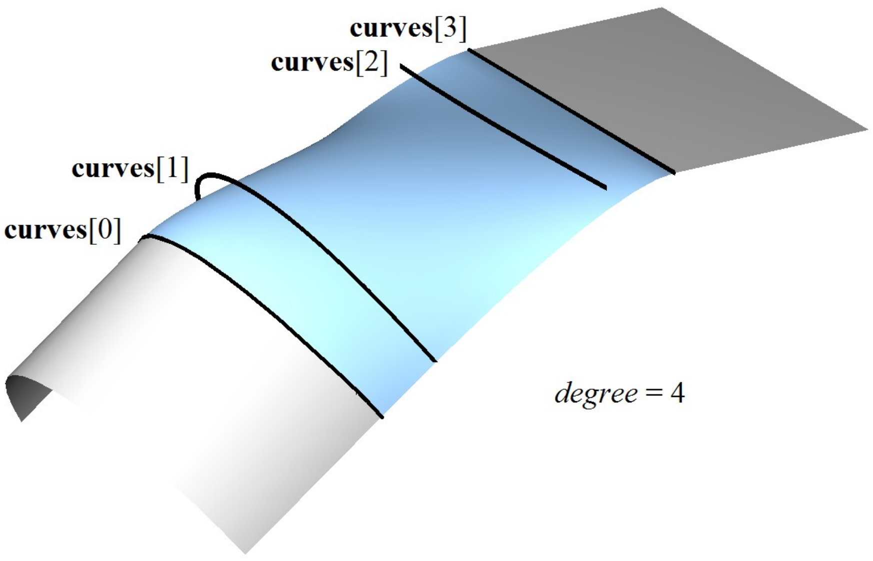


Figure O.5.21.1.

Each **s**(*const*,*v*) curve with fixed first parameter (*u*=const) is a NURBS curve of *degree* order constructed based on **curves**[*i*](*const*) points.

O.5.22. MbSplineSurface NURBS Surface

MbSplineSurface class is declared in surf_spline_surface.h file.

MbSplineSurface NURBS (NonUniform Rational B-Spline surface) surface is described by SArray<MbCartPoint3D>**points**[*i*][*j*], *i*=0,1,...,*vcount*–1, *j*=0,1,...,*ucount*-1 control points that are conventionally located in the nodes of a rectangular table having *ucount* columns and *vcount* rows, weights of control pints defined in *weight*[*i*][*j*] table, *udegree* order of B-splines along surface first parameter, *vdegree* degree of B-splines along surface second parameter, *uknots* nodal vector along the first parameter, *vknots* nodal vector along the second parameter, *uclosed* and *vclosed* surface periodicity signs for the first and for the second parameters. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

The order of B-splines along surface parameters coincides with the order of divided difference that was used to calculate corresponding B-splines. *uknots* and *vknots* nodal vectors are non-decreasing sequences of real numbers that define definition area of surface parameter and the form of the surface. Let *uknots* nodal vector contain *uknotsCount* elements, and the number of points in each row of rectangular table be equal to *ucount*. For a NURBS surface that is non-periodic by the first parameter, the following equation holds for the numbers of elements in the sets: *ucount*+*degree*=*uknotsCount*. For a periodic NURBS surface the following equation holds for the numbers of elements in the sets: *knotsCount*+2*degree*–1=*knotsCount*.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$r(u,v) = \frac{\sum_{i=0}^{\text{vcount}-1}\sum_{j=0}^{\text{ucount}-1} N_i^{\text{vdgree}}(v) N_j^{\text{udgree}}(u)\,\text{weight}[i][j]\,\text{points}[i][j]}{\sum_{i=0}^{\text{vcount}-1}\sum_{j=0}^{\text{ucount}-1} N_i^{\text{vdgree}}(v) N_j^{\text{udgree}}(u)\,\text{weight}[i][j]},$$

where $N_i^{vdegree}(v)$ and $N_j^{udegree}(u)$ are B--splains. In Fig. O.5.22.1 you can see segments that connect adjacent **points**[*i*][*j*] control points.

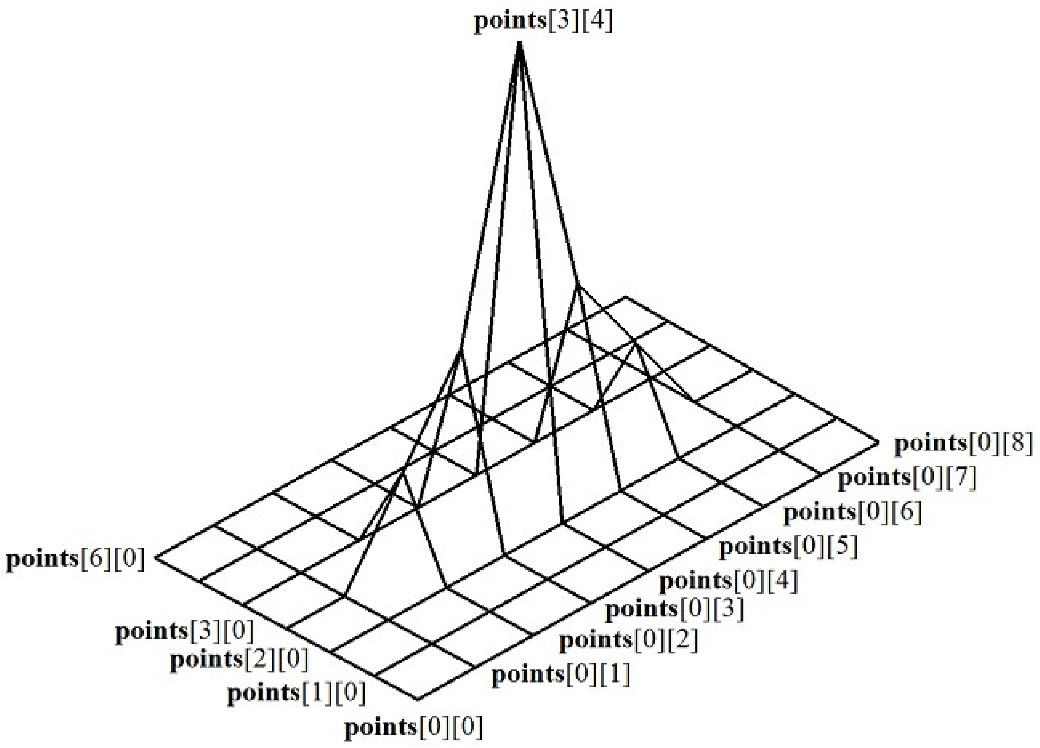


Figure O.5.22.1.

Figure O.5.22.1 demonstrates control points that are conventionally located in the nodes of a rectangular table. NURBS surface that is constructed based on the same control points is shown in Figure O.5.22.2.

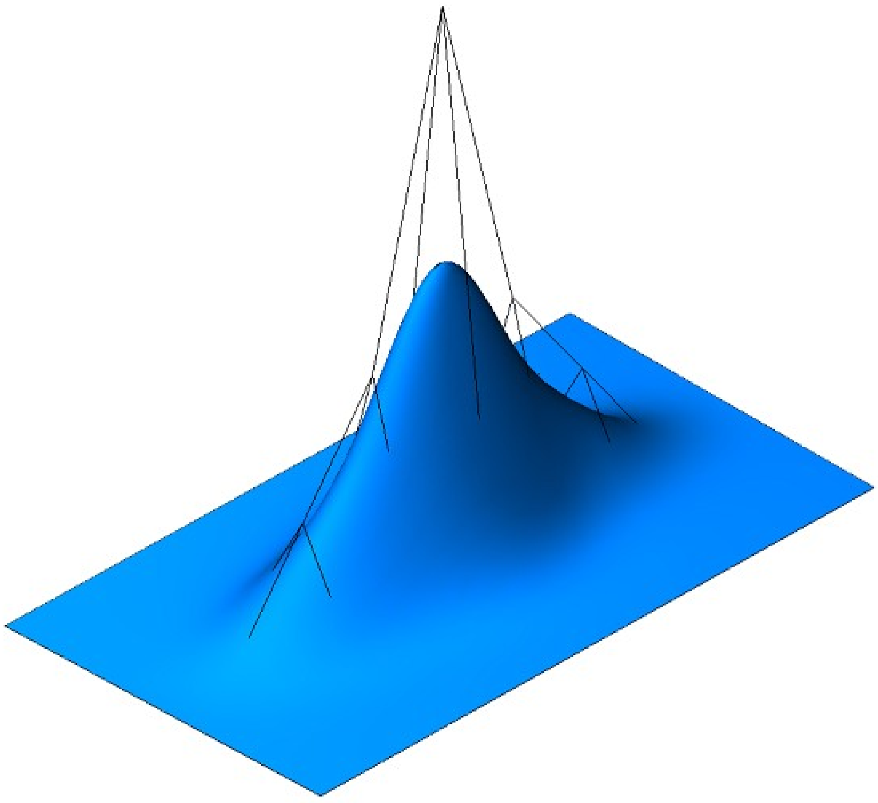


Figure O.5.22.2.

s(*const*,*v*) and **s**(*u*,*const*) curves on the surface with *u*=const or *v*=const parameters are *B*-curves having *udegree* order and *vdegree* order respectively. *udegree* is surface degree by the first parameter; *vdegree* is surface degree by the second parameter. Parameter variation area of non-periodic NURBS surface is a rectangle: *uknots*[*udegree*–1]≤*u*≤*uknots*[*ucount*], *vknots*[*vdegree*–1]≤*v*≤*vknots*[*vcount*].

A surface may be periodic both by the first parameter and by the second parameter. *uknots* and *vknots* nodal vectors for periodically closed surface have *udegree*–1 and *vdegree*–1 more elements, respectively. If *NURBS* surface is periodic by both parameters then parameter variation area is a rectangle: *uknots*[*udegree*–1]≤*u*≤*uknots*[*ucount*+*udegree*–1], *vknots*[*vdegree*–1]≤*v*≤*vknots*[*vcount*+*vdegree*–1].

Every surface can construct its NURBS copy using **NurbsSurface**(const MbNurbsParameters & *uParam*, const MbNurbsParameters & *vParam*) virtual method.

O.5.23. MbOffsetSurface Equidistant Surface

MbOffsetSurface class is declared in surf_offset_surface.h file.

MbOffsetSurface equidistant surface is described by MbSurface* **basisSurface** base surface, *distance* offset along the normal to the base suface, *u*0*min*, *u*0*max* limits of base surface first parameter, *v*0*min*, *v*0*max* limits of base surface second parameter, *u*0*closed*, *v*0*closed* base surface periodicity signs, *dumin*, *dumax* increments of the limits of base surface first parameter, *dvmin*, *dvmax* increments of the limits of base surface second parameter. There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

Point radius vector of equidistant surface is calculated as follows. The point of the base surface and the normal in this point are calculated for a preset parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

r(*u*,*v*) = **basisSurface**(*u*,*v*) + **normal**(*u*,*v*) · *distance* vector function,

where **normal**(*u*,*v*) is normal to the surface in the preset point. An equidistant surface and its base surface are shown in Figure O.5.23.1.

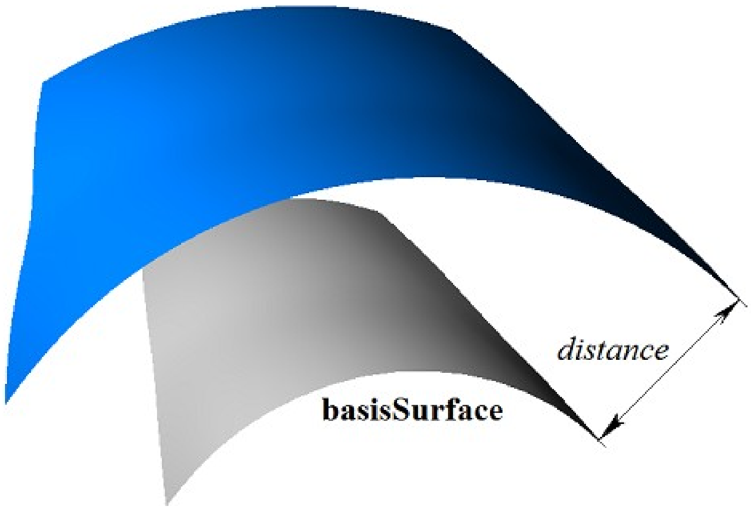


Figure O.5.23.1.

Parameter variation area of equidistant surface and parameter variation area of its base surface may differ. Parameter variation area of the first parameter of equidistant surface is determined by the following inequalities: $u0min+dumin \le u \le u0max+dumax$. Parameter variation area of the second parameter of equidistant surface is determined by the following inequalities: $v0min+dvmin \le v \le v0max+dvmax$. An equidistant surface with a negative offset and expanded parameter definition area and its base surface are shown in Figure O.5.23.2.

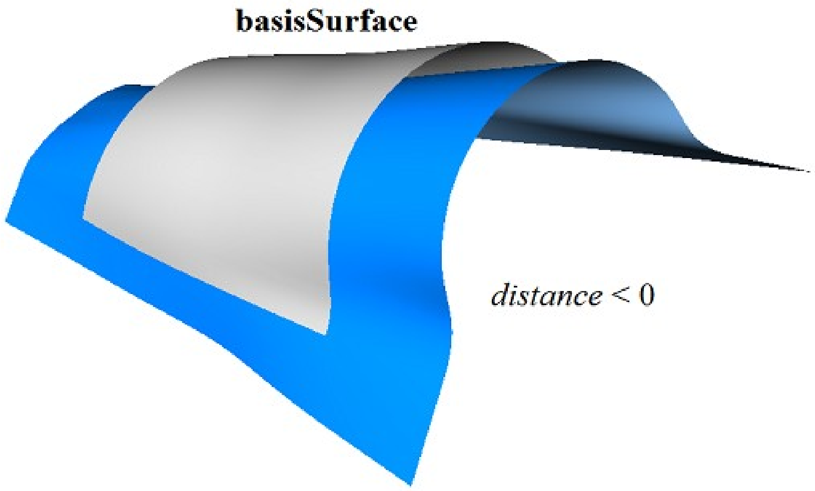


Figure O.5.23.2.

An equidistant surface can't use other equidistant surface as a base surface; rather base surface of the other equidistant surface should be used subject to corresponding recalculation of offset value.

Each surface can construct an equidistant surface using **Offset**(double *distance*, bool *sense*) virtual method.

O.5.24. MbChamferSurface Chamfer Surface

MbChamferSurface class is declared in surf_chamfer_surface.h file.

MbChamferSurface chamfer surface belongs to MbSmoothSurface group of mating surfaces. Chamfer surface is described by MbSurfaceCurve* **curve**1 curve on the first mated surface, MbSurfaceCurve* **curve**2 curve on the second mated surface, *form* chamfer construction method, *distance*1 and *distance*2 chamfer sides, *umin* and *umax* parameter limits **curve**1 and **curve**2 curve parameters, *vmin* and *vmax* limits of surface first parameter, *uclosed* periodicity sign of surface first parameter, *poleMin* sign indicating a surface pole for the initial value of the first parameter, *poleMax* sign indicating a surface pole for the end value of the first parameter.

curve1 and **curve**2 are aligned with each other, they have the same parameter range. *u* is the first surface parameter that coincides with parameter of **curve**1 and **curve**2 curves that is common for them. The first surface parameter takes values in *umin*≤*u*≤*umax* range that corresponds to parameter definition area of **curve**1 and **curve**2 curves. If **curve**1 and **curve**2 curves are periodic, then the surface is periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=*vmin* corresponds to a point in **curve**1 curve, *v*=*vmax* corresponds to a point at **curve**2 curve. The surface can't be periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{curve}1(u)\,(1-w) + \mathbf{curve}2(u)\,w \text{ vector function,}$$

where $w = \frac{v - \text{vmin}}{\text{vmax} - \text{vmin}}$. A chamfer surface is shown in Figure O.5.24.1.

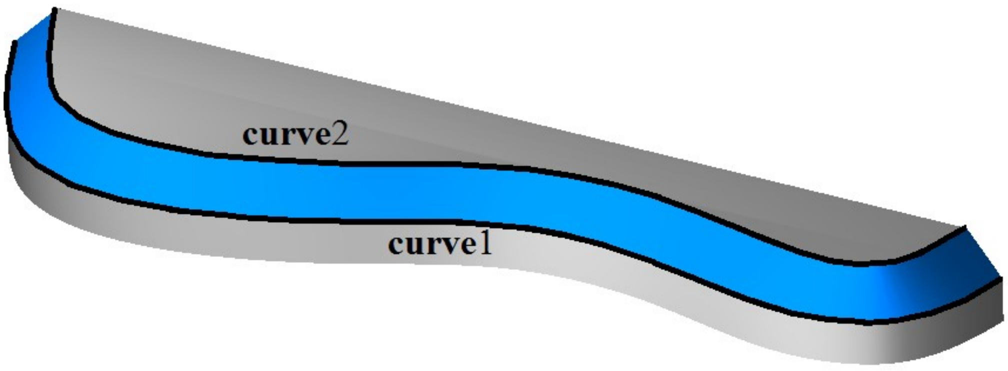


Figure O.5.24.1.

Curves at **s**(*const*,*v*) surface with *u*=const parameters are straight line segments. A surface may have a pole at *u*=*umin* and *u*=*umax* if corresponding edges of **curve**1 and **curve**2 curves coincide.

O.5.25. MbFilletSurface Fillet Surface

MbFilletSurface class is declared in surf_fillet_surface.h file.

MbFilletSurface fillet surface belongs to MbSmoothSurface group of mating surfaces. MbFilletSurface fillet surface is described by MbSurfaceCurve* **curve**1 curve on first mated surface, MbSurfaceCurve* **curve**2 curve on second mated surface, MbCurve3D* **curve**0 curve, MbFunction* **weights**0 weight function of **curve**0 curve, *form* filleting method, *distance*1 and *distance*2 filleting radii, *conic* shape coefficient, *umin* and *umax* limits of **curve**1, **curve**2, **curve**0 curve parameters and **weights**0 functions, *vmin* and *vmax* limits of surface second parameter, *uclosed* periodicity sign of surface first parameter, *poleMin* sign indicating a surface pole at the initial value of the first parameter, *poleMax* sign indicating a surface pole at the terminal value of the first parameter, *even* sign indicating uniform surface parameterization for the second parameter, *equable* sign indicating smooth mating of the surface with mated surfaces, *byCurve*1 sign indicating an edge along **curve**2 or**curve**1 curve (*equable*=false). There are some other surface parameters that are not mandatory, they are used to speed up surface methods.

curve1, **curve**2, **curve**0 curves and **weights**0 function are aligned with each other and have the same parameter range. *u* is the first surface parameter that coincides with **curve**1, **curve**2, **curve**0 curve parameter and **weights**0 function that is common for them. Surface first parameter takes values in *umin≤u≤umax* range that corresponds to parameter range of **curve**1, **curve**2, **curve**0 curves and **weights**0 function. If **curve**1, **curve**2, **curve**0 curves and **weights**0 function are periodic, then the surface is periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin≤v≤vmax* range. *v=vmin* corresponds to a point in **curve**1 curve, *v=vmax* corresponds to a point at **curve**2 curve. The surface can't be periodic by the second parameter.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by

$$s(u,v) = \frac{(1-v)^2\text{curve}\,1(u) + 2t(1-v)\text{weight}\,0(u)\text{curve}\,0(u) + v^2\text{curve}\,2(u)}{(1-v)^2 + 2v(1-v)\text{weight}\,0(u) + v^2}.$$

Each **s**(*const*,*v*) curve with fixed surface first parameter (*u*=const) is third-order NURBS curve constructed based on **curve**1(*u*), **curve**0(*u*), **curve**2(*u*) points; the weights of the outermost points of this NURBS curve are 1, the weight of **curve**0(*u*) midpoint is equal to **weights**0(*u*). If *conic*=_ARC_, then **weights**0(*u*) function is calculated from condition that all **s**(*const*,*v*) curves are circular arcs. If *conic*≠_ARC_, then **weights**0 function is a constant and it is equal to *conic* shape coefficient. A fillet surface is shown in Figure O.5.25.1.

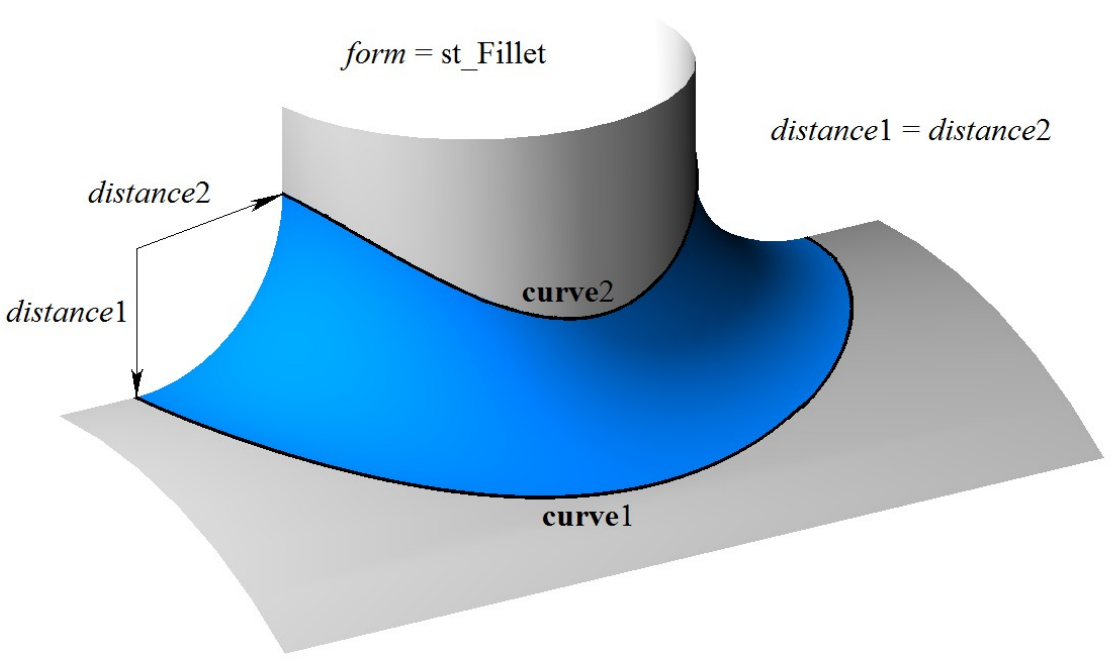


Figure O.5.25.1.

In general case, this surface smoothly mates with the surfaces where **curve**1(*u*) and **curve**2(*u*) curves are located. In this case *equable* parameter is "true". If *equable*=false, then for **curve**1 or **curve**2 mating is smooth, and the other curve is face edge. If *byCurve*1=true, then mating of **curve**1 curve is smooth, otherwise this is true for **curve**2. A fillet surface with kept edge is shown in Figure O.5.25.2.

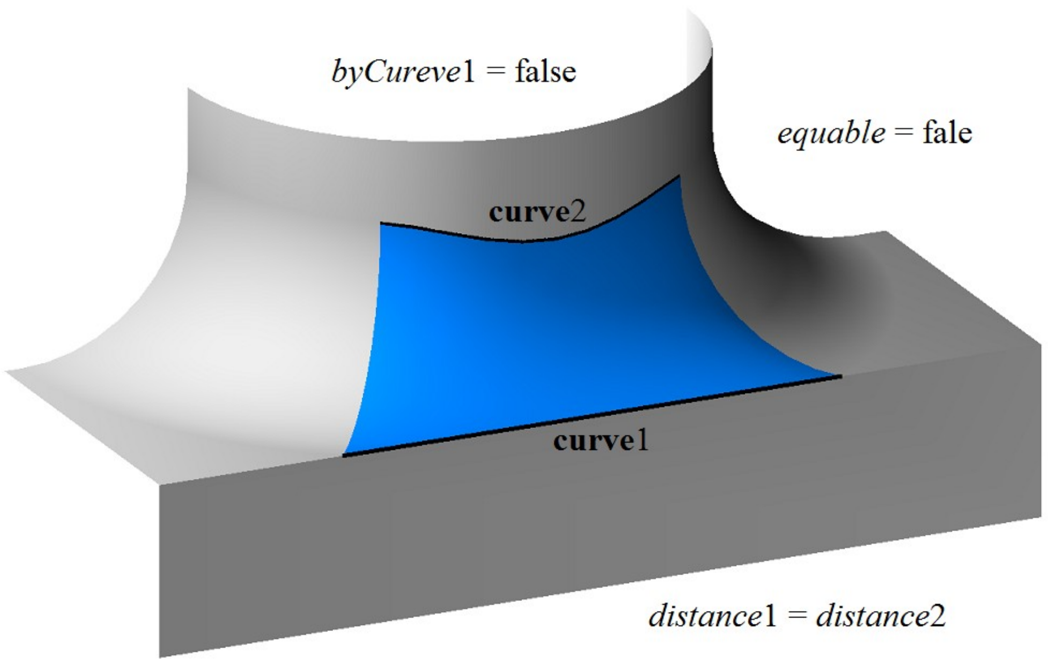


Figure O.5.25.2.

If *distance*1 and *distance*2 filleting radii are not equal, then fillet surface in **s**(*const*,*v*) section with *u*=const parameters circumscribes an ellipse. An elliptical fillet surface is shown in Figure O.5.25.3.

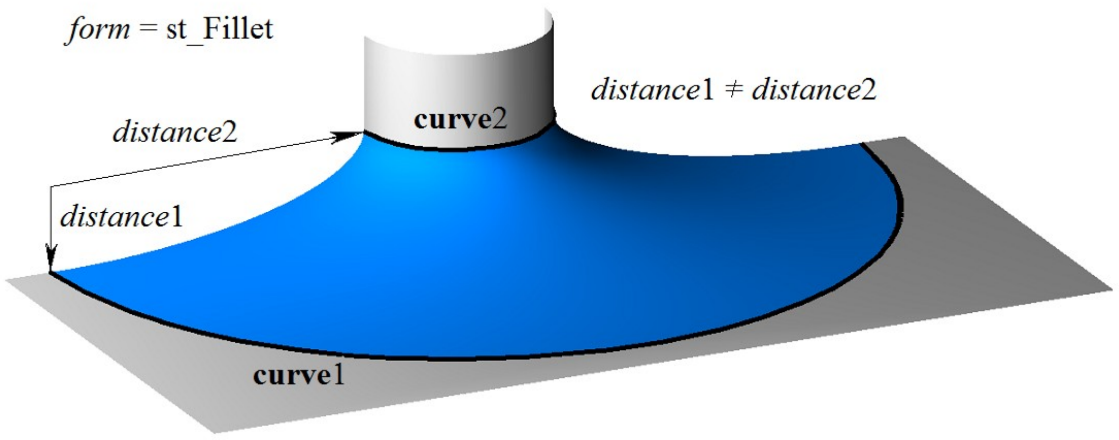


Figure O.5.25.3.

In general case, filleting method is *form*=st_Fillet. If *form*=st_Span, then *distance*1=*distance*2 and they are equal to the distance between **curve**1 and **curve**2, and surface filleting radius is variable. A fillet surface with preserved distance between reference curves is shown in Figure O.5.25.4.

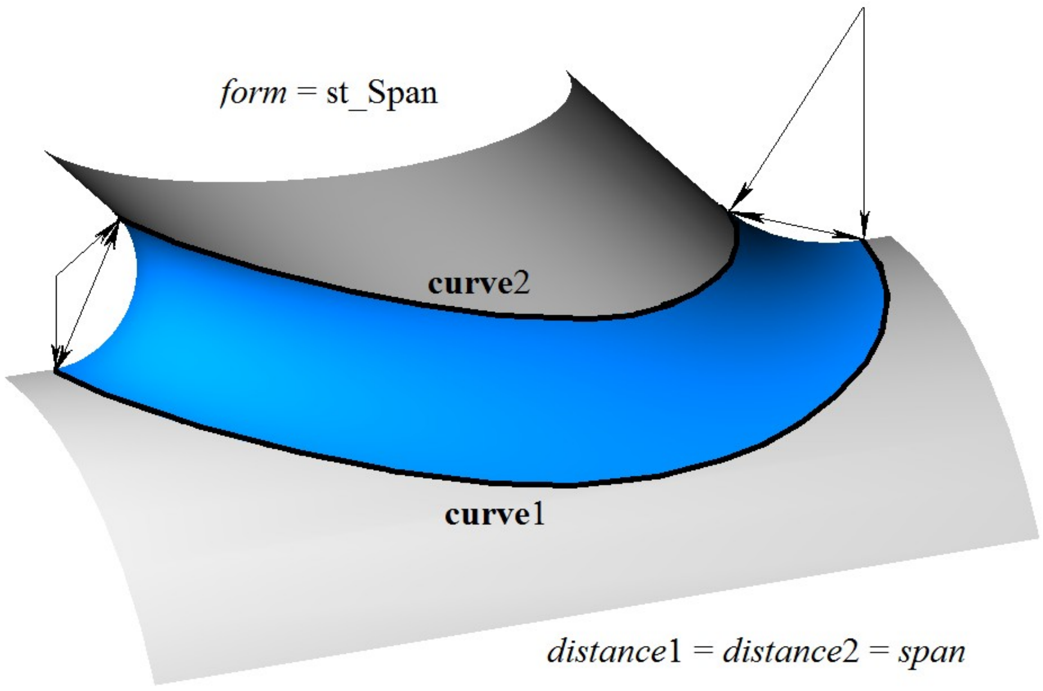


Figure O.5.25.4.

Curves of **s**(*const*,*v*) surface with *u*=const parameters are conic sections, their shape depends on *conic* parameter. If *conic*=_ARC_=0 then the curves of **s**(*const*,*v*) surface are circular arcs. If *conic*=0.5 then the curves of **s**(*const*,*v*) surface are parabolic arcs. If 0.05<*conic*<0.5 then the curves of **s**(*const*,*v*) surface are elliptical arcs. If 0.5<*conic*<0.95 then the curves of **s**(*const*,*v*) surface are hyperbolic arcs. A surface may have a pole at *u*=*umin* and at *u*=*umax* if the corresponding edges of **curve**0, **curve**1 and **curve**2 curves coincide.

O.5.26. MbChannelSurface Fillet Surface

MbChannelSurface class is declared in surf_channel_surface.h file.

MbChannelSurface fillet surface is an inheritor of MbFilletSurface fillet surface. MbChannelSurface fillet surface is described by MbSurfaceCurve* **curve**1 curve on the first mated surface, MbSurfaceCurve* **curve**2 curve on the second mated surface, MbCurve3D* **curve**0 curve, MbFunction* **weights**0 weight function of **curve**0 curve, MbFunction* **function** radius change function, *form* filleting method, *distance*1 and *distance*2 filleting radii, *conic* shape coefficient, *umin* and *umax* limits of **curve**1, **curve**2, **curve**0 curve parameters and **weights**0 and **function** functions, *vmin* and *vmax* limits for surface second parameter, *uclosed* periodicity sign for surface first parameter, *poleMin* sign indicating a surface pole at the initial value of the first parameter, *poleMax* sign indicating a surface pole at the end value of the first parameter. There are some other parameters of the surface that are not mandatory, they are used to speed up the methods.

curve1, **curve**2, **curve**0 curves and **weights**0, **function** functions are aligned with each other and have the same parameter range. *u* is the first surface parameter, it coincides with parameter of **curve**1, **curve**2, **curve**0 curves and**weights**0 and **function** functions that is common for them. The first surface parameter takes values in *umin*≤*u*≤*umax* range that corresponds to the range of **curve**1, **curve**2, **curve**0 curves and **weights**0 and **function** functions. If **curve**1, **curve**2, **curve**0 curves and **weights**0 and **function** functions are periodic then the surface may be is periodic by the first parameter.

Surface second parameter (*v*) takes values in *vmin*≤*v*≤*vmax* range. *v*=*vmin* corresponds to a point in **curve**1 curve, *v*=*vmax* corresponds to a point at **curve**2 curve. The surface can't be periodic by the second parameter.

In **PointOn** (double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by vector function

$$\mathbf{s}(u,v) = \frac{(1-v)^2\mathbf{curve}1(u) + 2t(1-v)weight0(u)\mathbf{curve}0(u) + v^2\mathbf{curve}2(u)}{(1-v)^2 + 2v(1-v)weight0(u) + v^2}$$

Each **s**(*const*,*v*) curve with fixed surface first parameter (*u*=const) is third-order NURBS curve constructed based on **curve**1(*u*), **curve**0(*u*), **curve**2(*u*) points; the weights of the outermost points of this NURBS curve are 1, the weight of **curve**0(*u*) mid point is equal to **weights**0(*u*). If surface first parameter is modified, then filleting radii are changed as follows: R1(*u*)=*distance*1·**function**(*u*) and R2(*u*)=*distance*2·**function**(*u*). If *conic*=_ARC_, then **weights**0(*u*) function is calculated from condition that all **s**(*const*,*v*) curves are circular arcs. If *conic*≠_ARC_, then **weights**0 function is a constant and it is equal to *conic* shape coefficient. A fillet surface with variable radius is shown in Figure O.5.26.1.

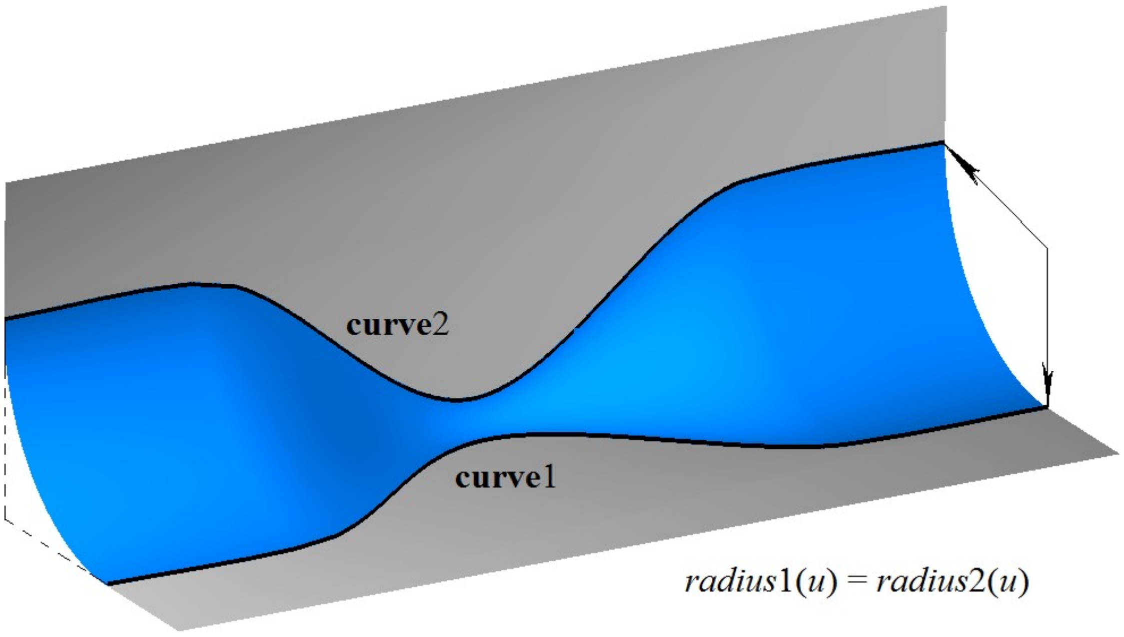


Figure O.5.26.1.

A fillet surface with variable radius is smoothly mated with surfaces, where **curve**1(*u*) and **curve**2(*u* curves are found; in this case *equable*=true and *form*=st_Fillet.

Curves of **s**(*const*,*v*) surface with *u*=const parameters are conic sections, their shape depends on *conic* parameter. If *conic*=_ARC_=0, then the curves of **s**(*const*,*v*) surface are circular arcs. If *conic*=0.5, then the curves of **s**(*const*,*v*) surface are parabolic arcs. If 0.05<*conic*<0.5, then the curves of **s**(*const*,*v*) surface are elliptical arcs. If 0.5<*conic*<0.95, then the curves of **s**(*const*,*v*) surface are hyperbolic arcs. A surface may have a pole at *u*=*umin* and at *u*=*umax* if the corresponding edges of **curve**0, **curve**1 and **curve**2 curves coincide.

O.5.27. MbCurveBoundedSurface Surface with Arbitrary Borders

MbCurveBoundedSurface class is declared in surf_curve_bounded_surface.h file.

MbCurveBoundedSurface surface with arbitrary borders is described by MbSurface* **basisSurface** base surface, **RPArray**<MbContourOnSurface>**curves** set of boundary curves (contours on the surface) as well as by *umin*, *umax*, *vmin* and *vmax* parameter limits that define a dimensional rectangle of parameter definition area.

MbCurveBoundedSurface surface has curved edges and can have arbitrary cutouts inside. Surface boundaries describe contours on the surface of **curves** container.

In **PointOn**(double *u*, double *v*, MbCartPoint3D & **s**) method, **s** surface radius vector is described by vector function

$$\mathbf{s}(u,v) = \mathbf{basisSurface}(u,v) \text{ vector function}, \quad u,v\in\Omega ,$$

where Ω is parameter definition area represented by a connected two-dimensional area. Radius vector of the surface bounded by the contours is described according to the same law as **basisSurface** surface; but parameter definition area is different. A base surface and contours on it are shown in Figure O.5.27.1.

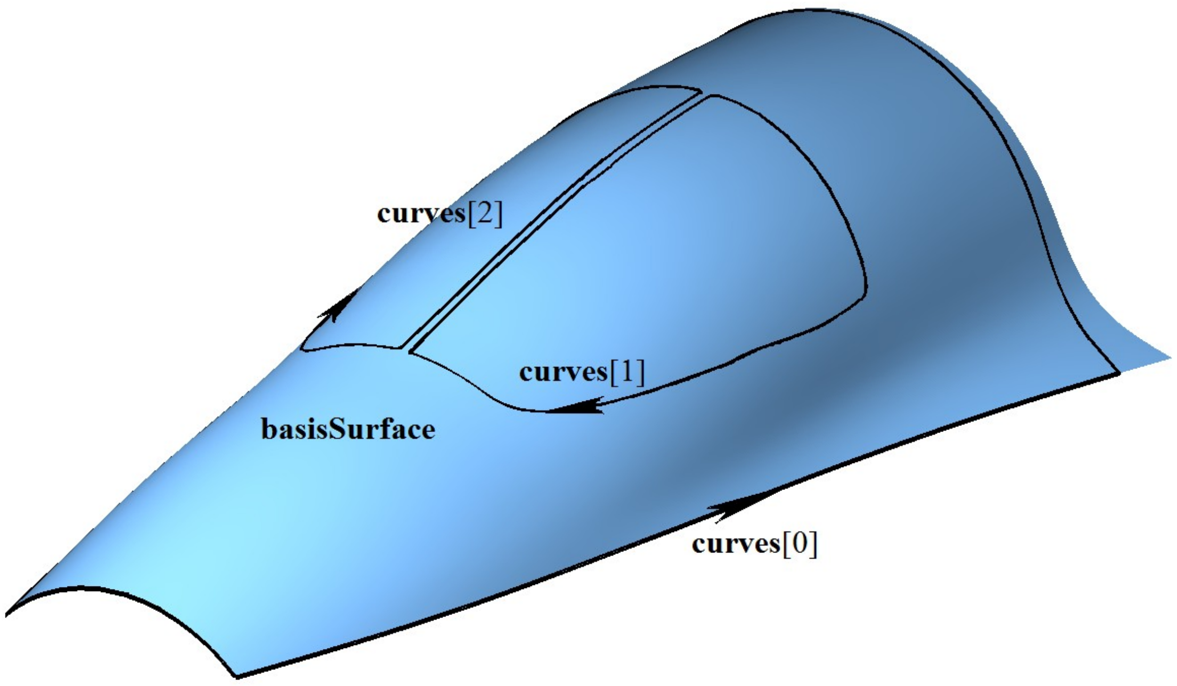


Figure O.5.27.1

In general case, parameter definition area of Ω surface that is bounded by surface contours can go beyond **basisSurface** parameter definition area. Outside of **basisSurface** surface parameter definition area, **r**(*u*,*v*) radius vector is calculated using **_PointOn**(*u*,*v*,**s**) method according to **basisSurface** surface extension rules. A surface with an arbitrary border is shown in Fig. O.5.27.2.

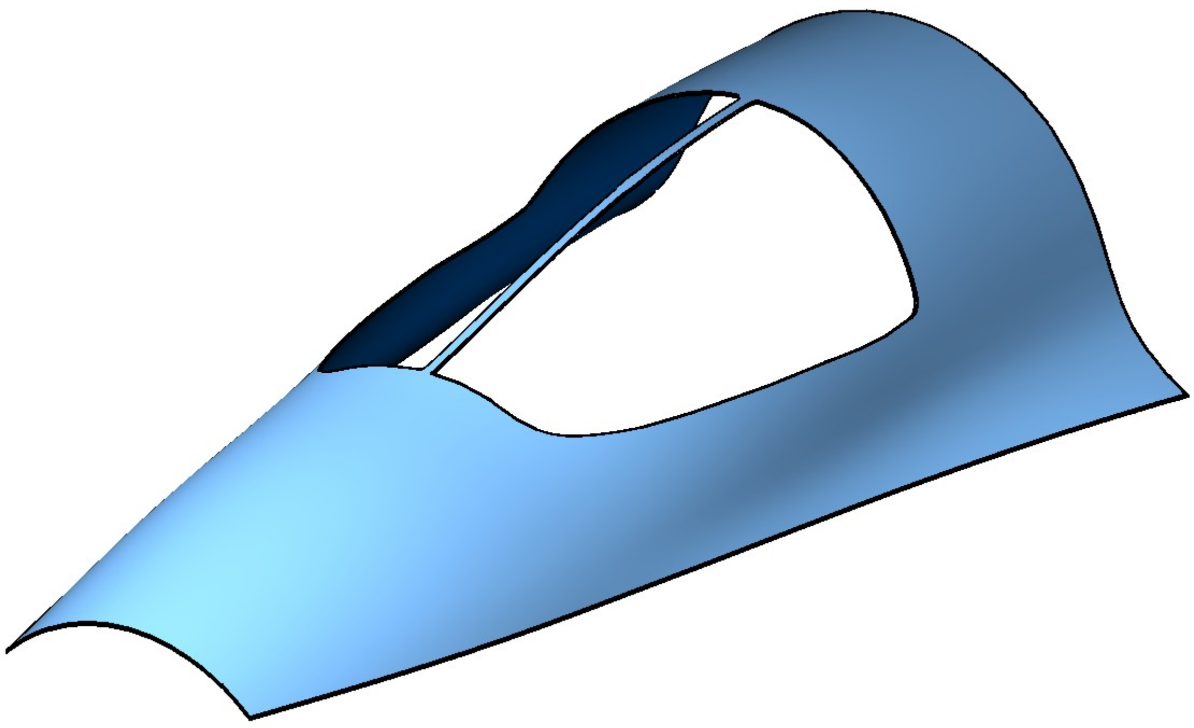


Figure O.5.27.2.

Each curve in **curves** container describes one closed surface border. Each curve in **curves** container is a contour on MbContourOnSurface surface. Each contour on the surface is described by **surface** that coincides with **basisSurface,** and ***contour*** 2D contour. ***contour*** is a closed 2D composite curve. ***contour*** contour segments may by any MbCurve 2D curves, except for MbContour composite curves. In general case, contour derivatives have discontinuities by length and direction in the points where the segments join. Two-dimensional contours describe borders of Ω definition area of MbCurveBoundedSurface surface. Boundary contours of **curves** container meet the following conditions: they do not intersect themselves and each other, the first **curves**[0] contour of the container describes the external border and it contains all other contours that describe internal cutouts in the surface. Internal contours can't be nested. For quick location of a 2D point relative to the surface parameter definition area, boundary contours are oriented so that if you move along the border, then the surface is always on the left side if we look opposite to surface normal. So, external contour is oriented so that movement along the border is executed counter-clockwise when gaze direction is opposite to its normal, and inner contours are oriented in the opposite direction.

MbCurveBoundedSurface surface with arbitrary borders can't be used as **basisSurface** base surface. If you need to construct a surface with arbitrary borders based on other surface with arbitrary borders, then you should use a base surface of the latter.

O.6. SPECIAL OBJECTS

Scalar functions that are similar to curves in one-dimensional space are special objects. Special objects are used for specific purposes, for example, in order to describe the change of fillet surface radius as a function of one surface parameter. In two-dimensional space, multiline and area are special objects. Contour with breaks was created to work with multilines based on a two-dimensional contour. In three-dimensional space, special objects describe base points of other objects, threads, extension lines, unevenness and other symbols.

O.6.1. MbFunction Function

MbFunction class is declared in function.h file.
MbFunction is an abstract class, it is an inheritor of MbRefItem and TapeBase classes, see Figure O.6.1.1.

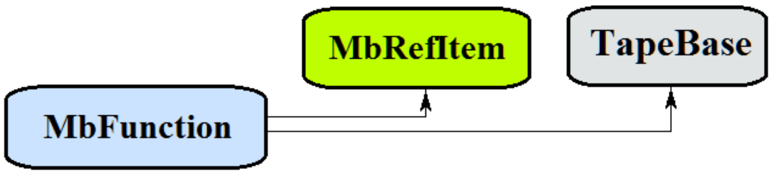


Fig. O.6.1.1.

In C3D geometric kernel, the following scalar functions are realized that are inheritors of MbFunction class:
MbConstFunction – constant function
MbLineFunction – linear function
MbCubicFunction – cubic Hermite function,
MbCubicSplineFunction – cubic spline function.
MbCharacterFunction is an analytic function.
MbFunction is

$$function(t) = f(t) \text{ scalar function}$$

of t scalar parameter that takes values in [$t_{\min}$, $t_{\max}$] range. Function parameter range is [$t_{\min}$, $t_{\max}$] range in one-dimensional space. $f(t)$ should be one-valued continuous function.

$t_{\min}$ and $t_{\max}$ limit values of parameter range are received using double **GetTMin**() and double **GetTMax**() function methods respectively.

The function is referred as periodic if there is $p>0$ such that $f(t \pm kp)=f(t)$, where k *is* an integer. **IsClosed**() method returns "true" for periodic function. double **GetPeriod**() method for periodic function or method of for function that can be extended and made periodic returns period 9p). Periodic function parameter range is always limited by one period.
Double **Value**(double & t) is the main method of the function.
This method returns function value for specified parameter (t).
double **FirstDer**(double & t),
double **SecondDer**(double & t),
double **ThirdDer**(double & t) methods
return respectively the first, second and third function derivatives for specified t parameter. These methods adjust function parameter if it goes beyond the range. If t parameter goes beyond [$t_{\min}$, $t_{\max}$] range, then non-periodic function moves t parameter to the nearest $t_{\min}$ or $t_{\max}$ limit, and the periodic function adds or subtracts the required number of periods.
double **_Value**(double t)
method returns function value for specified t parameter both inside and outside the parameter range. In the

general case, a non-periodic function is extended outside of the parameter range along the tangent in the end point. Analytic functions are exceptions. Periodic functions are extended cyclically outside of the parameter range.
double **_FirstDer**(double *t*),
double **_SecondDer**(double *t*),
double **_ThirdDer**(double *t*)
methods return respectively the first, second and third derivatives of function for specified *t* parameter both inside and outside of the parameter range.

The functions reload the following methods:
the methods that permit to copy, check objects for coincidence, check whether it's possible to make objects coinciding and make them coinciding:
MbFunction & **Duplicate**(),
bool **IsSame**(const MbFunction & **item**),
bool **IsSimilar**(const MbFunction & **item**),
bool **SetEqual**(const MbFunction & **item**),
the method that returns a type from function enumeration,
MbeFunctionType **IsA**(),
the methods that provide access to object internal data and to edit them,
MbProperty & **CreateProperty**(MbePrompt name),
void **GetProperties**(MbProperties & *properties*),
void **SetProperties**(MbProperties & *properties*).

As a rule, all functions have no kinks. MbAnaliticalFunction function defined by user should be continuous and it should not have critical points.

O.6.2. MbConstFunction Constant Function

MbConctFunction class is declared in func_const_function.h file.
MbConctFunction constant function is described by one value of *value* function.
double **Value**(double & *t*) method uses

$$f(t) = value \text{ function.}$$

Function parameter range is within $0 \le t \le 1$ range. The function can't be periodic.

O.6.3. MbLineFunction Linear Function

MbLineFunction class is declared in func_line_function.h file.
MbLineFunction linear function is described by two function limit values (*value*1, *value*2) and parameter limit values *tmin*, *tmax*.
double **Value**(double & *t*) method uses

$$f(t) = value1\ (1-t) + value2\ t \text{ function.}$$

Function parameter range is within $tmin \le t \le tmax$ range. The function can't be periodic.

O.6.4. MbCubicFunction Cubic Hermite Function

MbCubicFunction class is declared in cur_cubic_function.h file.
MbCubicFunction cubic Hermite function is described by *valueList* set of control points, *firctList* set of function derivatives in control points, *tList* set of function parameter values in control points and *closed*

function periodicity sign. There are some other function parameters that are not mandatory, they are used to speed up function methods.

If *tList*[*i*], *i*=0,1,...,*splinesCount*, where *splinesCount*=*tList*.MaxIndex(), then MbCubicFunction cubic Hermite function goes through *valueList*[*i*] control point and has *firctList*[*i*] derivative in it. The function is constructed on base of *splinesCount* smoothly joined third-order Hermite splines. Each Hermite cubic spline describes a function segment between two neighboring control points. Each Hermite cubic spline is defined by two extreme points and two curve derivatives in these points.

To calculate the function, we first use the value of *t* parameter to find the number of the working segment (the number of Hermite cubic spline) *i* from *tList*[*i*]≤*t*≤*tList*[*i*+1]. The function is calculated for the found working segment using its *w local parameter* that is determined from *tList*[*i*] and *tList*[*i*+1].

double **Value**(double & *t*) method uses

$$f(t) = (1 - 3w^2 + 2w^3)\text{valueList}[i] + (3w^2 + 2w^3)\text{valueList}[i+1] +$$
$$+ ((w - 2w^2 + w^3)\text{valueList}[i] + (-w^2 + w^3)\text{valueList}[i+1])(\text{tList}[i+1] - \text{tList}[i]) \text{ function,}$$

where $w = \dfrac{t - \text{tList}[i]}{\text{tList}[i+1] - \text{tList}[i]}$ is the local parameter of *tList*[*i*]≤*t*≤*tList*[*i*+1] working segment. Cubic Hermite function is shown in Figure O.6.4.1.

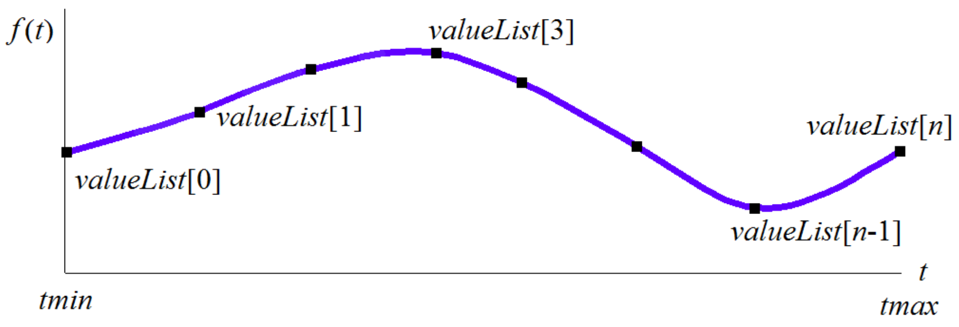


Fig. O.6.4.1.

Function parameter range is within *tmin*≤*t*≤*tmax* range, where *tmin*=*tList*[0], *tmax*=*tList*[*splinesCount*]. The function may be periodic.

Function shape depends on location of control points, function derivatives in control points, as well as on *tList* set of parameter values in control points. When a function is constructed using control points only, *firctList*[*i*] derivatives are calculated by constructing a parabola that passes through three adjacent points *valueList*[*i*–1], *valueList*[*i*], *valueList*[*i*+1] for *tList*[*i*–1], *tList*[*i*], *tList*[*i*+1], then parabola derivative is calculated in the midpoint.

O.6.5. MbCubicSplineFunction Cubic Spline Function

MbCubicSplineFunction class is declared in cur_cubic_spline_function.h file.

MbCubicSplineFunction cubic spline function is described by *valueList* set of control points, *secondList* set of function second derivatives in control points, *tList* set of function parameter values in control points and *closed* function periodicity sign. There are some other function parameters that are not mandatory, they are used to speed up function methods.

If *tList*[*i*], *i*=0,1,...,*splinesCount*, where *splinesCount*=*tList*.MaxIndex(), then cubic function goes through *valueList*[*i*] control point and has *secondList*[*i*] second derivative in it.

To calculate the function, we first use *t* parameter value to find *i* number of the working segment from *tList*[*i*]≤*t*≤*tList*[*i*+1]. The function is calculated for the found working segment using its *w local parameter* that is determined from *tList*[*i*] and *tList*[*i*+1].

double **Value**(double & *t*) method uses

$$r(t) = (1-w)\text{valueList}[i] + w\text{valueList}[i+1] +$$
$$+ ((-2w + 3w^2 - w^3)\text{valueList}[i] + (-w + w^3)\text{valueList}[i+1])\frac{(\text{tList}[i+1] - \text{tList}[i])^2}{6}$$

function,where $w = \dfrac{t - \text{tList}[i]}{\text{tList}[i+1] - \text{tList}[i]}$ is the local parameter of *tList*[*i*]≤*t*≤*tList*[*i*+1] working segment. Cubic spline function is shown in Figure O.6.5.1.

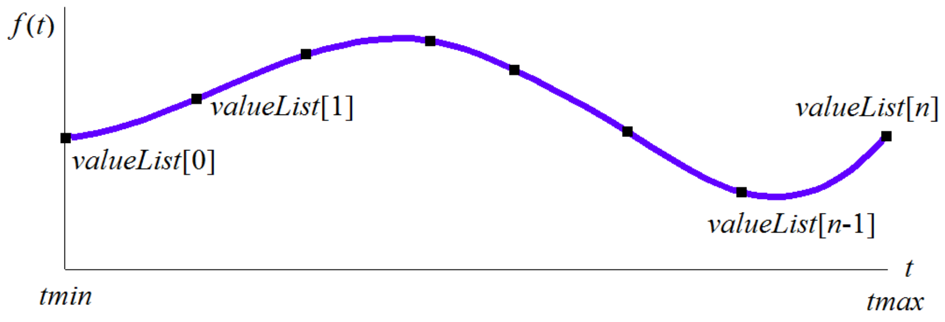


Fig. O.6.5.1.

Function parameter range is within *tmin*≤*t*≤*tmax* range, where *tmin*=*tList*[0], *tmax*=*tList*[*splinesCount*]. The function may be periodic.

Function shape depends on location of control points and on *tList* set of parameter values in control points. When the function is constructed using the control points, *secondList*[*i*] derivatives are calculated from the fact that the second derivatives vary linearly between the control points.

O.6.6. MbCharacterFunction Character Function

MbCharacterFunction class is declared in func_analytical_function.h file.

MdCharacterFunction character function is described by function string expression, action tree used to calculate the expression, *tmin*, *tmax* parameter limits and *sense* direction. There are some other function parameters that are not mandatory, they are used to speed up function methods.

Character function permits to describe any function of *t* parameter as a string expression that contains analytical functions and arithmetic operations.

double **Value**(double & *t*) method uses an action tree to calculate string expression values.

Function parameter range is within *tmin*≤*t*≤*tmax* range. The function may be periodic.

O.6.7. MbMultiline Multiline

MbMultiline class is declared in multiline.h file.

MbMultiline multiline is an inheritor of MbPlaneItem class, see Figure O.6.7.1.

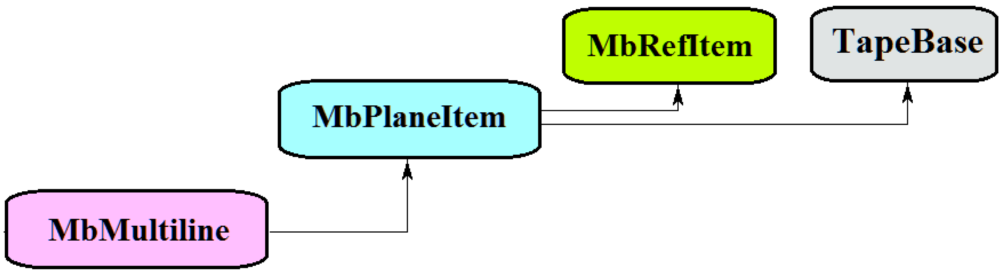


Fig. O.6.7.1.

A multiline is described by ***basisCurve*** base contour, ***vertices*** set of vertices, *equidRadii* set of radii, begTipParams multiline start edge parameters, endTipParams multiline end edge parameters, *processClosed* periodicity processing sign, *isTransparent* transparency sign, ***curves*** set of curves, ***tipCurves*** set of end edges in multiline vertices, ***begTipCurves*** end edge in multiline initial vertex, ***endTipCurves*** end edge in multiline end vertex.

A multiline is a contour that has thickness. Contour thickness is variable. Contour edges and connection points of contour segments may have various forms.

A multiline is shown in Figure O.6.7.2.

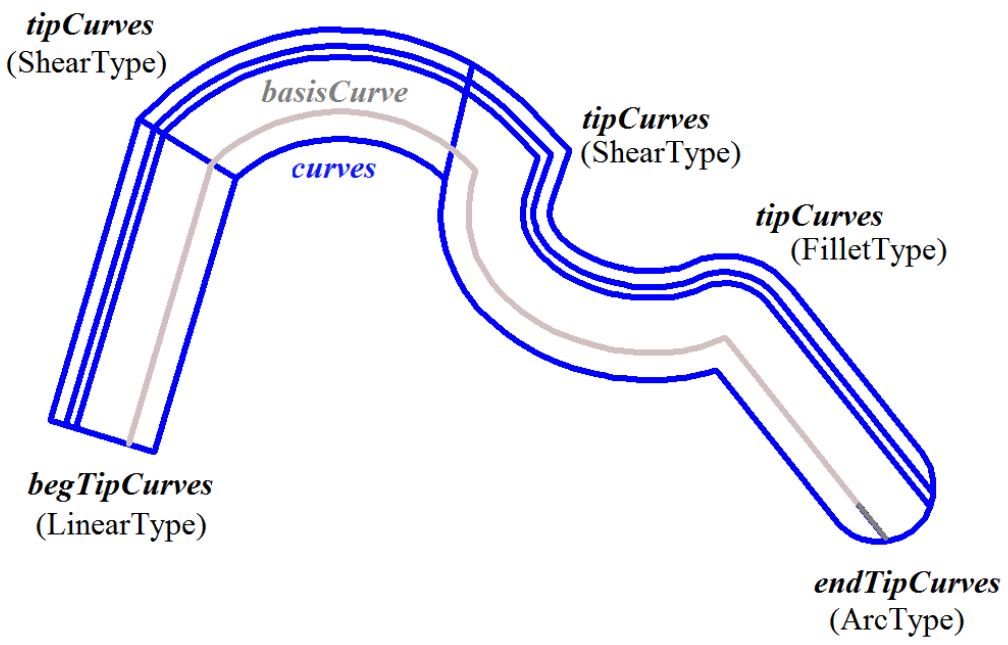


Fig. O.6.7.2.

A multiline is used to exchange data with other systems.

O.6.8. MbContourWithBreaks Two-Dimensional Contour with Breaks

MbContourWithBreaks class is declared in cur_contour_with_breaks.h file.

Two-dimensional contour with breaks is an inheritor of MbContour contour, see Figure O.6.8.1.

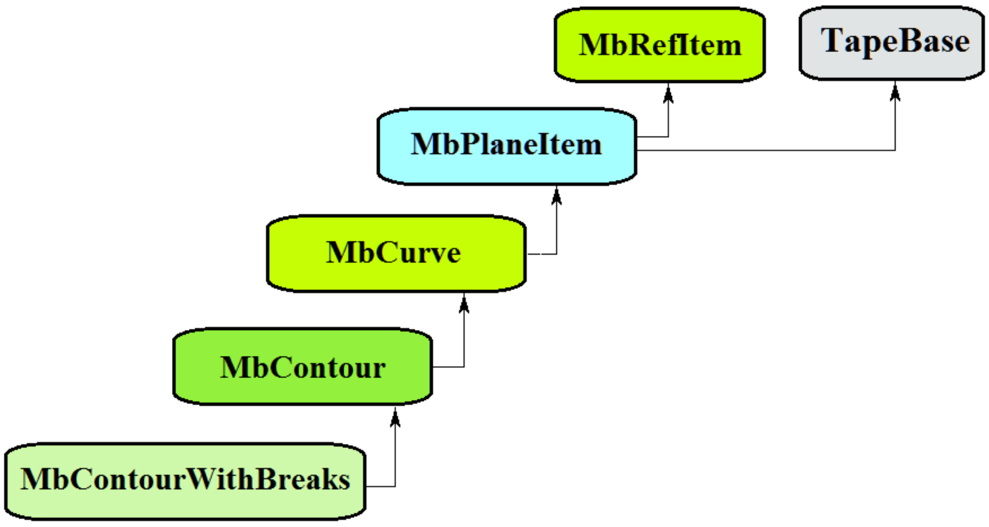


Fig. O.6.8.1.

Two-dimensional contour with breaks is described by ***segments*** set of sequentially joined curves, *closed* curve periodicity sign, ***breaks***, ***visibleContours*** set of contour visible sections , and *baseSegNumbers* set of contour segment numbers used to define fixed break points.

A contour with breaks is shown in Figure O.6.8.2.

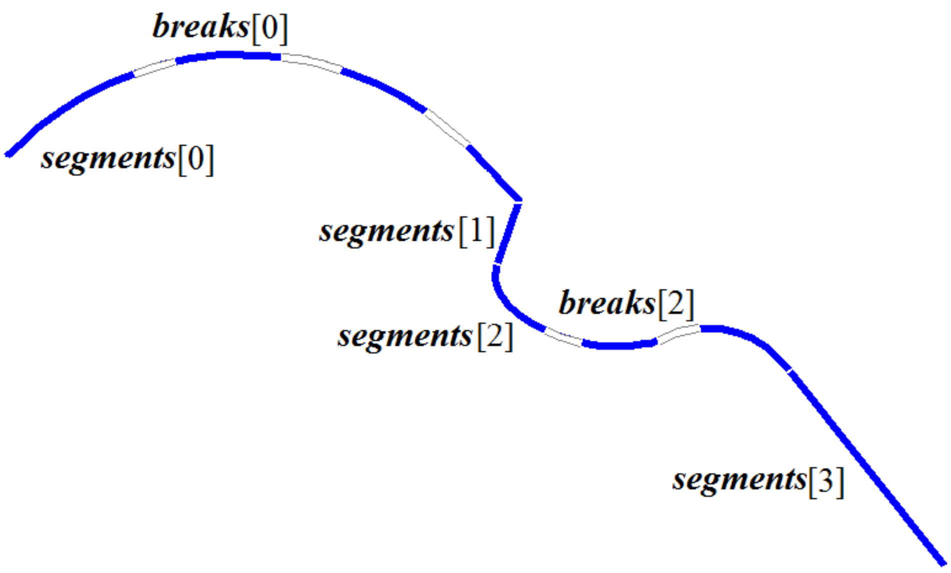


Fig. O.6.8.2.

A two-dimensional contour shouldn't be used as a segment of other 2D contour segments with breaks. Two-dimensional contour with breaks is used to construct multilines only.

O.6.9. MbRegion Region

MbRegion class is declared in region.h file.

MbRegion region is an inheritor of MbPlaneItem class, see Figure O.6.9.1.

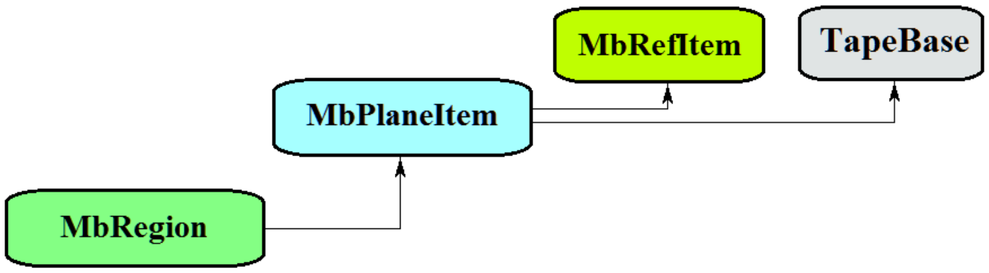


Fig. O.6.9.1.

A region is described by ***contours*** set of contours. Region contours are periodic and they don't cross each other and themselves. One contour in a set is an external contour, and all other contours are located inside it, they are internal contours. The first contour in ***contours*** set is always external.

A region is an interconnected set of points in 2D space, its boundaries are described by 2D periodic contours. Contours of the region are oriented so that when one moves along any contour, circumscribed set of points is located to the left from the contour. That is, external contour of the region is oriented counter-clockwise, and internal contours are oriented clockwise. A region is shown in Figure O.6.9.2.

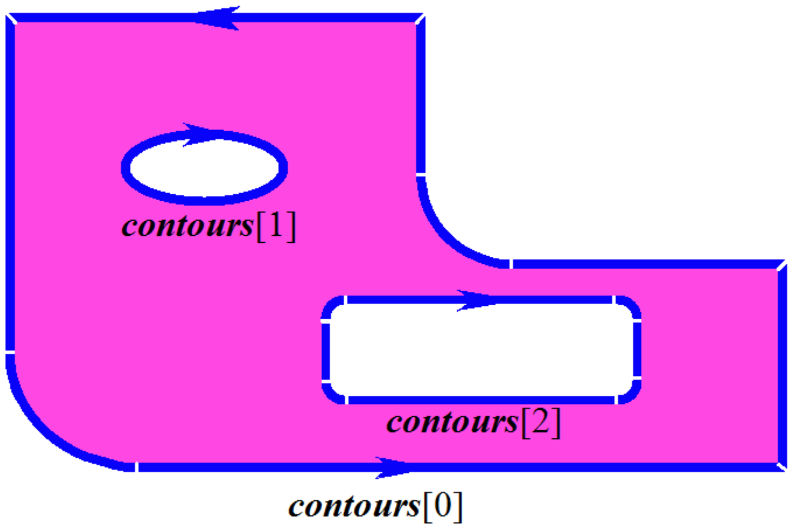


Fig. O.6.9.2.

Regions are used to describe 2D interconnected areas. Boolean operations can be performed on regions.

O.6.10. MbLegend Auxiliary Geometric Object

MbLegend class is declared in legend.h file.

MbLegend auxiliary geometric object is an inheritor of MbSpaceItem class, see Figure O.6.10.1.

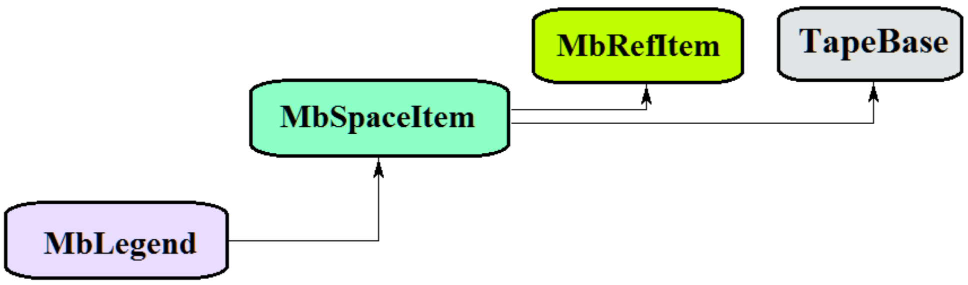


Fig. O.6.10.1.

An auxiliary geometric object is an abstract class. The following topological objects are inhertors of MbLegend class in C3D geometric kernel:
MbMarker — a point and two orthonormal vectors,
MbThread — a thread,
MbPointsSymbol— a symbol in base points,
MbRough— surface finish symbol,
MbLeader— a leader line.

Auxiliary objects are used for various purposes, however, all of them all interact with curves, surfaces and objects of the geometric model.

Auxiliary objects reload the following methods of 3D geometrical object:
the methods that serve geometrical object transformation,
Move(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
Transform(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),
the methods that permit to copy, check objects for coincidence, check whether it's possible to make objects coinciding and make them coinciding:
MbSpaceItem& **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbSpaceItem & **item**),
bool **IsSimilar**(const MbSpaceItem & **item**),
bool **SetEqual**(const MbSpaceItem & **item**),
the methods that return a type from enumeration of geometric objects,
MbeSpaceType **IsA**(),
MbeSpaceType **Type**(),
MbeSpaceType **Family**(),
the methods that provide access to object internal data and to edit them,
MbProperty & **CreateProperty**(MbePrompt name),
GetProperties(MbProperties & *properties*),
SetProperties(MbProperties & *properties*),
the method that fills up a polygonal copy of the geometrical object,
CalculateWire(double sag, MbMesh & **mesh**).

O.6.11. MbMarker Marker

MbMarker class is declared in marker.h file.

MbMarker auxiliary object is described by **origin** point and two orthogonal vectors (**axisZ**, **axisX**).

A marker is used to set restrictions on geometric objects in 3D space. A marker is a representative of the geometric object, it can replace a 3D point, a line, a plane, local coordinate system and other objects during work with geometric constraints.

O.6.12. MbThread Thread Graphic Symbol

MbThread class is declared in mb_thread.h file.

MbThread thread graphic symbol is described by **place** thread local coordinate system, *radObj* initial radius of thread in the surface, *radThr* initial radius of thread in the solid, *length* thread length, *angle* thread cone angle, **name** thread name, and **solids** set of solids. There are some other parameters of this object that are not mandatory and that are used to speed up object methods.

The axis of a threaded joint goes along **place.axisZ** axis. A name is used to identify thread graphic symbol among flat projections of solid faces from **solids** set. Thread graphic symbol is shown in Figure O.6.12.1.

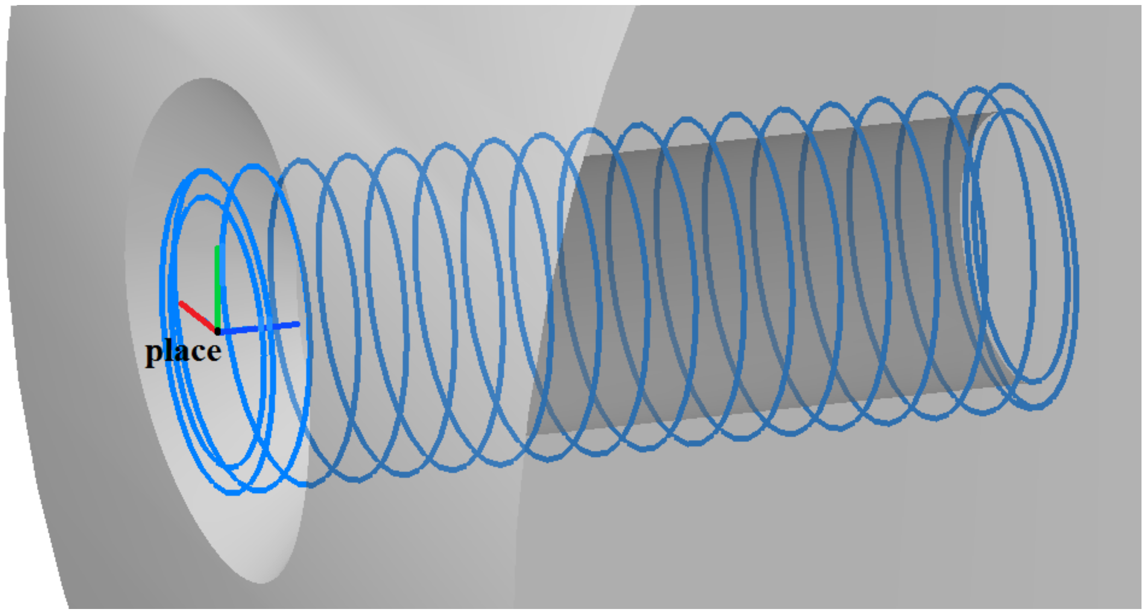


Fig. O.6.12.1.

Thread graphic symbol describes a threaded joint element of the geometric model, it is used to construct flat projections of threaded joints.

O.6.13. MbPointsSymbol Symbol

MbPointsSymbol class is declared in mb_symbol.h file.

MbPointsSymbol symbol is an inheritor of MbSymbol class. MbPointsSymbol is described by **points** set of identifiers and *steps* parameters that contain data on complex cut lines.

The object contains data on base points of the symbols associated with elements of the geometric model.

The object is used to construct flat projections of symbols, for which it is sufficient to know the location of the base points. It defines the base points of symbols in the elements of the geometric model.

O.6.14. MbRough Surface Finish Symbol

MbRough surface finish symbol.

MbRough class is declared in mb_rough.h file.

MbRough surface finish symbol is an inheritor of MbPointsSymbol class. Surface finish symbol is described by **points** set of 3D points, *steps* data on complex cut sections, and **item** topological binding object.

The object contains data on the base points of surface finish symbol that are associated with **item** topological object.

The object is used to construct flat projections of surface finish symbol for **item** geometric model element.

O.6.15. MbLeader Leader Line Symbol

MbLeader class is declared in mb_rough.h file.

MbLeader leader line symbol is an inheritor of MbSymbol class. Leader line symbol is described by a set of identifiers and **branches** set of surface finish symbols.

The object contains data on leader line used to show surface finish for topological objects.

The object is used to construct flat projections of surface finish leader line symbol for geometric model elements.

O.7. TOPOLOGICAL OBJECTS

Geometric properties that don't depend on quantitative characteristics (lengths and angles) and that reflect continuous relationship of object elements and its environment are called topological properties. Topological objects of C3D geometric kernel also describe geometrical properties of the object that depend on quantitative characteristics, as well as geometrical properties that reflect continuous relationship of the object with the neighboring elements. Topological objects are constructed based on surfaces, curves and points by adding to their data, properties, and methods new data, properties, and methods that reflect connections of the object with its environment.

O.7.1. MbTopologyItem Topological Object

MbTopologyItem class is declared in topology_item.h file.

Unlike other topological objects, MbTopologyItem has **name**, *changed* change sign and *label*. MbTopologyItem is an inheritor of MbTopItem topological object, see Figure O.7.1.1.

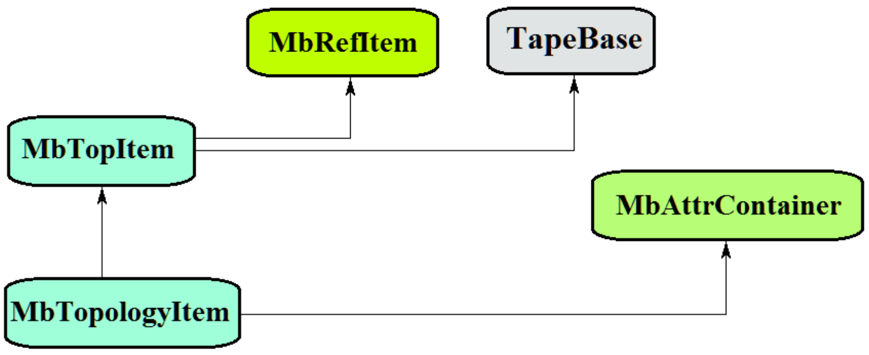


Fig. O.7.1.1

MbAttrContainer container provides attributes for named topological objects.

The following topological objects that inherit MbTopologyItem class are implemented in C3D geometric kernel:
MbFace – a face,
MbEdge – an edge,
MbVertex – a vertex.

MbEdge edge has MbCurveEdge an inheritor: a face that joins edges.

A named topological object has the following methods:
methods used to transform the topological object:
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),
methods used to work with topological object name:
MbName & **GetName**(),
SimpleName & **GetMainName**(),
SimpleName & **GetFirstName**(),
a method that returns type from enumeration of topological objects,
MbeTopologyType **IsA**().

Named topological objects are used as elements to construct objects of geometric model.

O.7.2. MbFace Face

MbFace class is declared in topology.h file.

MbFace face is an inheritor of MbTopologyItem topological object; it is a limited section of a surface that was assigned normal direction and has defined borders. We'll call the side of surface and face, gaze direction to which is directed opposite to the normal "outer side"; we'll call the other side "inner side". The sides of MbSurface surface are not equal relative to the normal, since a surface always has an outer side and an inner side. Unlike surface, a face permits to assign normal direction, and hence to assign outer side and inner side.

Data set of the face includes a pointer to MbSurface* **surface**, **sameSense** coincidence sign of face normal direction and surface normal direction, as well as RPArray<MbLoop> **loops** set of face cycles. A face has some other parameters that are not mandatory, they are used to speed up face methods.

A pointer to a surface can't be zero. Normal to face and normal to surface coincidence sign takes "true" value if the normals coincide, otherwise the sign takes "false" value. We'll call the side of face "outer side" if gaze direction to it opposite is to normal direction; we'll call the other side "inner side".

Face cycles describe face borders. Each face border is closed. Each cycle is described by MbOrientedEdge sequence order of edges along the border. The number of face cycles is equal to the number of face borders at the surface. One face border is the outer border, it contains the borders of internal cutouts. The first cycle in the container cycle describes the outer border of the face and it contains inner cycles that describe inner face borders. Outer face cycle is oriented counterclockwise, and inner cycles are oriented clockwise when the gaze direction is opposite to face normal. Thus, when we move along the outer side of face cycle, the face is always on the left side. In Fig. O.7.2.1, arrows indicate the directions of face cycles and face normal.

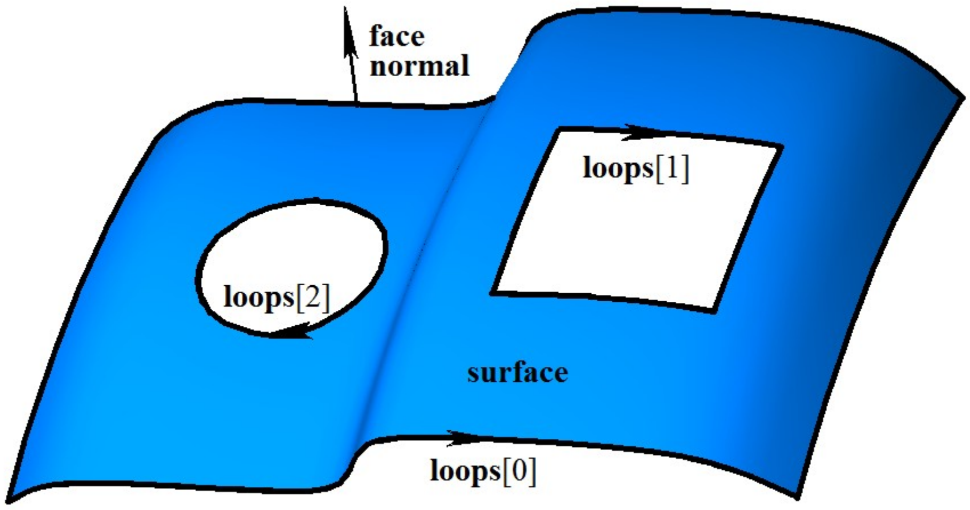


Fig. O.7.2.1.

Cycles of one face should not cross each other and themselves.

The face can be named and it can have attributes. A name can be used to identify the face, and attributes can provide additional face data, such as color, transparency, origin, etc.

A face is used for both solid-state and hybrid simulation.

O.7.3. MbEdge Edge

MbEdge class is declared in topology.h file.

MbEdge is an inheritor of MbTopologyItem topological object; it is a curve with assigned direction. Direction of MbCurve3D curve is strictly related to its parameter increase direction. Unlike a curve, an edge may be directed either in curve parameter increase direction or curve parameter decrease direction. An edge always starts and ends in some MbVertex vertex.

Edge data set contains MbCurve3D* **curve** pointer to curve, *sameSense* edge direction and curve direction coincidence sign, MbVertex* **begVertex** pointer to start vertex, and MbVertex* **endVertex** pointer to end vertex. In Fig. O.7.3.1, you can see an edge.

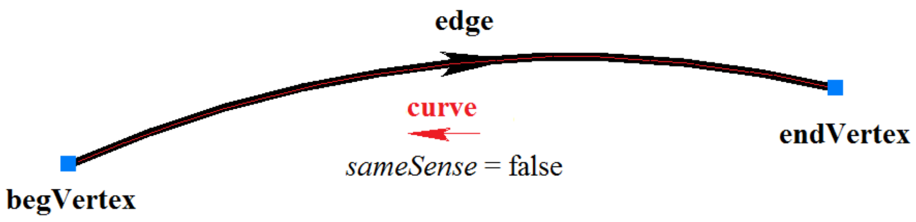


Fig. O.7.3.1.

A pointers to curve or vertex can't be zero. Coincidence sign of edge curve directions takes "true" value, if edge direction and curve direction coincide, if edge direction and curve direction are opposite then it takes "false" value. If an edge begins and ends in the same vertex then such edge is closed one. Pointers to start vertex and end vertex of a closed edge are equal.

An edge can be named and it can have attributes. A name can be used to identify the edge, attributes can provide additional data on edge, e.g., color, display style, origin, etc.

Edges are used for wireframe simulation.

O.7.4. MbVertex Vertex

MbVertex class is declared in topology.h file.

MbVertex is an inheritor of MbTopologyItem topological object; it is a point with known location error. Vertex data set includes MbCartPoint **point** and *tolerance* location error for this point.

A vertex can describe single wireframe point or edge junction point. Any number of edges can meet in a vertex. Meeting edges point to the same shared vertex. In Fig. O.7.4.1, you can see a vertex that is the meeting point of three edges.

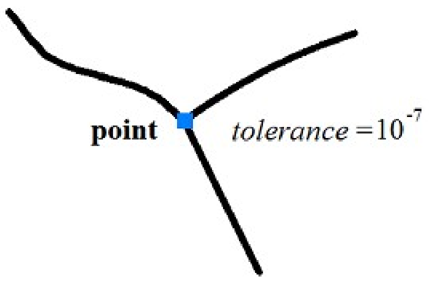


Fig. O.7.4.1.

If edges are joined inaccurately, then vertex location error is the distance from the vertex point to the most remote edge. In Fig. O.7.4.2, you can see a tolerant vertex, where four edges meet.

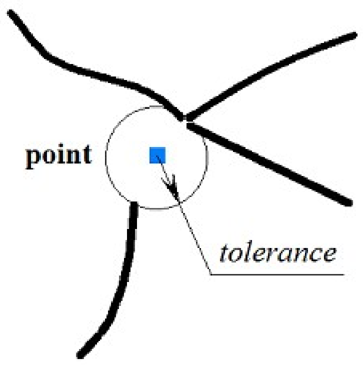


Fig. O.7.4.2.

A vertex can be named and it can have attributes. A name can be used to identify the vertex, and attributes can provide additional data on the vertex, e.g. color, display style, origin, etc.

Vertices are used in all simulation methods.

O.7.5. MbCurveEdge Face Edge

MbCurveEdge class is declared in topology.h file.

MbCurveEdge is an inheritor of MbEdge edge; it is an edge constructed on MbSurfaceIntersectionCurve surface intersection curve. MbCurveEdge edge is designed to describe a segment of face border. Unlike MbEdge edge, MbCurveEdge describes not just a curve, but a segment joining two faces or a segment of face edge.

Data set of face edge includes a pointer to MbSurfaceIntersectionCurve * **curve** surface intersection curve, *sameSense* edge direction and curve direction coincidence sign, MbVertex* **begVertex** pointer to start vertex, MbVertex* **endVertex** pointer to end vertex, MbFace* **facePlus** pointer to a face located to the left from the edge, and MbFace* **faceMinus** pointer to a face located to the right from the edge. In Fig. O.7.5.1, you can see an edge joining two faces.

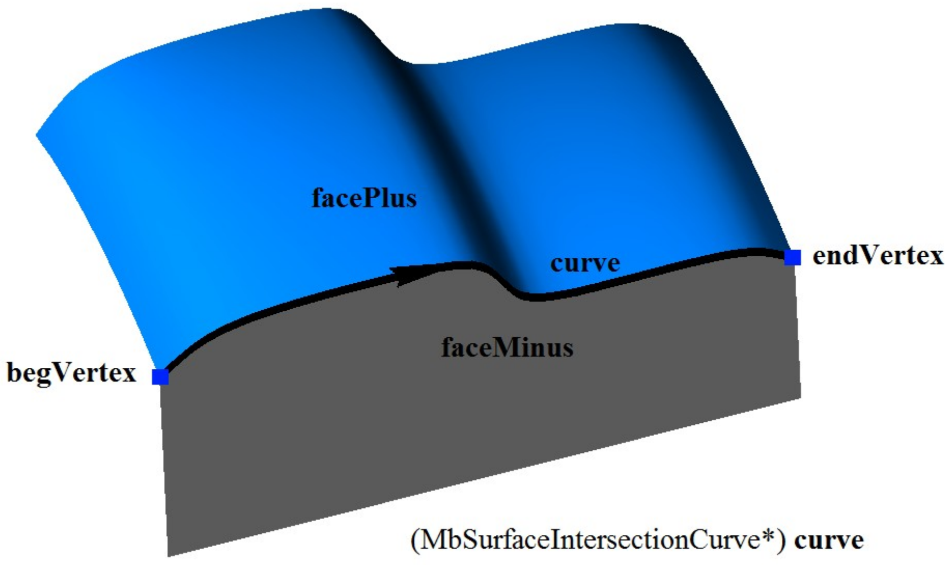


Fig. O.7.5.1.

Face edge may describe a segment that joins two different faces. In this case, pointers to faces located to the left and to the right from the edge are not zero and they are not equal to each other. Surfaces located in

data structures of faces connected by an edge coincide with surfaces located in edge curve data set:
facePlus->surface->GetSurface() == curve->curveOne.suface and
faceMinus->surface->GetSurface() == curve->curveTwo.suface
or
facePlus->surface->GetSurface() == curve->curveTwo.suface and
faceMinus->surface->GetSurface() == curve->curveOne.suface.

If a face is closed by one or both face surface parameters, there are border segments where the face meets with itself. Such edge is a seam. In this case, pointers to faces located to the left and to the right from the edge are equal to each other, see Figure O.7.5.2.

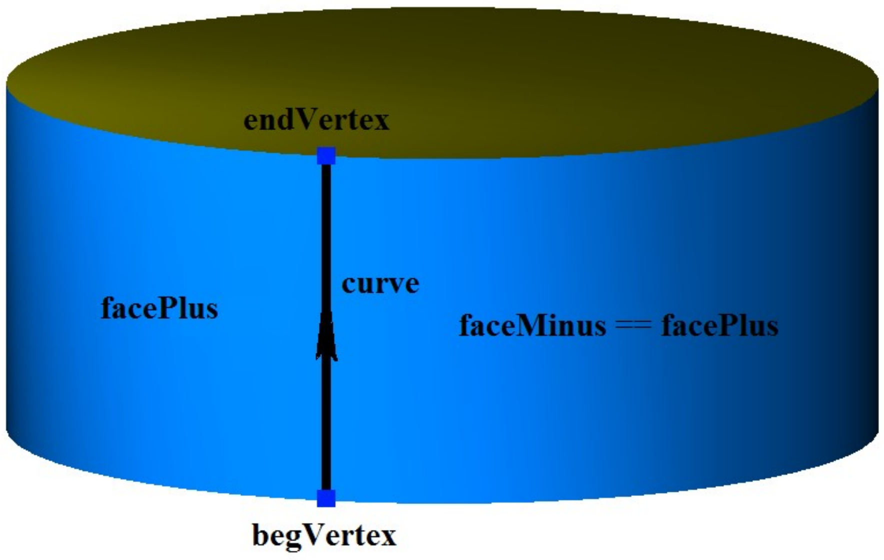


Fig. O.7.5.2.

Face edge can describe a segment of face border. Such edge is a boundary one. In this case, pointer to the face located to the left or to the right of the edge is equal to zero, see Figure O.7.5.3.

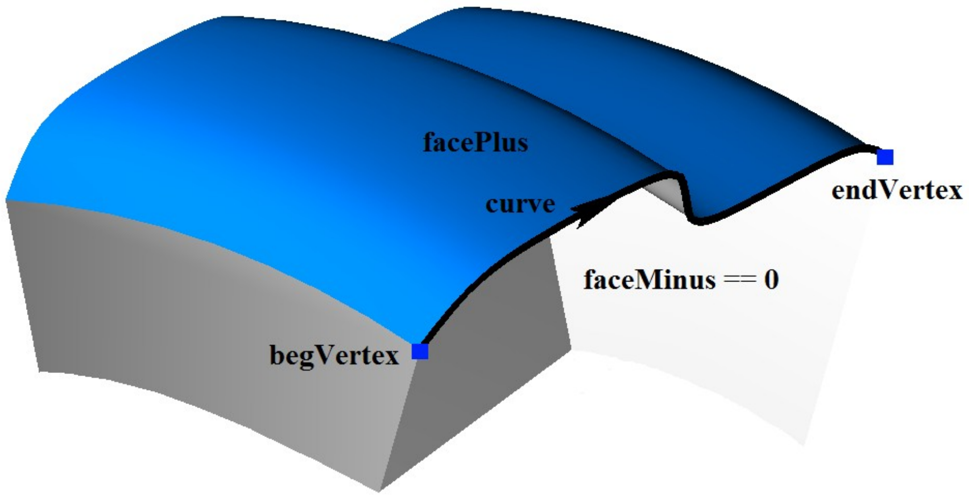


Fig. O.7.5.3.

Edge having zero length is a polar one and it describes face pole. Pole edge is not a boundary edge, as the face of polar edge has no border. As a rule, a polar edge is located in specific face surface points. Some curve corresponds to a polar segment in parameters area of face surface. Pointers to start and end vertices of a polar edge are equal. Polar edge is shown in Figure O.7.5.4.

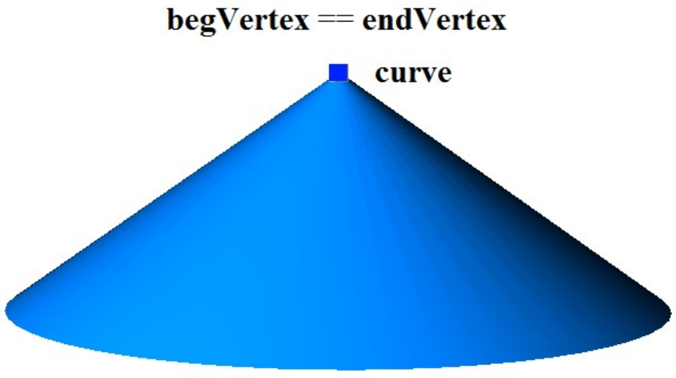


Fig. O.7.5.4.

As for the polar edge, a pointer to the face located to the left or to the right from the edge is equal to zero. Intersectional curve of polar edge has the following data:
curve->curveOne.suface == **curve->curveTwo.suface**, and
curve->curveOne.curve segment is a copy of the **curve->curveTwo.curve** segment.

A face edge can be named and it can have attributes. A name can be used to identify the edge, and attributes can provide additional data, such as color, style, origin, etc.

Edge face is used for solid-state and hybrid simulation.

O.7.6. MbLoop Face Cycle

MbLoop class is declared in topology.h file.

MbLoop face cycle is an inheritor of MbTopItem topological object and it describes a sequence of edges that completely fill some face border.

Cycle data set includes RPArray<MbOrientedEdge> **edgeList** set of oriented edges in their order along the face border. Cycle has some other parameters that are not mandatory and that are used to speed up cycle methods.

Directions of oriented edges and cycle direction coincide. End point of each cycle oriented edge is joined with the start point of the next oriented edge, see Figure O.7.6.1.

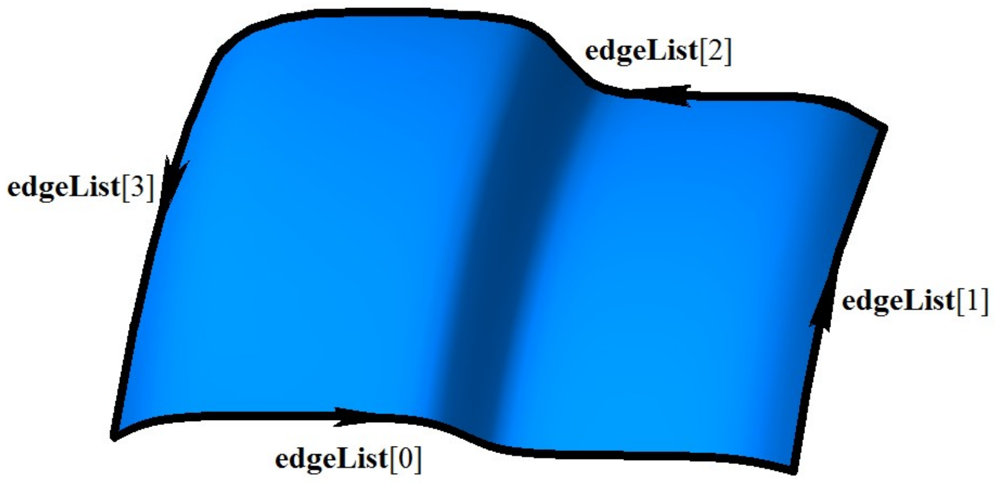


Fig. O.7.6.1.

End point of the last oriented cycle edge is joined with the start point of the first oriented edge.

In Fig. O.7.6.2, you can see a cycle of a spherical face that consists of four oriented edges, two of these edges are constructed on seam edge, and two others are pole edges.

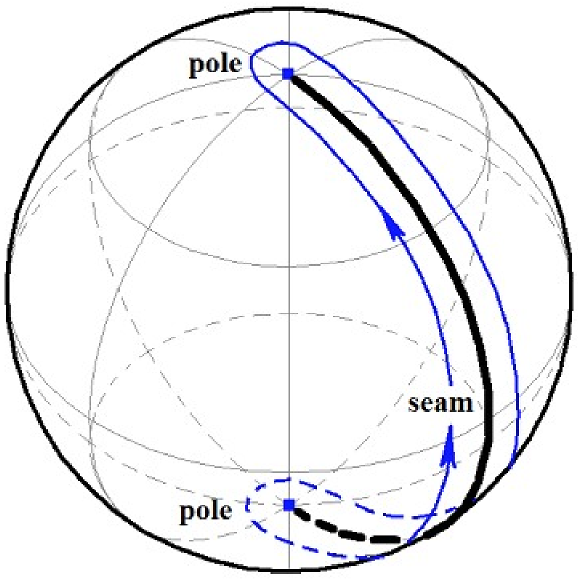


Fig. O.7.6.2.

In sphere parameter face, spherical face cycle will be a rectangular quadrangle.

A cycle is always closed just like a face border. A cycle is directed so that a face is always to the left when we move along the cycle from face outside side.

O.7.7. MbOrientedEdge Oriented Face Edge

MbOrientedEdge class is declared in topology.h file.

MbOrientedEdge oriented face edge is an inheritor of MbTopItem topological object and it describes face border segment. Data structure of oriented edge contains MbCurveEdge* **curveEdge** face edge coincidence sign of face edge direction with oriented edge direction or face cycle (in this case direction is defined by *orientation* parameter).

If **curveEdge** face edge direction coincides with cycle direction, then *orientation*==true holds for the corresponding oriented edge of this face. If **curveEdge** face edge direction does not coincide with cycle direction, then *orientation*==false holds for corresponding oriented edge of this face. In Fig. O.7.7.1, you can see two oriented edges constructed on the same **curveEdge** face edge.

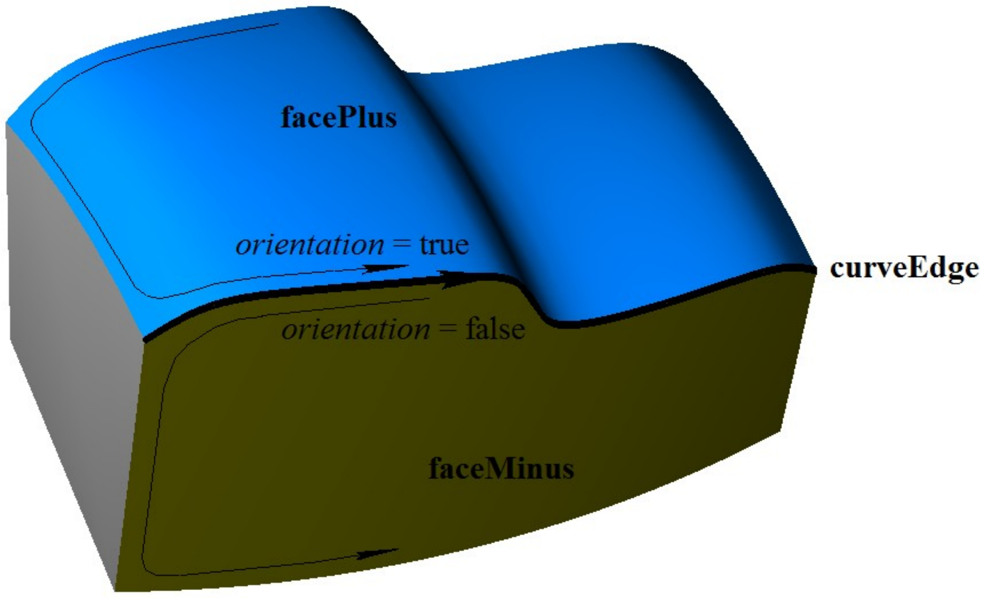


Fig. O.7.7.1.

In general, MbCurveEdge face edge is included in two cycles that belong to faces joined by this edge. Face edge is included in one cycle with *orientation*==true parameter; this face is located to the left from the edge, and **facePlus** data field points to this face. As for other cycle, face edge is included in it with *orientation*==false orientation sign; this face is located to the right from the edge, and **faceMinus** data field points to this face. Thus, adjacent faces and edges joining them are interrelated.

O.7.8. MbFaceShell Set of Faces

MbFaceShell class is declared in topology_faceset.h file.

MbFaceShell set of faces is an inheritor of MbTopItem topological object. MbFaceShell data set includes RPArray<MbFace> **faceSet** set of faces and *closed* closure sign for a set of faces. There are some other data for a set of faces that are not mandatory and that are used to speed up the methods for a set of faces.

Usually a set of faces describes an interconnected segment of simulated object surface. Interconnected faces that belong to a set of faces meet the following conditions: the faces are joined by shared edges, each edge joins only two faces so that the outer side of one face transfers to the outer side of another face, so the shared surface of the faces does not intersect itself.

In Fig. O.7.8.1, you can see a set of interconnected faces. All edges belong to two cycles, all **facePlus** and **faceMinus** edge pointers are not zero, and MbSurfaceIntersectionCurve curves that were used to construct face edges meet the following condition: **curveOne.surface**!=**curveTwo.surface**.

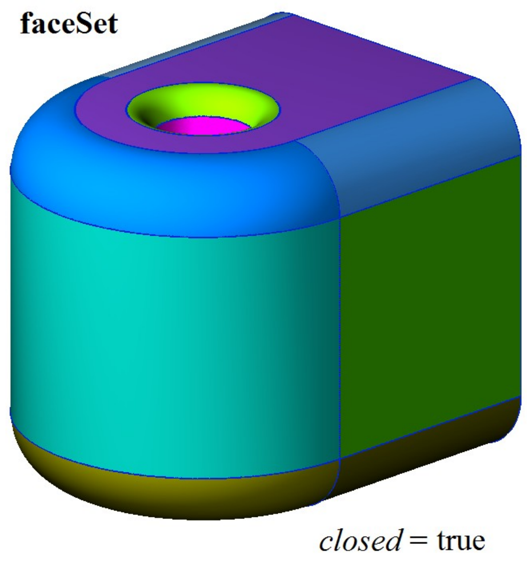


Fig. O.7.8.1.

Faces shown in Figure O.7.8.1 form a shared closed surface, the outer side of each face transfers to the outer side of the adjacent face. *closed* sign has "true" value for a set of faces that have no boundary edges.

If at least one face in the set has at least one boundary edge, then *closed* sign of the set of faces takes "false" value. In Fig. O.7.8.2, you can see a set of faces, some faces in the set have boundary edges. A boundary edge is included in one cycle, and only one of **facePlus** or **faceMinus** pointers of face boundary edge is not equal to zero, for MbSurfaceIntersectionCurve curves that were used to construct edges **curveOne.surface==curveTwo.surface** and *buildType parameter =cbt_Boundary*. A polar edge is not a boundary edge, as it does not form an edge.

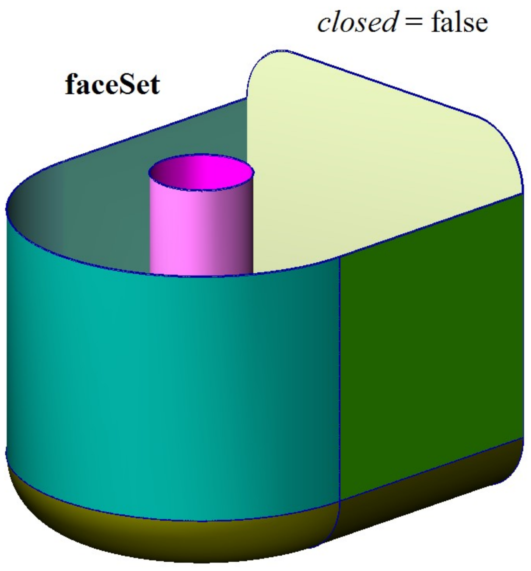


Fig. O.7.8.2.

Interconnected set is called a *closed shell* if the faces of the set have no borders. If connected faces have at least one boundary edge, then such interconnected set is called *open shell*. Please note that closed shell and open shell make a connected set of faces joined with each other.

Set of faces may consist of several not interconnected parts. In Fig. O.7.8.3, you can see a set of faces describing two open shells.

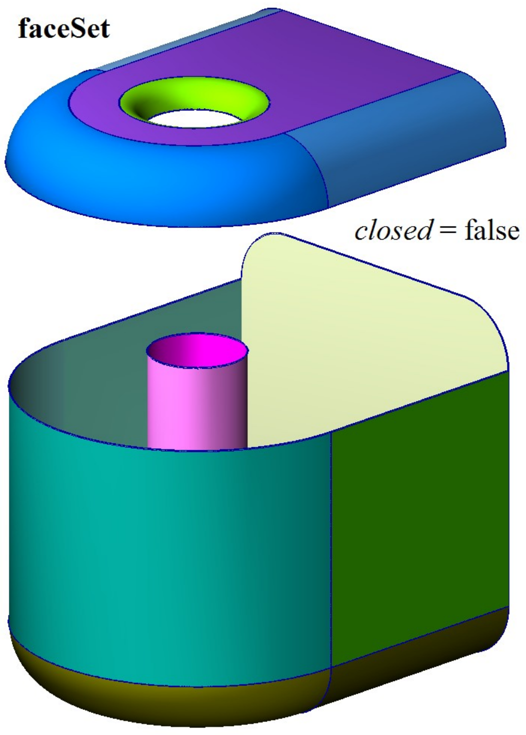


Fig. O.7.8.3.

Formally, there are no restrictions for a set of faces. A set may contain separate faces. In Fig. O.7.8.4, you can see a set of separate faces. Each face shown in Figure O.7.8.4 forms a separate shell, all face edges are boundary edges.

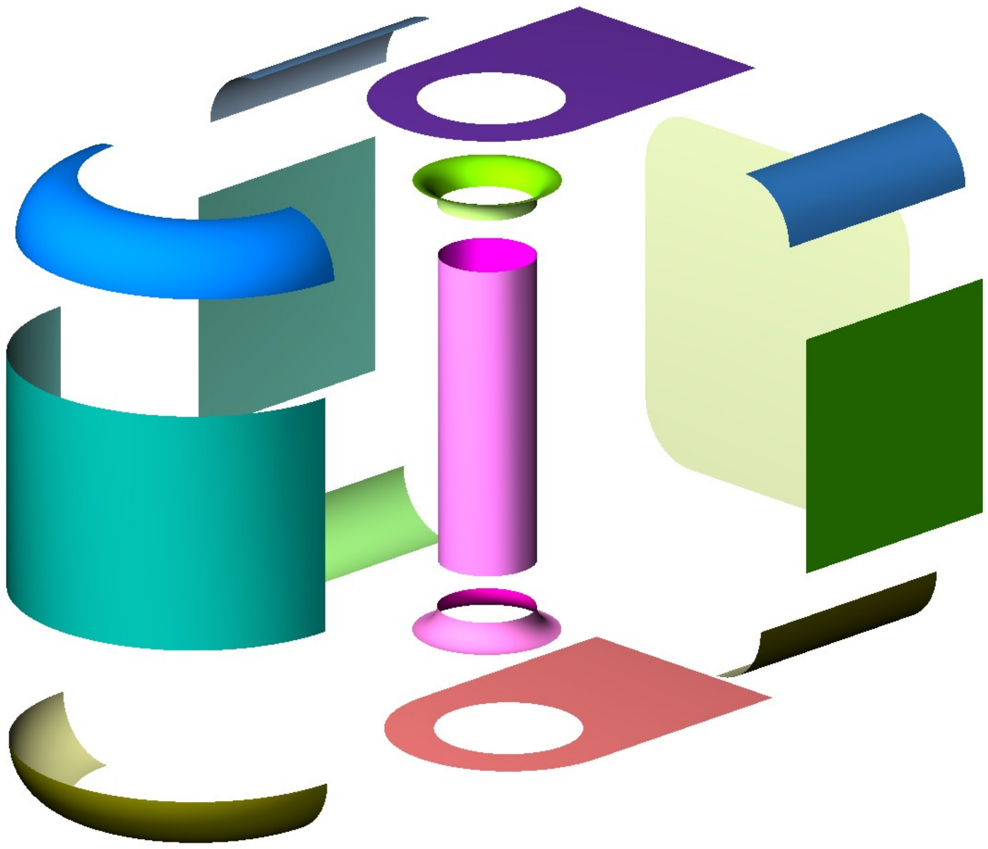


Fig. O.7.8.4.

The main shell methods are methods of its transformation in space:
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),
faces, edges and vertices search methods, as well as methods used to determine the location of a point with respect to a closed shell:
bool **DistanceToBound**(const MbCartPoint3D &,...),
bool **PointClassification**(const MbCartPoint3D &,...).

An open shell covers only a part of boundary surface of the simulated object. Open shells are used to simulate a surface. If we add to a closed shell a set of its inner points, then we'll receive a *solid body*. Closed shells are used to simulate solid solids. Faces are cut and joined when a shell is constructed. C3D geometric kernel methods provide this. Closed and open shells are used to construct objects in a geometric model.

O.7.9. Copying a Set of Faces

Each construction method that uses a set of faces represented as MbFaceShell or MbSolid modifies some vertices, edges and faces of original objects. In order to speed up construction and keep the original set of faces, MbFaceShell object data are copied completely or partly. C3D geometric kernel uses four methods to copy MbFaceShell set of faces; these methods are defined in MbeCopyMode enumeration. As a rule, construction methods together with modified set of faces transfer MbeCopyMode type parameter that controls transmission of faces, edges and vertices from the original object to the constructed object.

MbeCopyMode enumeration is declared in mb_enum.h file. MbeCopyMode type parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*.

If MbeCopyMode = *cm_Copy* then the original set of faces of modified object is fully copied, so a new object and the original one will not have shared surfaces, curves, faces, edges, vertices and other objects. In this case, the new object and the original one will not be interrelated.

If MbeCopyMode = *cm_KeepSurface*, then the new object and the original one will have the same base surfaces of faces. This option is used when high construction speed is required.

If MbeCopyMode = *cm_KeepHistory*, then the new object and the original one will have the same vertices, base surfaces of faces and faces that were not modified by construction or other action. This option is used to provide the lowest memory usage.

If MbeCopyMode = *cm_Same*, then all required data of the original object will be moved to the newly constructed object, so the original object should be deleted after construction. This option is used when the original object wouldn't be required and it was constructed specifically for this construction.

MbeCopyMode enumeration is included in the method used to copy MbFaceShell* set of faces: MbFaceShell::**Copy**(MbeCopyMode, MbShellHistory*). This method is used in solid construct operations with input parameters containing other solids.

For enumeration value *cm_Copy*, the original set of faces and its copy don't have shared data. The option when the original set of faces and its copy have shared base surfaces corresponds to *cm_KeepSurface* enumeration value. The option when the initial set of faces and its copy have the same base surfaces, vertices and faces not modified by any operation corresponds to *cm_KeepHistory* enumeration value. For this purpose, **Copy**(...) method uses a pointer to MbShellHistory object that stores correspondence between the original set of faces and its copy. After the operation, a copy of the set of faces is transferred by a parameter in MbShellHistory::**SetOrigins**(MbFaceShell&) method to replace unchanged faces in the copy with the initial faces from the original set of faces. The option when the set of faces is not copied in **Copy**(...) method corresponds to *cm_Same* enumeration value. One should note that if the operation fails, then the original set of faces will be modified.

MbFaceShell* **Duplicate**(MbRegDuplicate* iReg) method can be used to copy MbFaceShell set of faces. MbRegDuplicate object is used to save in the copy the structure of reciprocal links available in the original set of faces. The copied set of faces and its copy will not be connected.

O.7.10. Naming of Faces, Edges and Vertices

Faces, edges and vertices have MbName name in a data set. MbName class is declared in name_item.h file. Faces, edges and vertices are named by C3D geometric kernel during construction of MbFaceShellset of faces in all shape-generating operations. Parameters of each shape-generating operation contain MbSNameMaker object. MbSNameMaker object contains the main operation name and a belonging to the SimpleName type container of simple names that are used to name faces. MbSNameMaker object names newly constructed faces using the container of simple names. Face names will be unique if the elements of the container of simple names are unique. Edges are named by hashing the names of the faces joined by them. Vertices are named by hashing the names of edges joined by them. In addition, MbSNameMaker object contains construction method version and ensures storage of old construction methods when they are modified during development of the geometric kernel.

O.8. OBJECTS OF GEOMETRIC MODEL

A class of geometrical model objects belongs to the class of three-dimensional geometric objects. For example, the geometric model has an object called "solid" that is used for solid, surface and direct simulation. Solids are also constructed when objects made of sheet metal are simulated. Besides a solid, objects of geometric model include wire frame, point frame and polygonal object. Assemblies can be constructed using geometric model objects. Such geometric model objects as local coordinate system can be used for auxiliary constructions. In addition, geometric model provides an object to construct sketches.

O.8.1. MbItem Geometric Model Object

MbItem class is declared in model_item.h file.

MbItem object of geometric model is an inheritor of MbSpaceItem, MbTransactions and MbAttributeContainer classes. C3D geometric kernel works with geometric model objects shown in Figure O.8.1.1.

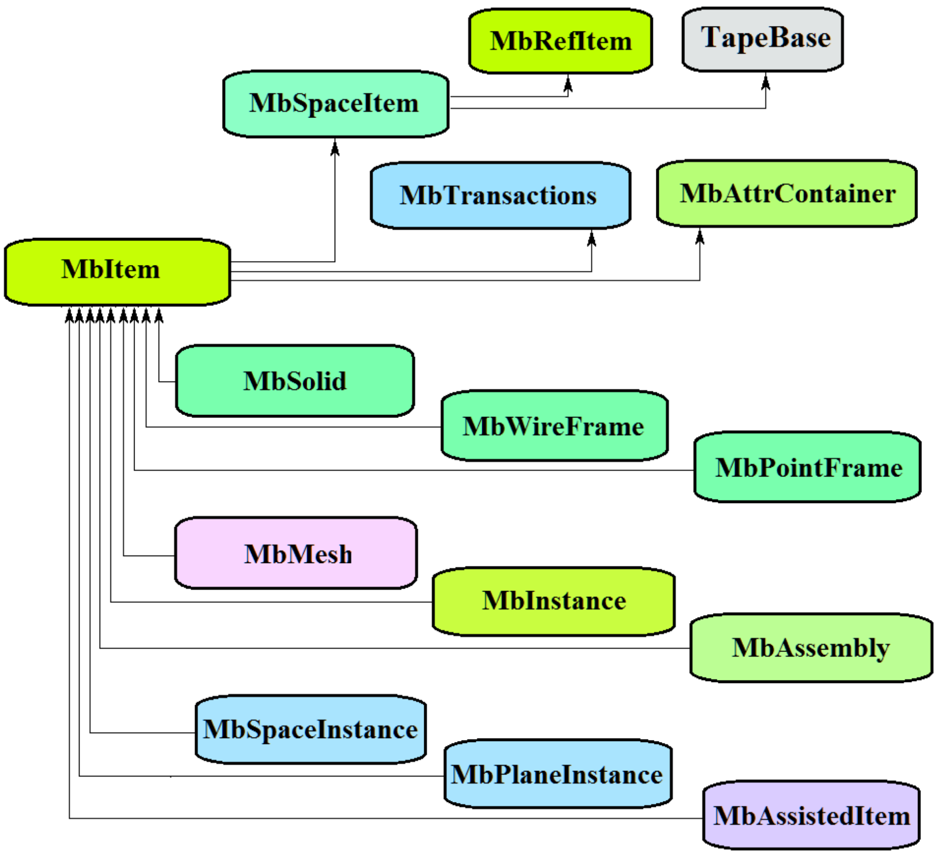


Figure O.8.1.1.

MbTransactions construction log contains the data required to construct the object and it permits to repeat object construction with edited parameters. MbAttrContainer container provides the attributes to geometric model objects. Thus, geometric model objects contain the following data in addition to their own specific data:
size_t *m_countRegistrable* is the number of object registrations during writing and reading,
ptrdiff_t *useCount* is the number of times the object was used by other objects,
std::vector<MbCreator *> **transactions** is the ordered set of object constructors,

std::multimap<int, MbAttribute*> **attributes** is object attributes set.

The following objects of geometric model are inheritors of MbItem class:
MbSolid – solid body,
MbWireFrame – wire frame,
MbPointFrame – point frame,
MbMesh – polygonal object,
MbInstance – insertion of geometric model object,
MbAssembly – assembly unit of geometric model objects,
MbAssistingItem – auxiliary object,
MbSpaceInstance – insertion of a 3D object,
MbPlaneInstance – insertion of a set of 2D objects.

The main methods of geometric model objects are methods providing editing and visualization of the objects.

bool **RebuildItem**(MbeCopyMode sameShell, RPArray<MbSpaceItem> * **items**) method repeats construction of the object using the construction log. This method is called after editing object internal data.

bItem * **CreateMesh**(MbeStepData data, bool wire, bool grid, MbRegDuplicate * iReg) method constructs a polygonal copy of the object. If the object is an assembly unit or an insertion, then object copy will also be an assembly unit or an insertion with polygonal objects.

bool **AddYourMesh**(MbeStepData data, bool wire, bool grid, MbMesh& **mesh**) method adds a polygonal copy into **mesh** object.

```
bool NearestMesh( MbeSpaceType sType, MbeTopologyType tType, MbePlaneType pType,
                  const MbAxis3D & axis, double maxDistance, double & t, double & dMin,
                  MbItem *& find, SimpleName & findName,
                  MbRefItem *& element, SimpleName & elementName,
                  MbPath & path, MbMatrix3D & from ) method
```

searches for the nearest **find** polygon object, its **element**, their names (findName and elementName), as well as path in assembly unit structure, and **from** transformation matrix into global coordinate system.

Objects of geometric model reload the following 3D object methods:
the methods involved in transformation of geometrical objects,
void **Move**(const MbVector3D & **v**, MbRegTransform * iReg = NULL),
void **Rotate**(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),
void **Transform**(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),
the methods that permit to copy, check for coinciding objects, check whether it's possible to make objects coinciding and make them coinciding:
MbSpaceItem & **Duplicate**(MbRegDuplicate * iReg = NULL),
bool **IsSame**(const MbSpaceItem & **item**),
bool **IsSimilar**(const MbSpaceItem & **item**),
bool **SetEqual**(const MbSpaceItem & **item**),
the methods that return a type from enumeration of geometric objects,
MbeSpaceType **IsA**(),
MbeSpaceType **Type**(),
MbeSpaceType **Family**(),
the methods that ensure access to object internal data and their editing:
MbProperty & **CreateProperty**(MbePrompt name),
GetProperties(MbProperties & *properties*),
SetProperties(MbProperties & *properties*).

O.8.2. MbSolid Solid Body

MbSolid class is declared in solid.h file.

MbSolid solid body (or just solid) is an inheritor of MbItem class, and it is described by MbFaceShell* **outer** set of faces and *multiState* connectivity type.

A solid is a set of edges that are joined together by edges and describe the surface of the simulated object. A solid can describe one or more interconnected sets of points. *multiState* connectivity type indicates that a solid describes one or several interconnected sets of points (in the latter case, a solid can be divided into several solids).

outer set of solid faces can describe two fundamentally different sets of points, depending on the presence of boundary edges. If the set of faces has no edge then the solid describes a set of points located on surfaces of faces and on inside surfaces of these faces. Such solid is called *closed* and is described by a closed shell. A closed solid is shown in Figure O.8.2.1.

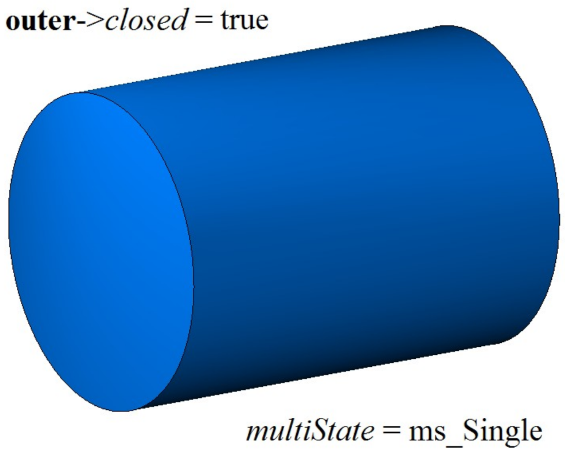


Fig. O.8.2.1.

If the set of faces has one or several edges, then the solid describes a set of points located on face surfaces of only. Such a solid is called *open* and is described by an open shell. An open solid is shown in Figure O.8.2.1.

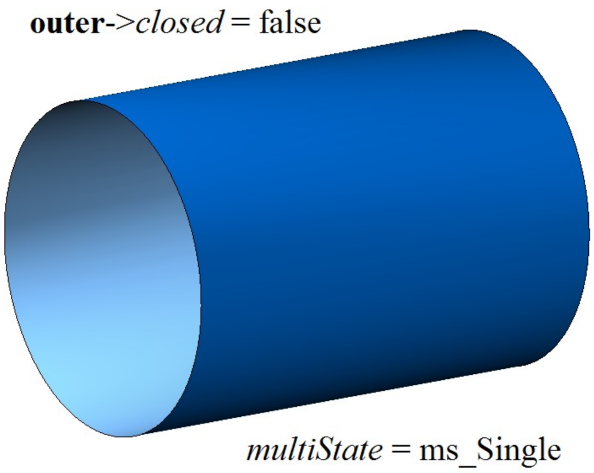


Fig. O.8.2.2.

In most cases, a closed solid is described by one interconnected set of faces, that is, by one shell. If a solid has cavities, then they are described by several interconnected sets of faces. In Figure O.8.2.3, you can see a closed solid described by two closed shells, an external shell and an internal one (the latter shell is located inside the first one).

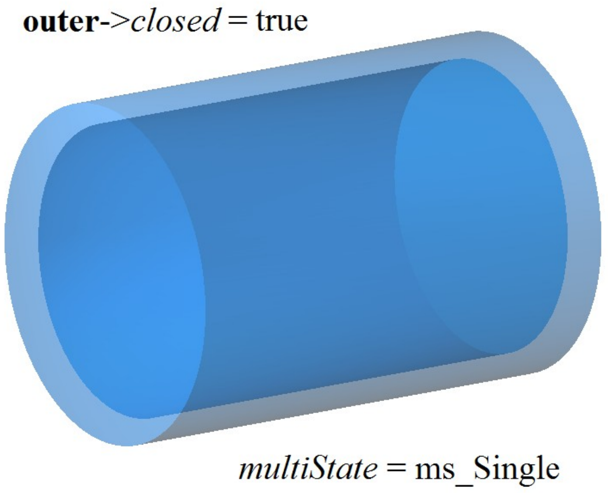


Fig. O.8.2.3.

The solid is semitransparent, so you can see a cavity inside it.

Solids support various operations (such as Boolean operations) on them, these are sets of actions that result in construction of solids having different shape. The Result of subtraction of two closed solids is shown in Fig. O.8.2.4.

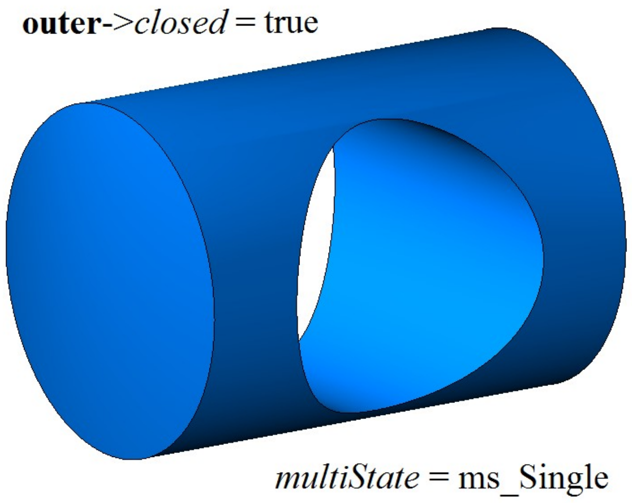


Fig. O.8.2.4.

Results of operation on a closed solid and a non-closed solid are completely different as the operations are executed on different sets of points. A result of subtraction of two closed solids is shown in Figure O.8.2.5.

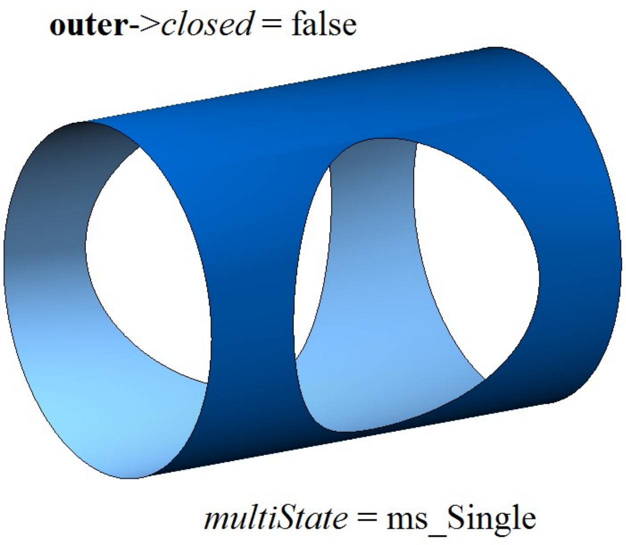


Fig. O.8.2.5.

A solid can be multiply-connected (multi-part), that is, they can consist of several separate parts. In this case, *multiState* is equal to ms_Multiple. Doubly-connected solid described by two closed shells is shown in Figure O.8.2.6.

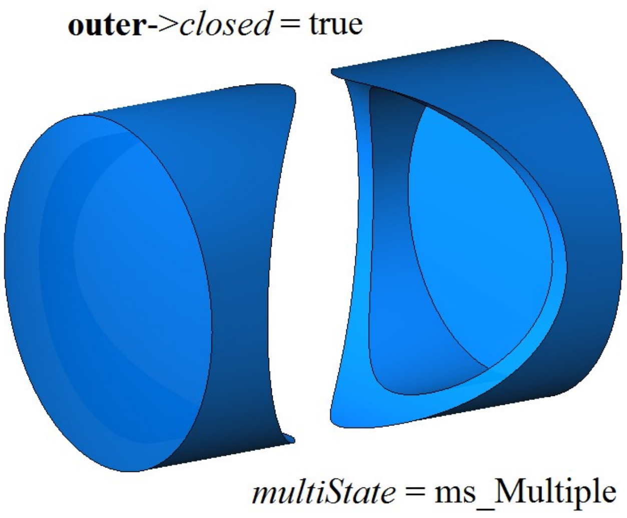


Fig. O.8.2.6.

Such a solid can be divided into two simply connected closed solids using **::DetachParts** (...) method. A solid in Figure O.8.2.6 is semitransparent. A doubly-connected solid described by two non-closed shells is shown in Figure O.8.2.7.

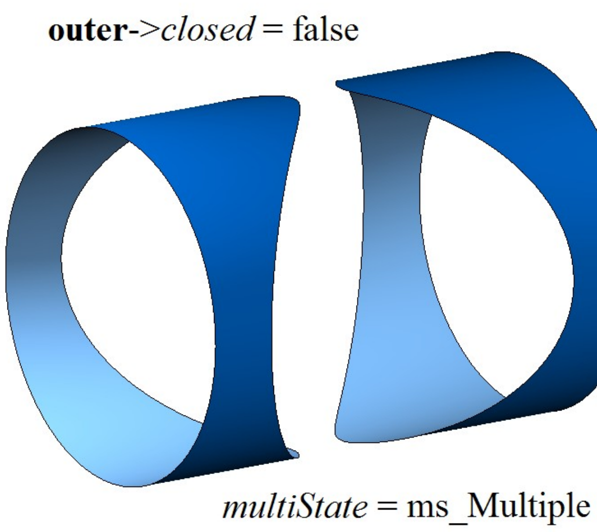


Fig. O.8.2.7.

Such a solid can be divided into two simply connected non-closed solids.

Solids are constructed using the methods provided by C3D geometric kernel. Solids with simple shapes are constructed by points, curves and surfaces. Operations permit you to construct more complex solids based on simple ones. You can edit initial solids and construct modified ones by changing the parameters in MbTransactions construction log or by directly modifying the elements of previously constructed solids. Open solids are used for surface simulation. Open solid permits you to focus on complex shapes of simulated objects.

O.8.3. MbWireFrame Wireframe

MbWireFrame class is declared in wire_frame.h file.

MbWireFrame wireframe is an inheritor of MbItem class and is described by std::vector<MbEdge*>**edges** set of edges, *parts* number of interconnected parts and *closed* boundary vertices absence sign.

A wireframe is a set of edges that are joined at vertices and describe frame structure of simulated object. A wireframe can describe one or more interconnected sets of points. *parts* number of interconnected parts indicates that wireframe describes one or several interconnected sets of points (in the latter case, a wireframe can be divided into several wire frame).

A wireframe can be *closed* or *open*, depending on the presence or absence of boundary vertices. Closed wireframe is shown in Figure O.8.3.1.

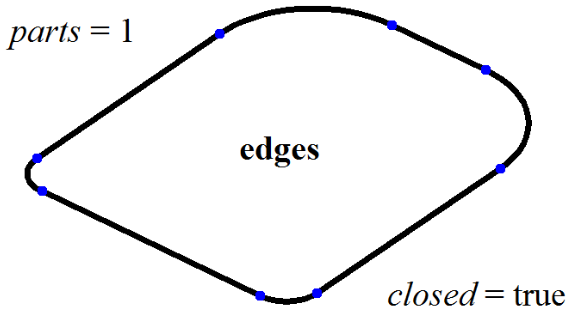


Fig. O.8.3.1.

Open wireframe is shown in Figure O.8.3.2.

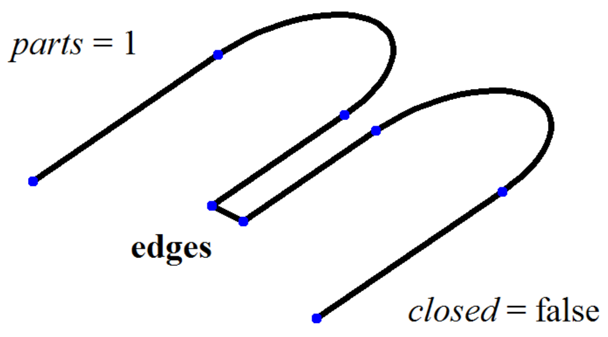


Fig. O.8.3.2.

An open wireframe consisting of two interconnected sets of edges is shown in Figure O.8.3.3.

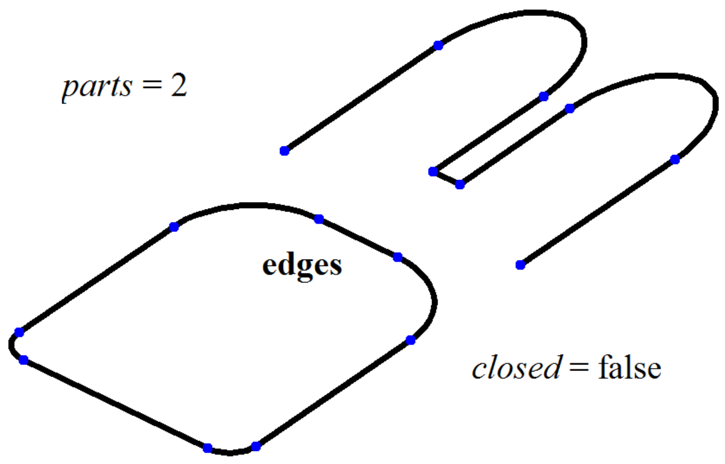


Fig. O.8.3.3.

Closed wireframe consisting of two interconnected sets of edges is shown in Figure O.8.3.4.

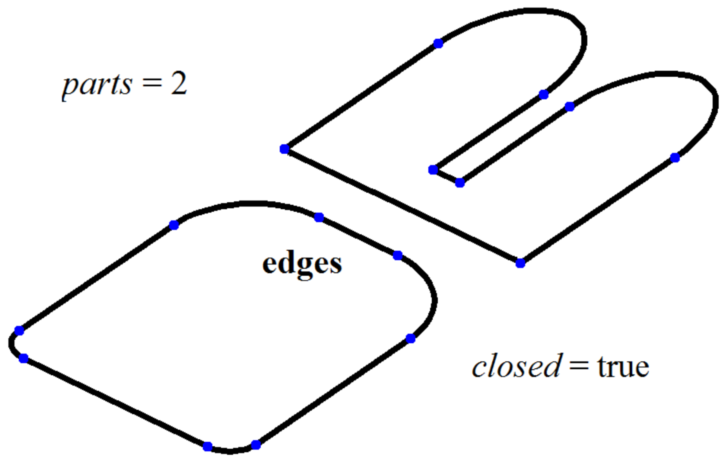


Fig. O.8.3.4.

A wireframe can be used to construct trajectories, space sketches, as well as for auxiliary constructions.

O.8.4 MbPointFrame Point Frame

MbPointFrame class is declared in point_frame.h file.

MbWireFrame point frame is an inheritor of MbItem class and it is described by std::vector<MbVertex*>**vertices** set of points presented as vertices.

Point frame is shown in Figure O.8.4.1.

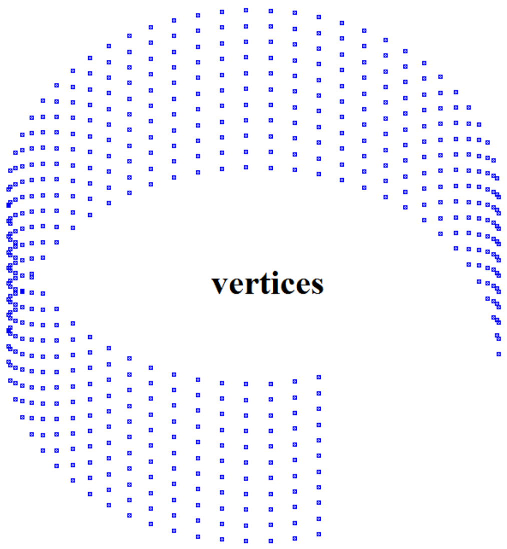


Fig. O.8.4.1.

Point frame can be used either to position other objects or for auxiliary constructions.

O.8.5. MbMesh Polygonal Object

MbMesh class is declared in mesh.h file.

MbMesh polygonal object is an inheritor of MbItem class and it is described by RPArray<MbGrid>**grids** set of triangulations, RPArray<MbPolygon3D>**wires** set of polygons, RPArray<MbApex3D>**peaks** set of apexes, MbRefItem* **item** pointer to original object, *type*, **cube** dimensional cube and *closed* closure sign.

Polygonal object is a set of triangular and quadrangular plates, polylines and individual points. One of the methods used to construct a polygonal object is associated with approximation of other geometric model objects, for example, a solid. Every *i*th solid face is approximated by **grids**[*i*] triangulation, every *j*th solid edge is approximated by **wires**[*j*] polygon, every *k*th vertex is associated with **peaks**[*k*] apex, *closed* closure sign corresponds to solid closure. One other way to construct a polygonal object is to import data, using polygon representation converter.

MbGrid triangulation is Sarray<MbFloatPoint3D>**points** set of points and Sarray<MbFloatVector3D>**normals** set of normals (the number of points is equal to the number of normals), Sarray<MbFloatPoint>**params** set of two-dimensional points of surface parametric area (the number of 2D points is equal to the number of 3D points, otherwise it is equal to zero, i. e., the set can be empty), Sarray<MbTriangle> *triangles* set of triangular plates in the form of three indices of **points** set of points, SArray<MbQuadrangle>*quadrangles* set of quadrangular plates in the form of four indices of **points** set of points. Triangulation plates approximate some surface.

MbPolygon3D polygon is an ordered set of points, their serial connection permits you to construct a polyline that approximates some curve.

MbApex3D apex is a point that contains additional data.

item pointer to the original object may be equal to zero, *type* type may be undefined.

A vector image of a polygonal object is shown in Figure O.8.5.1.

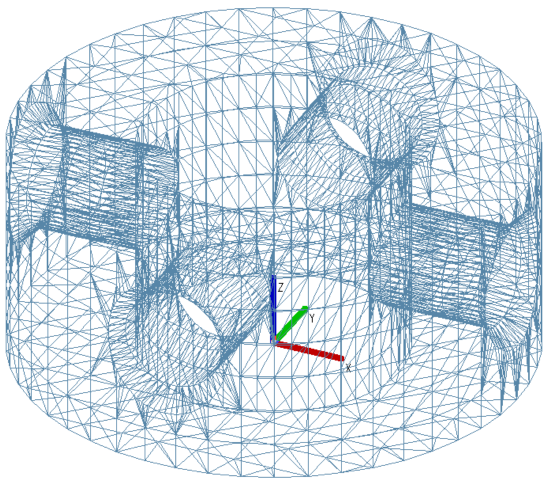


Fig. O.8.5.1.

A toned image of a polygonal object is shown in Figure O.8.5.2.

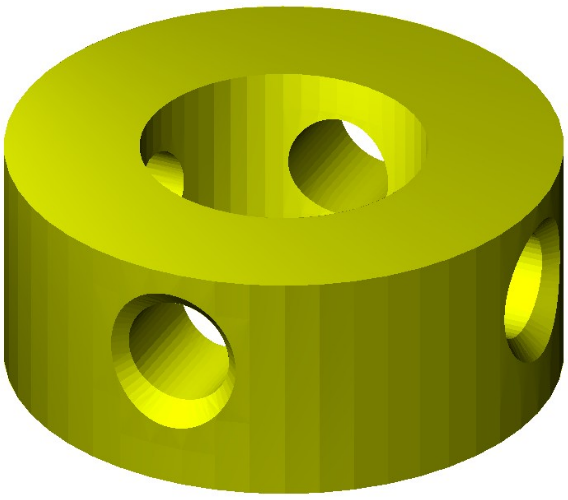


Fig. O.8.5.2.

In Figure O.8.5.2, you can see individual triangles of the object, this is due to the fact that direction of normals in each triangle is constant. If direction of normals in triangulation triangles is permanently changing, then individual object triangles become invisible, see Figure O.8.5.3.

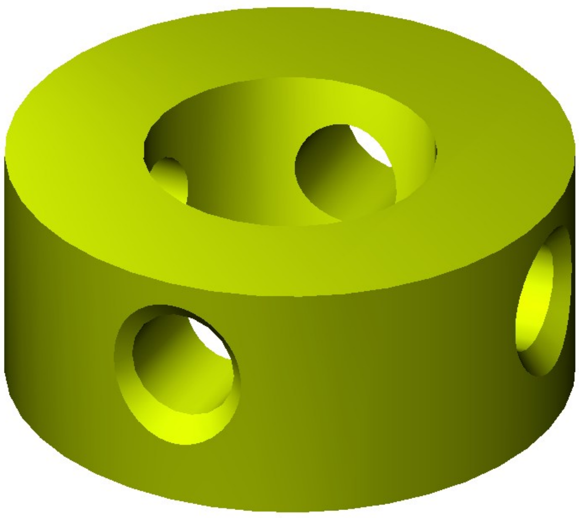


Fig. O.8.5.3.

You can use MbItem::**CreateMesh**(...) or MbItem::**AddYourMesh**(...) method to construct a polygonal object. Polygon object is used to visualize, calculate and manufacture simulated objects. The main advantages of polygons are ease of use and high speed calculations, for example, for intersection with a straight line.

O.8.6 MbInstance Insertion

MbInstance class is declared in instance.h file.

MbInstance insertion is an inheritor of MbItem class, it is described by MbItem* **item** object of geometric model and MbPlacement3D **place** local coordinate system. Insertion is **item** object that was moved into **place** local coordinate system.

Insertion has all properties of **item** object. The difference is that **Move**(...), **Rotate**(...), **Transform**(...) methods modify **place** local coordinate system not modifying **item** object.

Object insertion may contain a solid, a wireframe, a point frame and a polygonal object, but it can't contain other insertion or assembly unit.

O.8.7. MbAssembly Assembly Unit

MbAssembly class is declared in assembly.h file.

MbAssembly assembly unit or assembly is an inheritor of MbItem class and it is described by std::vector<MbItem*>**assemblyItems** set of objects of geometric model and MbPlacement3D **place** local coordinate system.

Assembly is a set of geometric model objects that can be processed as a single entity.

Assembly unit is shown in Figure O.8.7.1.

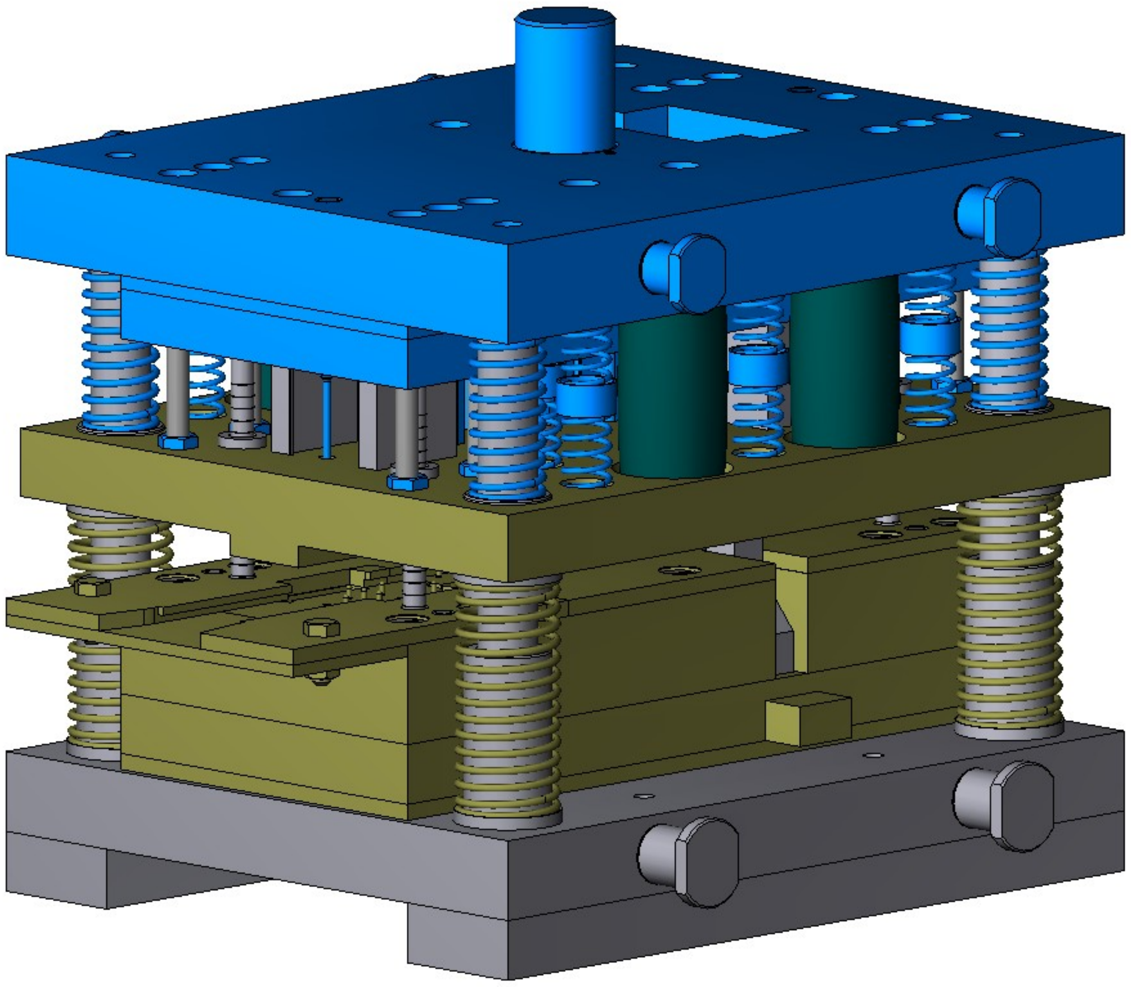


Fig. O.8.7.1.

An assembly unit may contain other assembly units, i.e., it may have a tree structure.

O.8.8. MbSpaceInstance Three-Dimensional Object Insertion

MbSpaceInstance class is declared in space_instanse.h file.

MbSpaceInstance 3D object insertion is an inheritor of MbItem class, and it is described by MbSpaceItem* **spaceItem** geometric object.

Insertion acts as geometric object wrapper that permits you to process it as geometric model object. An insert adds to an ordinary geometric object a construction log, attributes and MbItem geometric model object methods. 3D object insertion is intended for auxiliary constructions. Surface insertion is shown in Figure O.8.8.1.

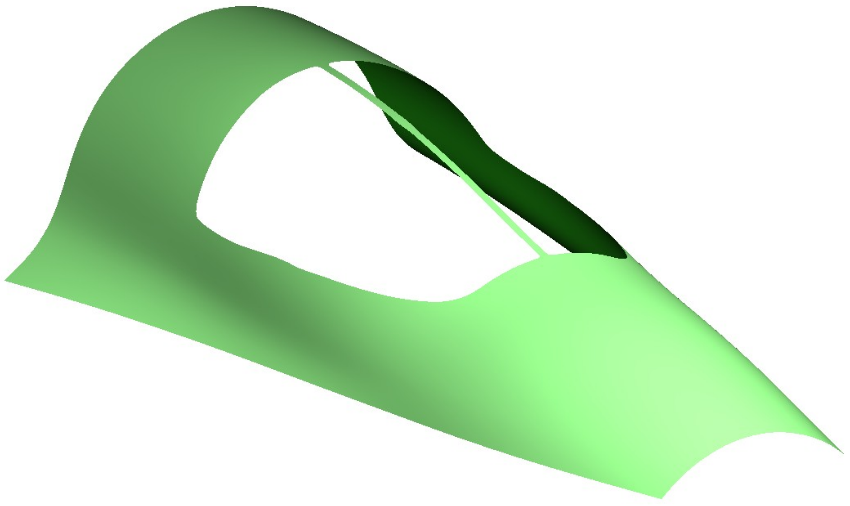


Fig. O.8.8.1.

Object insertion can contain MbSurface surface, MbCurve3D curve, MbPoint3D point or MbLegend auxiliary geometric object. Geometric object insertion is not used for objects that inherit an object of MbItem geometric model object (a solid, a wireframe, a point frame, a polygon, an assembly or an insertion).

O.8.9. MbPlaneInstance Two-Dimensional Object Insertion

MbPlaneInstance class is declared in plane_instanse.h file.

MbPlaneInstance 2D object insertion is an inheritor of MbItem class and it is described by std::vector<MbPlaneItem>**planeItems** set of 2D geometric objects and MbPlacement3D **place** local coordinate system. Two-dimensional objects are located in XY plane of the local coordinate system.

Insertion acts as a wrapper of 2D geometric objects that permits you to process them as geometric model object. An insert adds construction log, attributes and methods of MbItem geometric model object to 2D geometric objects. Two-dimensional object insertion is intended for auxiliary constructions. An insertion of 2D curves is shown in Figure O.8.9.1.

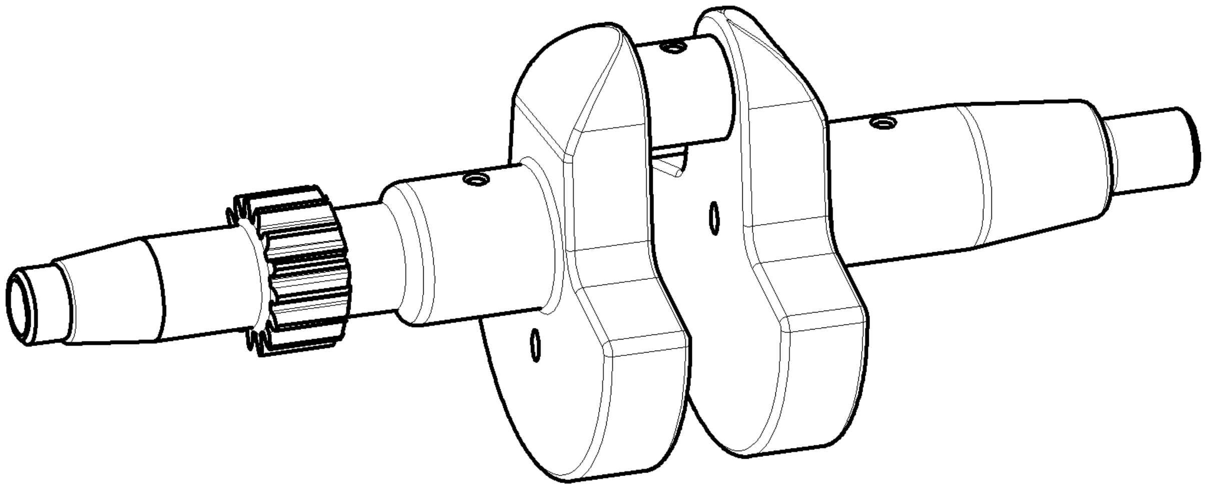


Fig. O.8.9.1.

An insertion of 2D objects can contain MbCurve 2D curve, MbMultiline multiline or MbRegion region.

O.8.10. MbAssistingItem Auxiliary Object

MbAssistingItem class is declared in assisting_item.h file.

MbAssistingItem auxiliary object of geometric model is an inheritor of MbItem class and it is described by MbPlacement3D **place** local coordinate system. An auxiliary object is used to position other objects. An auxiliary object has a construction log, attributes and methods of MbItem geometric model object.
auxiliary constructions. An auxiliary object is shown in Figure O.8.10.1.

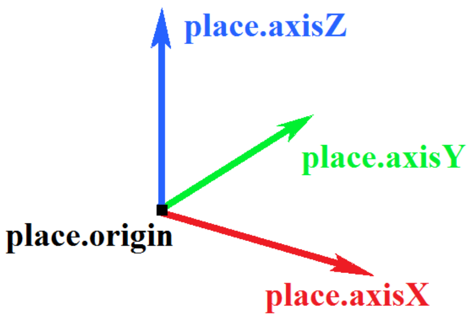


Fig. O.8.10.1.

R.1. CONSTRUCTING TRIANGULATION

C3D geometric kernel constructs a polygonal representation of geometric model based on its boundary representation. A polygonal representation contains a set of triangulations. Every triangulation approximates a single face of the modeled object by rectangular and triangular flat plates. Polygonal representation is used to visualize a geometric model, calculate inertial characteristics and detect collisions of model elements.

R.1.1. Triangulation Calculation Control

Methods that construct polygonal representations use **MbStepData** structure shown in Figure R.1.1.1 as their input.

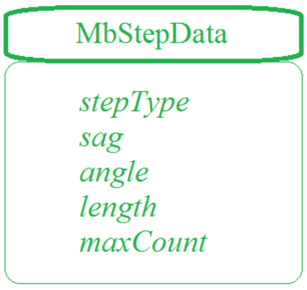


Figure R.1.1.1.

MbStepData structure is declared in mb_data.h file. **MbStepData** structure contains the following data:

- unsigned char *stepType* is the field that defines the method used to calculate parameter increments,
- double *sag* is the maximum allowable deviation of deflection,
- double *angle* is the maximum allowable deviation of tangents or normals by angle,
- double *length* is the maximum allowable distance between two adjacent points,
- unsigned int *maxCount* is the maximum number of cells per row or column of triangulation grid.

MbStepData structure controls grid density of a polygonal object, it contains all the data required to calculate parameter increment when moving along model curves and surfaces. *stepType* field defines the method used to calculate the parameter increment when moving along a curve or a surface. This field may contain masks of MbeStepTypy enumeration declared in mb_enum.h file:

- *ist_SpaceStep* is used to visualize the geometric shape;
- *ist_DeviationStep* is used for construction operations;
- *ist_MetricStep* is used for 3D printers;
- *ist_ParamStep* is used to visualize the geometric shape of the objects with snapping texture to surface parameters;
- *ist_CollisionStep* is used to detect collisions of model elements;
- *ist_MipStep* is used to calculate inertial characteristics.

sag parameter limits the increment of curve or surface parameter taking into account the maximum allowable deviation from the original polygon object by deflection. *angle* parameter limits the increment of curve or surface parameter taking into account the maximum allowable deviation from the original polygon object by angular deflection of tangential curves or surface normals at two adjacent points, their separation is equal to the increment. *length* parameter limits the increment of curve or surface parameter taking into account the maximum allowable size of polygon element (triangle side or polygon segment). *maxCount* parameter limits the increment of curve or surface parameter taking into account the maximum allowable number of splittings per row or column of the triangulation grid.

Methods that construct polygonal representations use **MbFormNote** structure shown in Figure R.1.1.2 as their input.

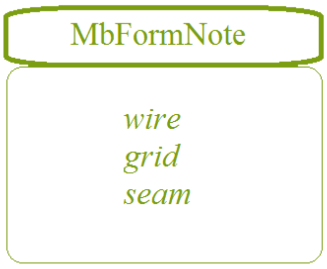


Figure R.1.1.2.

MbFormNote structure is declared in mb_data.h file. **MbFormNote** contains the following data:

- bool *wire* is the flag for constructing the polygonal object,
- bool *grid* is the flag for constructing the polygonal object,
- bool *seam* is the flag indicating that seam edges are not ignored.

MbFormNote structure defines the method for constructing the polygonal object: If *wire*==true then the polygonal object is filled with broken lines, if *grid*==true then the polygonal object is filled with triangulations. *seam* parameter defines the method for representing seams in the polygonal object: If *seam*==true then triangulation is not closed by seams and seam edges are treated as ordinary edges. Their points are considered to be edge in triangulation and polygons are constructed for them, if *seam*==false then triangulation is closed by seams and seam edges are ignored, closed triangulations are constructed and polygons are not constructed for edges.

R.1.2. Constructing a Polygonal Object

Virtual method for geometric model objects

```
MbItem *
MbItem::CalculateMesh ( const MbStepData & stepData,
                        const MbFormNote & note,
                        MbRegDuplicate * iReg ) const
```

constructs a polygon object approximating the specified object of the geometric model.

Input parameters of the method are:

- *stepData* are the data required to calculate approximation step,
- *note* is the method used to construct the polygonal object,
- iReg is the registrar of the copied objects.

In case of success, the method returns a pointer to the newly constructed object of the geometric modelMbItem*, otherwise zero is returned.

The method is declared in item.h file and header files of MbItem descendant files.

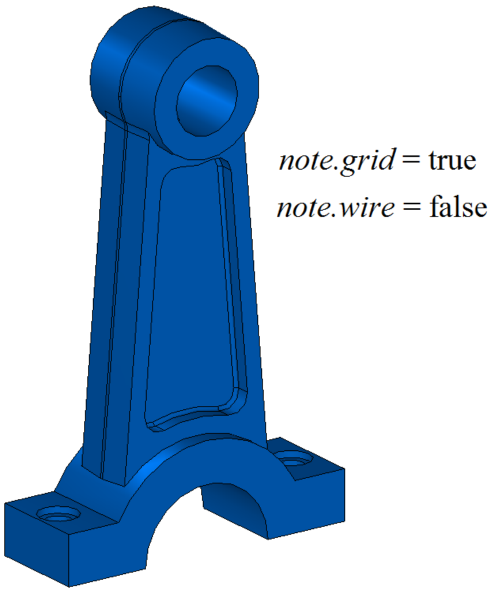
This method approximates model objects and creates polygonal copies having similar structures. For a solid, a wire frame or a point frame, this method will create a polygonal object MbMesh that approximates the original object, then the method will return a pointer to the newly created object. Each **grids**[*i*] triangulation of the object MbMesh will approximate the *i*th face; each **wires**[*i*] polygon of the object MbMesh will approximate the *i*th edge, each **peaks**[*i*] apex of the object MbMesh will approximate the *i*th vertex. For a polygonal object MbMesh, this method will create a polygonal copy object. For an insertion, this method will create the insertion with the object that will create the same method for insertion content. For an assembly unit, this method will create the assembly unit with the objects that will create the same method for assembly unit objects.

stepData parameter controls the density of polygonal object grid and contains all the data required to calculate parameter increment when moving along model curves and surfaces. *note* parameter defines the method for constructing a polygonal object. *stepData* and *note* parameters are described in Item R.1.1. Triangulation Calculation Control. If *note.wire*==true, then the method creates a set of pointers to **mesh**.wires polygons; if *note.grid*==true, then the method fills a set of pointers to **mesh**.grids triangulations (for faces and surfaces), a set of pointers to **mesh**.wires polygons (for edges) and a set of pointers to

mesh.peakes apexes (for vertices).

iReg parameter may be equal to zero. This parameter is used to provide nested methods data on already processed objects.

In Figure R.1.2.1, you can see a polygonal solid object that was constructed with the following parameters: *note.wire*==false, *note.grid*==true. This object contains triangulations, polygons and apexes. In Figure R.1.2.2, you can see a polygonal solid object that was constructed with the following parameters: *note.wire*==true, *note.grid*==false, the object contains only polygons.



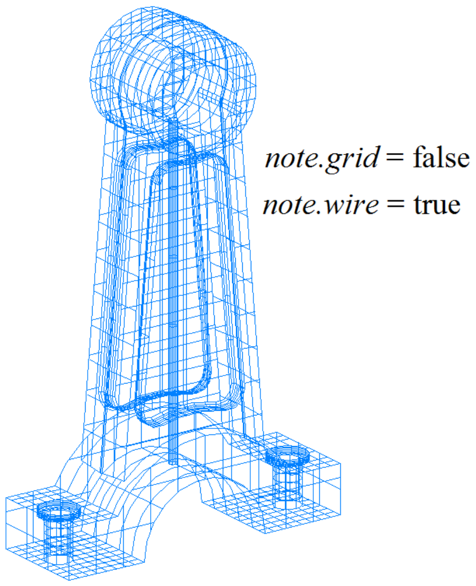


Figure R.1.2.1. *Figure R.1.2.2.*

In Figure R.1.2.3, you can see a polygonal object of an assembly unit that was constructed with the following parameters: *note*.wire==false, *note*.grid==true. This object consists of an assembly unit of polygonal objects that approximate the parts.

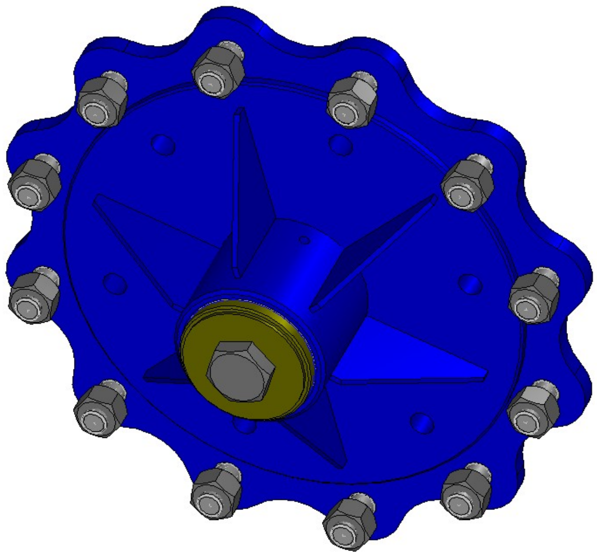


Figure R.1.2.3.

The method is used to visualize objects of a geometric model. You can easily transform polygonal objects and quickly find an intersection with a straight line. It is a polygonal copy of geometric model object that is displayed on the screen, and the original object remains offscreen.

R.1.3. Adding a Polygonal Object

Virtual method for geometric model objects
bool
MbItem::**AddYourMesh** (const MbStepData & *stepData*,
const MbFormNote & *note*,
MbMesh & **mesh**) const
constructs and adds its own polygonal copy to received **mesh** polygonal object.
Input parameters of the method are:

- *stepData* are the data required to calculate approximation step,
- *note* is the method used to construct the polygonal object.

The output parameter of the method is **mesh** polygonal object.
In case of success, the method returns true.
The method is declared in item.h file and descendant header files MbItem
The method approximates model objects by polygonal copies and adds them to the received **mesh** original object. For an insertion, the method will create a polygonal copy of insertion content, transform it to a global coordinate system and then will add it to received **mesh** object. For an assembly unit, the method will create a polygonal copy of assembly unit content, transform it to a global coordinate system and then will add it to **mesh** original object.
By analogy with **CalculateMesh** method, *stepData* parameter controls the density of polygonal object grid, and *note* parameter defines how to construct the polygonal object. *stepData* and *note* parameters are described in Item R.1.1. Triangulation Calculation Control. In contrast to the above method, this method creates a single polygon object for complex objects. In Figure R.1.3.1, you can see a polygonal object for the assembly unit shown in Figure R.1.2.3. This polygonal object was constructed using the considered method.

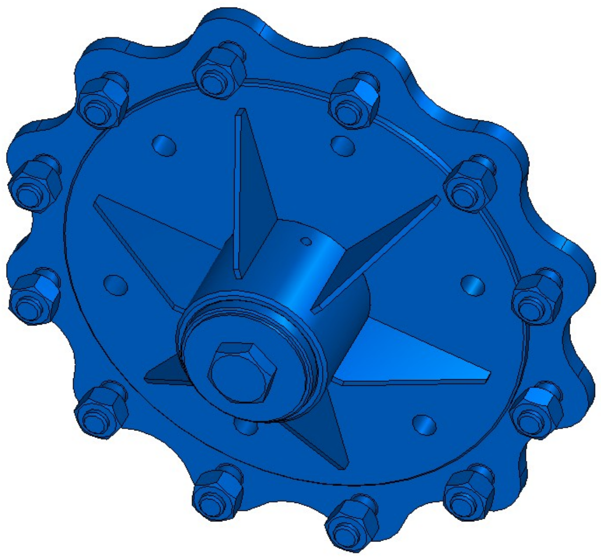


Figure R.1.3.1.

R.1.4. Constructing Polygons for an Object

Virtual method for three-dimensional geometric objects
void

MbSpaceItem::**CalculateWire** (double *sag*,
MbMesh & **mesh**)
fills received **mesh** polygonal object with the set of polygons approximating the geometric object.

Input parameter of the method is:

- *sag* is the maximum allowable deviation from the original object in terms of deflection.

The output parameter of the method is **mesh** polygonal object.

The method is declared in space_item.h file and MbSpaceItem descendant header files.

The polygonal object is described in Item O.8.5. MbMesh Polygonal Object. *sag* parameter determines the maximum allowable distance between the object and the broken line that goes by the points of polygons. The method creates only a set of pointers to **mesh.wires** polygons (broken lines).

The method uses a single polygon to approximate a curve. As to contours, this method uses several polygons, each contour approximates a corresponding segment of the contour. In Figure R.1.4.1, you can see a curve and its polygonal object consisting of one polygon.

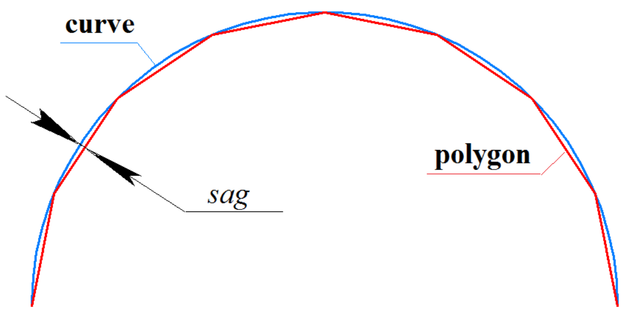


Figure R.1.4.1.

In order to approximate a surface, the method uses a set of polygons that go through *u* lines, *v* lines and along a surface border. In Figure R.1.4.2, you can see a polygonal object of a surface that consists of several broken lines.

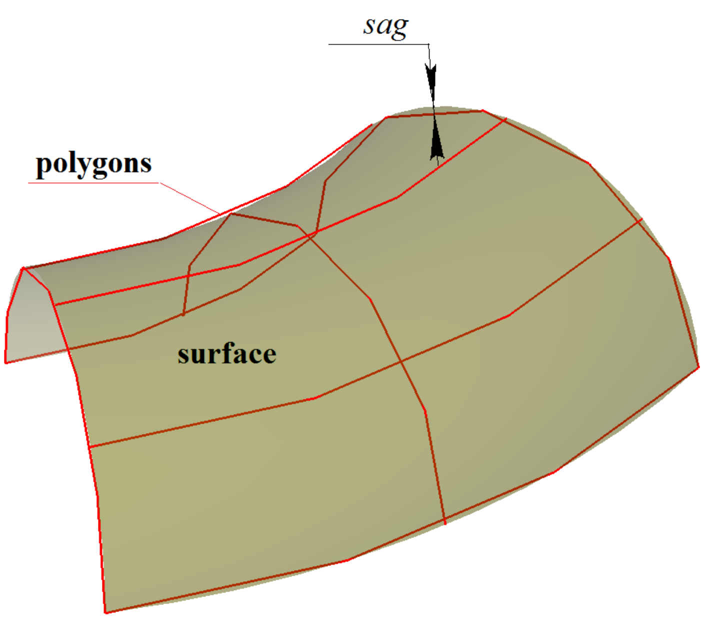


Figure R.1.4.2.

The method uses a set of broken lines to approximate a solid. These lines go inside a face along *u* lines and *v* lines of face surfaces. A set of broken lines is also used to approximate the curves at which solid edges are based. In Figure R.1.4.3 (right), you can see a polygon object for the solid.

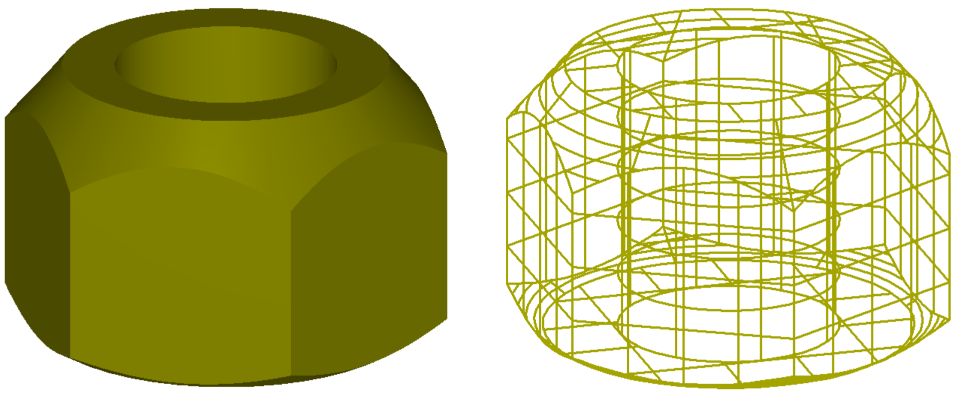


Figure R.1.4.3.

For objects of the geometric model, the method works exactly as **CalculateMesh** metod if *stepData.stepType*==ist_SpaceStep, *stepData.sag*==*sag*, note.wire==true and note.seam==true.

R.1.5. Constructing Triangulation for a Face

Method
void
CalculateGrid (const MbFace & **face**,
const MbStepData & *stepData*,
bool *edgePoints*,
MbGrid & **grid**,
bool *dualSeams* = true)
approximates a face using triangular and quadrangular plates.

Input parameters of the method are:

- **face** is the face itself,
- *stepData* are the data required to calculate approximation step,
- *edgePoints* is the flag indicating that spatial points will be used,
- *dualSeams* is the flag for processing seams.

grid triangulation is the output parameter of the method.

This method is declared in tri_face.h file.

stepData parameter manages triangulation density, it contains data used to calculate the increment for moving along face surface. *stepData* parameter is described in Item R.1.1. Triangulation Calculation Control. For various values of *stepData.stepType*, various fields of **grid** triangulation are filled. If *stepData.stepType* contains *ist_MipStep* mask, then **grid.params** set is created. If *stepData.stepType* contains *ist_CollisionStep* or *ist_ParamStep* mask, then **grid.params**, **grid.points** and **grid.normals** sets are created. In all other cases, **grid.params** and **grid.points** sets are created.

In Figure R.1.5.1, you can see triangulation of a flat face with *ist_SpaceStep* mask.

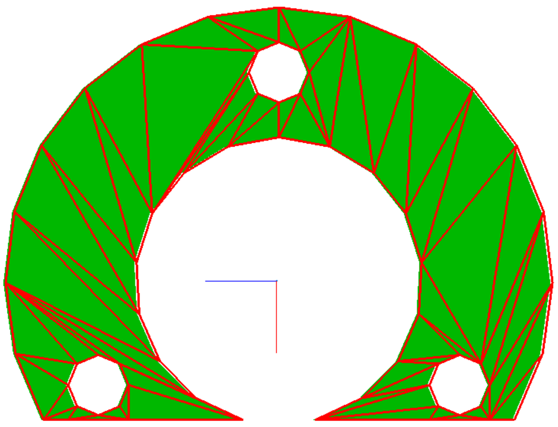


Figure R.1.5.1.

In Figure R.1.5.2, you can see triangulation of a curved face with *ist_SpaceStep* mask.

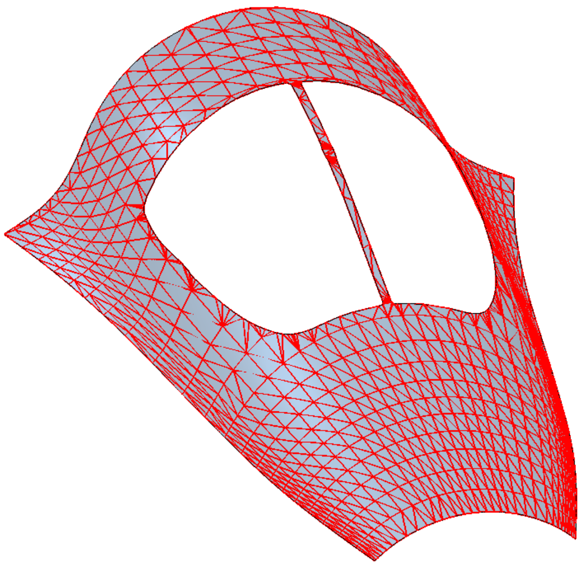


Figure R.1.5.2.

When polygonal objects are constructed, the method is used by **CalculateMesh** and **AddYourMesh** methods if *note.grid*==true.

R.1.6. Constructing Triangulation for a Solid

Method
void
CalculateGrid (const MbSolid & **solid**,
const MbStepData & *stepData*,
RPArray<MbGrid> & **grids**)
constructs a set of triangulations for the faces of a solid.

Input parameters of the method are:

- **solid** is the solid,
- *stepData* are the data needed to calculate approximation increment.

grids set is the output parameter of the method.

This method does not return any value.

The method is declared in mip_solid_area_volume.h file.

The method approximates faces of **solid** solid using **grids** triangulations. When the method is called, **meshs** set should be empty. When the method is called, **grids** set should be empty. Each *i*th face of **solid** solid has its own **grids**[*i*] object.

stepData parameter controls grid density of the polygonal object, it contains all data required to calculate parameter increment when moving along curves and surfaces of the solids. *stepData* parameter is described in Item R.1.1. Triangulation Calculation Control.

R.1.7. Constructing Polygonal Objects for a Set of Solids

Method

```
void
CalculateGrid ( const RPArray<MbSolid> & solids,
                const MbStepData & stepData,
                RPArray<MbMesh> & meshs )
```

constructs a set of polygonal objects for a set of solids.

Input parameters of the method are:

- **solids** is the set of solids,
- *stepData* are the data needed to calculate the approximation increment.

meshs set is the output parameter of the method.

This method does not return any value.

The method is declared in mip_solid_area_volume.h file.

The method approximates **solids** solids using **meshs** polygonal objects. When the method is called, **meshs** set should be empty. Each **solids**[*i*] solid has a corresponding newly constructed **meshs**[*i*] object.

stepData parameter controls grid density of the polygonal object, it contains all data required to calculate parameter increment when moving along curves and surfaces of the solids. *stepData* parameter is described in Item R.1.1. Triangulation Calculation Control.

R.2. CONSTRUCTING FLAT PROJECTIONS

C3D geometric kernel uses a wireframe model to construct a flat projection of the modeled object. We'll create a wireframe model from a boundary representation of the geometric model by taking edges and adding their outlines instead of faces. The outlines go through faces and divide them into parts that are visible or invisible from part observation point. Flat projections are more informative if all edges and outlines invisible from the observation point are hidden in wireframe model.

R.2.1. Data Required to Construct Flat Projections

MbLump structure shown in Figure R.2.1.1 is used as method input to construct flat projections to present the solids.

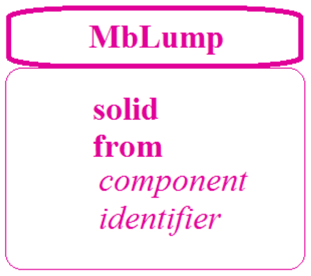


Figure R.2.1.1.

MbLump structure is declared in lump.h file. **MbLump** contains a pointer to **solid** solid, a matrix that transforms the solid from **from** local coordinate system, *component* and *identifier* solid identification parameters.

MbProjectionsObjects class shown in Figure R.2.1.2 is used by the method to construct flat projections in order to display supplementary objects.

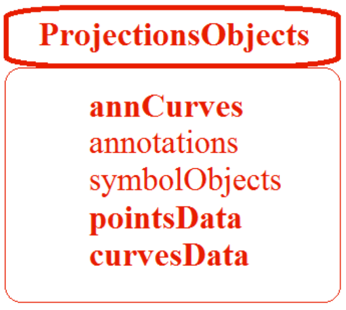


Figure R.2.1.2.

MbProjectionsObjects class is declared in map_create.h file. **MbProjectionsObjects** contains the following data:

- TPointer<PArray<MbAnnCurves> > **annCurves** are annotation curves,
- TPointer<RPArray<MbSimbolthThreadView> > annotations are subsidiary objects,
- TPointer<RPArray<MbSymbol> > symbolObjects are designations,
- TPointer<RPArray<MbSpacePoints> > **pointsData** are points,
- TPointer<RPArray<MbSpaceCurves> > **curvesData** are curves.

MbVEFVestiges structure shown in Figure R.2.1.3 is used to pass flat projection construction results.

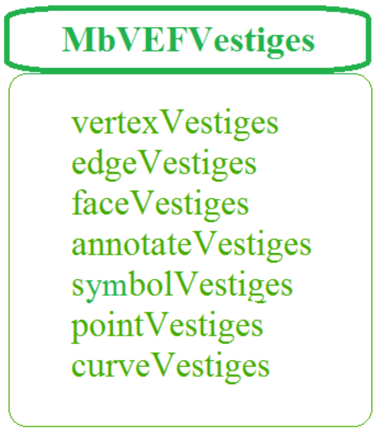


Figure R.2.1.3.

MbVEFVestiges structure is declared in map_vestiges.h file. **MbVEFVestiges** structure contains grouped flat projections of object elements. Each element in projection group contains a constructed projection, a pointer to projection parent object, data on projection visibility and other data on this projection. **MbVEFVestiges** structure contains the following groups:

• PArray<MbVertexVestige>	vertexVestiges is a set of vertex projections.
• PArray<MbEdgeVestige>	edgeVestiges is a set of edge projections,
• PArray<MbFaceVestige>	faceVestiges is a set of face projections,
• PArray<MbAnnotationVestige>	annotateVestiges is a set of annotation object projections,
• PArray<MbSymbolVestige>	symbolVestiges is a set of symbol projections,
• PArray<MbVertexVestige>	pointVestiges is a set of point projections,
• PArray<MbEdgeVestige>	curveVestiges is a set of curve projections.

R.2.2. Constructing Model Flat Projection

Method
void
GetVestiges (const MbPlacement3D & **place**,
double *znear*,
const RPArray<MbLump> & **lumps**,
const MbProjectionsObjects & **objects**,
MbVEFVestiges & **result**,
bool *invisible*,
VERSION version)
constructs flat projection for a set of solids and other objects.

Input parameters of the method are:

- **place** is the projection plane,
- *znear* is observation point parameter,
- **lumps** are the projected solids in local coordinate systems,
- **objects** are all other projected objects,
- *invisible* is a flag meaning that invisible lines are constructed,
- version is the version of the constructed object, it is the latest version of Math::DefaultMathVersion().

The output parameter of the method is **result**, it is the structure containing projection data.
The method does not return any value.
The method is declared in map_create.h file.
XY plane of **place** local coordinate system is the projection plane.
znear parameter defines image type. If *znear*=0, then a parallel projection of the objects is constructed.

lumps parameter (Figure R.2.1.1) contains the solids and matrices of their transformation from the local coordinate system. **objects** parameter (Figure R.2.1.2) contains auxiliary objects required to finalize the projection: auxiliary points, curves, designations and annotation objects. *invisible* parameter determines whether it is required to construct invisible lines. In some cases, the method works considerably faster if the construction of invisible lines is canceled.

In Figure R.2.2.1, you can see a solid, its projection lines to a plane parallel to the screen are shown in Figure R.2.2.2. In Figure R.2.2.3, you can see only visible lines of the projection of the solid shown in Figure R.2.2.1.

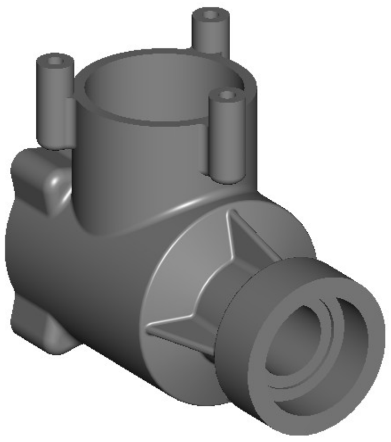
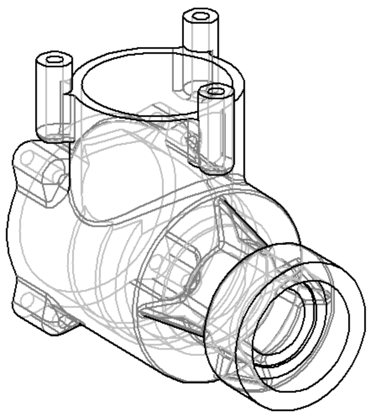
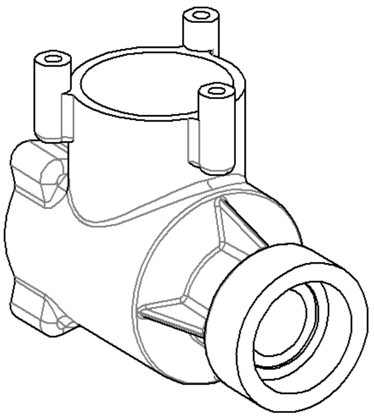




Figure R.2.2.1. *Figure R.2.2.2.* *Figure R.2.2.3.*

The last parameter of the method is used to support previous construction versions.

Method

void
VisualLinesMapping (const MbPlacement3D & **place**,
double *znear*,
const RPArray<MbLump> & **lumps**,
MbVEFVestiges & **result**,
bool *invisible* = true)

constructs a planar projection for a set of solids. It is similar to **GetVestiges** method, the difference is the absence of **objects** auxilliary objects.

R.2.3. Constructing Polygonal Projections of Solids

Method

void

HiddenLinesMapping (const RPArray<MbLump> & **lumps**,
const MbPlacement3D & **place**,
double *znear*,
double *sag*,
PArray<MbPolygon3DSolid> & visibleEdges,
PArray<MbPolygon3DSolid> & hiddenEdges,
PArray<MbPolygon3DSolid> & visibleTangs,
PArray<MbPolygon3DSolid> & hiddenTangs)

constructs a polygonal projection for a set of solids to the specified plane.

Input parameters of the method are:

- **lumps** are the projected solids in local coordinate systems,
- **place** is the projection plane,
- *znear* is observation point parameter,
- *sag* is the maximum allowable deviation by deflection.

Output parameters of the method are as follows:

- visibleEdges are polygons of visible non-smooth edges,
- hiddenEdges are polygons of invisible non-smooth edges,
- visibleTangs are polygons of visible smooth edges,
- hiddenTangs are polygons of invisible smooth edges.

The method does not return any value.

The method is declared in map_create.h file.

XY plane of **place** local coordinate system is the projection plane.

lumps parameter (Figure R.2.1.1) contains the solids and matrices of their transformation from the local coordinate system. *znear* parameter defines image type. If *znear*=0, then a parallel projection of the objects is constructed. MbPolygon3DSolid class contains component number and a pointer to the polygon consisting of a set of points that should be connected in series to approximate edge projection on the plane. *sag* parameter determines approximation accuracy. visibleEdges and hiddenEdges polygons are visible and invisible polygonal projections of non-smooth edges to XY plane of **place** local coordinate system. visibleTangs and hiddenTangs polygons are visible and invisible polygonal projections of smooth edges to XY plane in **place** local coordinate system.

Method

void

VisualLinesMapping (const RPArray<MbLump> & **lumps**,
const MbPlacement3D & **place**,
double *znear*,
double *sag*,
PArray<MbPolygon3DSolid> & visibleEdges,
PArray<MbPolygon3DSolid> & visibleTangs)

constructs only a visible polygonal projection of the set of solids to the specified plane. The method is similar to **HiddenLinesMapping** mehod, but it constructs visible projections only. In some cases, **VisualLinesMapping** works significantly faster than **HiddenLinesMapping**.

R.2.4. Constructing a Triangulation Outline

Method

void

CalculateBoundsSltFast (const MbGrid & **grid**,
const MbMatrix3D & **matrix**,
bool *perspective*,
RPArray<MbFloatPoint3D> & **points**)

constructs a triangulation outline.

Input parameters of the method are:

- **grid** is solid face triangulation,

- **matrix** is a matrix that defines a gaze vector,
- *perspective* is perspective representation flag.

The output parameter of the method is
points, it is a set of pointers to the points from **grid.points** triangulation.

The method does not return any value.

The method is declared in map_create.h file.

This method is intended to construct polygonal outlines and visualize triangulation silhouette.

R.3. CALCULATION OF INERTIAL CHARACTERISTICS

C3D geometric kernel calculates modeled object surface area, volume, center of gravity and moments of inertia. In the general case, numerical integration is used. Volume integration is reduced to modeled object surface integration using Gauss`s theorem. Surface integration uses triangulation of a two-dimensional area of definition of surface parameters. When the above characteristics of the model are calculated, it is possible to include ready-to-use data for particular model elements.

R.3.1. Inertial Characteristics of a Model

InertiaProperties class shown in Figure R.3.1.1 is used to set inertial characteristics of a model.

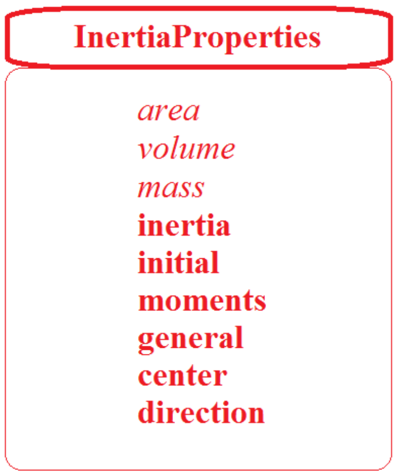


Figure R.3.1.1.

InertiaProperties class is declared in mip_solid_mass_inertia.h file. **InertiaProperties** contains the following model data:

- double *area* is surface area,
- double *volume* is volume,
- double *mass* is mass,
- double **inertia**[3] are static moments in the original coordinate system,
- double **initial**[3][3] are moments of inertia in the original coordinate system,
- double **moments**[3][3] are moments of inertia in the central coordinate system,
- double **general**[3] are principal central moments of inertia,
- MbCartPoint3D **center** is the center of gravity,
- MbVector3D **direction**[3] are the vectors of direction of principal axes of inertia.

When **InertiaProperties** class is initialized all data take zero values and gravity center coordinates are equal to NOT_INITIAL_DBL. **initial** moments of inertia are calculated in the coordinate system wherein the model is described. **moments** moments of inertia are calculated in the coordinate system with the origin located in **center** point and its coordinate axes coincide with the axes of the original coordinate system wherein the model is described. **general** inertia moments are calculated in principle center coordinate system, its origin is **center** point, and coordinate axes are calculated and they coincide with the model principal axes of inertia. Products of inertia in principle coordinate system are equal to zero.

direction vectors define the direction of principal axes of inertia. If all **general**[*i*] *i*=1,2,3 principle moments of inertia differ, then all **direction**[*i*] *i*=1,2,3 vectors are non-zero. If all principle moments of inertia in **general**[*i*] *i*=1,2,3 are the same, then all **direction**[*i*] *i*=1,2,3 vectors are equal to zero, and any three mutually orthogonal vectors may be used as principle directions. If two of three principal moments of inertia are equal, for example, **general**[*j*]==**general**[*k*], then two of the three vectors are equal to zero: **direction**[*j*]=**direction**[*k*]=0, and non-zero **direction**[*i*] vector defines the direction of the principal inertia axis, its moment for other axes is different from the others, any two vectors that are mutually orthogonal and orthogonal to non-zero **direction**[*i*] vector can be used as any two principle directions.

SolidMIAttire and **AssemblyMIAttire** classes shown in Figure R.3.1.2 and Figure R.3.1.3 provide an opportunity to use ready-to-use data for separate model elements. These classes are declared in mip_solid_mass_inertia.h file.

SolidMIAttire
solid
density
matrix
properties
ready

Figure R.3.1.2.

AssemblyMIAttire
assemblies
solids
matrix
properties
ready

Figure R.3.1.3.

SolidMIAttire class contains the following data:

- const MbSolid & **solid** is the solid,
- double *density* is the density or specific gravity of unit area,
- MbMatrix3D **matrix** is the matrix used to transform the solid to the nearest assembly system,
- **InertiaProperties** * **properties** are specified inertial characteristics of the solid, they can be equal to zero or they may be defined not completely,
- bool *ready* is the flag showing that it is not required to calculate the characteristics.

AssemblyMIAttire class contains the following data:

- RPArray<**AssemblyMIAttire**> **assemblies** is a set of assembly units at the next level,
- RPArray<**SolidMIAttire**> **solids** are the solids of the assembly unit,
- MbMatrix3D **matrix** is the matrix used to transform the solid into the nearest assembly system,
- **InertiaProperties** * **properties** are specified inertial characteristics of the assembly solid, they can be equal to zero or they may be defined not completely,
- bool *ready* is the flag showing that it is not required to calculate the characteristics.

If instances of **SolidMIAttire** and **AssemblyMIAttire** classes contain non-zero **properties** and *ready*==true then inertial characteristics of the corresponding object won't be calculated, and calculation result will contain data from **properties**.

If instances of **SolidMIAttire** and **AssemblyMIAttire** classes contain non-zero **properties** and *ready*==false, then calculation result will contain data from **properties** data that are not equal to NULL_EPSILON or NOT_INITIAL_DBL. Data that in **properties** are equal to NULL_EPSILON or NOT_INITIAL_DBL will be calculated. Considered classes permit to mix calculated and assigned data.

R.3.2. Inertial Solid Characteristics

Function
void
MassInertiaProperties (const MbSolid * **solid**,
double *density*,
double *deviateAngle*,
InertiaProperties & **properties**,
IfProgressIndicator * progress = 0)
calculates solid surface area, volume, mass, center of gravity and moments of inertia.

Method input parameters are:

- **solid** is the solid,
- *density* is density or specific gravity per unit area,
- *deviateAngle* is a parameter used to control calculation accuracy.

Output parameters of the method are:

- **properties** are calculated inertial characteristics,
- progress is calculation progress indicator.

This method returns no value.

The method is declared in mip_solid_mass_inertia.h file.

For **solid** closed solid, *density* parameter determines solid density. For a non-closed solid, **solid** *density* parameter determines specific gravity of solid unit area. In the general case, numerical integration is applied, definition area of face surface parameters are triangulated. Parametric area of faces is triangulated using **CalculateGrid** method described in item R.1.4. Constructing Polygons for an Object if *stepData.stepType=ist_MipStep* and *stepData.angle=deviateAngle*. *deviateAngle* parameter determines the maximum allowable angle between the normals of adjacent triangles and quadrangles of surface triangulation. *deviateAngle* parameter controls calculation accuracy. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian). Please note that for complex models small values of *deviateAngle* result in long calculation time.

properties inertial characteristics are described in Item R.1.4. Constructing Polygons for an Object. progress parameter provides information about calculation progress; it may be used to terminate the calculation.

R.3.3. Inertial Characteristics for a Set of Solids

Function
void
MassInertiaProperties (const RPArray<MbSolid> & **solids**,
const SArray<double> & *densities*,
const SArray<MbMatrix3D> & **matrices**,
const PArray<InertiaProperties> & **mpSolids**,
double *deviateAngle*,
InertiaProperties & **properties**,
IfProgressIndicator * progress = 0)
calculates surface area, volume, mass, center of gravity and moments of inertia for a set of solids.

Method input parameters are:

- **solids** is a set of solids,
- *densities* is a set of densities or specific gravities per unit area,
- **matrices** is the set of matrices used to transform solids into a common coordinate system,
- **mpSolids** is a set of available characteristics of solids (it may contain zeros),
- *deviateAngle* is a parameter used to control calculation accuracy.

Output parameters of the method are:

- **properties** are calculated inertial characteristics,
- progress is calculation progress indicator.

This method returns no value.

The method is declared in mip_solid_mass_inertia.h file.

densities, **matricies**, and **mpSolids** should contain a number of elements equal to the number of solids in **solids** set. The second parameter determines solid density for a closed solid or specific gravity of unit area for a non-closed solid. **matricies** parameter contains a set of matrices used to convert solids into a coordinate system where the calculation should to be used. **mpSolids** parameter contains the characteristics of the corresponding solids that should be used instead of calculated characteristics. Considered method can be used to calculate inertial characteristics of assembly unit; inertial characteristics of unit elements were previously calculated in local coordinate systems.

In the general case, numerical integration is applied, definition area of face surface parameters are triangulated. Parametric area of faces is triangulated using **CalculateGrid** method described in item R.1.4. Constructing Polygons for an Object if *stepData.stepType=ist_MipStep* and *stepData.angle=deviateAngle*. *deviateAngle* parameter determines the maximum allowable angle between the normals of adjacent triangles and quadrangles of surface triangulation. *deviateAngle* parameter controls calculation accuracy. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian). Please note that for complex models small values of *deviateAngle* result in long calculation time.

properties inertial characteristics are described in Item R.3.1. Inertial Characteristics of a Model. progress parameter provides information about calculation progress; it may be used to terminate the calculation.

R.3.4. Inertial Characteristics of a Model

Function
void
MassInertiaProperties (const AssemblyMIAttire & **assembly**,
double *deviateAngle*,
InertiaProperties & **properties**,
IfProgressIndicator * progress = 0)
calculates surface area, volume, mass, center of gravity and moments of inertia for the model described as assembly unit.

Method input parameters are:

- **assembly** is an assembly unit that may contain precalculated characteristics,
- *deviateAngle* is a parameter used to control calculation accuracy.

Output parameters of the method are:

- **properties** are calculated inertial characteristics,
- progress is calculation progress indicator.

This method returns no value.

The method is declared in mip_solid_mass_inertia.h file.

assembly parameter represents an analogue of the assembly unit containing other assembly units in local coordinate systems and solids with specified density in local coordinate systems. Elements of the assembly units may have precalculated inertial characteristics and corresponding control parameters. If an element has precalculated characteristics, then these characteristics are used in the total amount, thus reducing calculation time.

deviateAngle parameter controls calculation accuracy. Calculation time depends on this value. *deviateAngle* parameter determines the maximum allowable angle between the normals of adjacent triangles and quadrangles of surface triangulation. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian). Please note that for complex models small values of *deviateAngle* result in long calculation time.

properties inertial characteristics are described in Item R.3.1. Inertial Characteristics of a Model. progress parameter provides information about calculation progress; it may be used to terminate the calculation.

R.3.5. Calculation of Surface Area

Method
double
CalculateArea (const RPArray<MbFace> & **faces**,
double *deviateAngle*)
calculates the surface area for a set of faces.

Method input parameters are:

- **faces** is the set of faces,
- *deviateAngle* is a parameter used to control calculation accuracy.

This method returns the area of the specified set of faces.

The method is declared in mip_solid_mass_inertia.h file.

deviateAngle parameter controls the accuracy of calculation. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian). In the general case, numerical integration is applied, definition area of face surface parameters is triangulated. Parametric area of faces is triangulated using **CalculateGrid** method described in Item R.1.4. Constructing Polygons for an Object if *stepData.stepType=ist_MipStep* and *stepData.angle=deviateAngle*. *deviateAngle* parameter determines the maximum allowable angle between the normals of adjacent triangles and quadrangles of surface triangulation.

Method

double

CalculateArea (const MbFace & **face**,
double *deviateAngle*)

calculates the surface area of a single face.

Method input parameters are:

- **face** is the face,
- *deviateAngle* is a parameter used to control calculation accuracy.

This method returns the area of the specified **face**.

This method is declared in tri_face.h file.

deviateAngle parameter controls the accuracy of calculation. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian).

Method

double

CalculateArea (constMbSurface & **surface**,
double *deviateAngle*)

calculates surface area.

Method input parameters are:

- **surface** is the surface,
- *deviateAngle* is a parameter used to control calculation accuracy.

This method returns the area of the **surface**.

The method is declared in mip_solid_area_volume.h file.

deviateAngle parameter controls the accuracy of calculation. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian).

Method

double

CalculateAreaCentre (const MbFace & **face**,
double *deviateAngle*,
bool *byOuter*,
VERSION version,
MbCartPoint3D & **centre**)

calculates face surface area and the center of gravity.

Method input parameters are:

- **face** is the face,
- *deviateAngle* is the parameter that controls calculation accuracy,
- *byOuter* is the parameter indicating that internal cutouts of the face should be ignored,
- version is the parameter that control calculation version.

Method output parameter: **centre** is the center of gravity of the face.

This method returns face area.

The method is declared in mip_solid_area_volume.h file.

deviateAngle parameter controls the accuracy of calculation. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian).

byOuter parameter permits to ignore the internal cutouts of the face in the calculations. If *byOuter*=true, then it is assumed that the face does not have internal cutouts. If *byOuter*=false, then standard calculation of the surface area and the center of gravity are executed for the face. version parameter permits to ensure support of previous calculation versions.

Method
double
CalculateAreaCentre (const MbFaceShell & **shell**,
double *deviateAngle*,
MbCartPoint3D & **centre**)
calculates the surface area and the center of gravity for a set of faces.

Method input parameters are:

- **shell** is the set of faces,
- *deviateAngle* is a parameter used to control calculation accuracy.

Method output parameter: **centre** is the center of gravity of the set of faces.

This method returns the area of the specified set of faces.

The method is declared in mip_solid_area_volume.h file.

deviateAngle parameter controls the accuracy of calculation. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian).

R.3.6. Calculation of Solid Volume

Method
double
CalculateVolume (const MbSolid & **solid**,
double *deviateAngle*)
calculates solid volume.

Method input parameters are:

- **solid** is the solid,
- *deviateAngle* is a parameter used to control calculation accuracy.

This method returns the solid volume.

The method is declared in mip_solid_area_volume.h file.

deviateAngle parameter controls the accuracy of calculation. Calculation time depends on this value. *deviateAngle* parameter should fall within 0.01 (radian) <=*deviateAngle*<=0.35 (radian).