C3D Labs

C3D

Developer Manual

CONTENT TABLE

CONTENT

INTRODUCTION

General Information

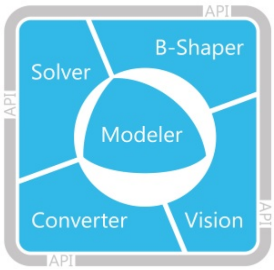
C3D Toolkit is a software development kit responsible for constructing and editing geometric models. C3D can be used as a software component in Computer-Aided Design systems.

C3D focuses in itself a software implementation of mathematical methods for constructing 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..

Structure and Distinctive Features

C3D consists of five modules shown in Figure: C3D Modeler geometric kernel, C3D Solver parametric kernel, C3D Converter exchange data module, C3D Vision visualization module and C3D B-Shaper.



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.

Using our C3D Visualization Module, software developers customize the graphical user interface of engineering applications and visualization parameters for three-dimensional geometric models displayed in it. C3D Vision controls the quality of rendering for 3D models using mathematical apparatus and software, and the workstation hardware.

C3D B-Shaper lets you work with polygonal models in MCAD, AEC, BIM, and other CAD applications by converting the models to boundary representation (b-rep) bodies.

So, C3D includes the geometric kernel, the geometric constraints solving component, and exchange data converter. This is the first distinctive feature of the C3D.

C3D API

All C3D modules provide programming interfaces (API) for using their functionality in an engineering application, which consist of constants, enumerations, functions, structures, classes.

The features of the C3D API are openness and extensibility. C3D provides direct access to objects, which allows you to extend the functionality by inheriting from C3D objects.

In addition, an important feature of the C3D API is its stability and backward compatibility. Stability means that API should not vary much over the longest possible period of time. Backward compatibility means that code written using the previous version works functionally the same with the next version of the API.

For cases when there is a need to modify an existing interface, there is a clear procedure for replacing it with a new interface, aimed primarily at maintaining the users' ease of use of the API.

An interface that needs to be changed is declared deprecated using the DEPRECATE_DECLARE and DEPRECATE_DECLARE_REPLACE macros defined in the math_x.h file. These macros display a warning message when compiling code that uses the deprecated function. In addition, on the Windows platform, the warning issued contains information about the timing of the removal of the deprecated interface, and the DEPRECATE_DECLARE_REPLACE macro additionally specifies the interface that replaces the deprecated function.

Functionality

C3D Modeler provides the following service for the geometric model: description of the shape of the modeled object; description of relations between geometric model elements; support of model construction history; and support of attributes of geometric model elements.

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

C3D Modeler constructs polygonal model based on its boundary representation model. A polygonal model is constructed by triangulating geometric model elements; it is used for visualization and calculations. In addition, C3D Modeler calculates Inertial Properties of the geometric model, maps flat projections for the geometric model, and detects collision of model elements.

C3D Modeler supports the construction log with operations, its input data, and sequence used for model construction. The construction log permits to edit the geometric model and to rebuild it with new parameters. Geometric model elements are supplied by attributes destined for the additional information about elements. Objects of geometric model as well as individual elements have attributes.

C3D Solver provide relations between model elements. C3D Solver guarantees 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.

C3D Converter uses the following formats to exchange geometric model data with other systems: STEP, IGES, SAT (ACIS), X_T, X_B (Parasolid), STL, VRML, and JT. STEP, IGES, SAT, X_T, X_B formats transmit the boundary representation of the geometric model. STL and VRML formats transmit the polygonal representation of the geometric model. JT format transmit the hybrid representation (both) of the geometric model. STEP format supports transmit of product and manufacturing information (PMI).

C3D Vision allows developers to significantly improve visualization capabilities of engineering software by increasing the quality of 3D model rendering and speeding the processing of large assemblies. The C3D visualization toolkit yields new opportunities for managing three-dimensional scenes and animations through its ready-to-use feature manager. Included is a design tree for 3D models, scene graph and interactive tools for scene manipulation. All of these functions have become an integral part of modern design products.

C3D B-Shaper converts polygonal models to boundary representation (b-rep) bodies. Polygonal models are the typical result from 3D scanners and non-CAD 3D modeling software, such as those used to develop movies and games. B-rep is the primary method of representing 3D models in geometric software such as CAD.

Theoretical Foundations

C3D 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 Modeler provide this.

Geometric constraints that describe relations between model elements and other conditions are formulated as equations. C3D Solver 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 Modeler.

C3D Modeler 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 and the objects in exchange formats.

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

Package

C3D package contains c3d.lib, c3d.dll, libc3d.so and libc3d.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 2012, VisualStudio 2013, VisualStudio 2015, VisualStudio 2017 and VisualStudio 2019 development environments. GCC and CLang compilers were 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. Clang compiler was used in amd64, Unicode/Multibyte, Release/Debug configurations in FreeBSD operating system.

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

The distribution kit contains C3D as well as C# wrapper that permits to use .NET technology when applications are developed in C#.

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

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 is stored in Example/Demo folder.

Test.vcxproj and Test.vcxproj.filters files contain C3D test application solution and project respectively for Microsoft VisualStudio 2015.

To create a project and compile the test application of 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, 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 can work in .NET environment. You should use the wrapper included in C3D package to develop applications in .NET environment.

C3D C# wrapper is NetC3D.dll file that was built using .NET Framework4.5.2 platform in 32bit/64bit, Debug/Release configurations and in VisualStudio 2012, VisualStudio 2013, VisualStudio 2015 development environments. The library was compiled with Strong Name signature support.

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

1. In C3D package, select NetC3D.dll file with the required configuration: 32bit/64bit or Debug/Release in VisualStudio 2012/VisualStudio 2013/VisualStudio 2015 development environment.
2. Copy c3d.dll file and place it text to NetC3D.dll file. The dll should be from the same package for the same configuration and development environment: 32bit/64bit, Debug/Release or VisualStudio 2012/VisualStudio 2013/VisualStudio 2015 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:

```
System.String key = Environment.GetEnvironmentVariable("C3Dkey");
System.String signature = Environment.GetEnvironmentVariable("C3Dsignature");
NetC3D.ToolEnabler.EnableMathModules(key, signature);
```

where C3Dkey and C3Dsignature are the environment variables containing the key and the signature.

M.1. METHODS USED TO BODIES CONSTRUCTING

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

M.1.1. Constructing an Elementary Body

The method
MbResultType
ElementarySolid (SArray<MbCartPoint3D> & **points**,
ElementaryShellType *solidType*,
const MbSNameMaker & names,
MbSolid*& **result**)
constructs an elementary body (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 body,
- names is faces namer.

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

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

This method is declared in action_body.h file.

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

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

Table M.1.1.1.

solidType	**Body 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 Fig M.1.1.1.

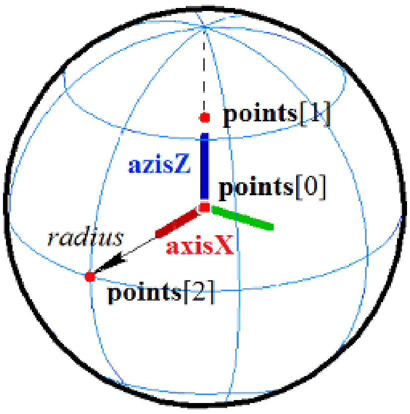


Fig M.1.1.1.

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 Fig M.1.1.2.

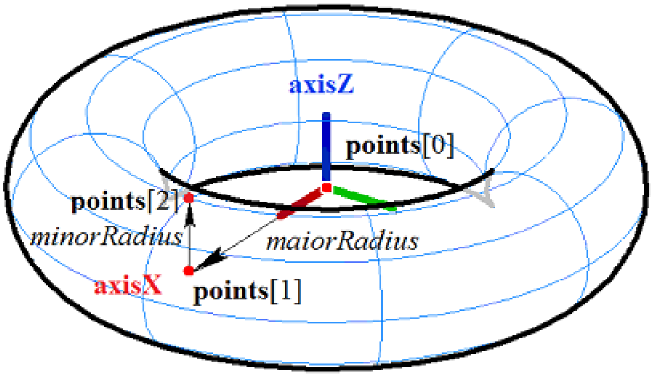


Fig M.1.1.2.

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 Fig M.1.1.3.

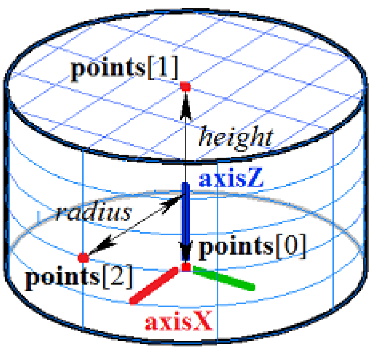


Fig M.1.1.3.

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 Fig M.1.1.4.

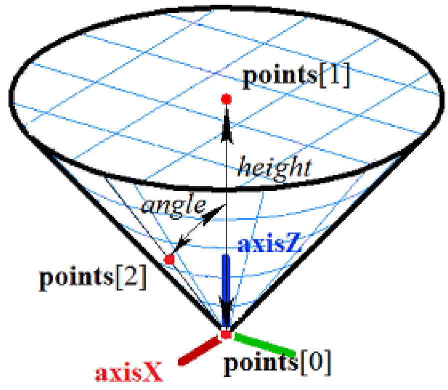


Fig M.1.1.4.

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 Fig M.1.1.5.

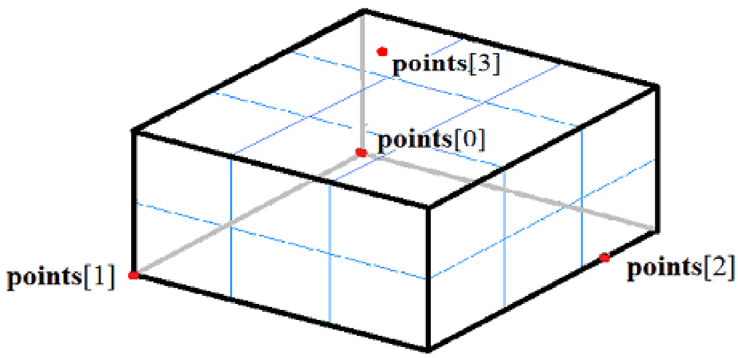


Fig M.1.1.5.

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 Fig M.1.1.6.

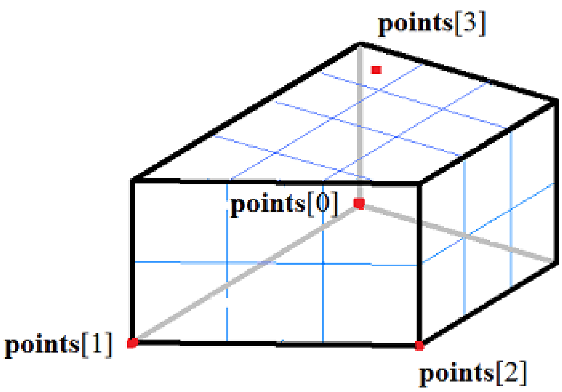


Fig M.1.1.6.

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 Fig. M.1.1.7.

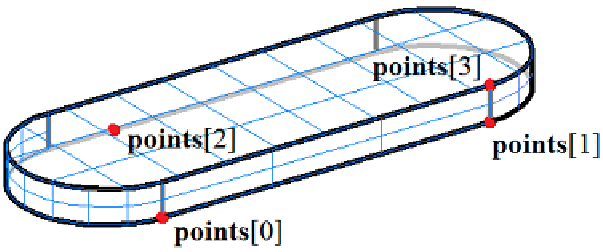


Fig. M.1.1.7.

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 Fig. M.1.1.8, you can see a right-angle prism with a pentagonal base.

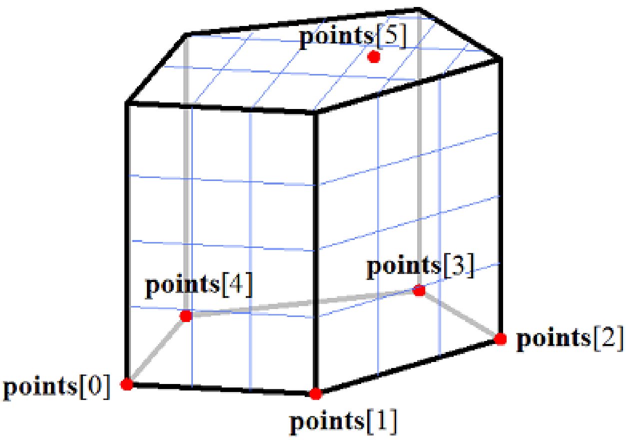


Fig. M.1.1.8.

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 Fig. M.1.1.9, you can see a pyramid with a pentagonal base.

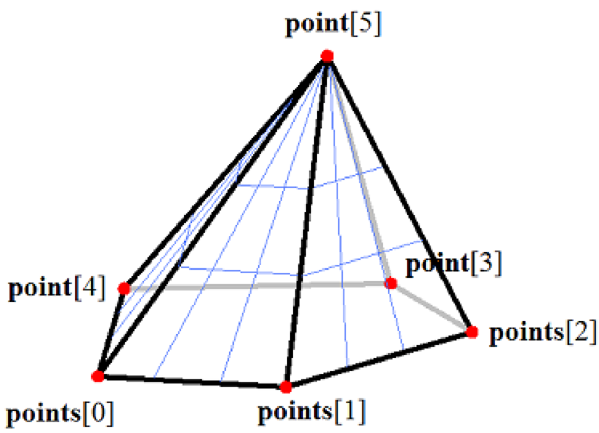


Fig. M.1.1.9.

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 body. This constructor contains all data required to construct the body. MbElementarySolid constructor is declared in cr_elementary_solid.h file.

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

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

The method

MbResultType

ElementarySolid (const MbSurface & **surface**,
const MbSNameMaker & names,
MbSolid *& **result**)

constructs an elementary body by a given surface.

Method input parameters are:

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

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

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 body.

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

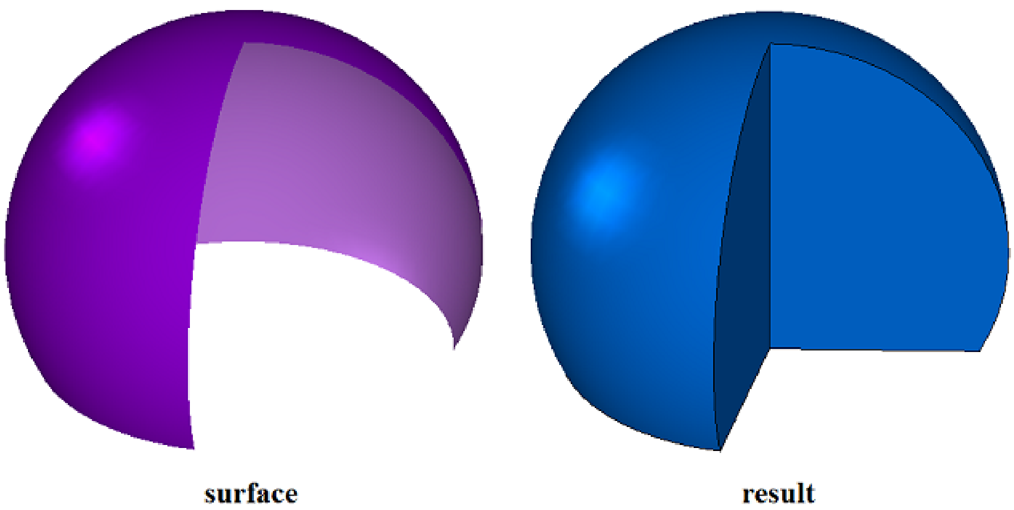


Fig. M.1.2.1.

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

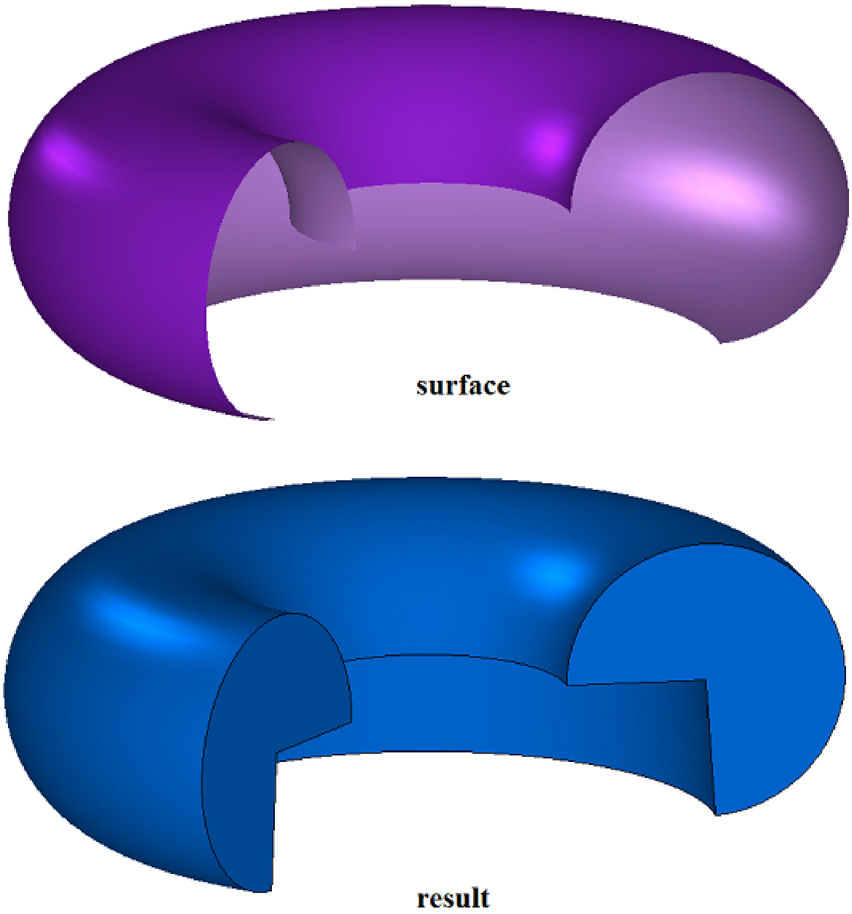


Fig. M.1.2.2.

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

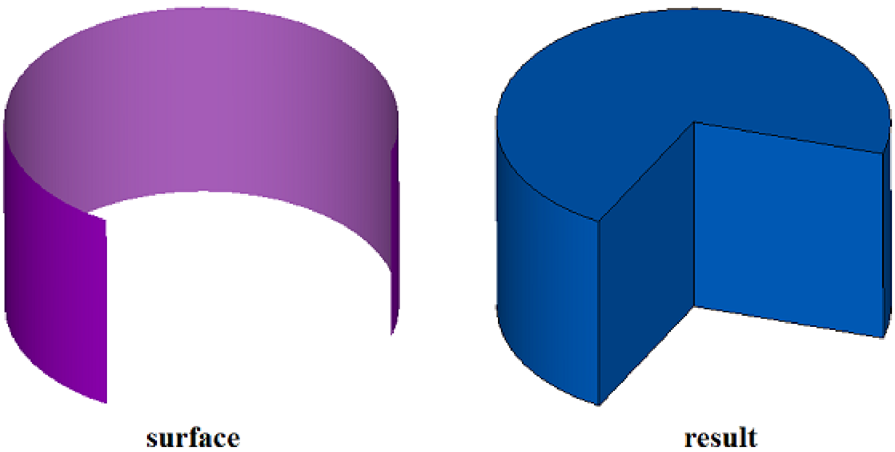


Fig. M.1.2.3.

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

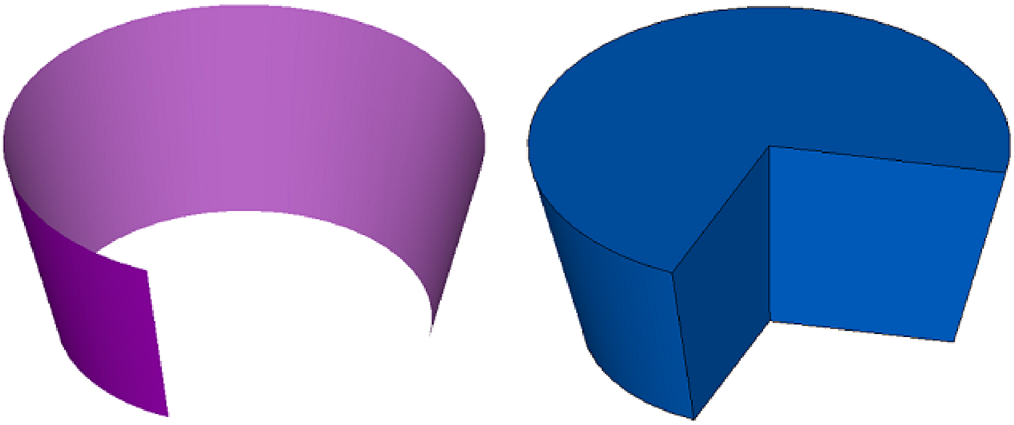


Fig. M.1.2.4.

If these are cyclically closed surfaces, then the bodies 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 body. This constructor contains all data required to construct the body. MbRevolutionSolid constructor is declared in cr_revolution_solid.h file.

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

M.1.3. Constructing an Extrusion Body

The 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 body.

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 bodies 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 body. 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 body belongs to the type of motion bodies, which are constructed by moving a generating curve along a guiding curve. A line segment is a guiding curve for an extruded body. An extruded body 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 body 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 body 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 Fig. M.1.3.1, you can see the data used for construction, as well as the scheme to inherit the parameters of constructed extrusion body (ExtrusionValues & *params*).

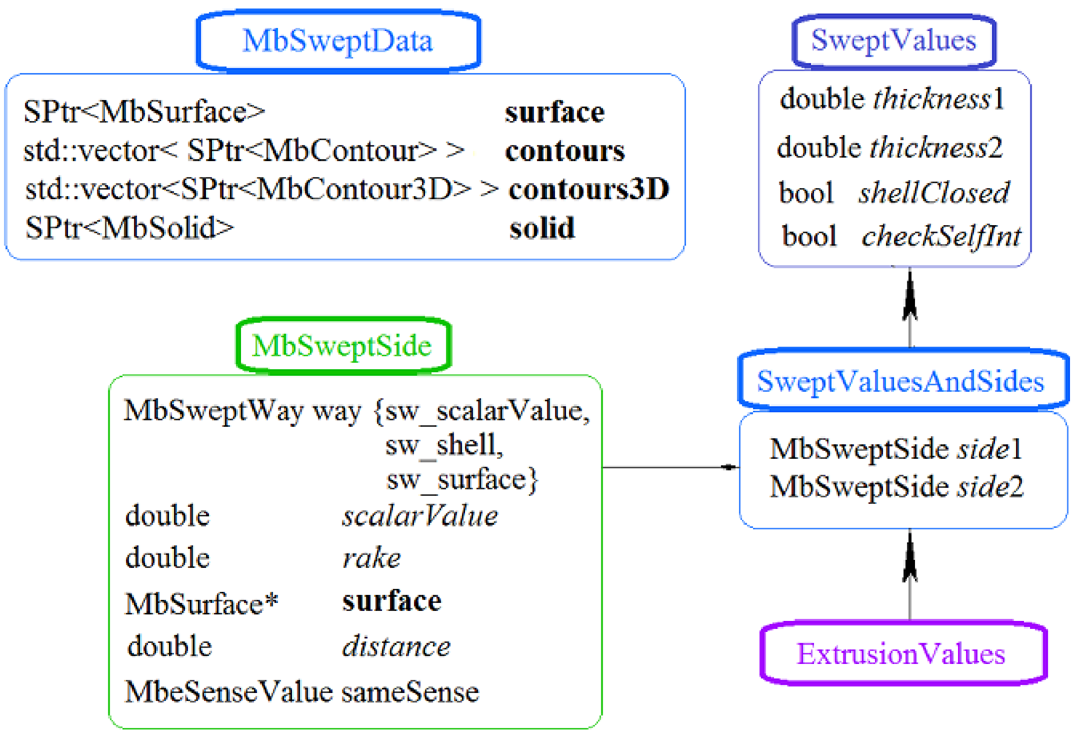


Fig. M.1.3.1.

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

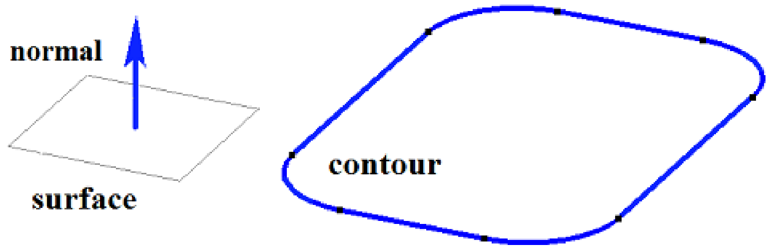


Fig. M.1.3.2.

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

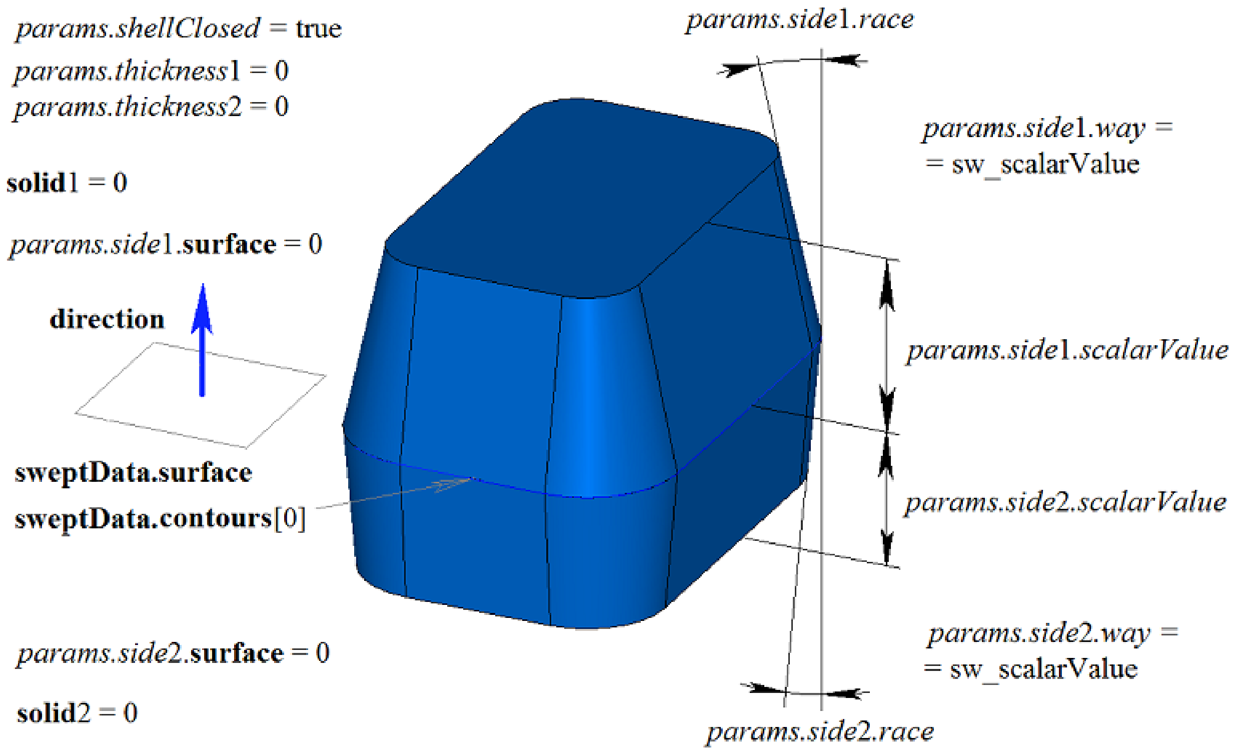


Fig. M.1.3.3.

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

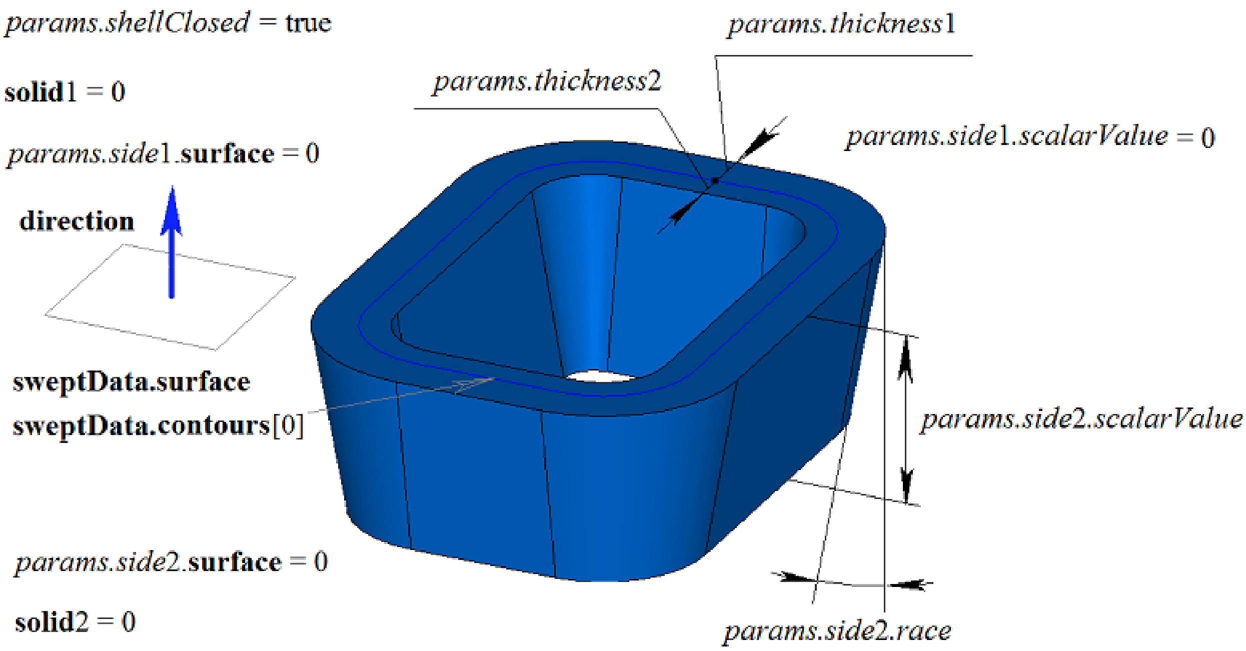


Fig. M.1.3.4.

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

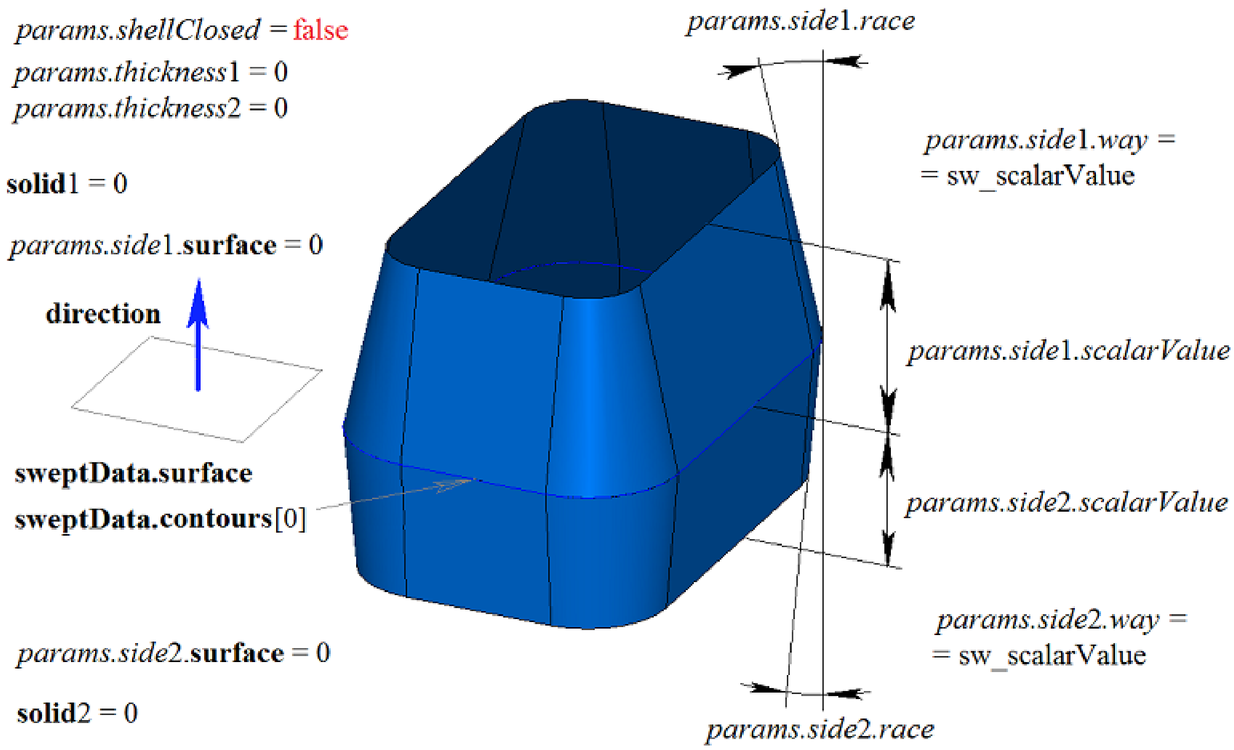


Fig. M.1.3.5.

In Fig. M.1.3.6, you can see two-dimensional **contour**, flat **surface** (MbPlane) as well as two bodies (**solid**1 and **solid**2) that will be used to construct an extruded body. For construction **solid**1 and **solid**2 bodies 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.

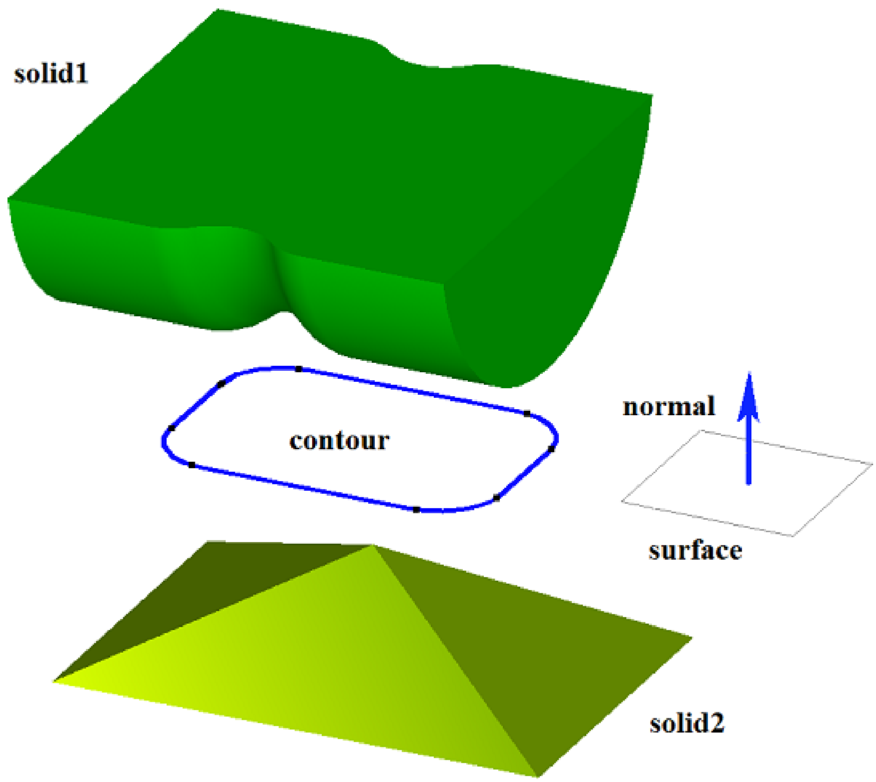


Fig. M.1.3.6.

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

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

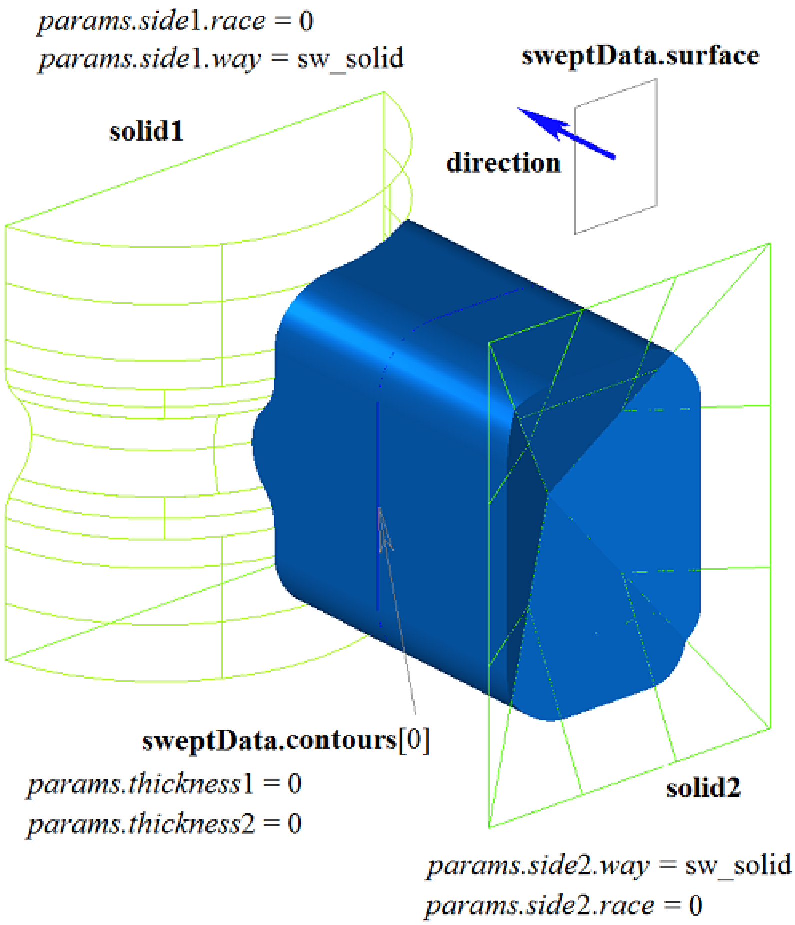


Fig. M.1.3.7.

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

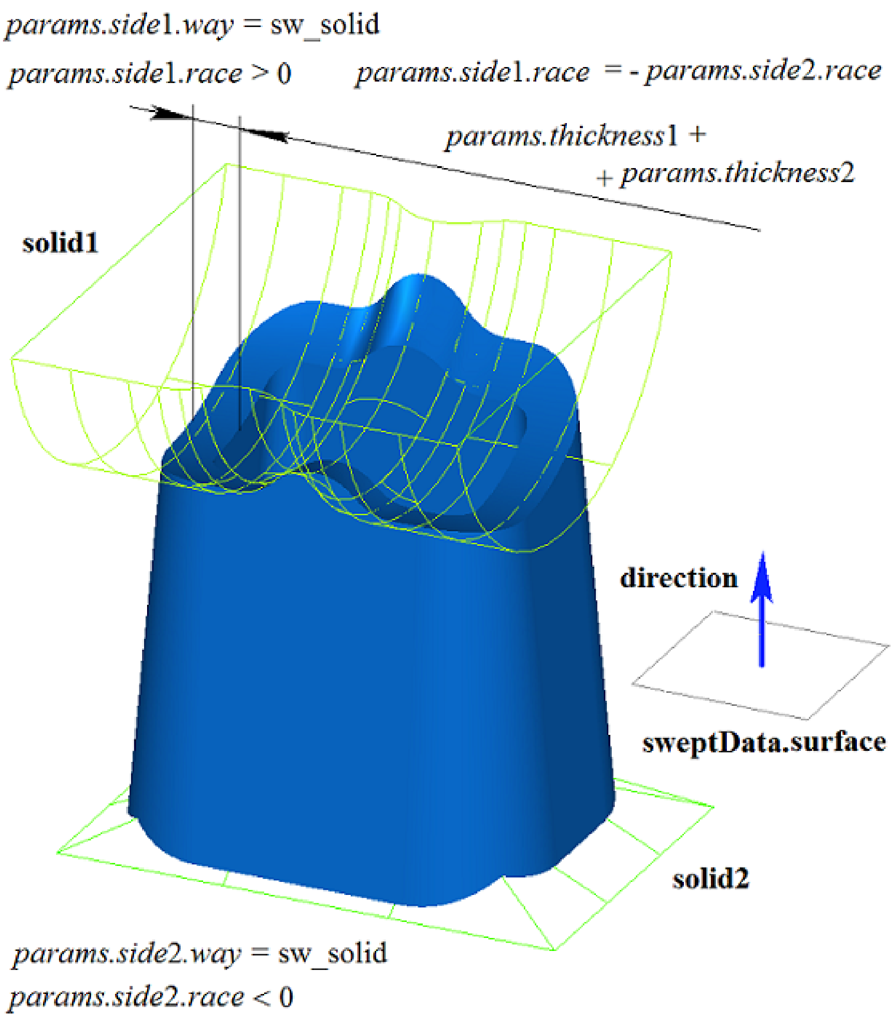


Fig. M.1.3.8.

In Fig. 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 body). 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 body 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.

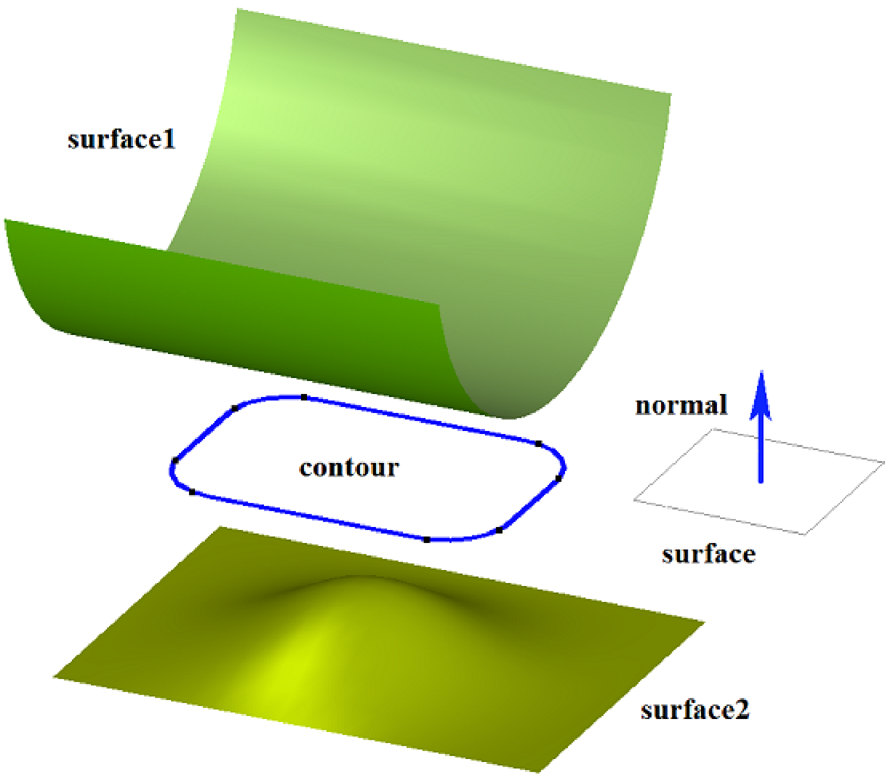


Fig. M.1.3.9.

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

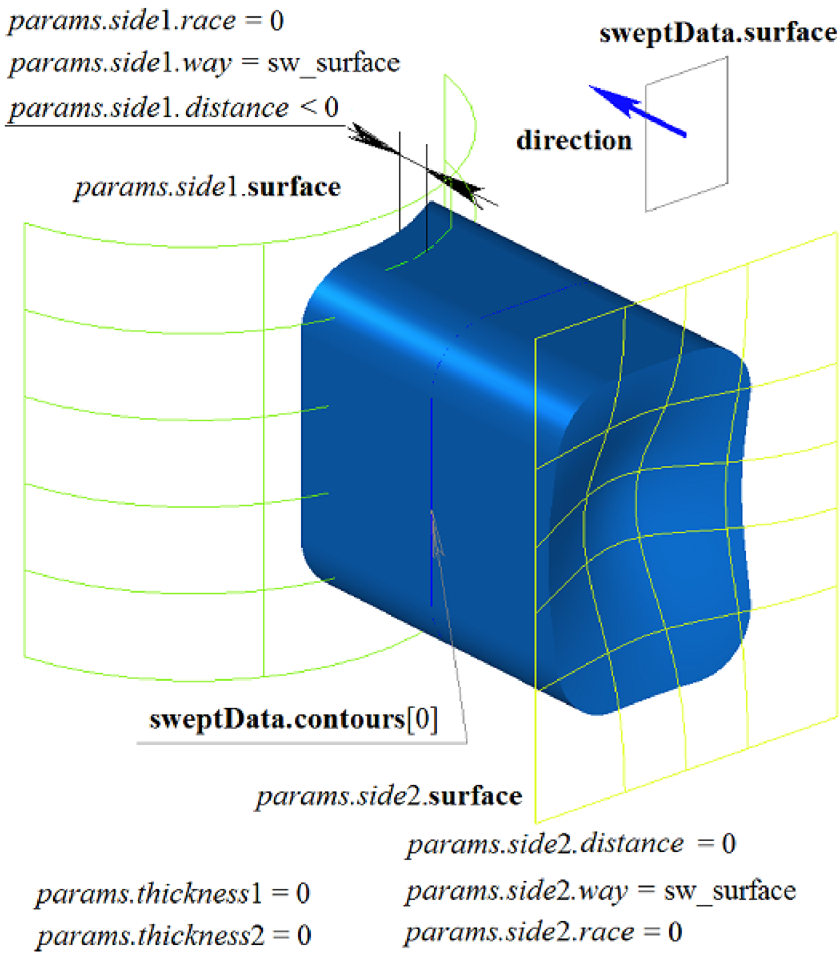


Fig. M.1.3.10.

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

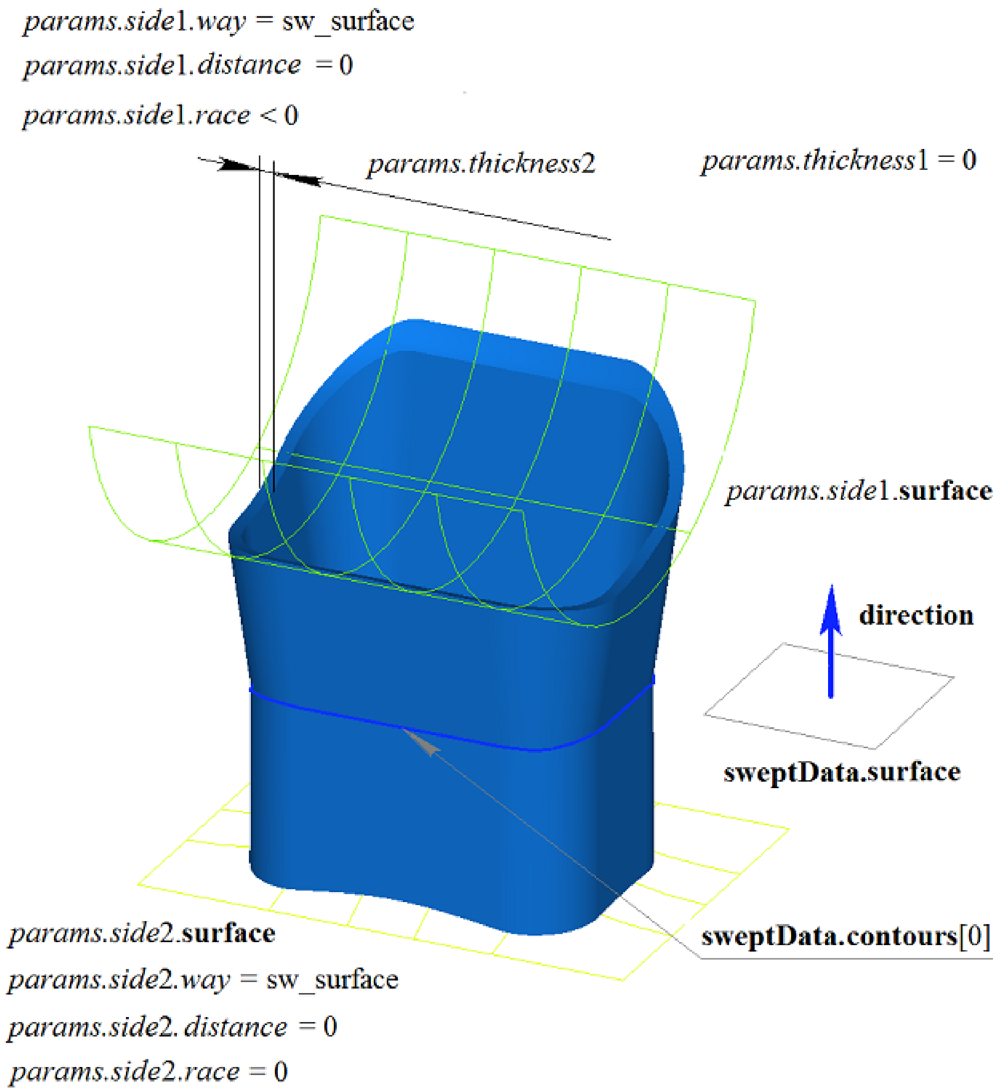


Fig. M.1.3.11.

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

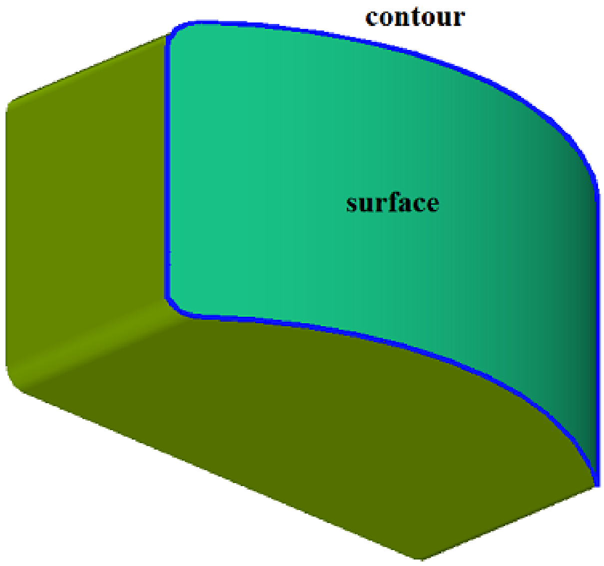


Fig. M.1.3.12.

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

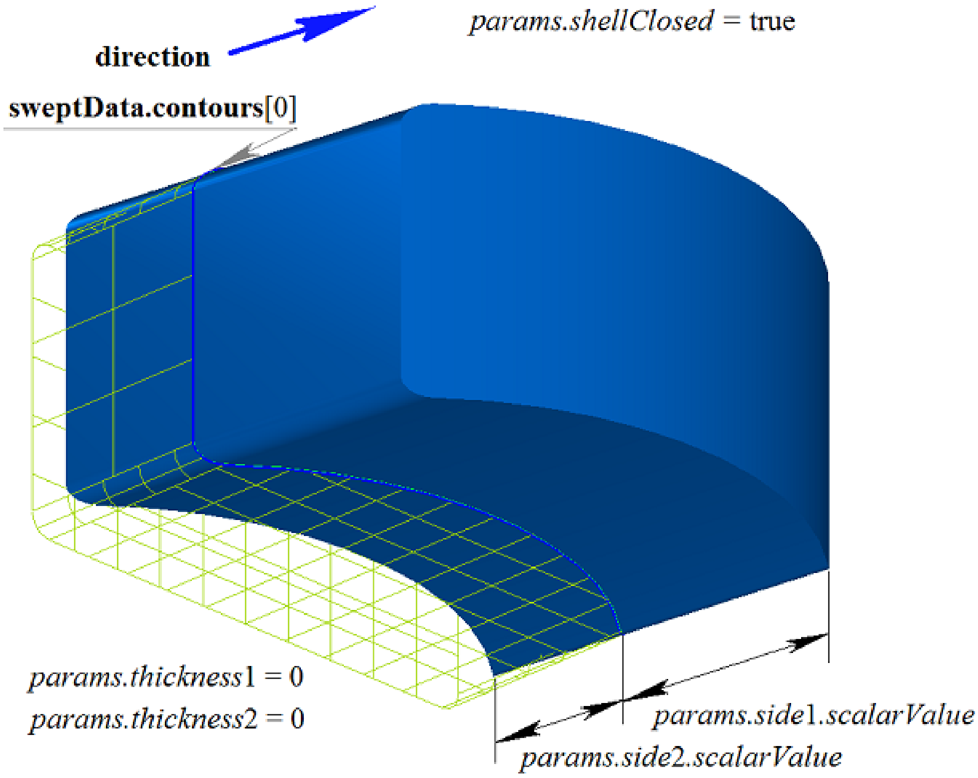


Fig. M.1.3.13.

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

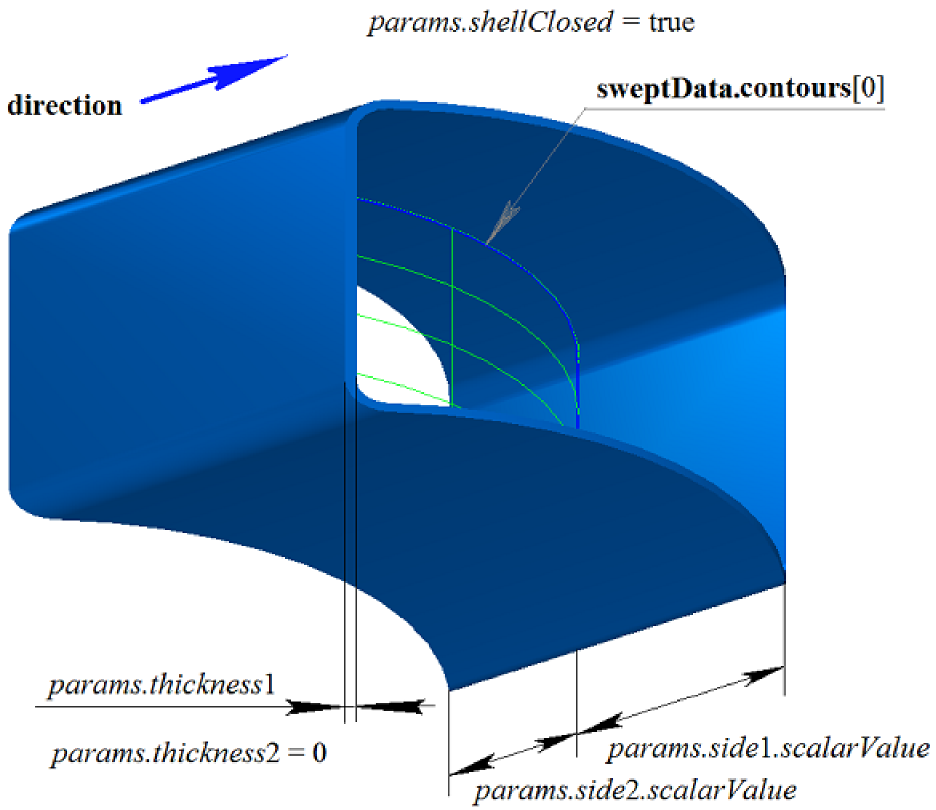


Fig. M.1.3.14.

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

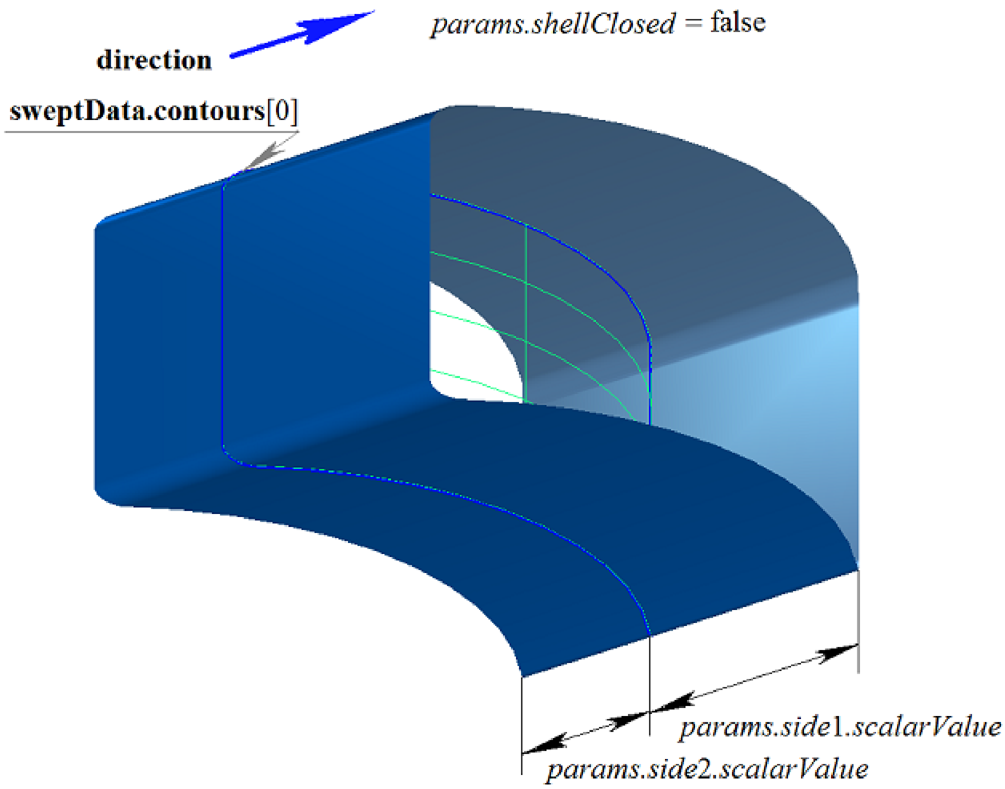


Fig. M.1.3.15.

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 Fig. M.1.3.16, you can see a set of non-intersecting two-dimensional **contours** and **surface** (MbPlane) flat surface.

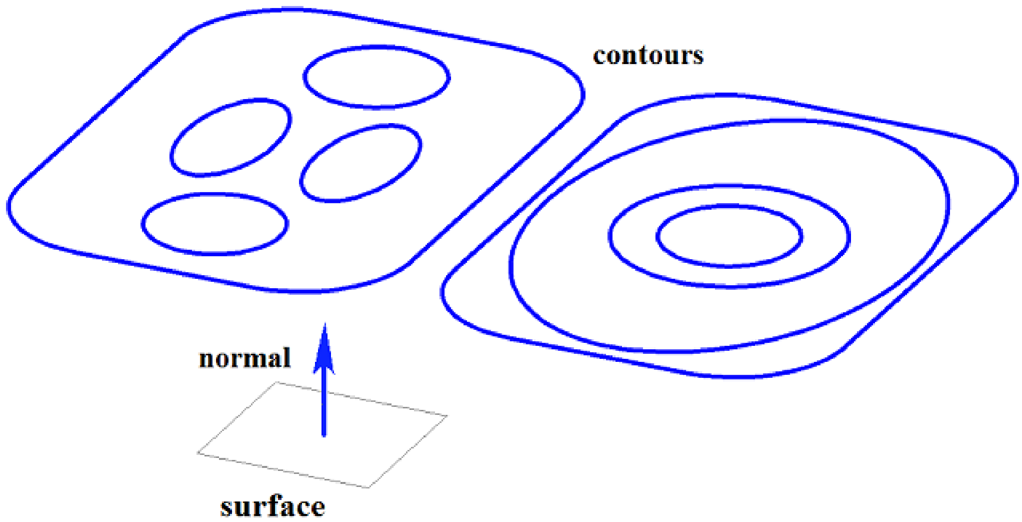


Fig. M.1.3.16.

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

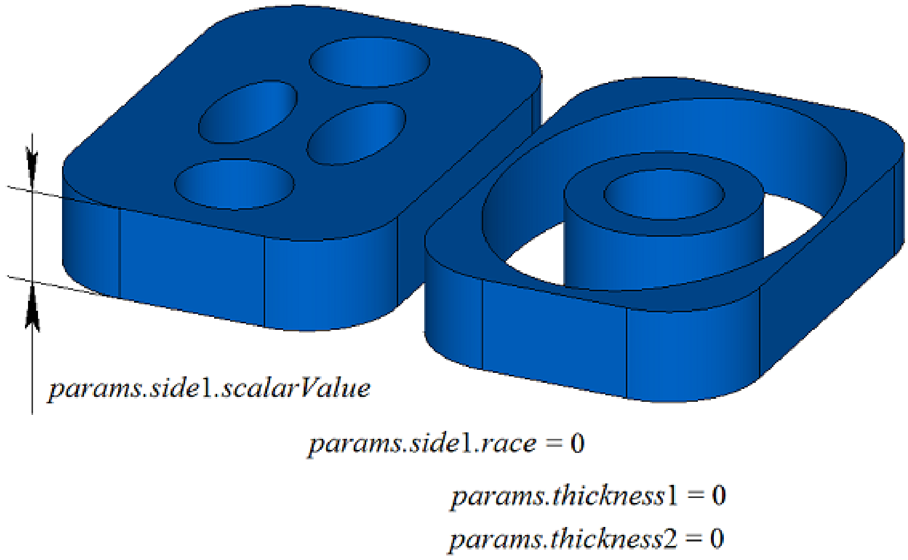


Fig. M.1.3.17.

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

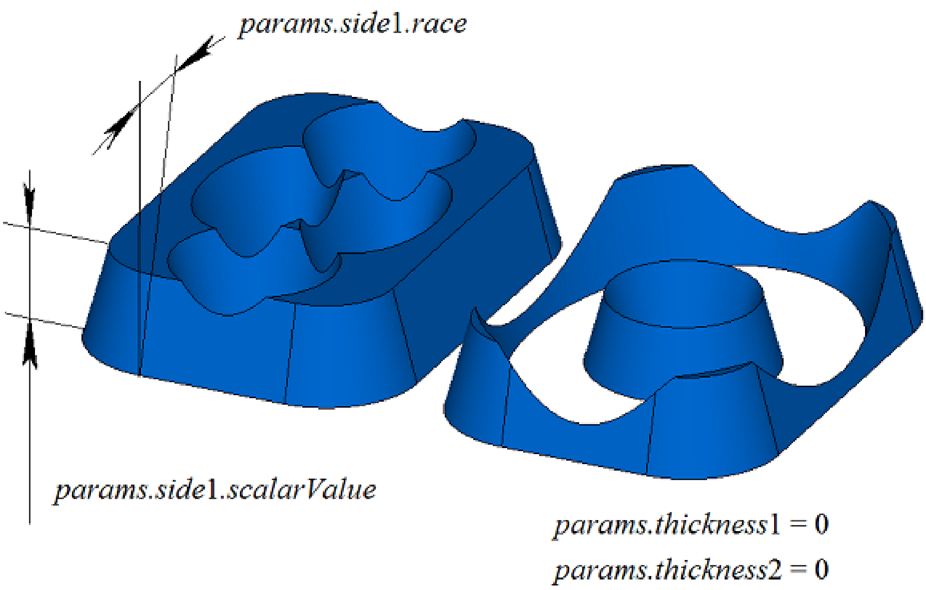


Fig. M.1.3.18.

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

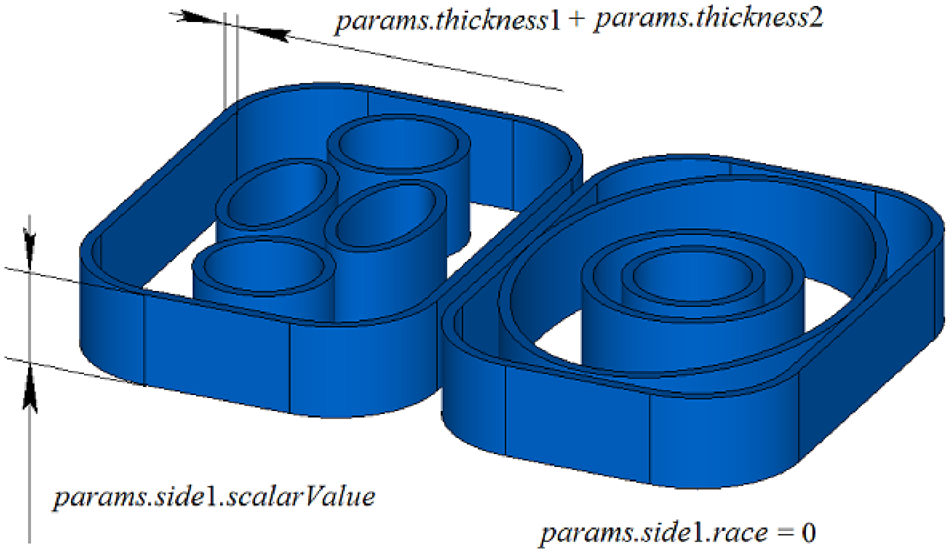


Fig. M.1.3.19.

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

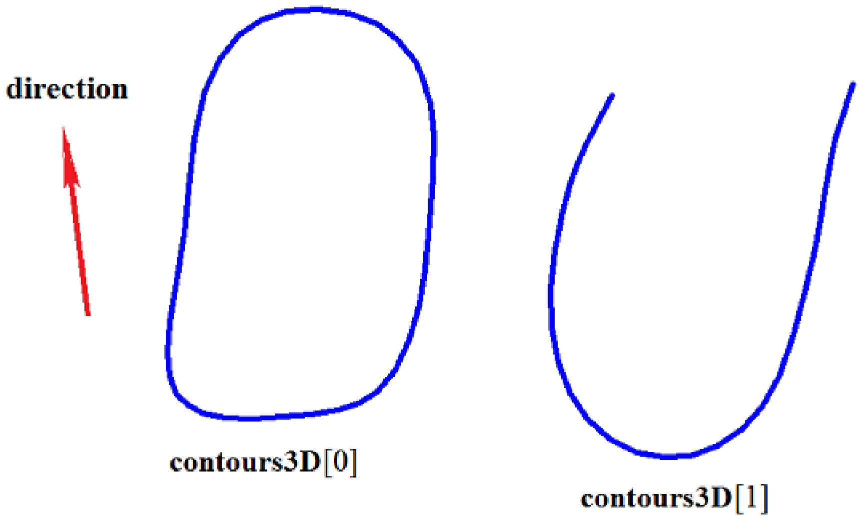


Fig. M.1.3.20.

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

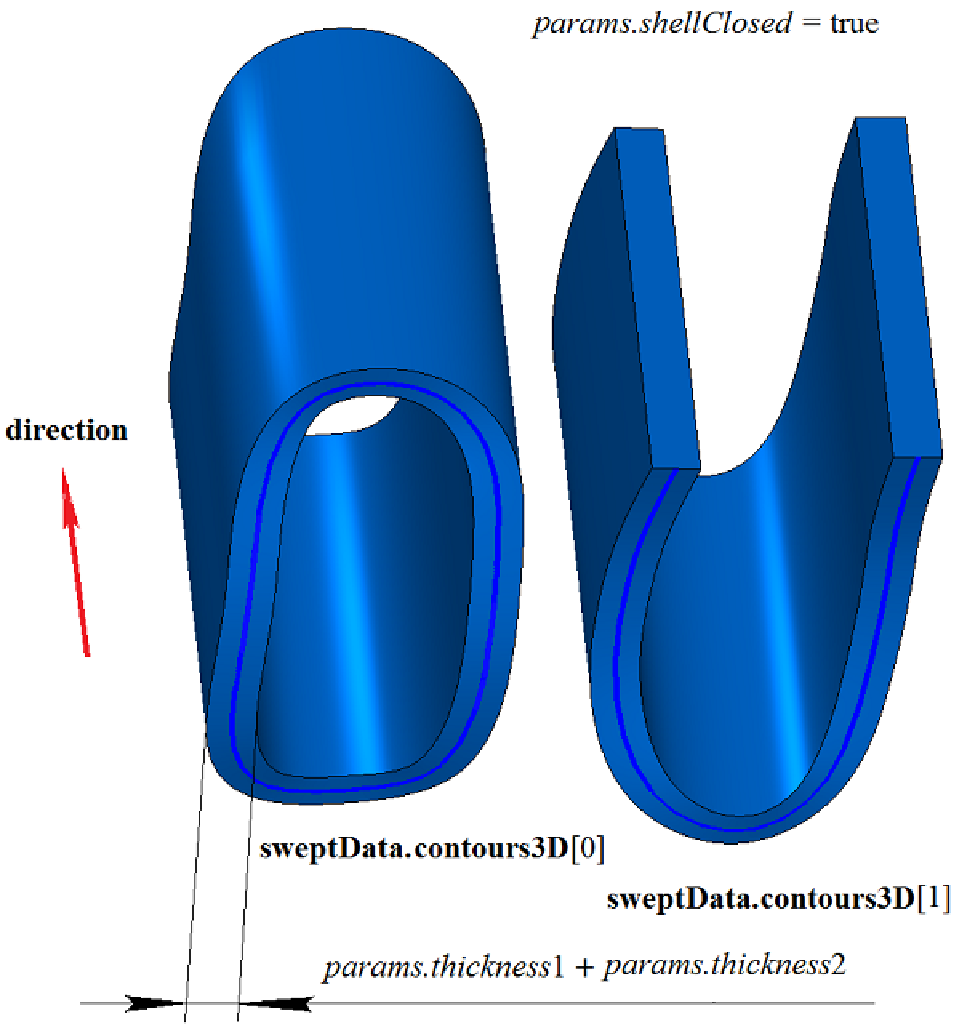


Fig. M.1.3.21.

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

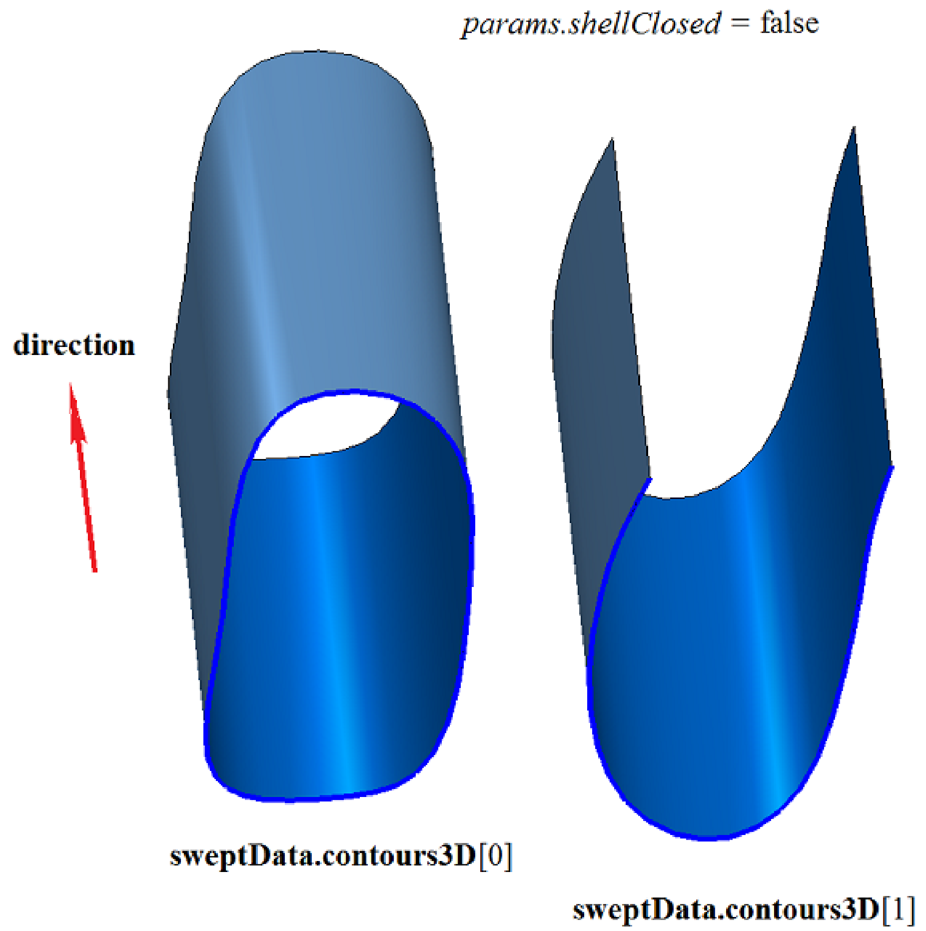


Fig. M.1.3.22.

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

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

M.1.4. Constructing a Revolution Body

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

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 body.

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 body belongs to the type of motion bodies which are constructed by moving a curve generator along a guiding curve. A circle or an arc can be a guidng curve for rotation body. Rotation body 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 body. *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 body 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 body. If *params.shape*=1, then the constructed body has torus topology. If *params.shape*=0, then the body has sphere topology.

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

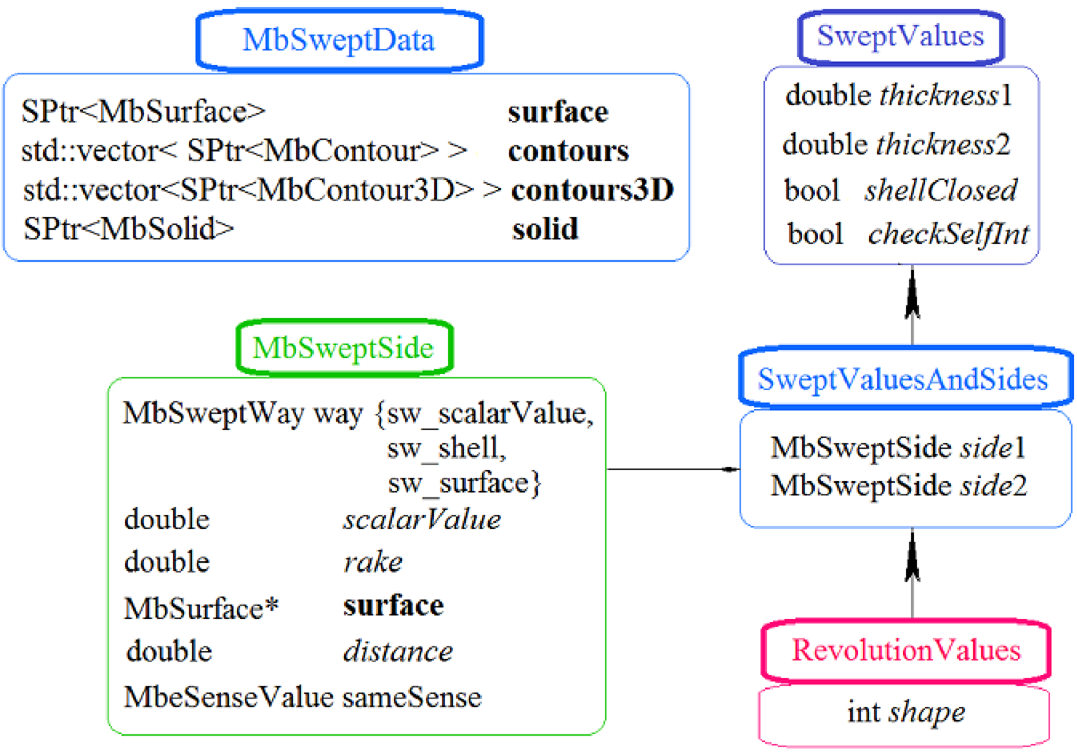


Fig. M.1.4.1.

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

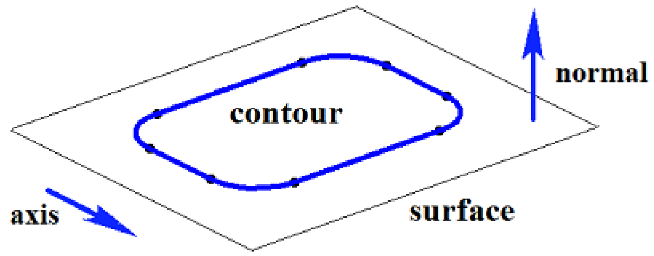


Fig. M.1.4.2.

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

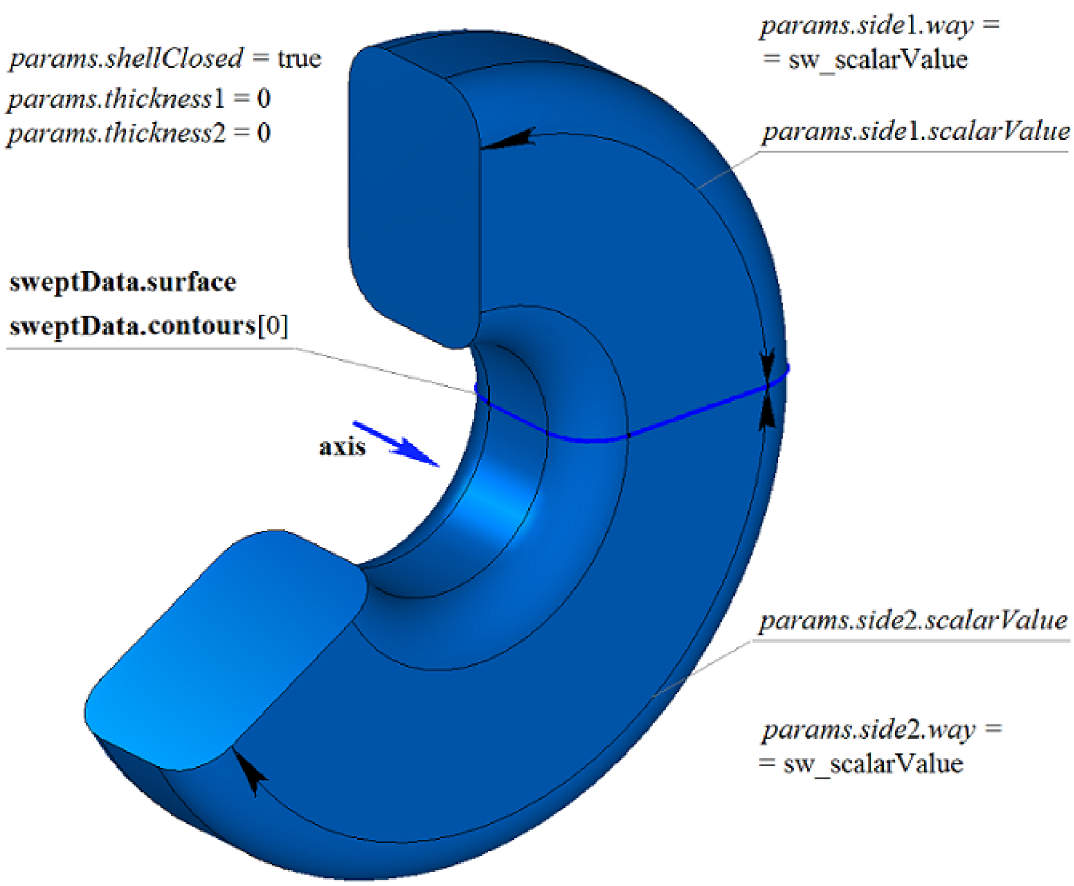


Fig. M.1.4.3.

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

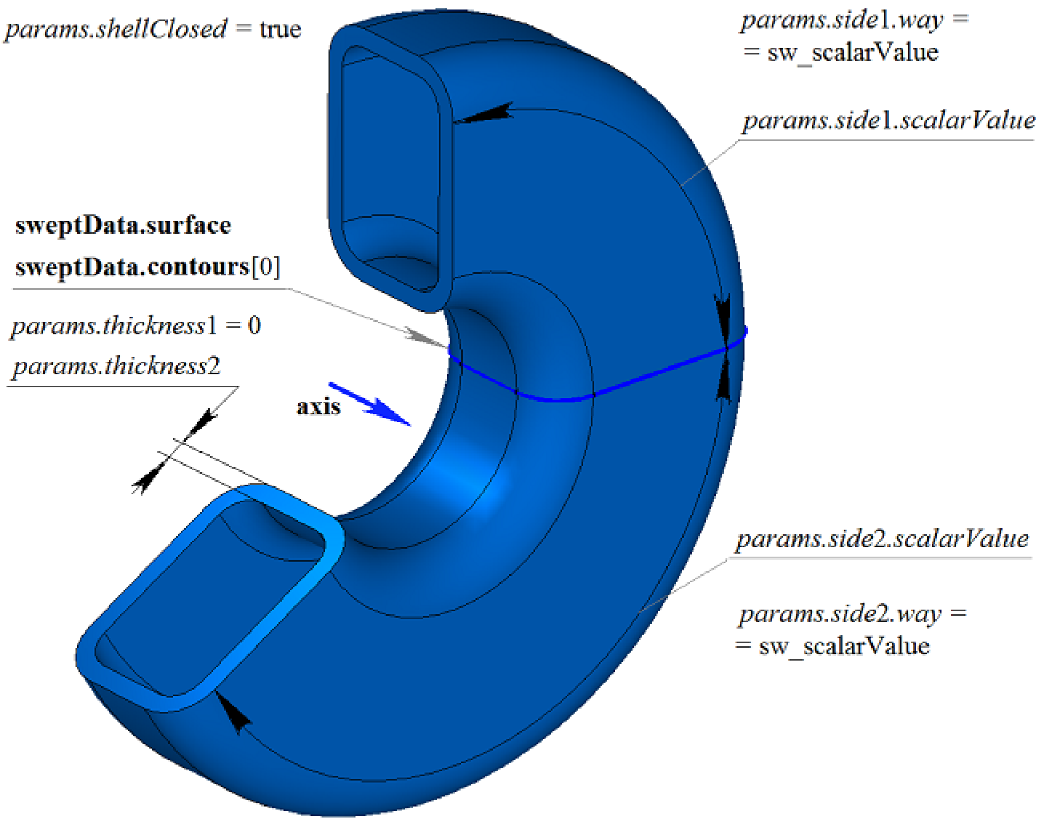


Fig. M.1.4.4.

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

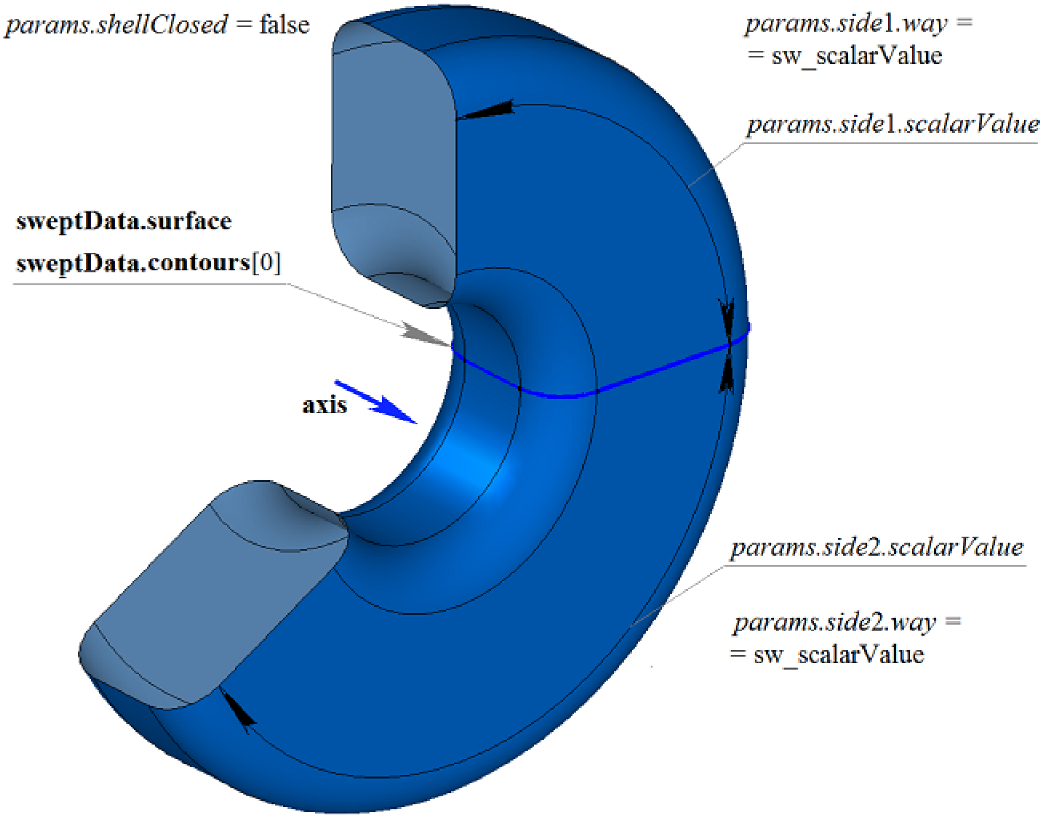


Fig. M.1.4.5.

In Fig. 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 body. 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 body is cut off by specified or equidistant surfaces if *params.side*1.*distance* or *params.side*2.*distance* are not equal to zero.

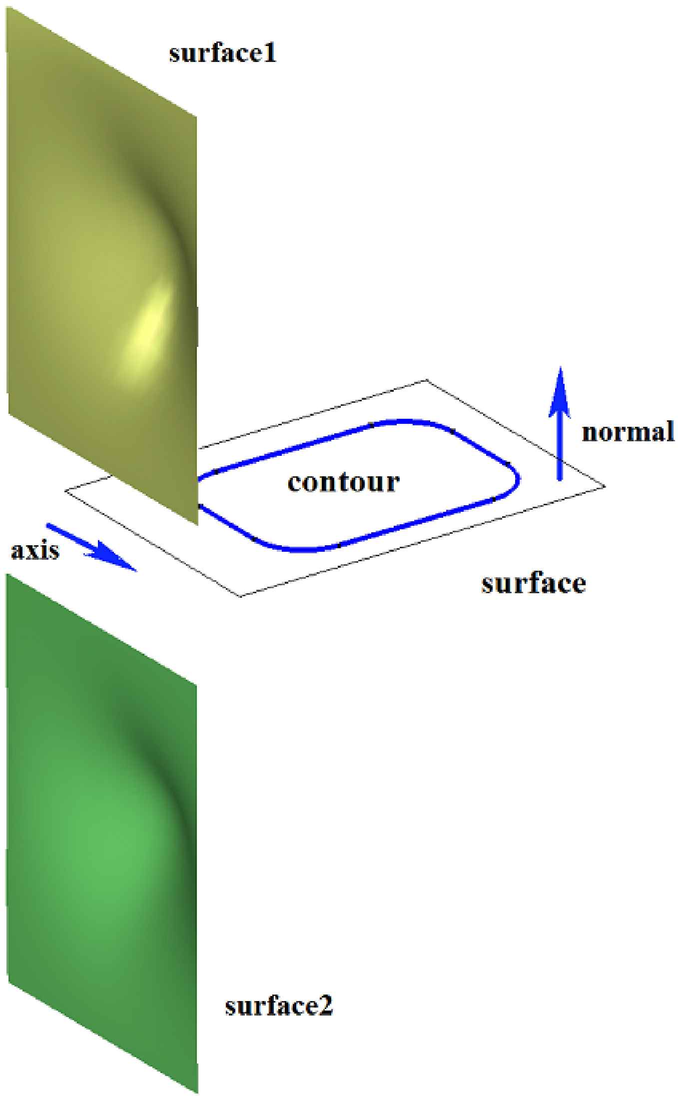


Fig. M.1.4.6.

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

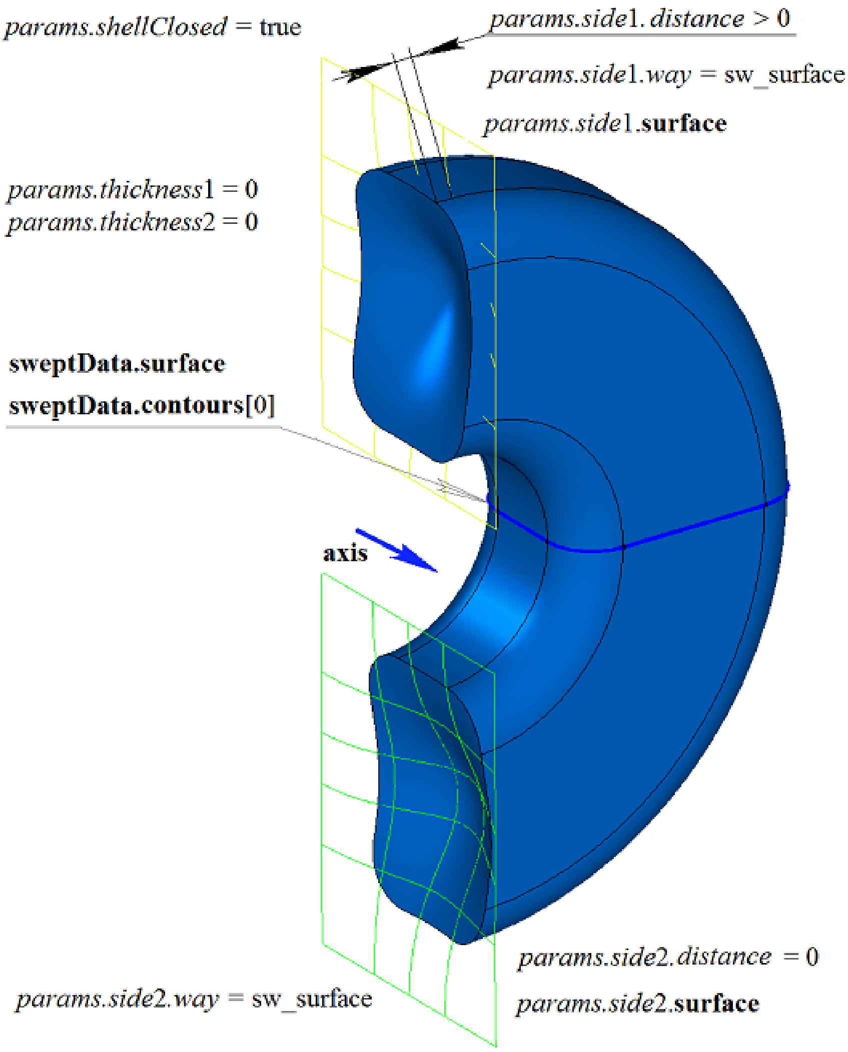


Fig. M.1.4.7.

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

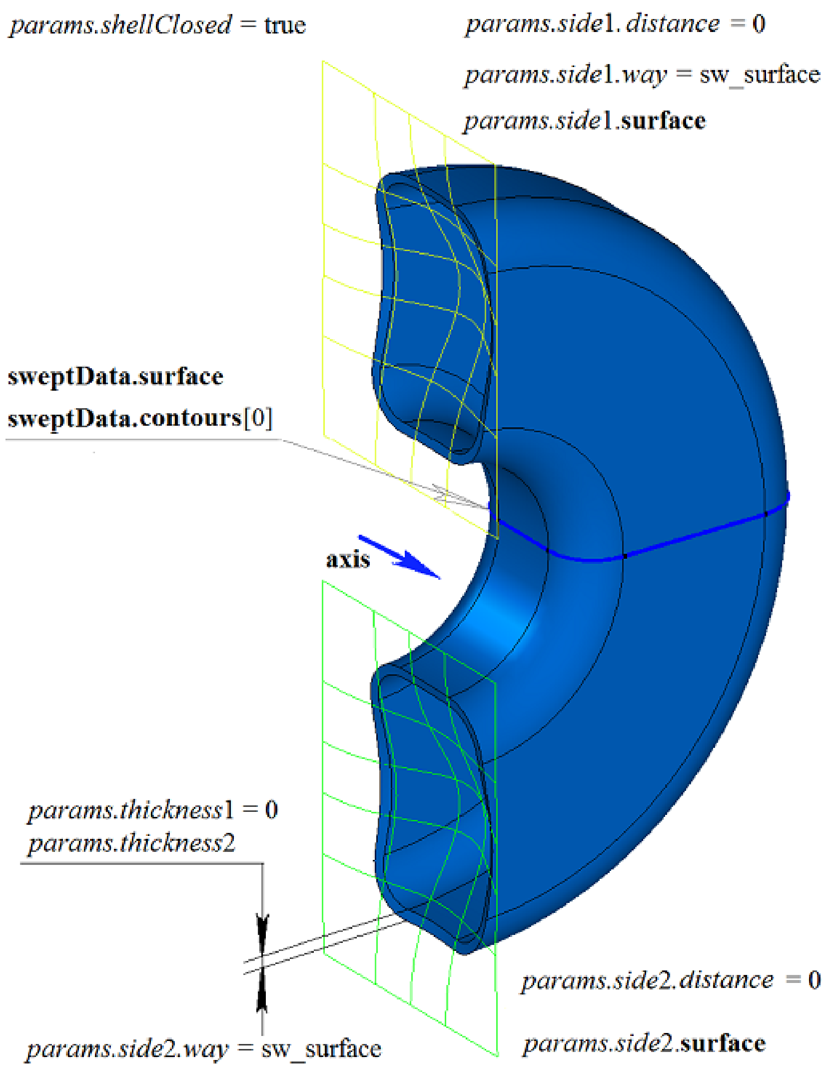


Fig. M.1.4.8.

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

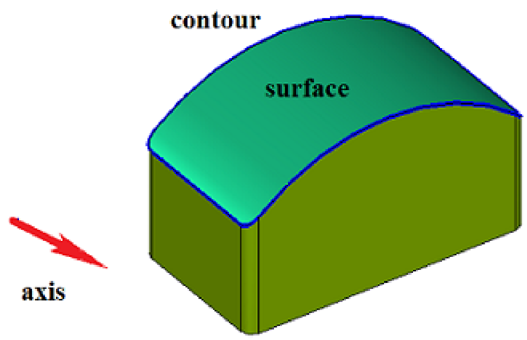


Fig. M.1.4.9

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

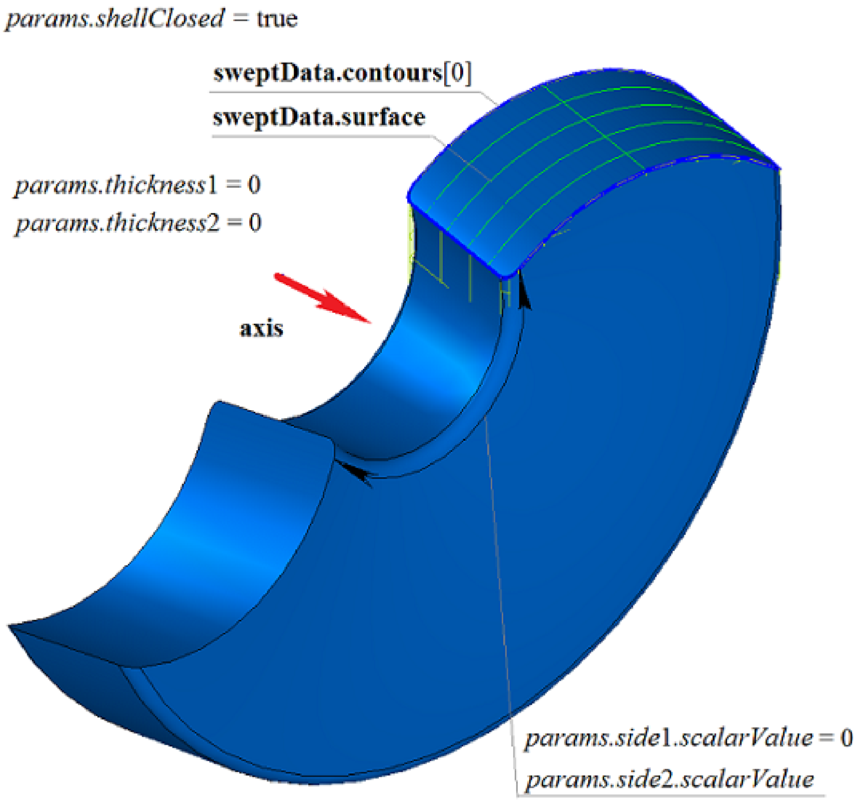


Fig. M.1.4.10.

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

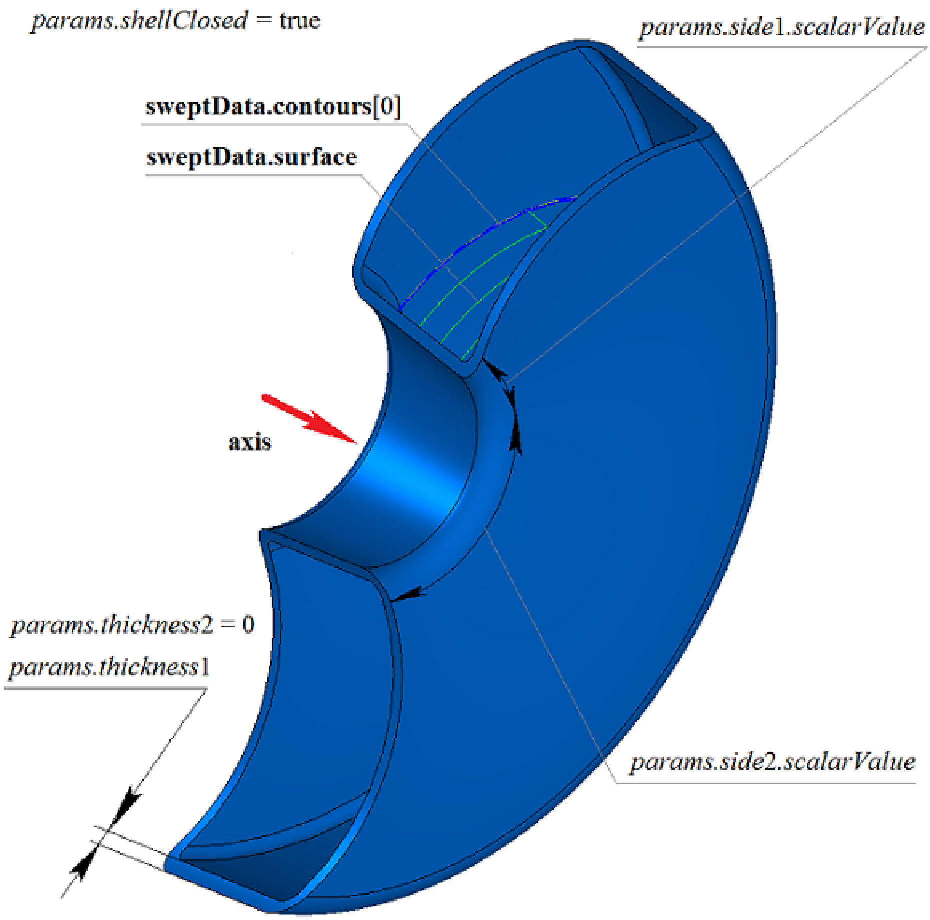


Fig. M.1.4.11.

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

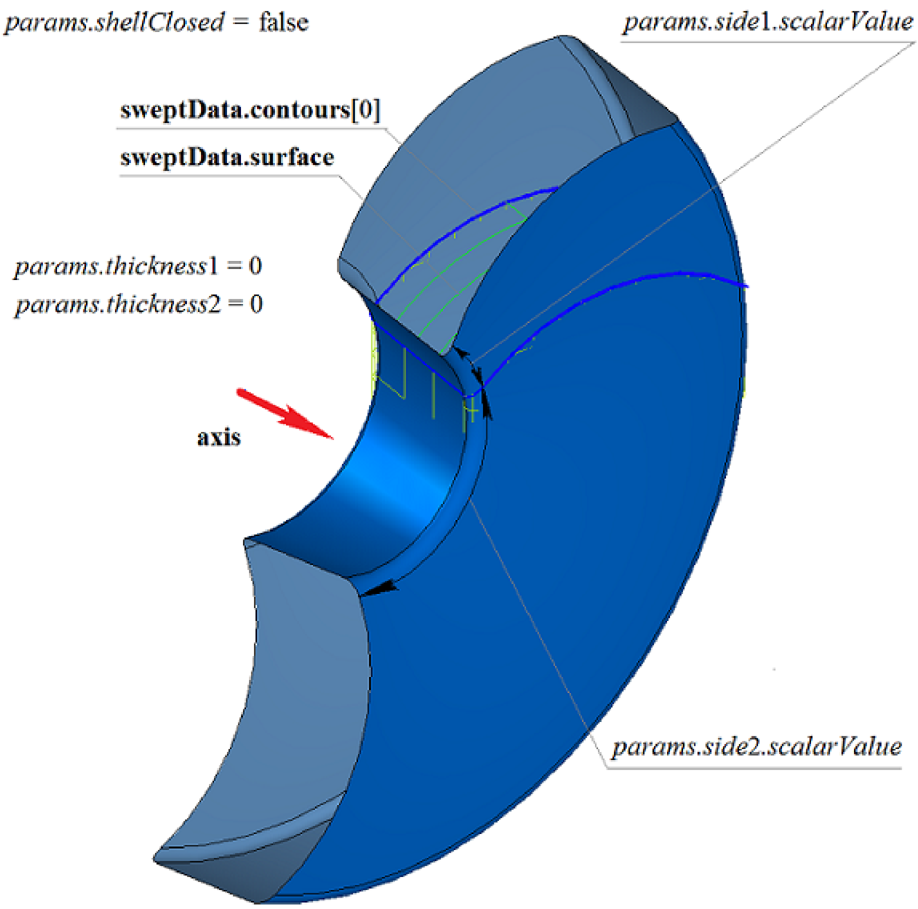


Fig. M.1.4.12.

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 Fig. M.1.4.13, you can see a set of non-intersecting two-dimensional **contours** and a flat **surface** (MbPlane).

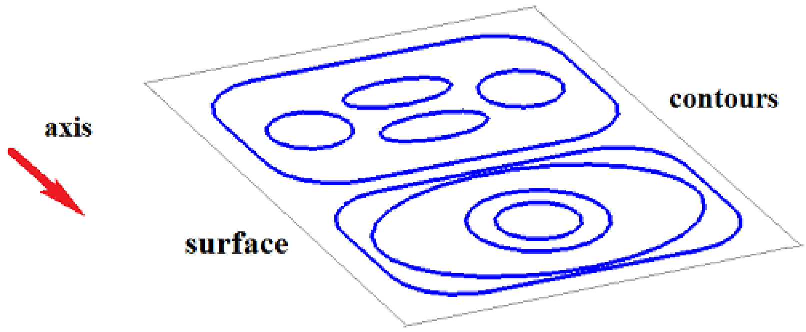


Fig. M.1.4.13.

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

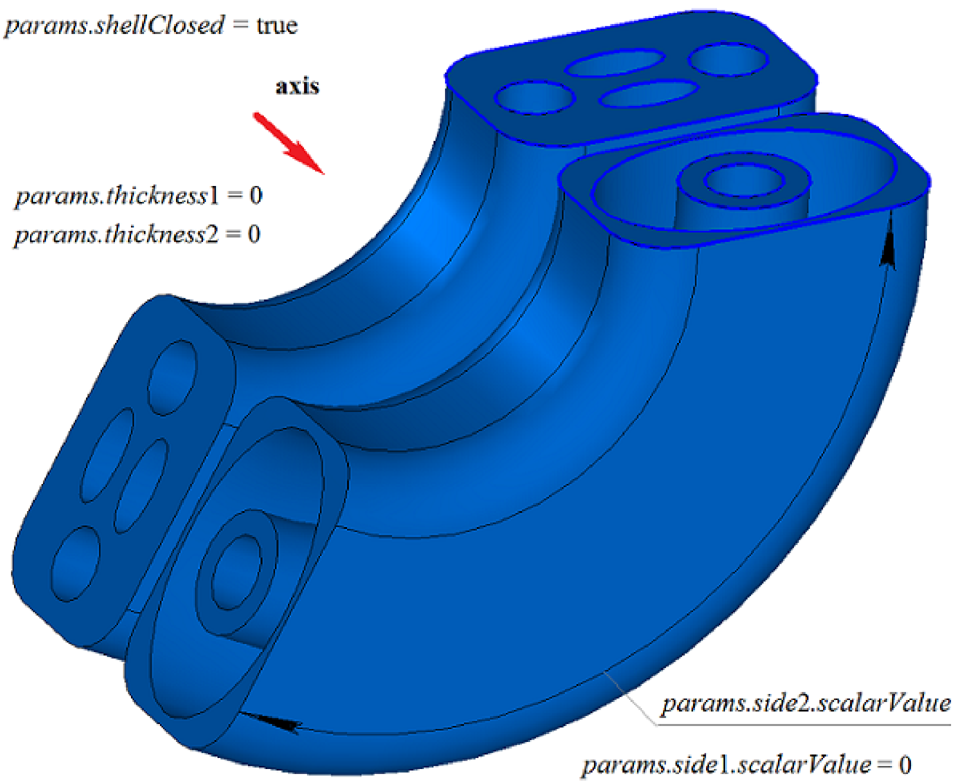


Fig. M.1.4.14.

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

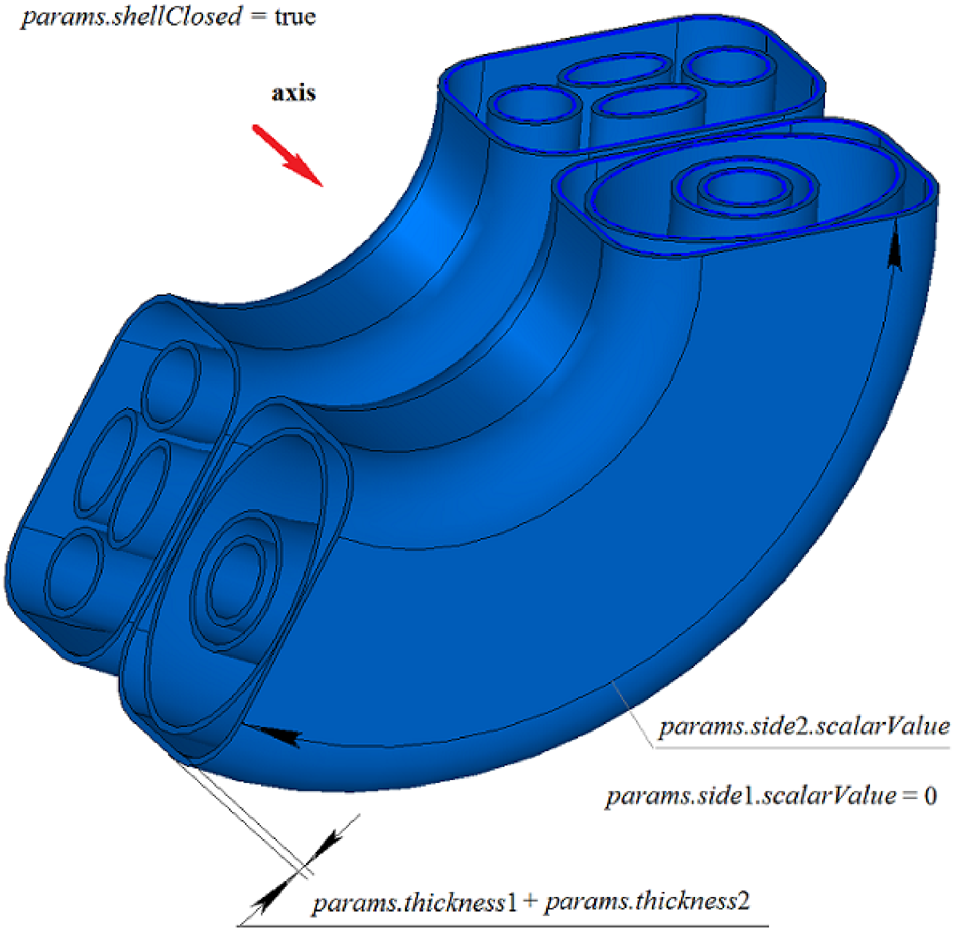


Fig. M.1.4.15.

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

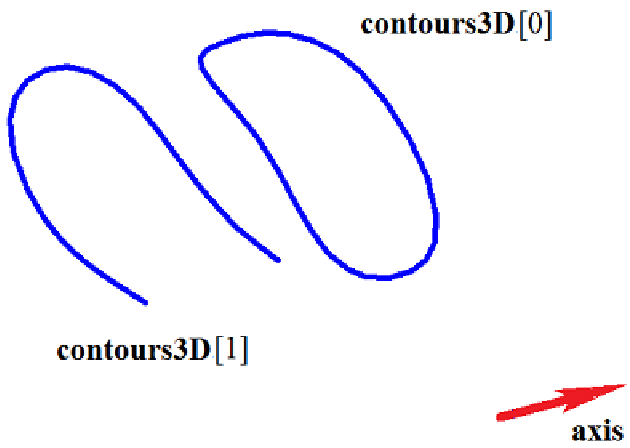


Fig. M.1.4.16.

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

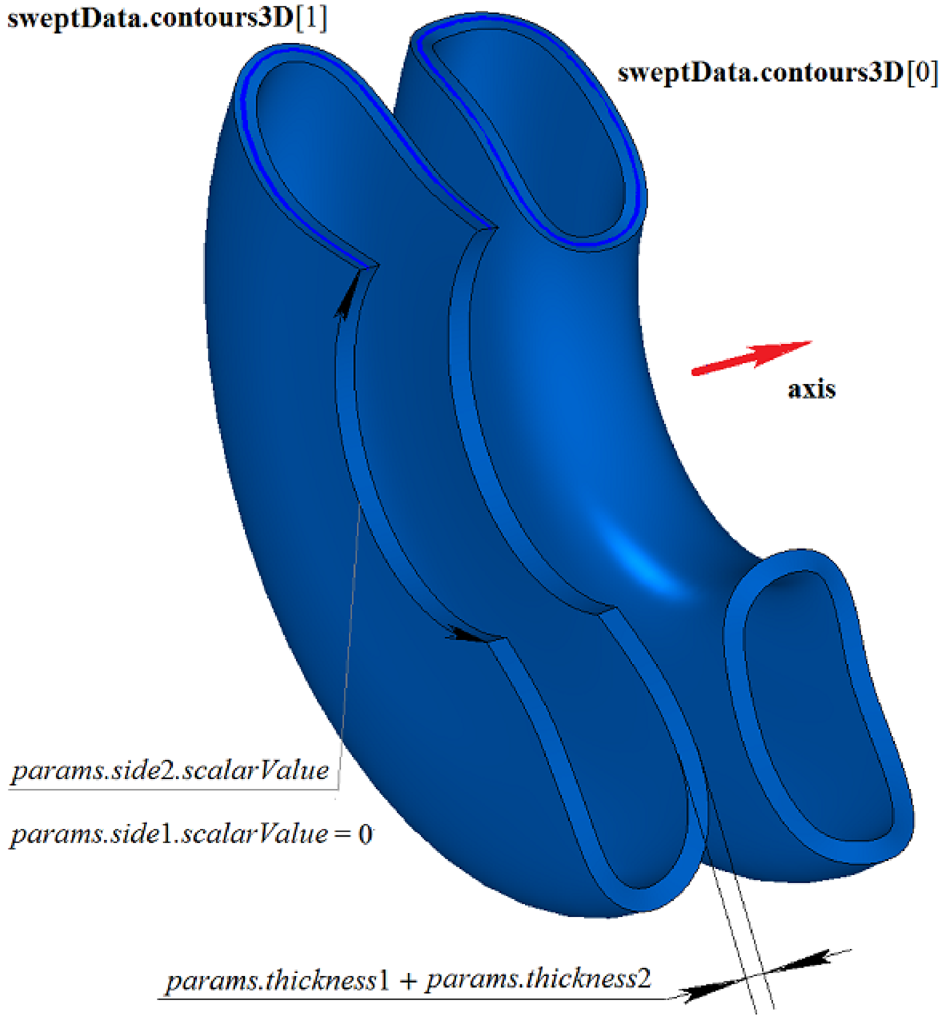


Fig. M.1.4.17.

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

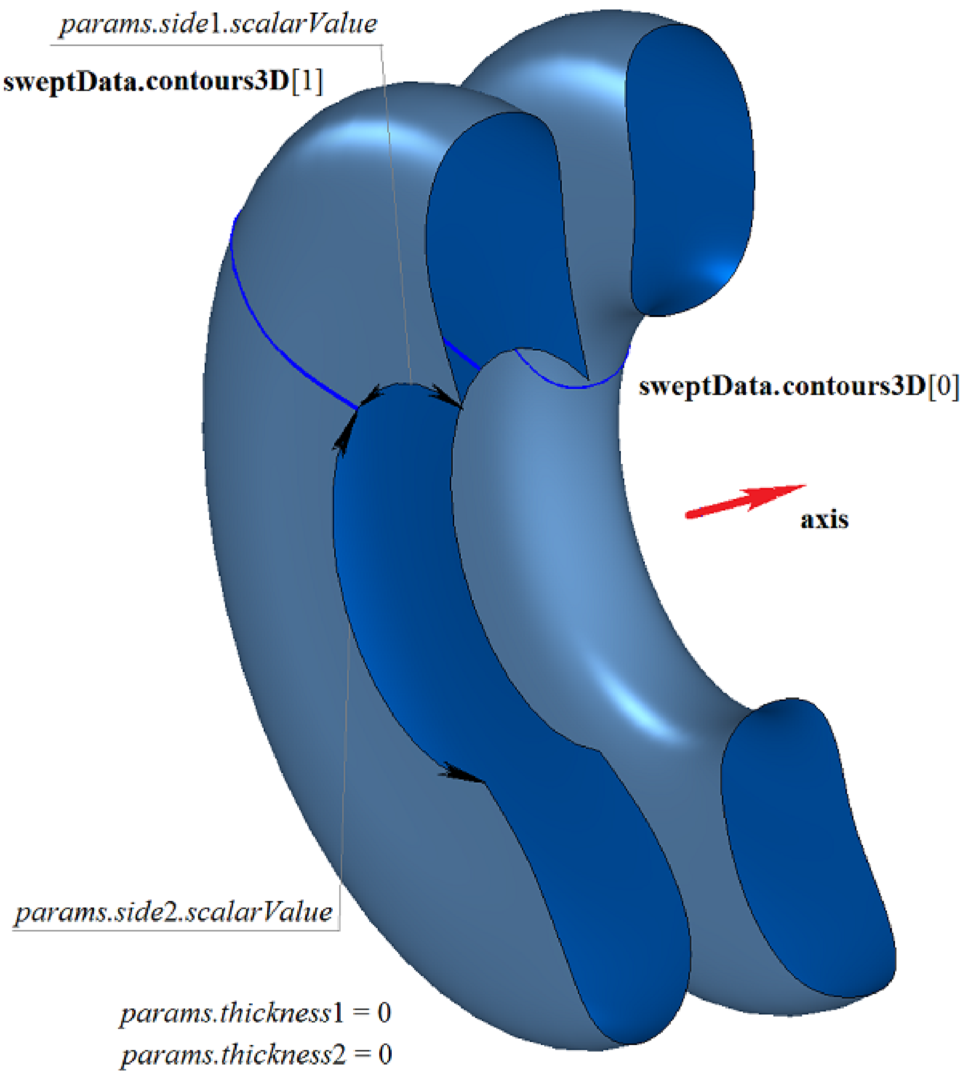


Fig. M.1.4.18.

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

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

M.1.5. Constructing a Swept Body

The 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 body 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 body.

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

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 body walls and their thickness and data whether the constructed body is closed. *params.thickness*1 and *params.thickness*2 parameters define wall thickness of the constructed thin-walled body. *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 body 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 bodies. The movement can be performed in three ways. Movement mode is defined by *params.mode* parameter. If *params.mode*=0, then the movement of generator curves is coplanar. If *params.mode*=1, then moving generator curves maintain their position in the local coordinate system, which is tangent to the generator curve. If *params.mode*=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 Fig. M.1.5.1, you can see the data used for construction, as well as parameters inheritance scheme for constructed swept body ExtrusionValues & *params*.

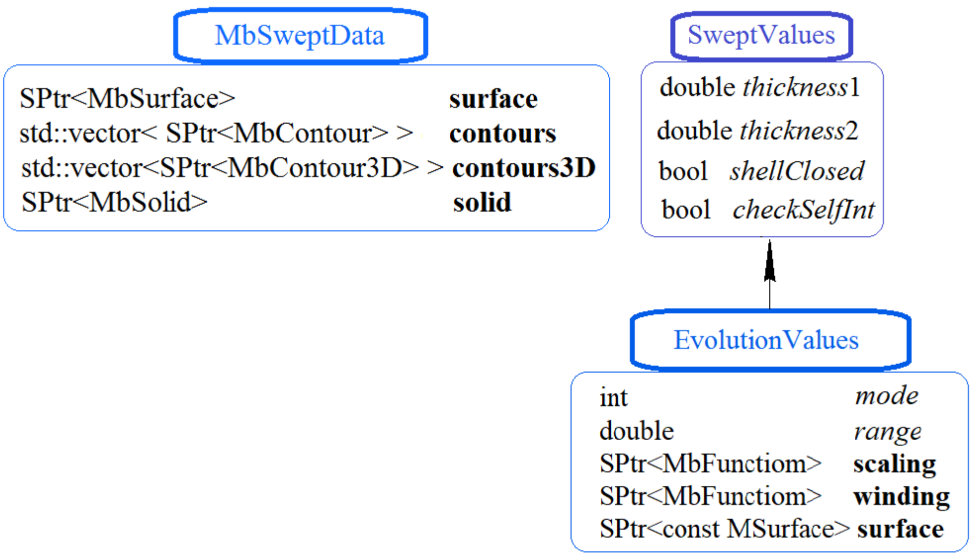


Fig. M.1.5.1.

The generating contour can change its shape in the process of moving along the trajectory, as well as change the scale according to a given law and rotate according to a given law. The *params.range* parameter determines the offset of the points of the generating contour at the end of the trajectory. The offset of the generating curve is performed according to the linear law. The *params.scaling* function defines the scaling law of the generating contour. The *params.winding* function defines the rotation law of the generating contour. The *params.surface* used to control the **spine** guide curve.

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

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

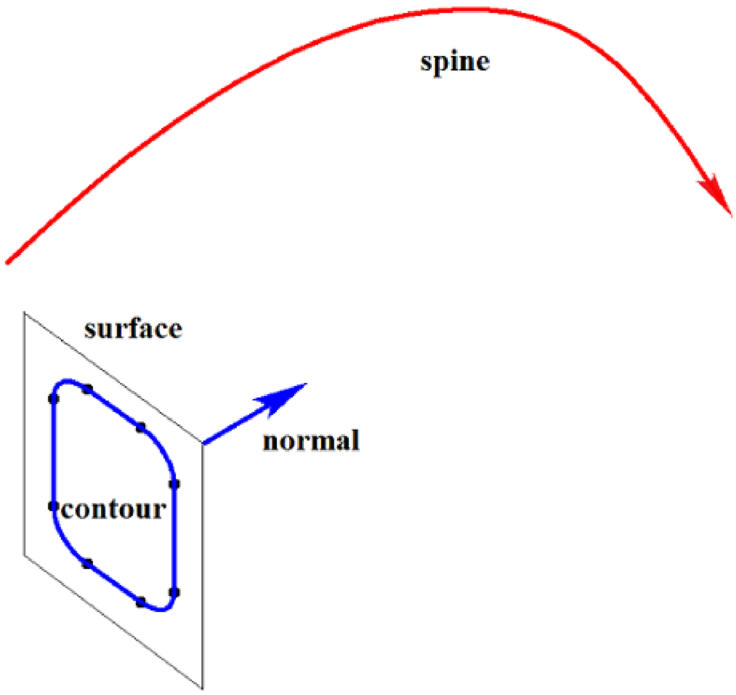


Fig. M.1.5.2.

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

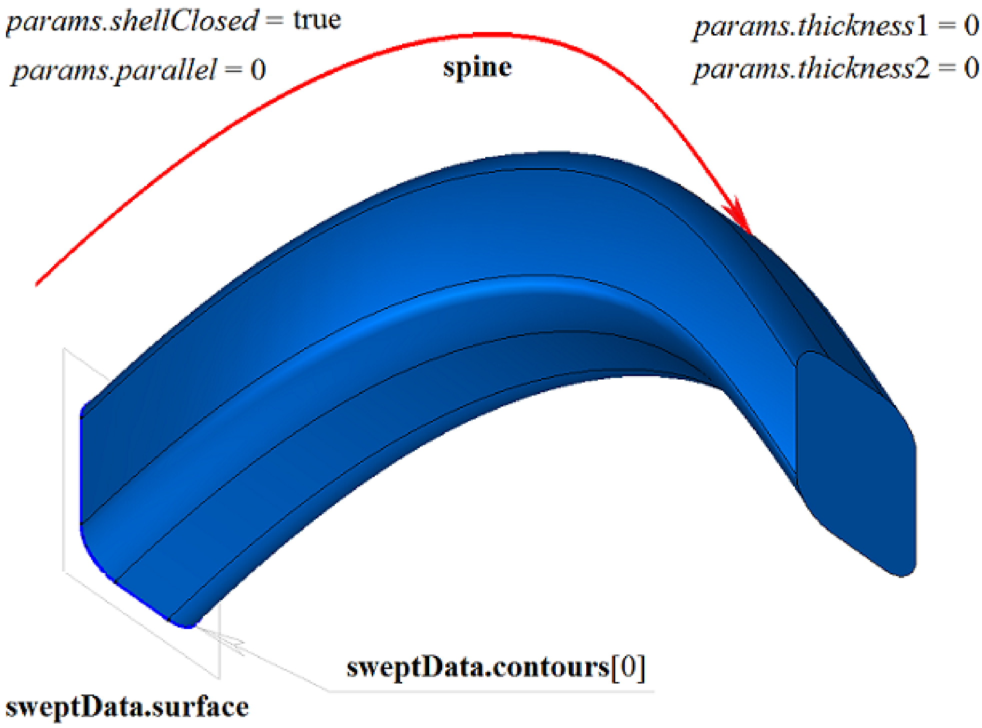


Fig. M.1.5.3.

In Fig. M.1.5.4, you can see a swept body that was constructed by moving a contour along the guiding curve shown in Fig. M.1.5.2, using the method defined by *params.mode*=1 parameter. In this case, the plane of the body 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.

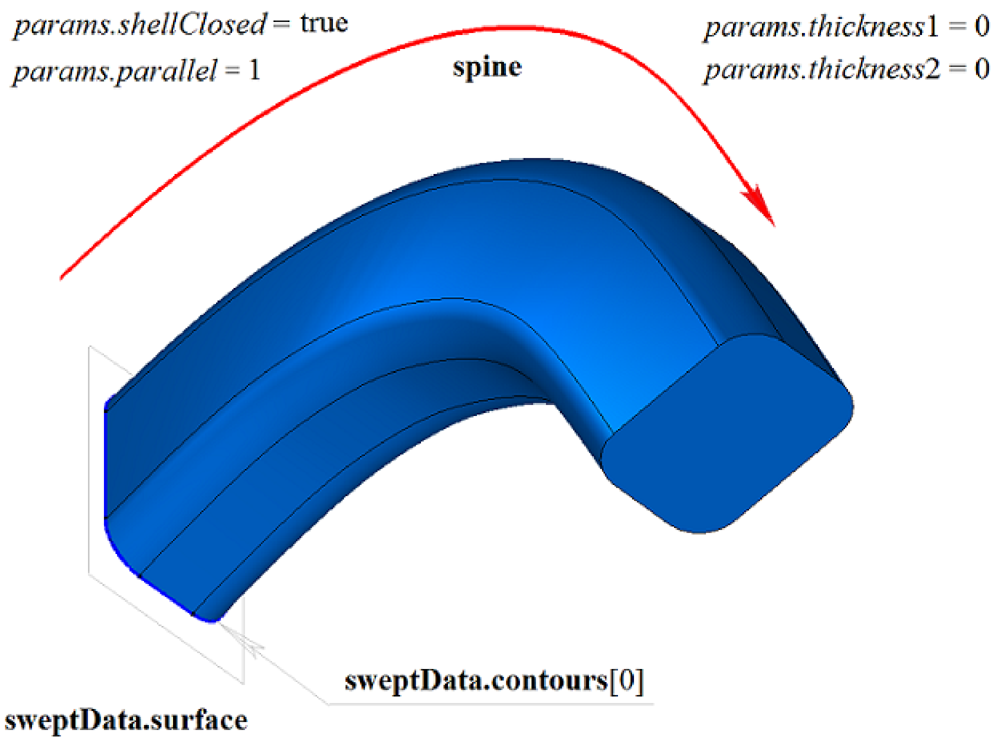


Fig. M.1.5.4.

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

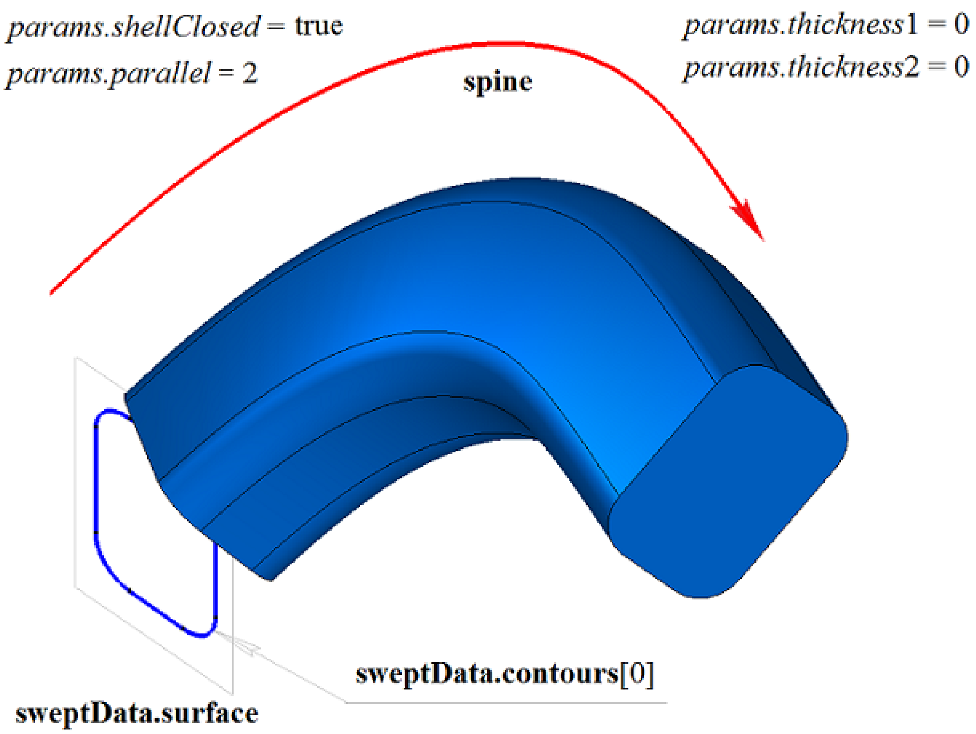


Fig. M.1.5.5.

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

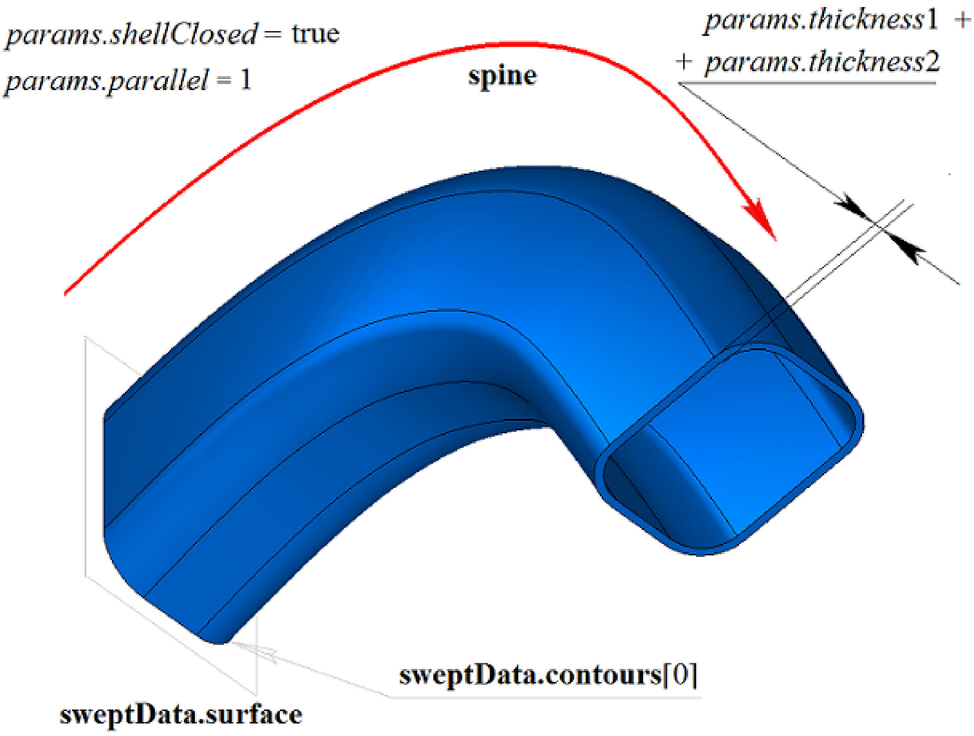


Fig. M.1.5.6.

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

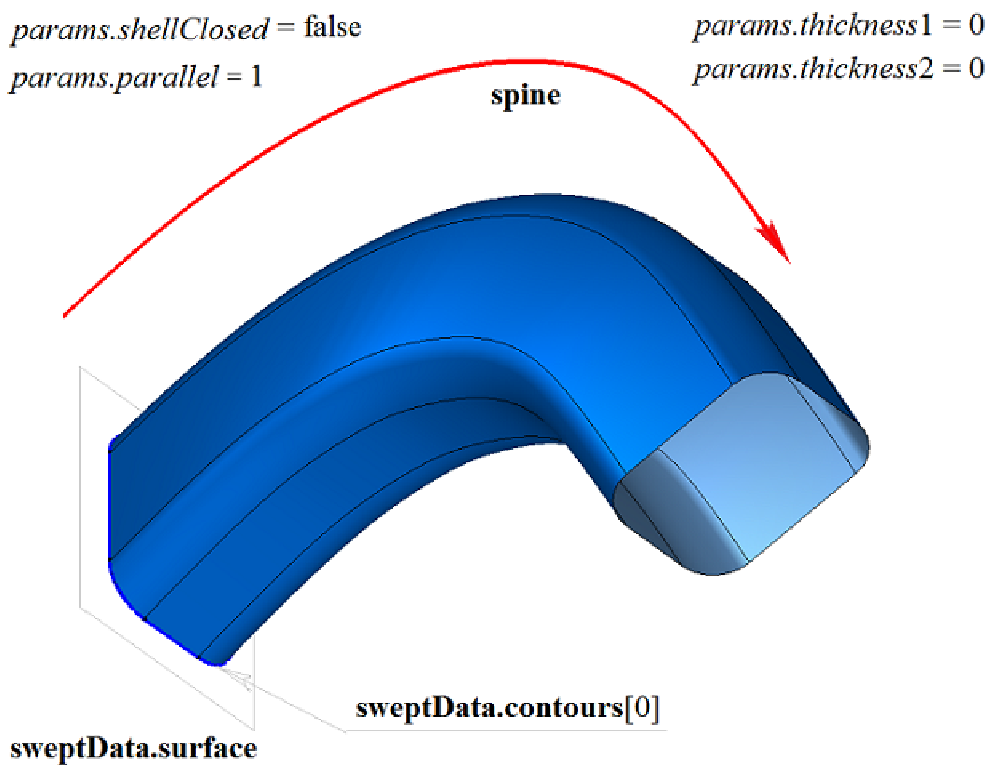


Fig. M.1.5.7.

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

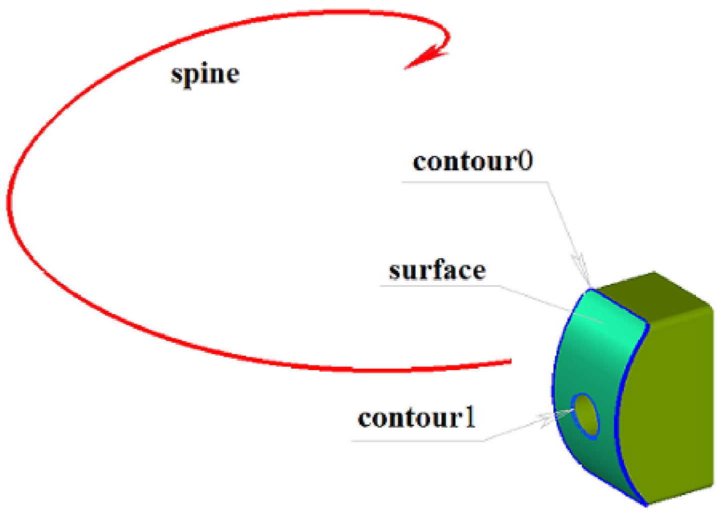


Fig. M.1.5.8.

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

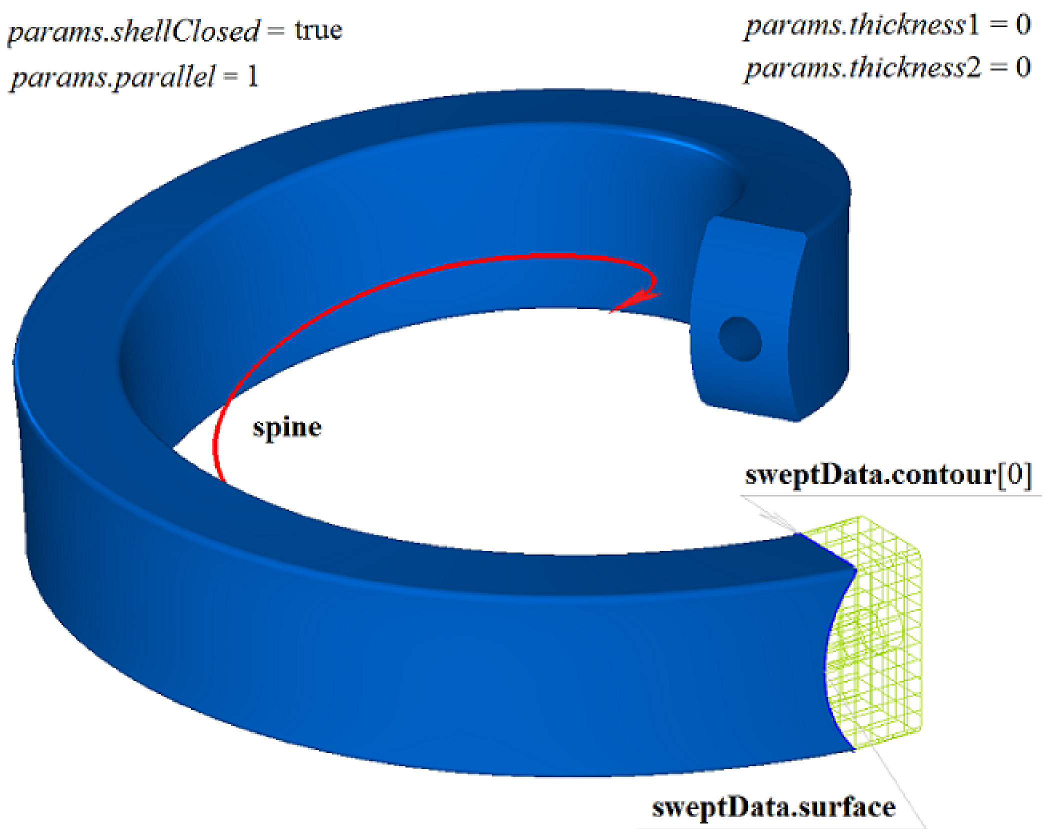


Fig. M.1.5.9.

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

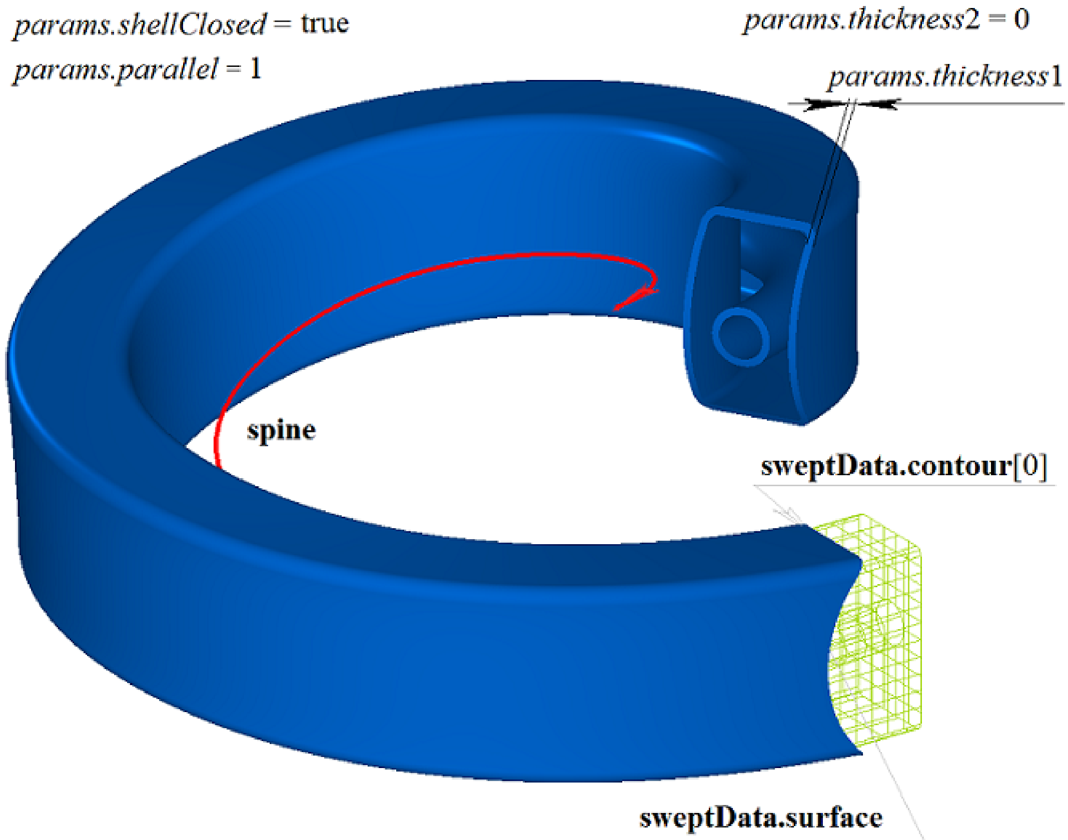


Fig. M.1.5.10.

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

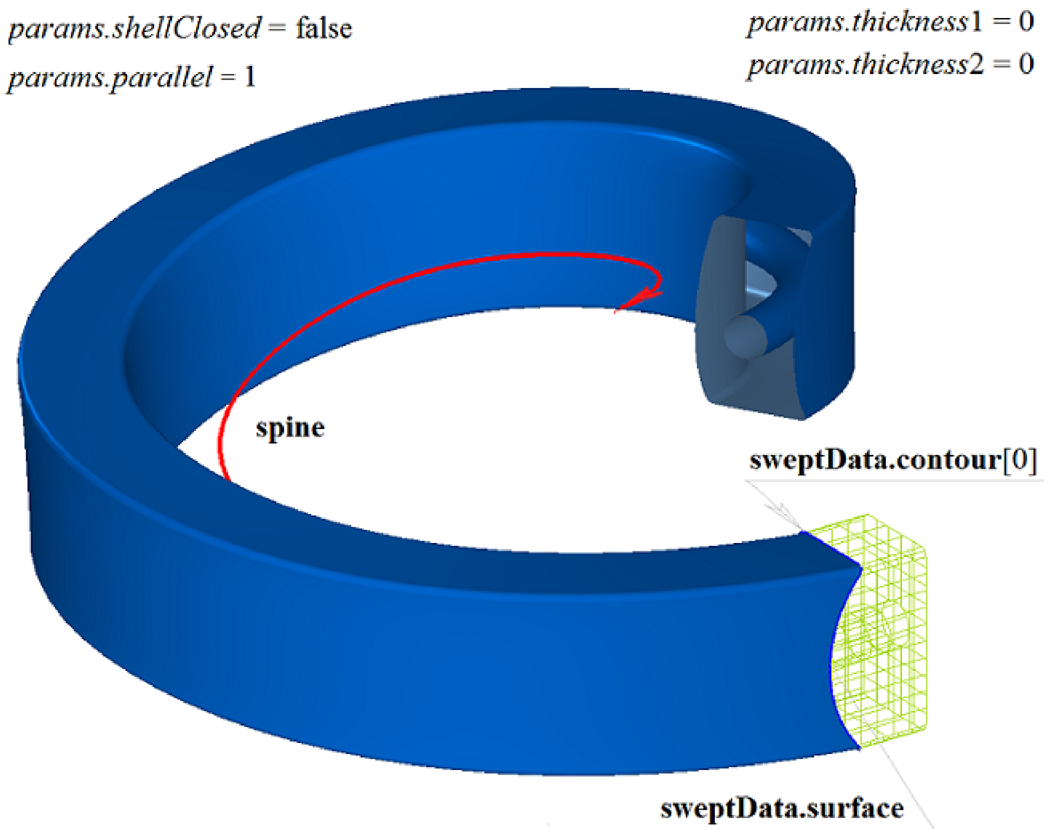


Fig. M.1.5.11.

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 Fig. M.1.4.12, you can see a set of non-intersecting two-dimensional **contours**, a flat **surface** (MbPlane) and **spine** guiding curve.

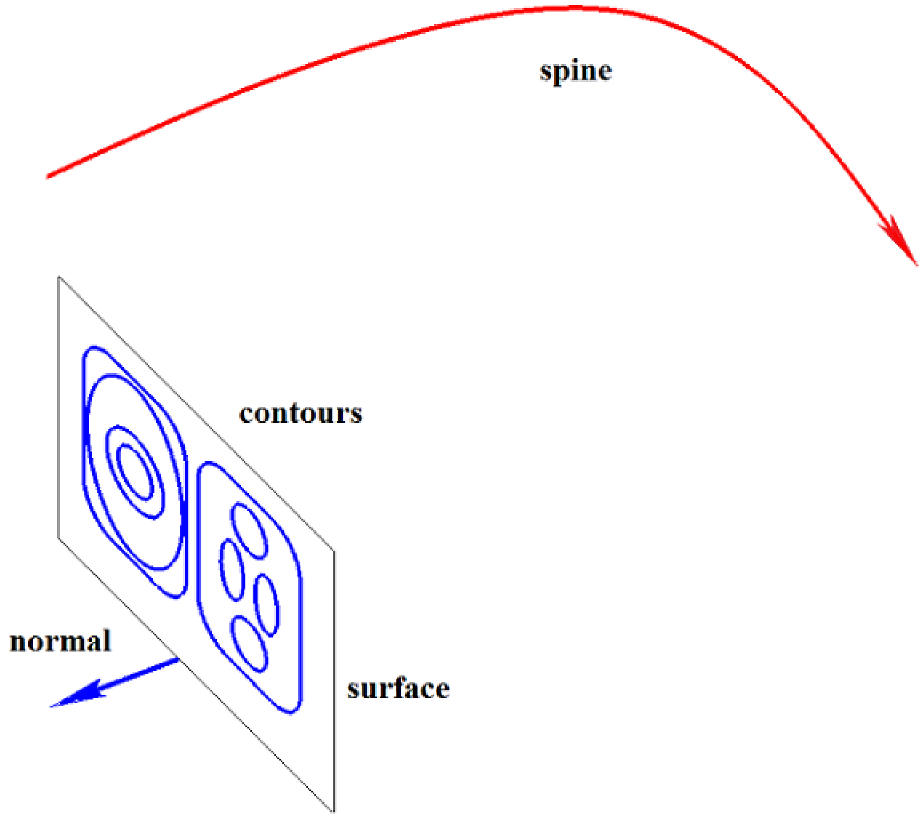


Fig. M.1.5.12.

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

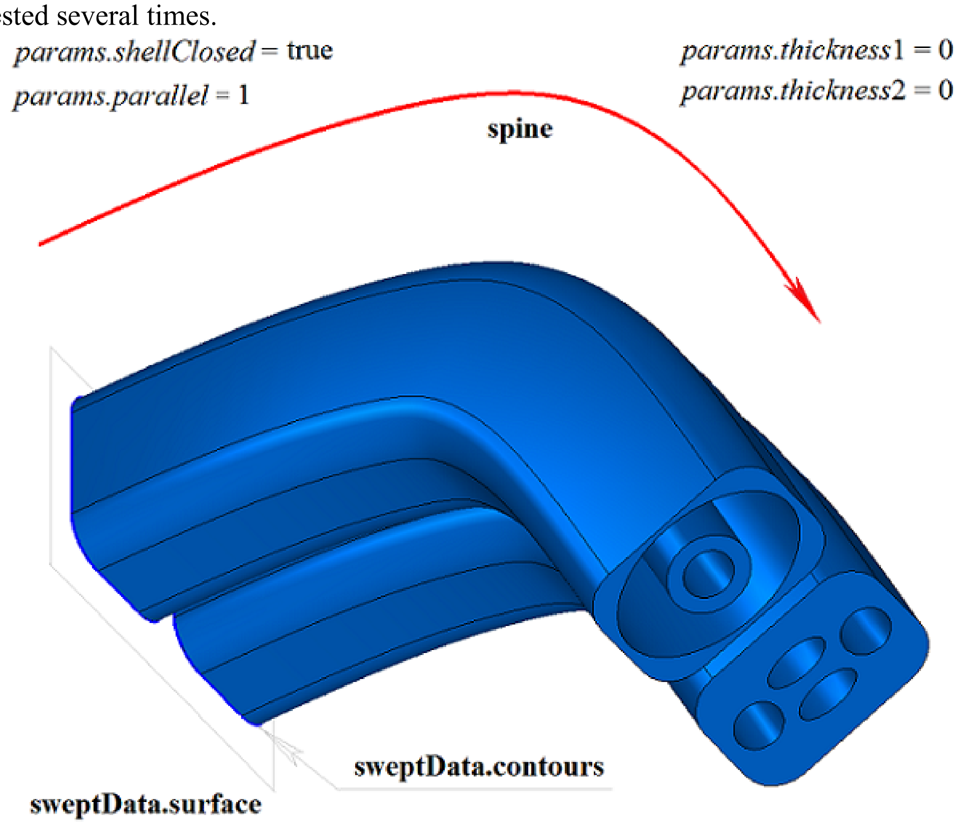


Fig. M.1.5.13.

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

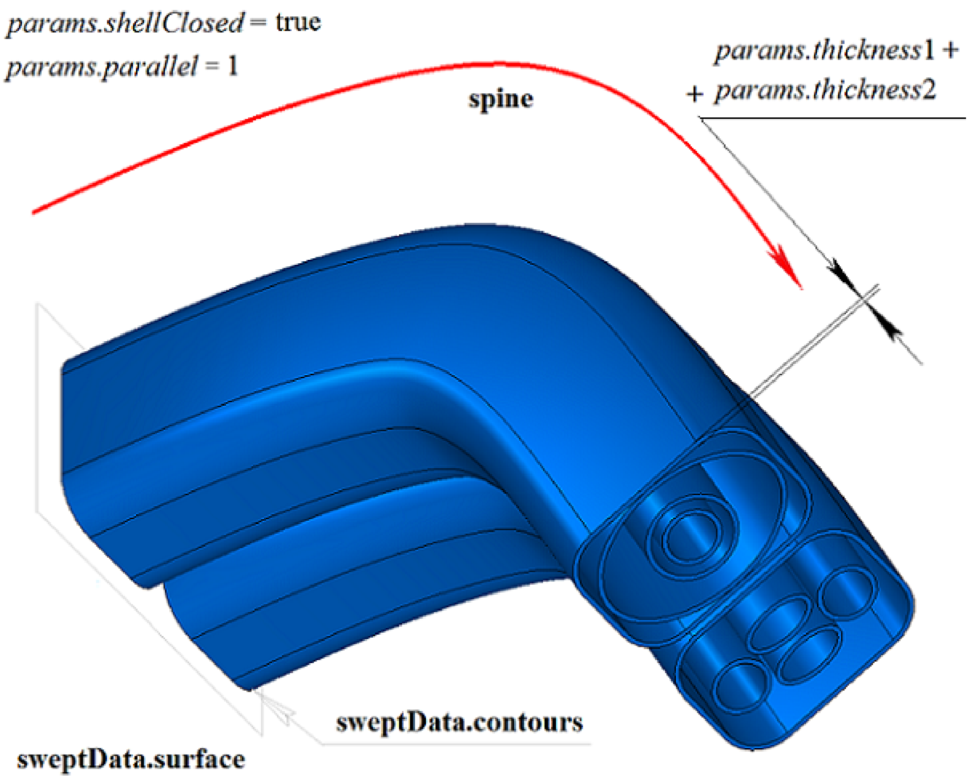


Fig. M.1.5.14.

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

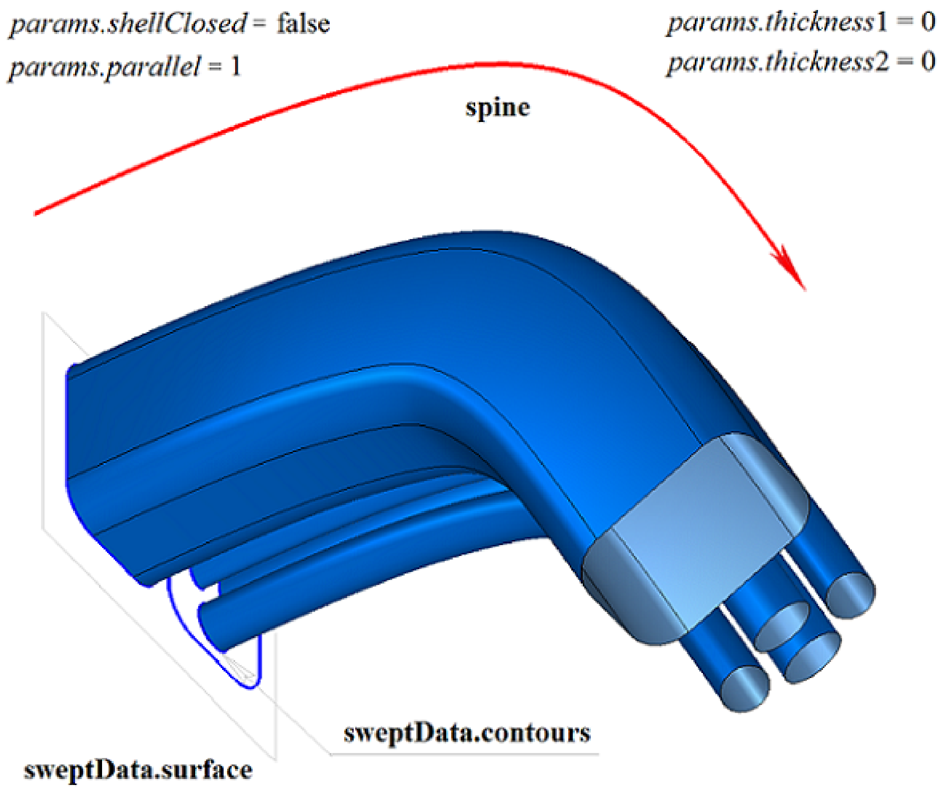


Fig. M.1.5.15.

In Fig. 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 bodies.

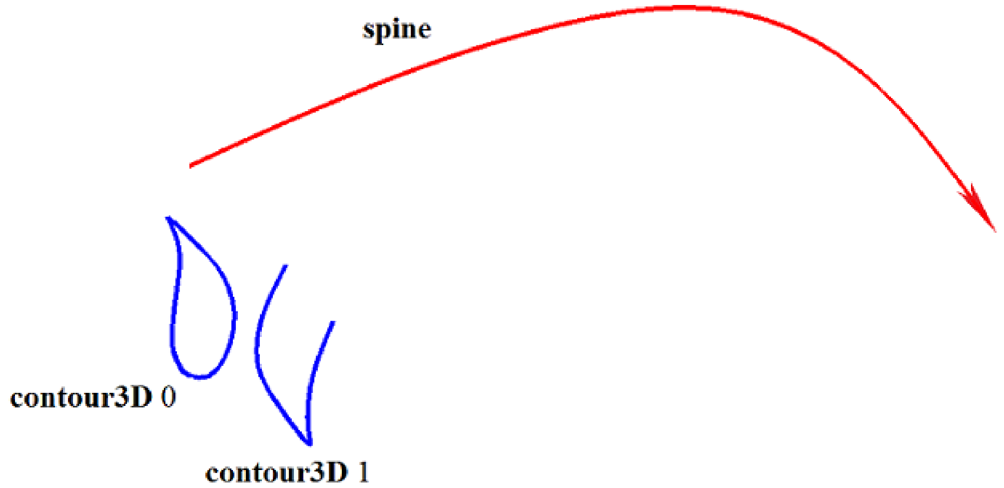


Fig. M.1.5.16.

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

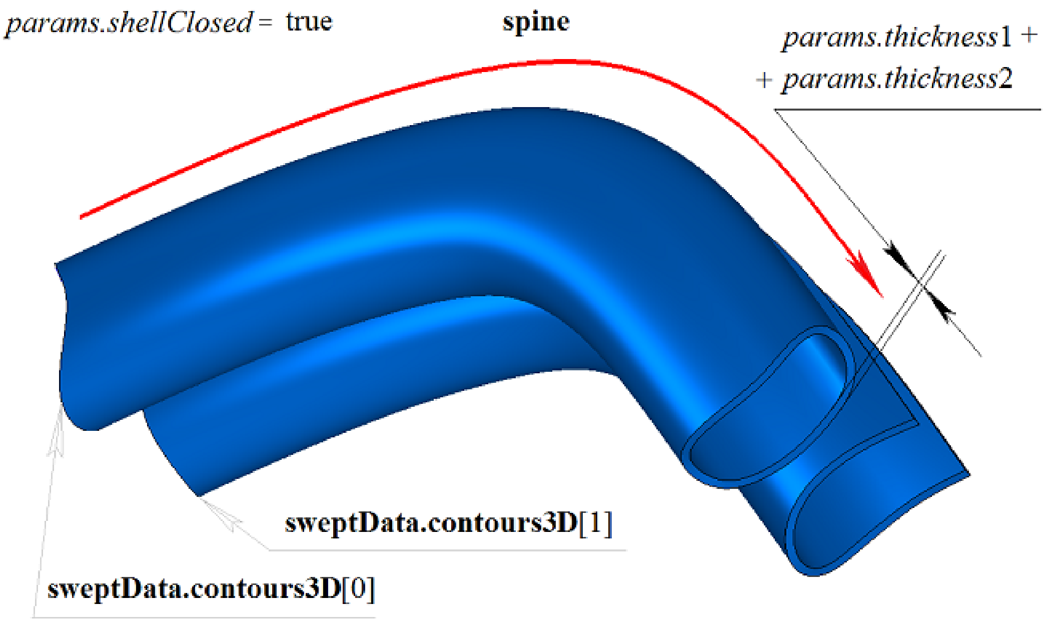


Fig. M.1.5.17.

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

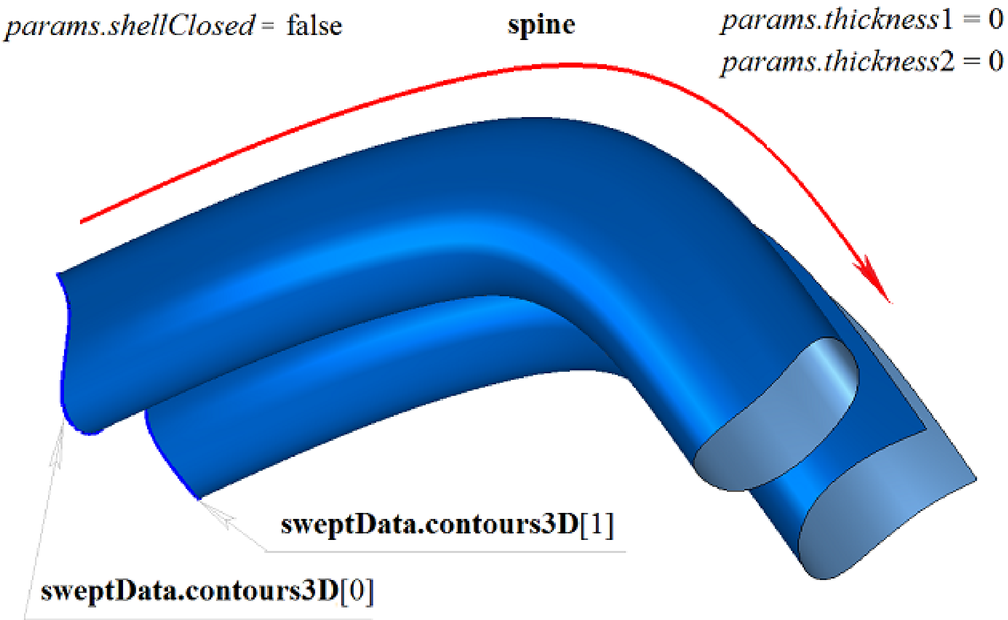


Fig. M.1.5.18.

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

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

M.1.6. Constructing a Body by Flat Sections

The 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 body 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 body.

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 body contains all the flat curves defining the body. **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 body shape between sections. Arbitrary curves can be used as guiding curve for a body.

params parameter contains information about movement mode, presence of body walls and their thickness and data whether the constructed body is closed. *params.thickness*1 and *params.thickness*2 parameters define wall thickness of the constructed thin-walled body. *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 body 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 body. If *params.closed*=true then there are no edges and the body has torus topology. *params.***vector**1 and *params.***vector**2 vectors define the direction of the body in the area of the start and end edges. For example, they permit you to define the direction of the body 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 Fig. M.1.6.1, you can see the data used for construction and parameters inheritance scheme for a constructed body by LoftedValues & *params* flat sections.

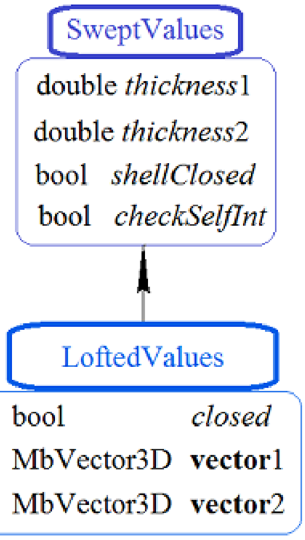


Fig. M.1.6.1.

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

In Fig. 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.

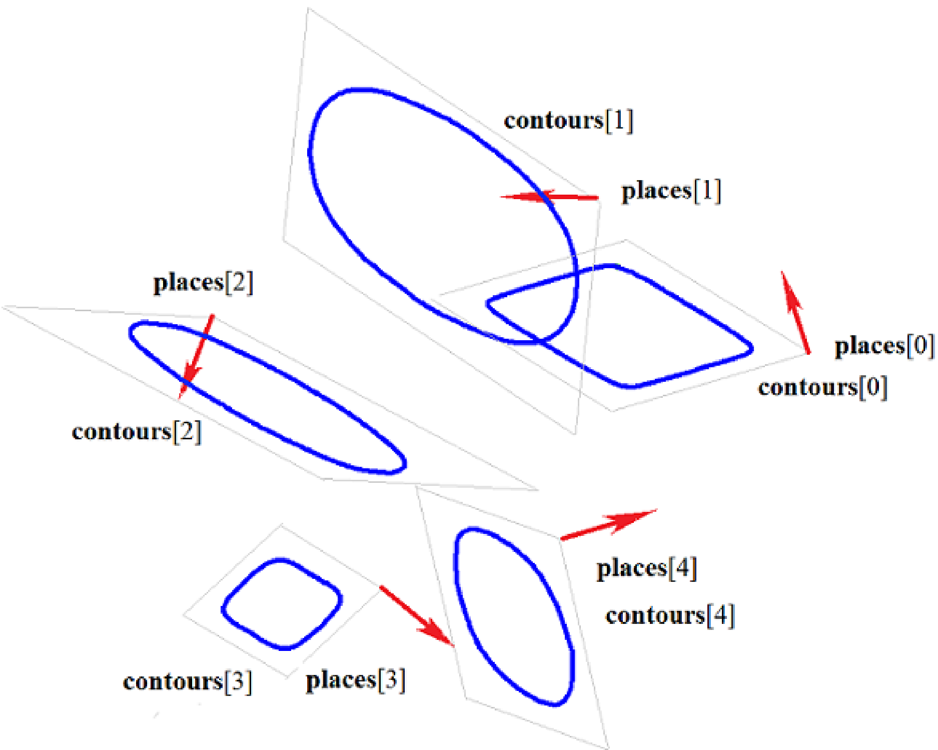


Fig. M.1.6.2.

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

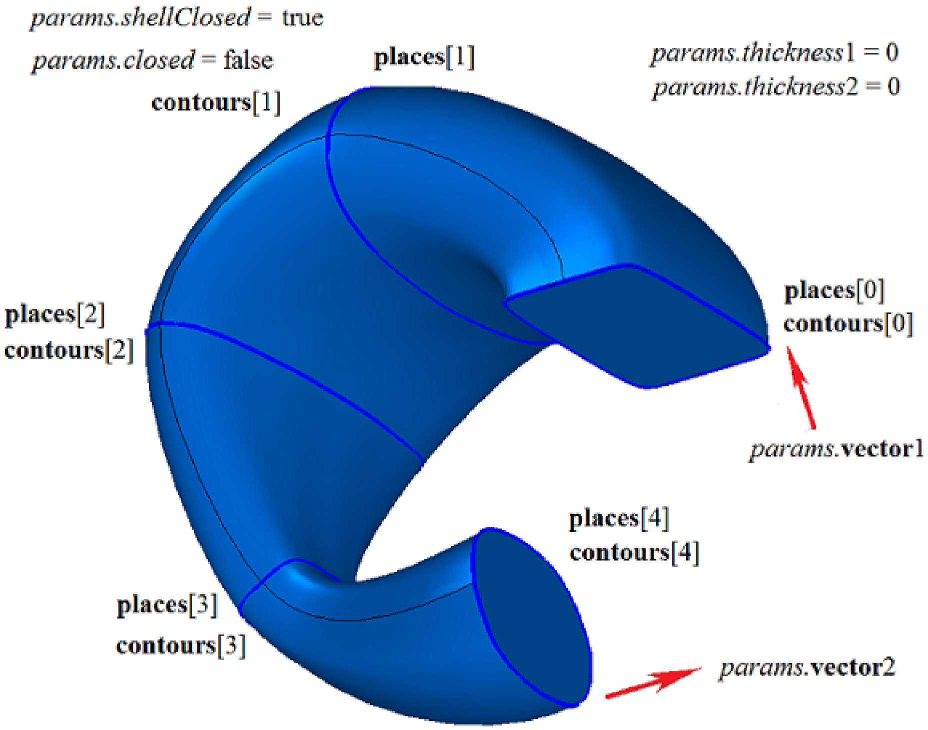


Fig. M.1.6.3.

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

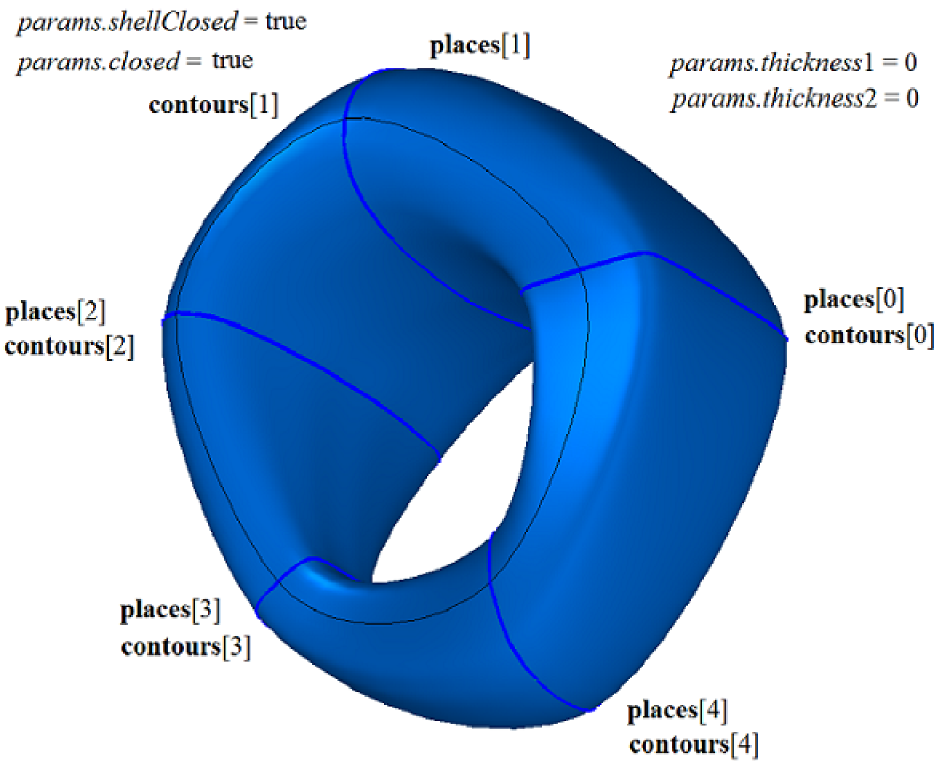


Fig. M.1.6.4.

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

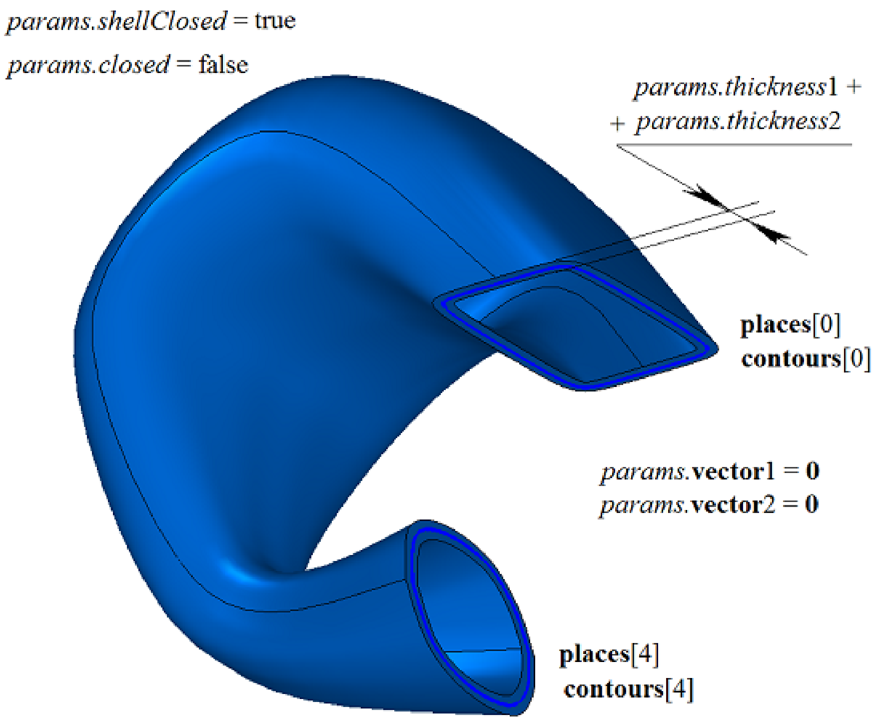


Fig. M.1.6.5.

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

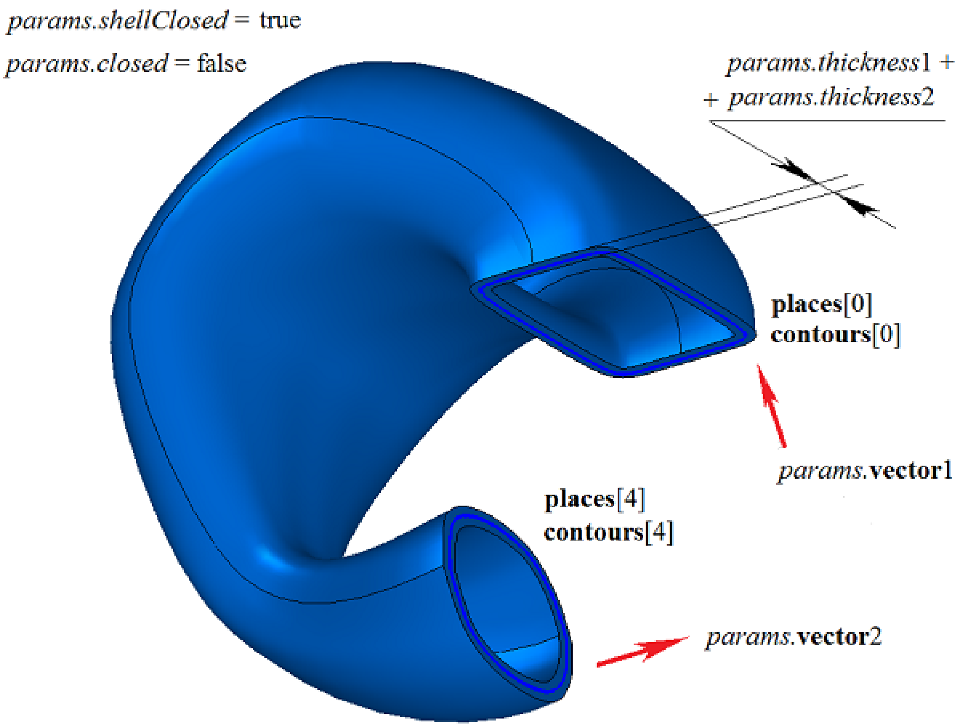


Fig. M.1.6.6.

In Fig. M.1.6.7, you can see a thin-walled body 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.

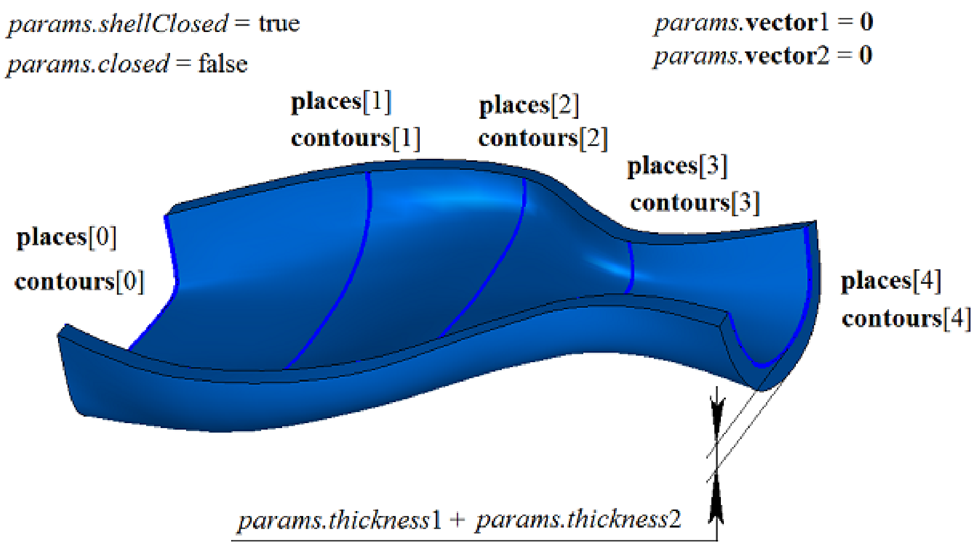


Fig. M.1.6.7.

In Fig. M.1.6.8, you can see a non-closed body 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.

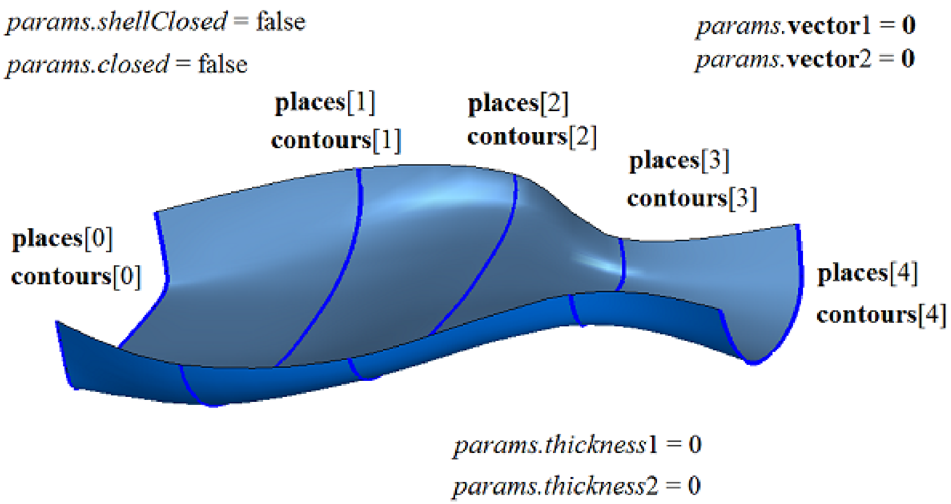


Fig. M.1.6.8.

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 Fig. M.1.6.9, you can see three contours that have unequal quantities of segments.

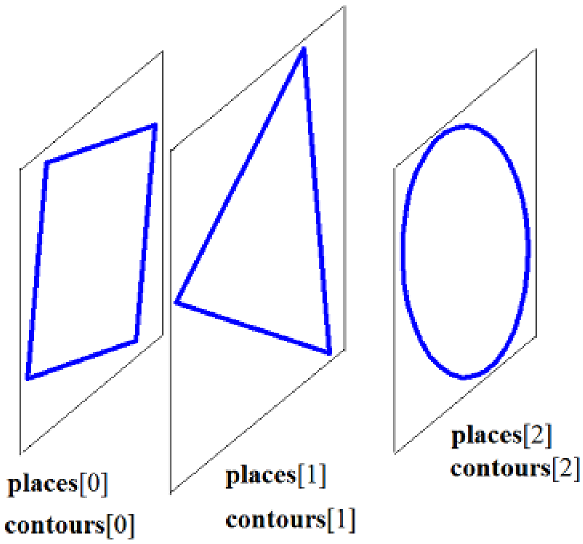


Fig. M.1.6.9.

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

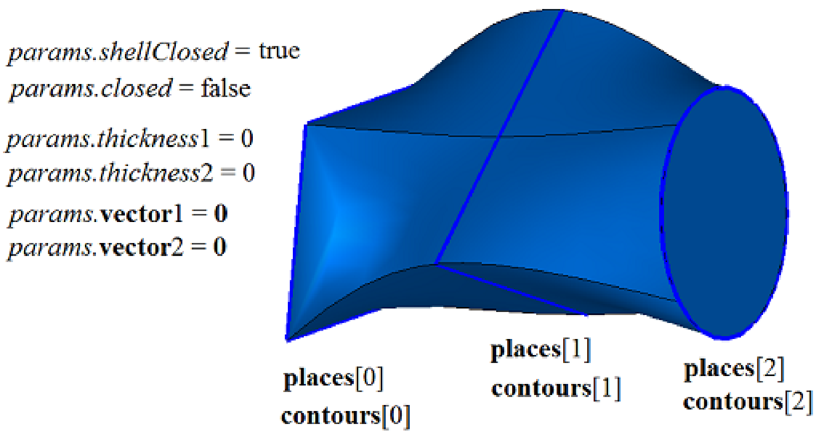


Fig. M.1.6.10.

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 body by flat sections shown in Fig. M.1.6.11.

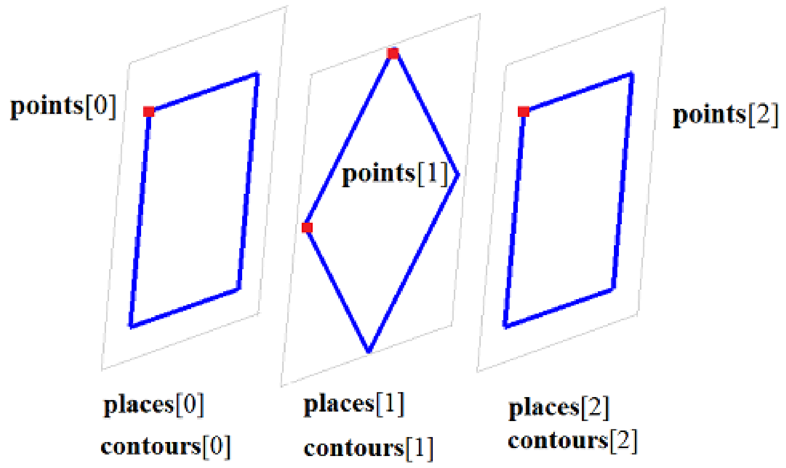


Fig. M.1.6.11.

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

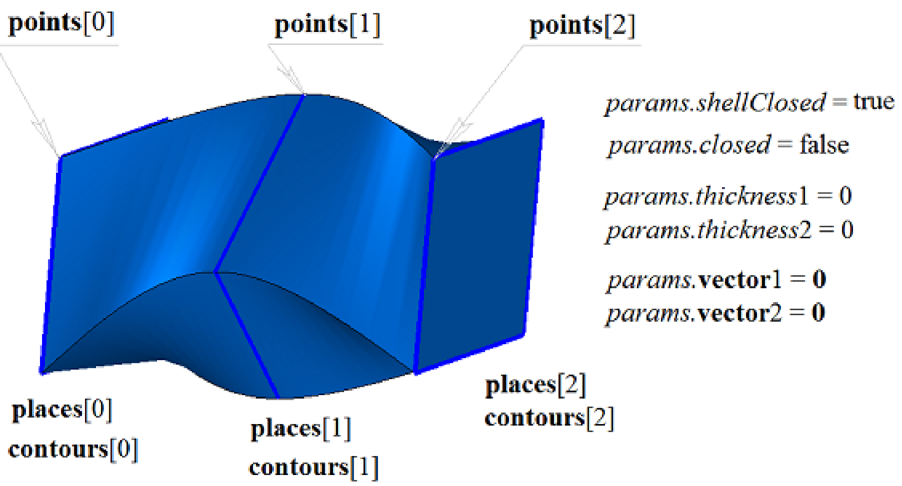


Fig. M.1.6.12.

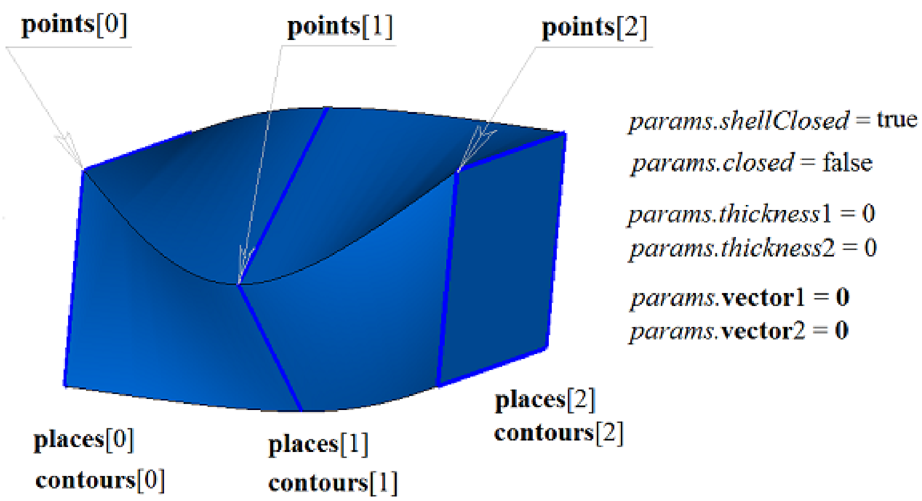


Fig. M.1.6.13.

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

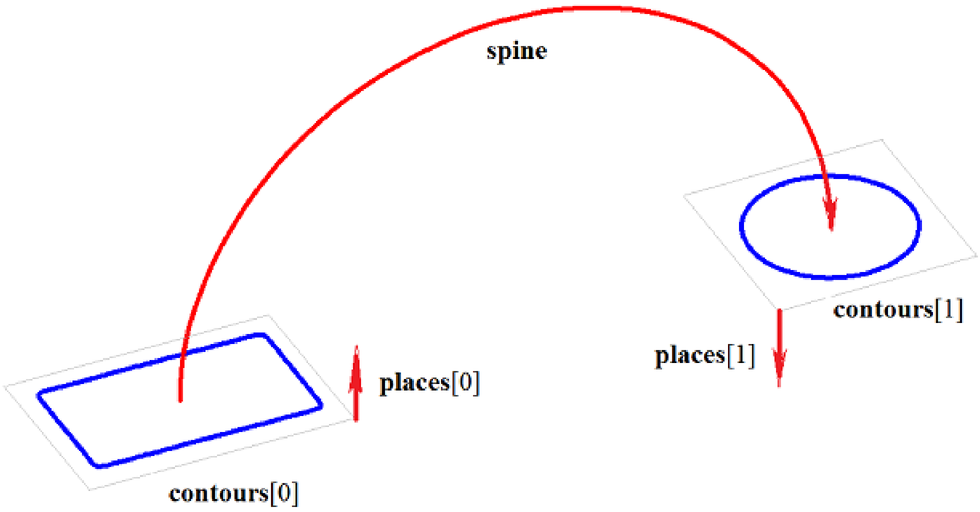


Fig. M.1.6.14.

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

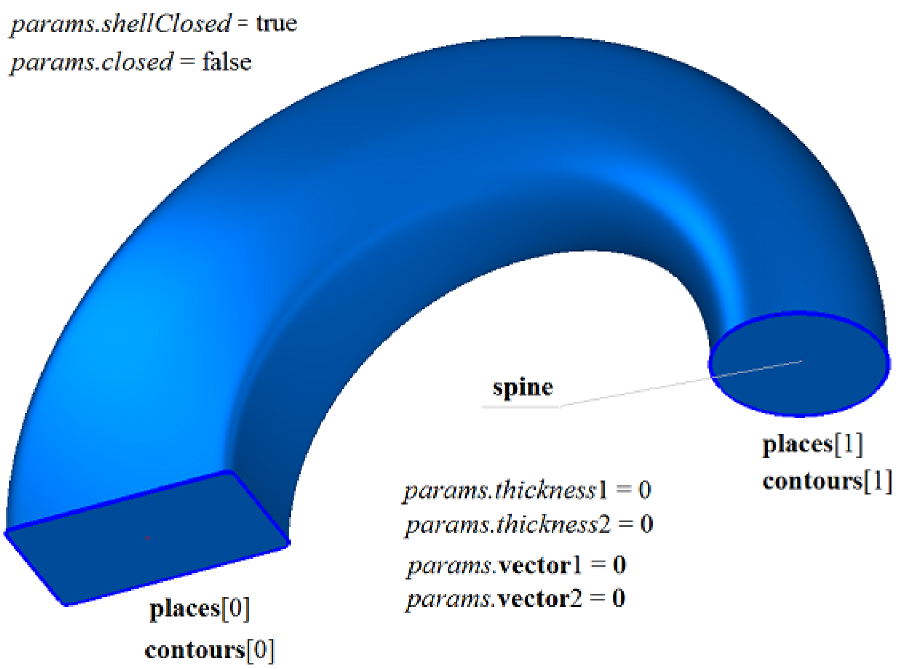


Fig. M.1.6.15.

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

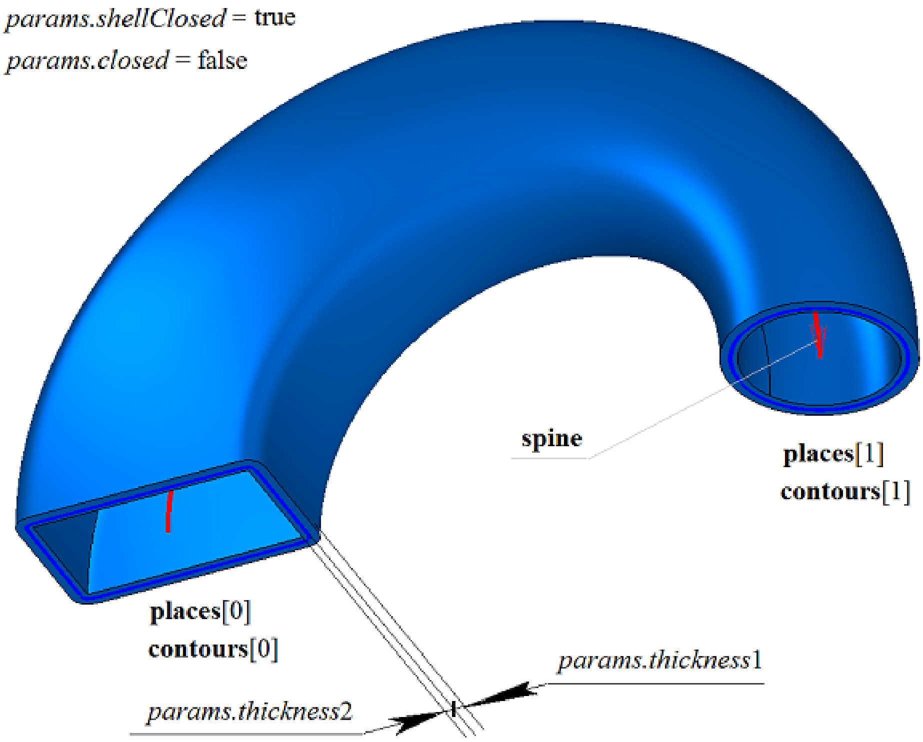


Fig. M.1.6.16.

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

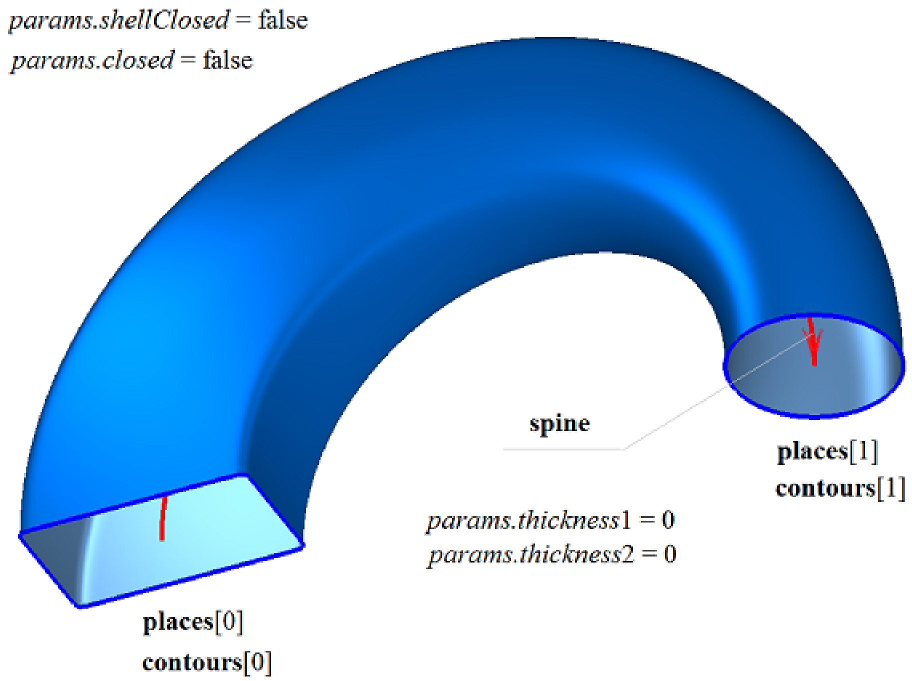


Fig. M.1.6.17.

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

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

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

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

Method input parameters are:

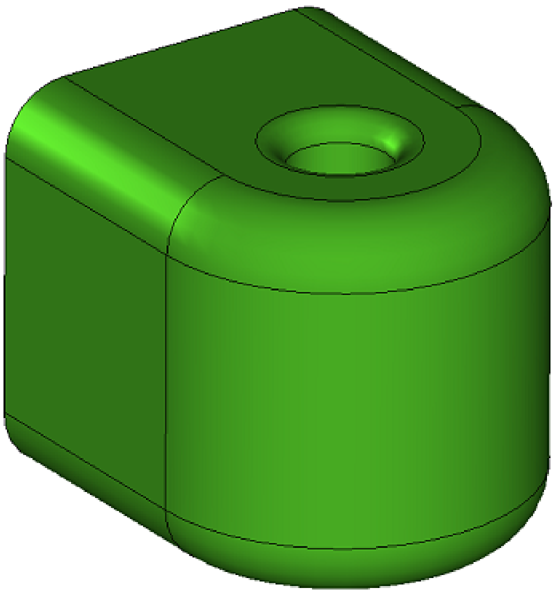
- **faceSet** is the set of faces,
- names is faces namer.

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

This method is declared in action_solid.h file.

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

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



faceSet

Fig. M.1.7.1.

This method gives names to unnamed faces, edges ribs and vertices and then it creates a body 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 body. **CreateSolid** method adds MbSimpleCreator simple constructor to the log of the newly constructed body. This constructor is declared in cr_simple_creator.h file.

M.1.8. Constructing a Body Based on a Surface

The method
MbResultType
SurfaceShell (const MbSurface & **surface**,

const MbSNameMaker & names,
MbSolid *& **result**)
constructs a body consisting of one face based on an original surface.

Input parameters of the method are as follows:

- **surface** is the original surface,
- names is a face namer.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

names parameter permits to assign names to faces, vertexes, and edges of the constructed body.

Fig. M.1.8.1 shows a body constructed from a surface cyclically closed in the first parametric direction.

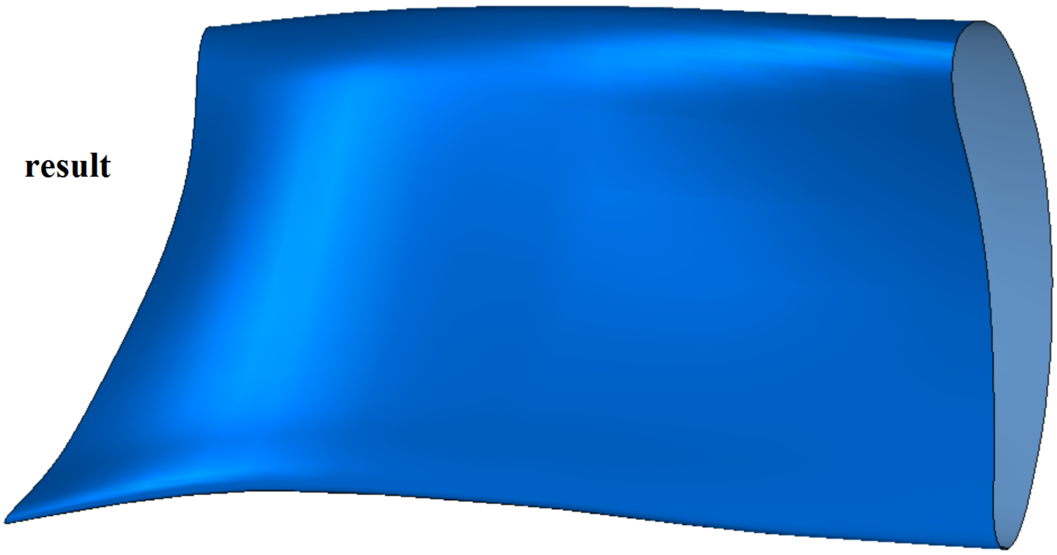


Fig. M.1.8.1.

This method doesn't check or calculate anything. The constructed body has only one face. The face differs from the initial surface in that it has edges on the borders and seams, and it has vertexes in edge mating points. If the initial surface is cyclically closed in one or both parametric directions, then the constructed body will have just one face cyclically closed by the corresponding edges. If the initial surface has borders, then the method will construct a non-closed body. The method adds MbSimpleCreator simple constructor declared in cr_simple_creator.h file in newly constructed body log.

M.1.9. Constructing a Ruled Body

The method
MbResultType
RuledShell (RuledSurfaceValues & **params**,
const MbSNameMaker & names,
bool *isPhantom*,
MbSolid *& **result**)
constructs a non-closed ruled body from two parametrically defined curves.

Input parameters of the method are as follows:

- **params** are construction parameters,
- names is a face namer,
- *isPhantom* is a construction goal flag (true means that the phantom mode is on).

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

params construction parameters contain information about the geometry required to construct the body, see Fig. M.1.9.1.

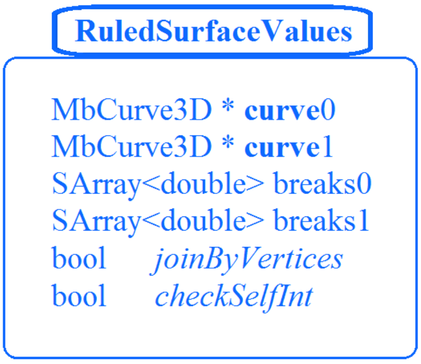


Fig. M.1.9.1.

A ruled body is constructed from two curves: **params.curve**0 and **params.curve**1. If compound curves are used then a ruled surface will be constructed based on each pair of compound curve segments, and a body face will be constructed on each surface. Adjacent faces will have common edges. **Params.curve**0 and **params.curve**1 curves can be additionally split into segments using **params.**breaks0 and **params.**breaks1 parameters. **params.***joinByVertices* is a parameter that indicates whether it is required to connect the contours having the same number of segments through the vertexes. **params.***checkSelfInt* is a parameter that indicates whether it is necessary to check the curves for self-intersection. By default, **params.***joinByVertices*=false and **params.***checkSelfInt*=false.

Fig. M.1.9.2 shows a ruled body constructed from compound curves and each curve has three segments.

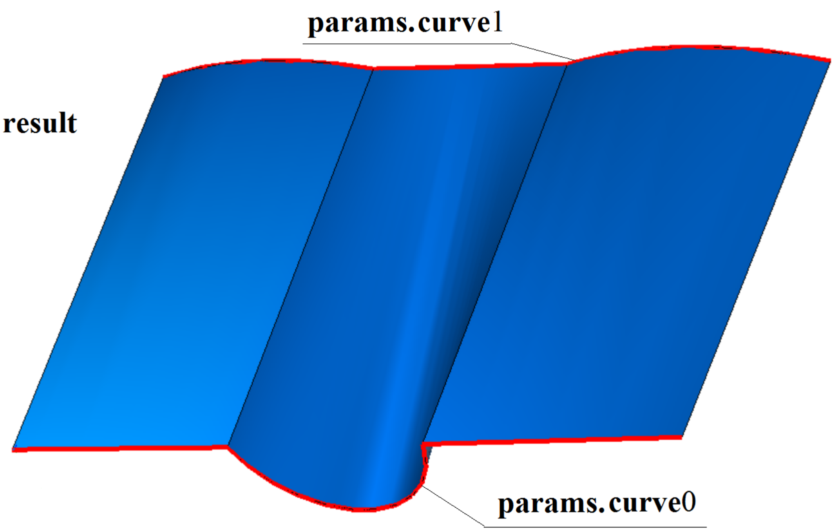


Fig. M.1.9.2.

The constructed body has one or several faces. A ruled surface always has borders formed by the initial curves. This method can check the faces for self-intersection. **params.curve**0 and **params.curve**1 curves can be cyclically closed. The method adds MbRuledShell constructor declared in cr_simple_creator.h file in a newly constructed body log.

M.1.10. Constructing a Body from a Curve Grid

The method

MbResultType
MeshShell (MeshSurfaceValues & **params**,
const MbSNameMaker & names,
bool *isPhantom*,
MbSolid *& **result**)

constructs a body from a curve grid defined in construction parameters.

Input parameters of the method are as follows:

- **params** are construction parameters,
- names is a face namer,
- *isPhantom* is a construction goal flag (true means that the phantom mode is on).

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

params construction parameters contain information about the geometry required to construct the body, see Fig. M.1.10.1.

MeshSurfaceValues	
RPArray<MbCurve3D>	**curvesU**
RPArray<MbCurve3D>	**curvesV**
bool	*uClosed*
bool	*vClosed*
bool	*checkSelfInt*
MbeMatingType	type0
MbeMatingType	type1
MbeMatingType	type2
MbeMatingType	type3
MbSurface *	**surface**0
MbSurface *	**surface**1
MbSurface *	**surface**2
MbSurface *	**surface**3
MbPoint3D *	point
bool	*defaultDir*0
bool	*defaultDir*1
bool	*defaultDir*2
bool	*defaultDir*3

Fig. M.1.10.1.

The body is constructed from two sets of curves: **params.curveU** and **params.curveV**. **params.curveU** curves are located along the first parametric direction of the body faces. **params.curveV** curves are located along the second parametric direction of the body faces. **params.***uClosed* is a parameter that indicates whether the surface is closed along the first parametric direction, it requires all curves from **params.curveU** curve set to be cyclically closed. **params.***vClosed* is a parameter that indicates whether the surface is closed along the second parametric direction, it requires all curves from **params.curveV** curve set to be cyclically closed. **params.***uClosed* and **params.***vClosed* permit to build non-closed surfaces on cyclically closed curves. **params.***checkSelfInt* indicates whether the surface of the constructed body should be checked for self-intersection. By default, **params.***checkSelfInt*=false.

params.type0, **params.**type1, **params.**type2, **params.**type3 parameters together with **params.surface**0, **params.surface**1, **params.surface**2, **params.surface**3 surfaces define body behavior on the borders, when the body parameters are *v=vMin*, *u=uMax*, *v=vMax*, *u=uMin*, correspondingly. **params.**type0, **params.**type1, **params.**type2, **params.**type3 parameters can take the values trt_Position, trt_Tangent, trt_Normal from MbeMatingType enumeration. By default, the parameters have trt_Position value. It means that the body surface contains the curves. **params.surface**0, **params.surface**1, **params.surface**2, **params.surface**3 surfaces can be equal to zero. trt_Tangent means that the corresponding body border is tangential to **params.surface**0, **params.surface**1, **params.surface**2, **params.surface**3 surfaces, correspondingly. trt_Normal means that the corresponding body border is orthogonal to **params.surface**0, **params.surface**1, **params.surface**2, **params.surface**3 surfaces, correspondingly.

Fig. M.1.10.2 shows a curve grid containing four cyclically closed curves of **params.curveU** set and two non-closed curves of **params.curveV** set.

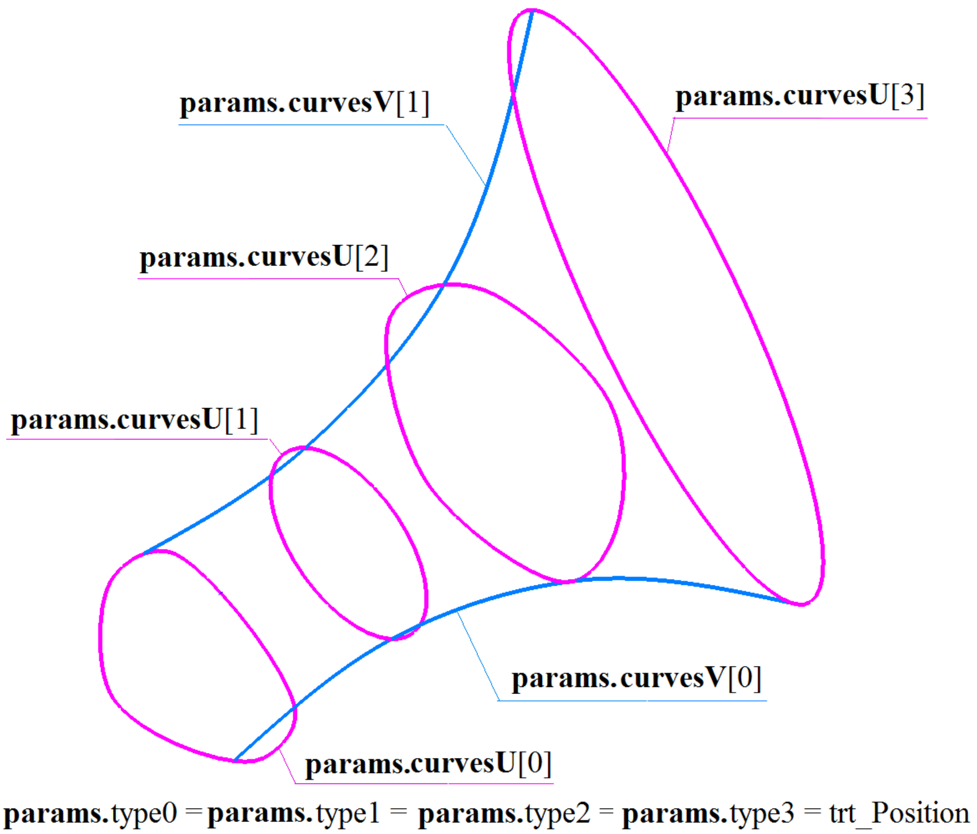


Fig. M.1.10.2.

Fig. M.1.10.3 shows a body constructed from a curve grid shown in Fig. M.1.10.2.

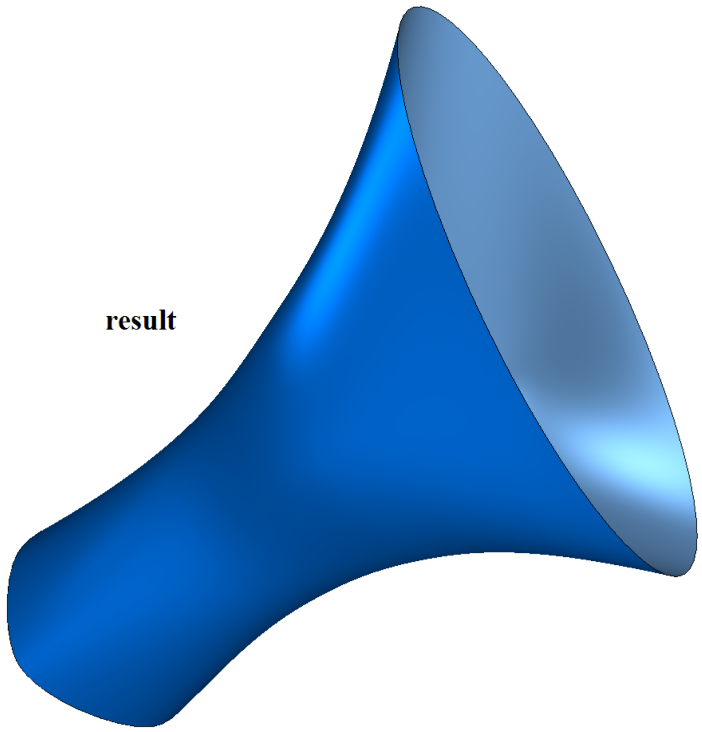


Fig. M.1.10.3.

Fig. M.1.10.4 shows a curve grid containing two non-closed curves of **params.curveU** set and three non-closed curves of **params.curveV**, and two end curves of which degenerate to points.

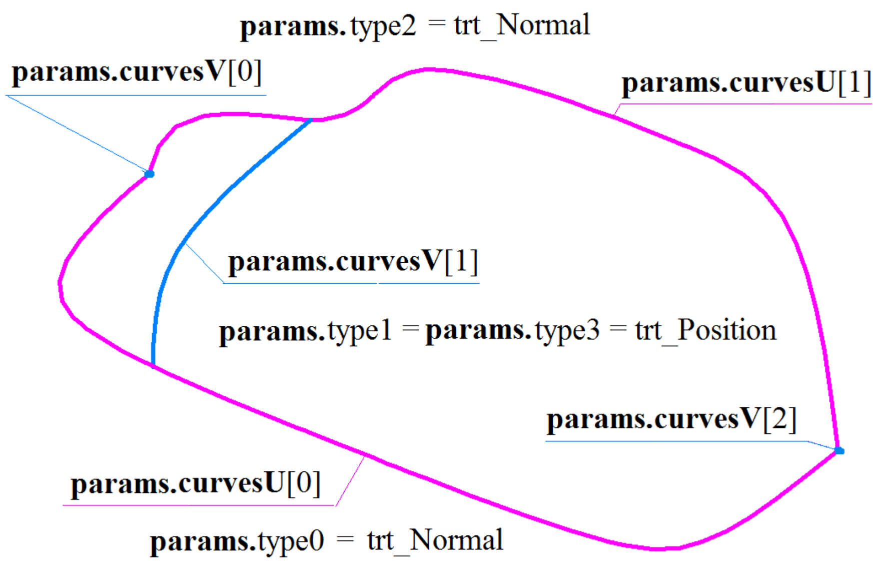


Fig. M.1.10.4.

Fig. M.1.10.5 shows a body constructed from curve grid shown in Fig. M.1.10.4. The borders of the constructed body corresponding to *v=vMin* and *v=vMax* parameters are orthogonal to flat surfaces **params.surface**0 and **params.surface**2, because **params.**type0 и **params.**type2 parameters are set to trt_Normal.

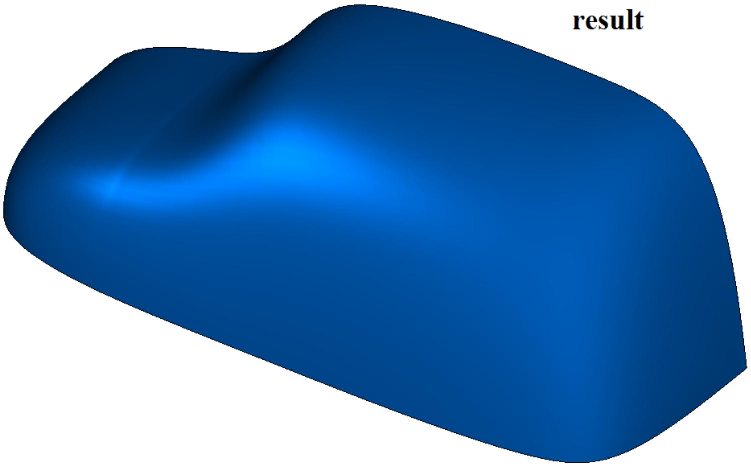


Fig. M.1.10.5.

The method adds MbMeshShell constructor declared in cr_simple_creator.h file in a newly constructed log of the body.

M.1.11. Constructing a Conjugating Body from Non-Connected Faces

The method
MbResultType
FacesFillet (const MbSolid & **solid**1,
const MbFace> & **face**1,
const MbSolid & **solid**2,
const MbFace> & **face**2,
const SmoothValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a non-closed body formed by a fillet face between two non-connected faces.

Input parameters of the method are as follows:

- **solid**1 is the first body,
- **face**1 is a mated border of the first body,
- **solid**2 is the second body,
- **face**2 is a mated border of the second body,
- *params* are construction parameters,
- names is a face namer.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

The method constructs a body formed by a mating face that enables smooth mating of **face**1 and **face**2. **solid**1 and **solid**2 parameters contain the initial bodies having **face**1 and **face**2, correspondingly.

The constructed face may have a shape of circle arc, ellipse, hyperbola or parabola in a cross-section. *params* parameter defines the shape of the mating face. *params* construction parameter contains information about the geometry required to construct the body, see Fig. M.1.11.1.

SmoothValues

double *distance*1
double *distance*2
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 **vector**1
MbVector3D **vector**2

Fig. M.1.11.1.

This method uses only four variables from the data shown in Fig. M.1.11.1: *params.distance*1, *params.distance*2, *params.conic*, and *params.prolong*. Other parameters are used to construct mating surfaces based on body edges. *params.form* takes st_Fillet value.

The body is constructed on the basis of **face**1 and **face**2. *params.distance*1 and *params.distance*2 are the parameters that determine arc radius (or radius of ellipse semi-axis) of mating surface cross-section. *params.conic* coefficient defines the shape of the mating surface. *params.conic* coefficient can range from 0.05 to 0.95. If *params.distance*1=*params.distance*2 and *params.conic*=0, then a mating surface with circular arc-shaped cross-section is constructed. The mating surface is constructed by moving a sphere tangential to **face**1 and **face**2. Reference borders of the mating body are located at contact points of the sphere and also **face**1 and **face**2. If *params.conic*=0.5, then cross-section of a mating face is a parabolic arc. If *params.conic*>0.5, then cross-section of the mating face is a hyperbolic arc. If *params.conic*<0.5, then cross-section of the mating face is an elliptical arc.

Fig. M.1.11.2 shows two bodies, the two faces of which will be used to construct mating bodies having various shapes.

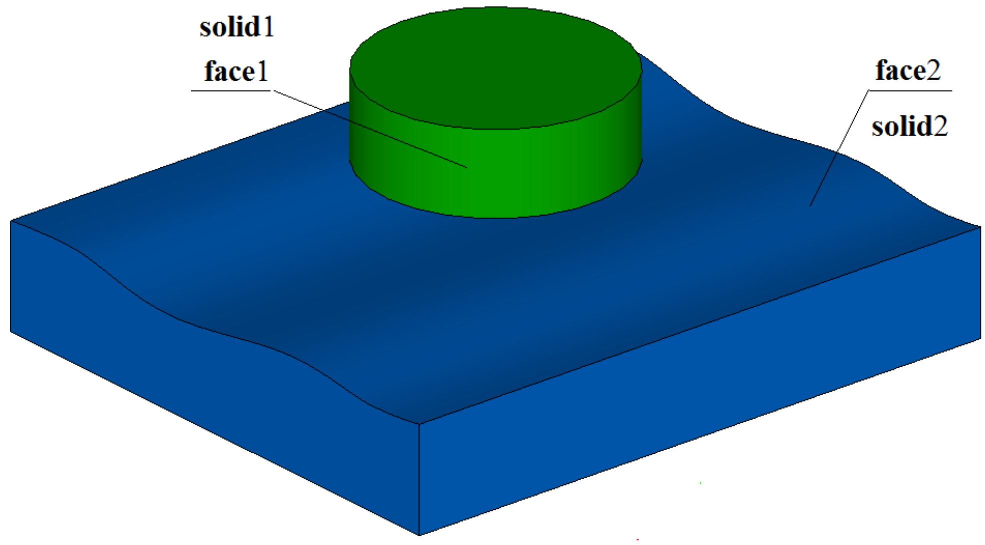


Fig. M.1.11.2.

Fig. M.1.11.3 shows a mating body with circular arc-shaped cross-section, this is achieved by the following equalities: *params.distance*1=*params.distance*2, *params.conic*=0.

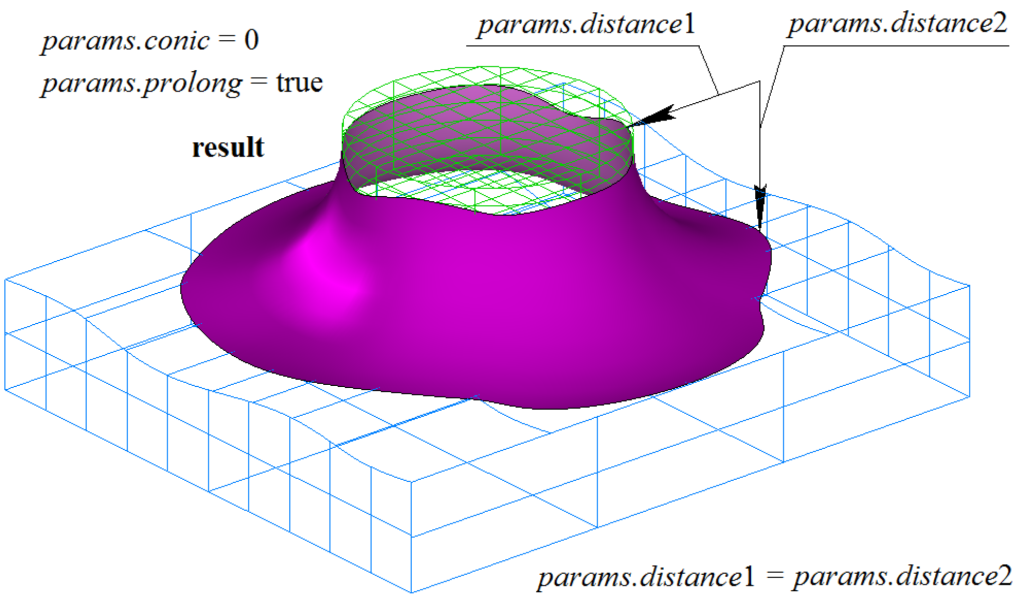


Fig. M.1.11.3.

Fig. M.1.11.4 shows a mating body with hyperbolic arc-shaped cross-section, which is achieved by*params.conic*=0.8 equality. If *params.prolong*=true, then a reference curve of the mating face goes beyond **face**2 at a specific section.

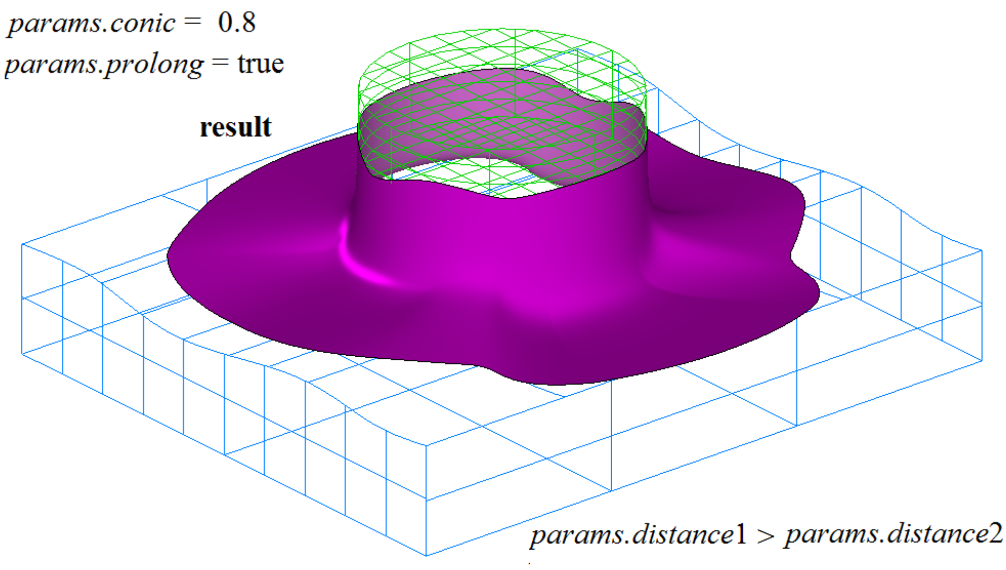


Fig. M.1.11.4.

Fig. M.1.11.5 shows a mating body with elliptical arc-shaped cross-section, that is achieved by *params.conic*=0.2 equality. If *params.prolong*=false, then a mating face is cut on the section where the reference curve goes beyond **face**2.

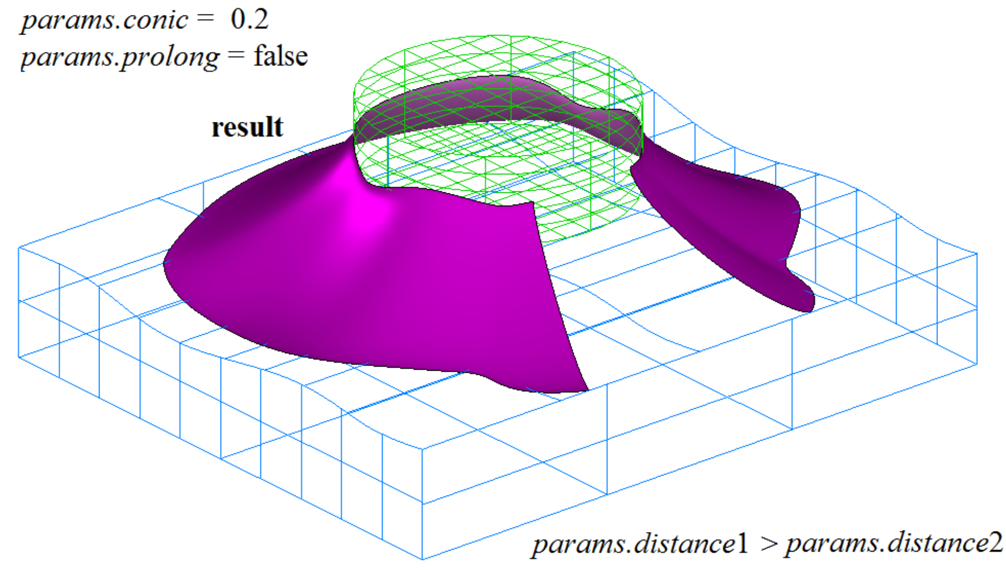


Fig. M.1.11.5.

FacesFillet method adds **solid**1 constructor, **solid**2 constructor and MbFilletShell constructor declared in cr_fillet_shell.h file in a log of the newly constructed body.

M.1.12. Constructing a Patch

The method
MbResultType
PatchShell (const RPArray<MbCurve3D> & **curves**,
const PatchValues & *params*,
const MbSNameMaker & names,

MbSolid *& **result**)

constructs a patch from a set of curves.

Input parameters of the method are as follows:

- **curves** is a set of curves,
- *params* are construction parameters,
- names is a face namer.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

curves and *params* parameters contain geometric information required to construct a patch. names parameter defines the names of faces of the constructed body.

The method constructs a body formed by a face, the borders of which contain defined curves. The body will be constructed correctly if the curves of **curves** set mate with each other and form a closed contour.

If the curves of **curves** set lie in one plane, then the patch will be a part of the plane lying within the contour formed by the curves. If all the curves of **curves** set lie in one curvilinear surface, then the patch will be a part of the common surface lying within the contour formed by the defined curves. If **curves** set contains a single curve, then this curve will be split into two parts, and patch surface will be a ruled surface having two poles. The patch is constructed the same way if **curves** set contains two curves; the patch surface will be a ruled surface with two poles. If **curves** set contains three curves, then patch surface will be MbCornerSurface built based on three curves, the surface has a pole. If **curves** set contains four curves, then patch surface will be MbCornerSurface constructed based on four curves. In other cases, construction can be denied.

Fig. M.1.12.1 shows eight curves lying on the edge of internal face cut. Fig. M.1.12.2 shows a patch constructed from the curves shown in Fig. M.1.12.1.

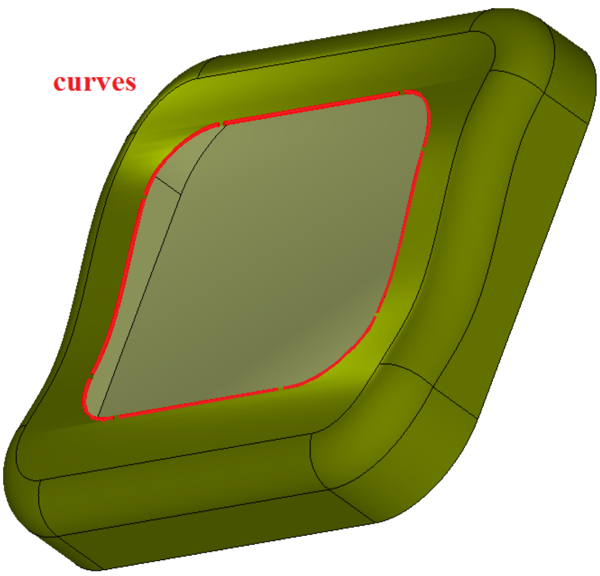


Fig. M.1.12.1.

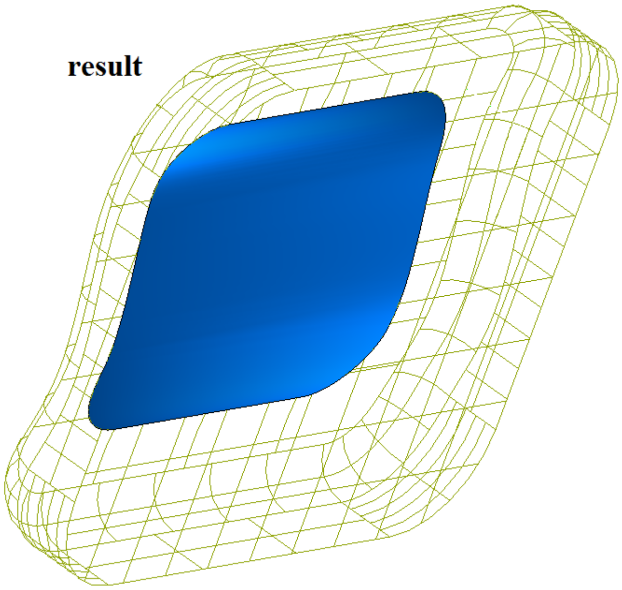


Fig. M.1.12.2.

Fig. M.1.12.3 shows six curves lying on the borders of various faces produced by cutting faces with a common surface.

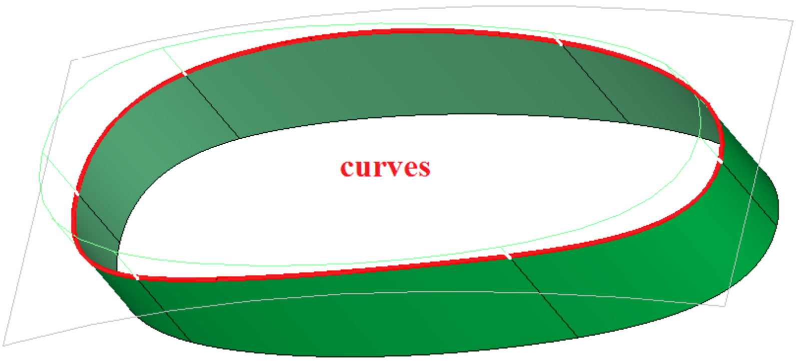


Fig. M.1.12.3.

If border edges keep information about the cutting surface after cutting, then the cutting surface might be used to construct the patch. Fig. M.1.12.4 shows an example of constructing a patch as a part of the cutting surface that formed border edges shown in Fig. M.1.12.3.

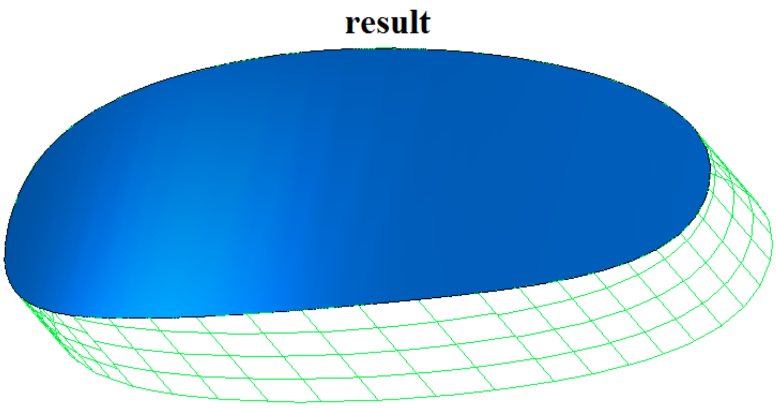


Fig. M.1.12.4.

As a rule, the curves for **curves** set are taken from border edges. After construction, the patch is often stiched with other bodies adjacent to its borders.

PatchShell method adds MbPatchCreator constructor into a log of the newly constructed body. MbPatchCreator constructor is declared in cr_patch_creator.h file.

M.1.13. Stiching Body Faces

The method

MbeStitchResType

StitchToOneSheetSolid (const RPArray<const MbSolid > & **solids**,
const MbSNameMaker & names,
bool *formSolidBody*,
double *stitchPrecision*,
MbSolid *& **result**)

stiches the faces of several bodies into a single body.

Input parameters of the method are as follows:

- **solids** are stiched bodies,
- names is operation namer,
- *formSolidBody* is a flag of stiching and nesting processing,
- *stitchPrecision* is a stiching precision.

The output parameter of the method is **result** constructed body.

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

This method is declared in action_shell.h file.

The method searches the pairs of border edges matching each other by length and position, and constructs a curve formed by intersection of faces of the edges found at the selected section. When intersection curve is successfully constructed, the pair of border edges is replaced by one edge that connects adjacent body faces. If required, the external sides of some faces are flipped. *stitchPrecision* parameter determines the maximum distance for corresponding points of stiched edges. **solids** set can contain any finite number of bodies. names parameter defines the names of the sewd body edges. If *formSolidBody*=true, then **result** constructed body is checked for absence of border edges, nested shells are searched, and internal shells are flipped if required. If *formSolidBody*=false, then **result** constructed body can have border edges, and nested shells are not determined.

Fig. M.1.13.1 shows two stiched bodies: **solids**[0] and **solids**[1]. Fig. M.1.13.2 shows **result** body, after stiching the edges are filleted.

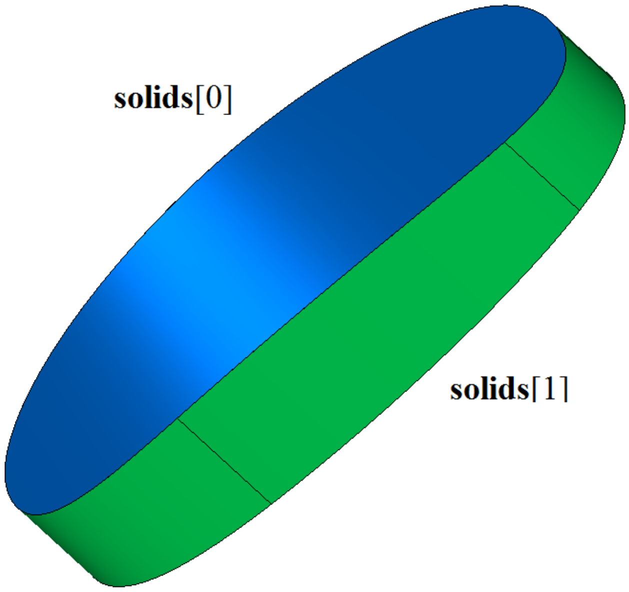


Fig. M.1.13.1.

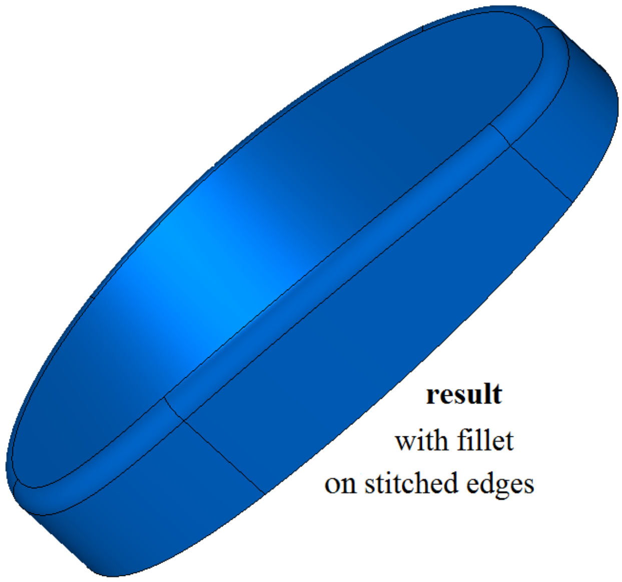


Fig. M.1.13.2.

StitchToOneSheetSolid method adds MbStitchedSolid constructor containing all data required to construct the body, into a log of newly constructed body. MbStitchedSolid constructor is declared in cr_stitch_solid.h file.

M.1.14. Construction of Body Based on Curves or Curve Points

The method
MbResultType
LoftedShell (const RPArray<MbCurve3D> & **curves**,
const MbSNameMaker & names,
SimpleName name,
MbSolid *& **result**)
constructs a body with one face, the surface of which contains a set of curves.

Input parameters of the method are as follows:

- **curves** is a set of curves,
- names is a face namer,
- name is an identifier.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

A surface of the constructed body contains the entire **curves** set. Surface shape of the body depends not only on the curve shape and relative position, but also on their orientation and also the positions of their starting points. In order to prevent body surface self-intersection and twisting, the adjacent curves of **curves** set should have the same direction, and closed curve starting points should be close to each other. If all curves in **curves** set are closed, then the surface of the constructed body will be closed along one of parametric directions.

The names and name parameters define a name of face in the constructed body.
Fig. M.1.14.1 shows **curves** set.

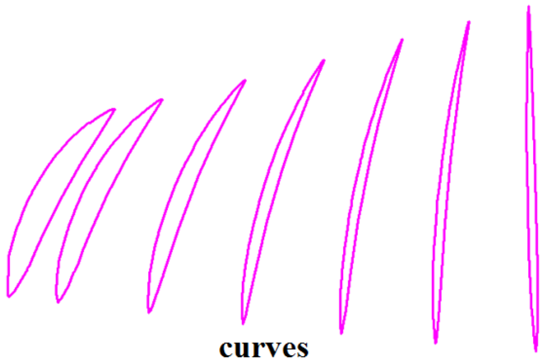


Fig. M.1.14.1.

Fig. M.1.14.2 shows a body constructed from the curves shown in Fig. M.1.14.1.

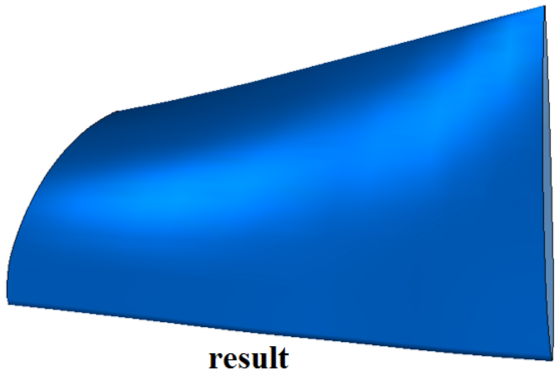


Fig. M.1.14.2.

Method
MbResultType
LoftedShell (const RPArray< SArray<MbCartPoint3D> > & **points**,
const MbSNameMaker & names,
SimpleName name,
MbSolid *& **result**)
constructs a non-closed body from one face passing through the control points.

Input parameters of the method are as follows:

- **points** are control point groups,
- names is a face namer,
- name is an identifier.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

The control points are grouped by SArray containers. Spline curves are constructed from point sets stored in each SArray container before the body is constructed. Then **LoftedShell** method described above constructs the body from the curve set. The surface of the constructed body passes through all the points defined by **points** parameter. The shape of body surface is influenced not only by the relative position of the points, but also by their order in groups. In order to prevent surfaces of the from self-intersection and twisting, point sequences in the adjacent groups should have the same direction, and starting points in adjacent groups should be close to each other. In order to construct a surface closed along one of parametric directions, first and last points in every SArray container should match.

The names and name parameters define a name of face in the constructed body.

Fig. M.1.14.3 shows a set of control points named **points**, each group has a different color.

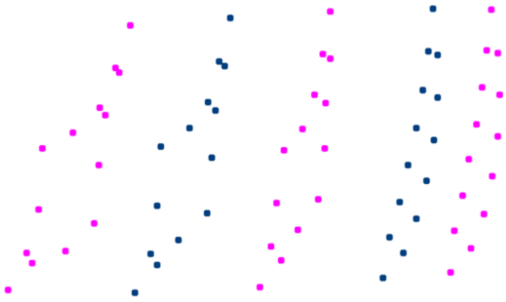


Fig. M.1.14.3.

Fig. M.1.14.4. shows a body and control points used to construct it.

Fig. M.1.14.4.

LoftedShell method adds LoftedShell constructor containing all data required to construct the body into a log of newly constructed body. MbThinShellCreator constructor is declared in cr_rib_solid.h file.

M.1.15. Construction of Equidistant Body

The method
MbResultType
OffsetShell (MbSolid & **solid**,
MbeCopyMode *sameShell*,
RPArray<MbFace> & **faces**,
bool *checkFacesConnection*,
SweptValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs an equidistant non-closed body based on a set of faces.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body copying method,
- **faces** are the faces of the original body,
- *checkFacesConnection* is a flag that checks the connection of selected faces,
- *params* are construction parameters,
- names is a face namer.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

The method constructs a body, the faces of which are equidistant to **faces** of **solid** original body. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original body to **result** constructed body. *params* parameter contains information about the distance from constructed faces from the faces of the original body. The distance may be equal to *params.thickness*1 in positive direction of the normal to the face or to *params.thickness*2 in negative direction of the normal to the face of the original body. If *params.shellClosed*=false, then a non-closed body will be constructed. names parameter is used to name the faces of the constructed body.

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

Fig. M.1.15.1 shows a body, its selected faces will be used to construct an equidistant body.

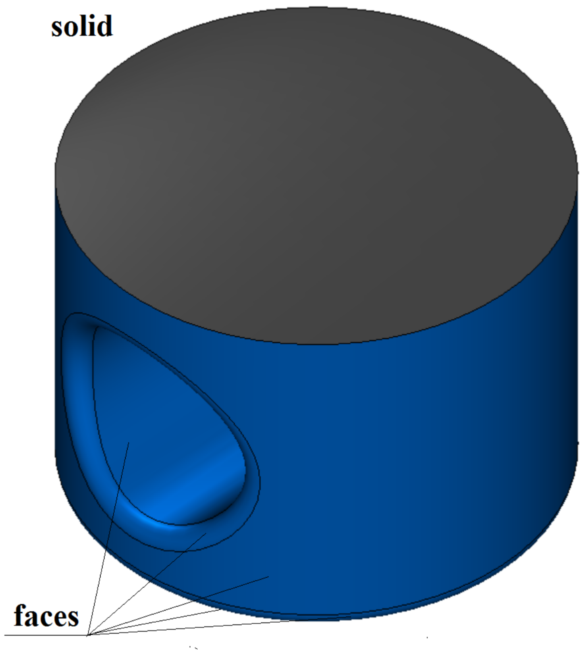
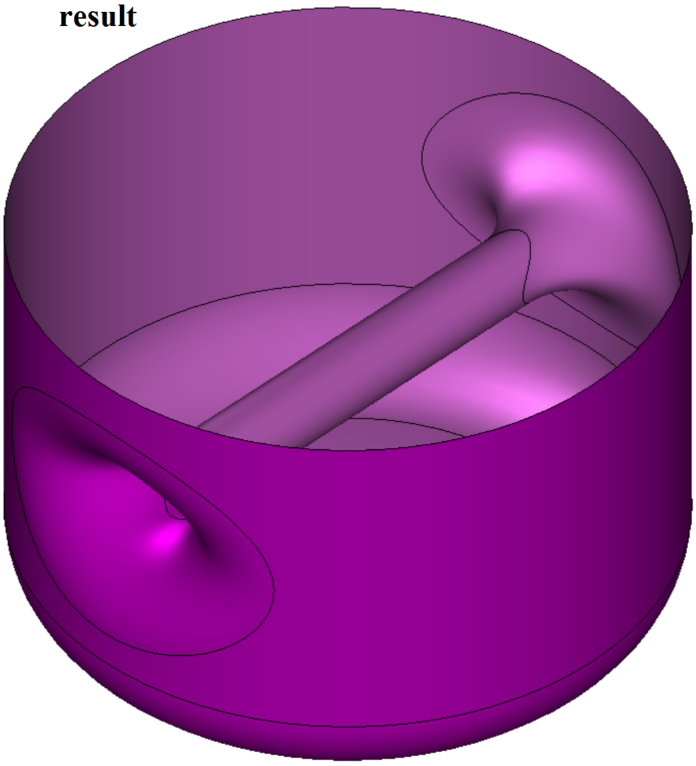


Fig. M.1.15.1.

Fig. M.1.15.2 shows constructed equidistant body.



*params.thickness*1 > 0 *params.thickness*2 = 0

Fig. M.1.15.2.

If **faces** set contains all faces of **solid** original body, **result** constructed body will be equidistant to the original body.

OffsetShell method adds MbShellSolid constructor containing all data required to construct the body into a log of newly constructed body. MbShellSolid constructor is declared in cr_thin_shell_solid.h file.

test.exe test application constructs an equidistant non-closed body using New -> Shell -> Based on Shell -> Equidistant to faces menu command.

M.1.16. Extension of Body Face

The method
MbResultType
ExtensionShell (MbSolid & **solid**,
MbeCopyMode *sameShell*,
MbFace & **face**,
const RPArray<MbCurveEdge> & **edges**,
const ExtensionValues & *params*,
const MbSNameMaker & names,
MbSolid * & **result**)
extends non-closed body by extruding the selected face of border edges of selected body face.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body copying method,
- **face** is the original body face,
- **edges** is a set of face edges that are extruded,

- *params* are construction parameters,
- names is a face namer.

The output parameter of the method is a **result** constructed body.

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

This method is declared in action_shell.h file.

The method extends **face** from the side of boundary **edges**. **face** is a part of **solid** body. *sameShell* parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory* and *cm_Same*. MbeCopyMode enumeration is described in Item O.7.9. Copying a Set of Faces.

params parameter defines face extension methods and contains data shown in Fig. M.1.16.1.

ExtensionValues

ExtentionType	type = { et_same, et_tangent, et_direction }
ExtentionWay	way = { ew_distance, ew_vertex, ew_surface }
LateralKind	kind = { le_normal, le_prolong }
MbCartPoint3D	**point**
MbVector3D	**direction**
double	*distance*
bool	*prolong*
bool	*combine*
MbFaceShell *	**shell**
MbItemIndex	faceIndex

Fig. M.1.16.1.

There are three types of face extension.

Extension type is defined by *params*.type parameter. If *params*.type=et_same, then the original **face** is extended by expanding its surface parameter definition area and moving border **edges**. If *params*.type=et_tangent, then new ruled faces that smoothly join with **face** are smoothly joined with **face** using **edges**. If *params*.type=et_direction, then new faces are joined to **face** of the original body using **edges**. New faces are received by extrusion of the curved **edges** in *params*.**direction**.

params.way parameter defines the distance used to extend **face** face. If *params*.way=ew_distance, then the face is extended by *params*.*distance*. If *params*.way=ew_vertex, then the face is extended by the distance defined by *params*.**point**. If *params*.way=ew_surface, then the face is extended up to *params*.**shell**. *params*.faceIndex defines the closest face of *params*.**shell** to **face**. If *params*.way!=ew_surface, then *params*.**shell** may equal zero.

params.kind parameter defines the method how extension border face will be cut in the beginning and in the end of each edge of **edges** set. If *params*.kind=le_normal, then **face** extension border is cut along the normal to the corresponding edge of **edges** set. If *params*.kind=le_prolong, then **face** extension border is defined by the edges adjacent to the edges of **edges** set.

names parameter is used to name the faces of the constructed body.

Fig. M.1.16.2 shows the body face, the two border edges of which will be extended.

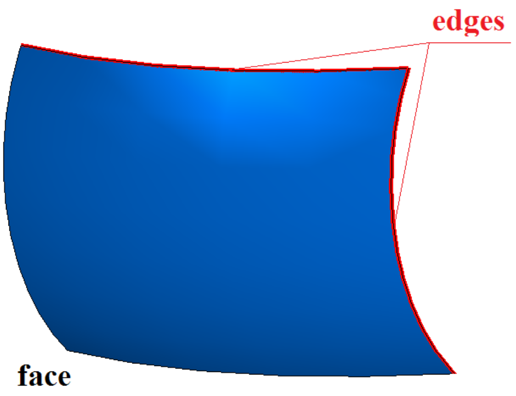


Fig. M.1.16.2.

Fig. M.1.16.3 and M.1.16.4 show the results of face extension by expanding surface parameter definition area using various values of *params*.kind parameter.

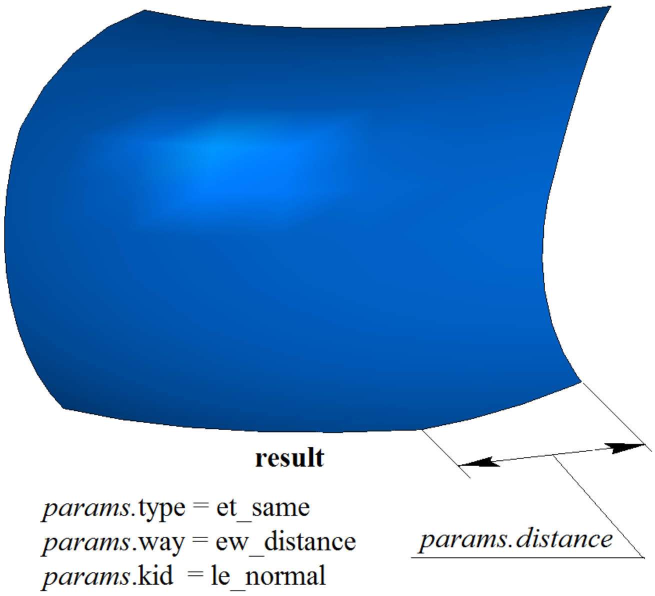


Fig. M.1.16.3.

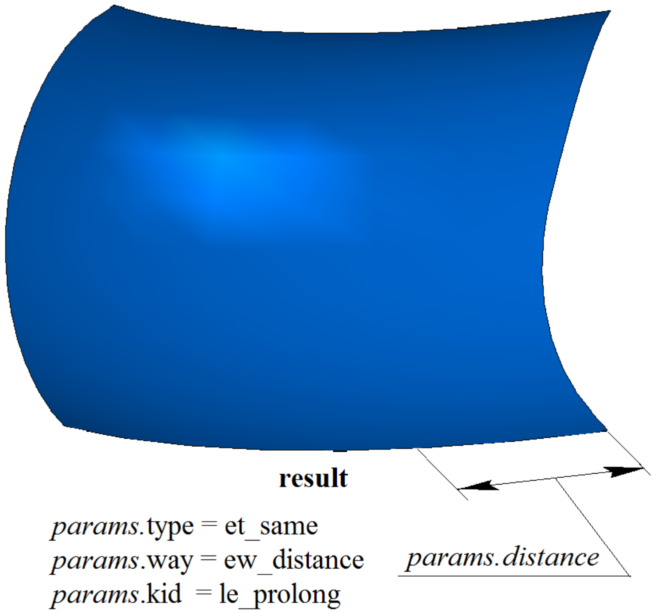


Fig. M.1.16.4.

Fig. M.1.16.5 shows the result of **face** extension by adding other faces that smoothly join with **face**. Fig. M.1.16.6 shows the result of extending **face** in the selected direction.

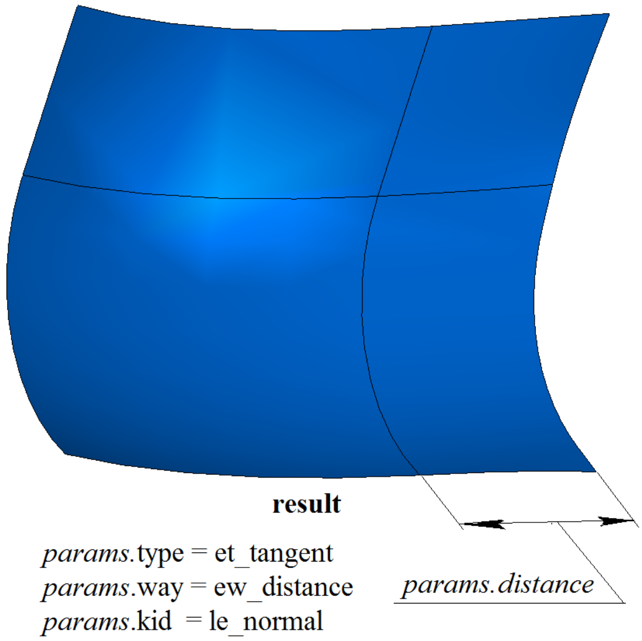


Fig. M.1.16.5.

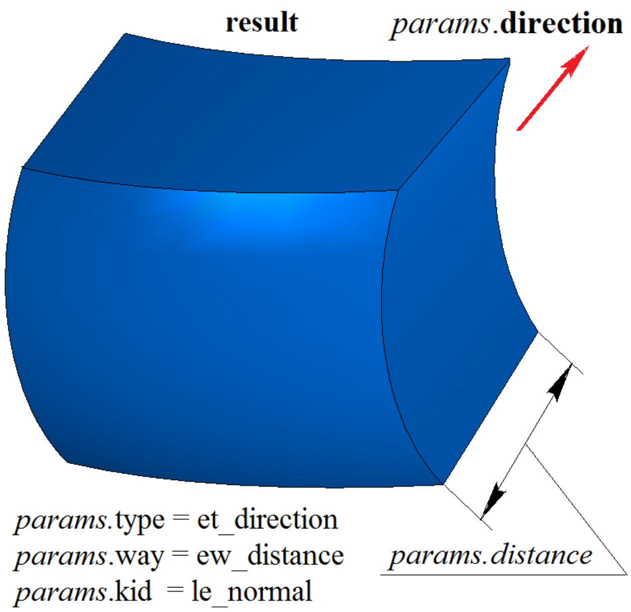


Fig. M.1.16.6.

ExtensionShell method adds MbExtensionShell constructor containing all data required to construct the body into a log of the newly constructed body. MbExtensionShell constructor is declared in cr_extension_shell.h file.

M.2. OPERATIONS ON BODIES

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

M.2.1. Boolean Operation on Bodies

The method

```
MbResultType BooleanResult( const c3d::SolidSPtr &           solid1,
                                  MbeCopyMode                sameShell1,
                            const c3d::SolidSPtr &           solid2,
                                  MbeCopyMode                sameShell2,
                            const MbBooleanOperationParams & params,
                                  c3d::SolidSPtr &           result );
```

constructs a new body by executing a Boolean operation on two specified bodies.

Input parameters of the method are as follows:

- **solid1** - the first body for the Boolean operation;
- **sameShell1** - copying method for the first body;
- **solid2** - the second body for the Boolean operation;
- **sameShell2** - copying method for the second body;
- **params** - the parameters of the boolean operation;

Method output parameter:

- **result** - result constructed body.

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 bodies (**solid1** and **solid2**). *sameShell*1 and *sameShell*2 parameters control transfer of faces, edges and vertices of **solid1** and **solid2** original bodies to **result** constructed body.

*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.

The `MbBooleanOperationParams` class contains parameters for performing a Boolean operation. When creating an object of the `MbBooleanOperationParams` class, one of the following constructors is used:

```
MbBooleanOperationParams(       OperationType        oType,
                          const MbBooleanFlags     & flags,
                          const MbSNameMaker       & operNames,
                                IProgressIndicator * progress = nullptr )

MbBooleanOperationParams(       OperationType        oType,
                                bool                 closed,
                          const MbSNameMaker       & operNames,
                                IProgressIndicator * progress = nullptr )
```

where,

- **oType** - type of boolean operation;
- **flags** - boolean operation control flags;
- **operName** - an object for naming the new objects;
- **closed** - closedness of operands' shells;
- **progress** - progress indicator.

The variant of the `MbBooleanOperationParams` object with the closed parameter works only for non-closed bodies and tells the operation to check the result for closedness.

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** bodies; if *oType*=bo_Intersect, then the method intersects **solid1** and **solid2** bodies; if *oType*=bo_Difference, then the method subtracts **solid2** body from **solid1** body.

The `MbSNameMaker operNames` parameter is used to name the resulting boolean operation objects. The `MbBooleanFlags flags` parameter contains the control flags for the boolean operation.

The object has the following flags, which can be set in the constructor, when the object is created, or with separate setter functions:

```
bool mergeFaces;              - Whether to merge similar faces;
bool mergeEdges;              - Whether to merge similar edges;
bool closed;                  - Closedness of operands' shells;
bool enclosureCheck;          - Check shell on nesting.;
bool allowNonIntersecting;    - Allow a final result if there is no intersection;
bool cutting;                 - Flag of cutting the shell in the construction of cuts and sections;
bool repairShellEdges;        - Flag of input shells edges repair.
```

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

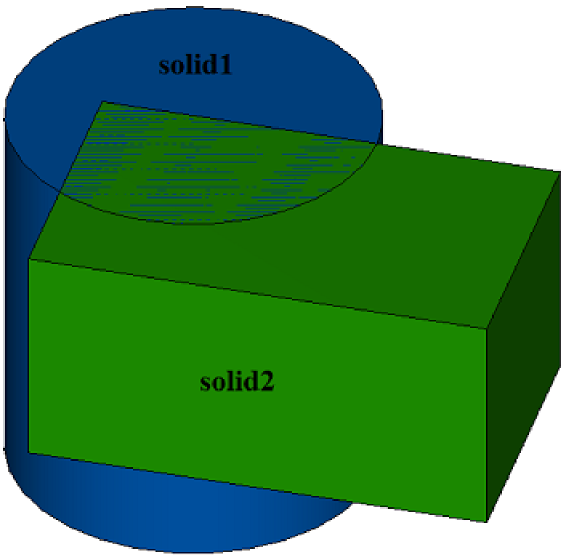
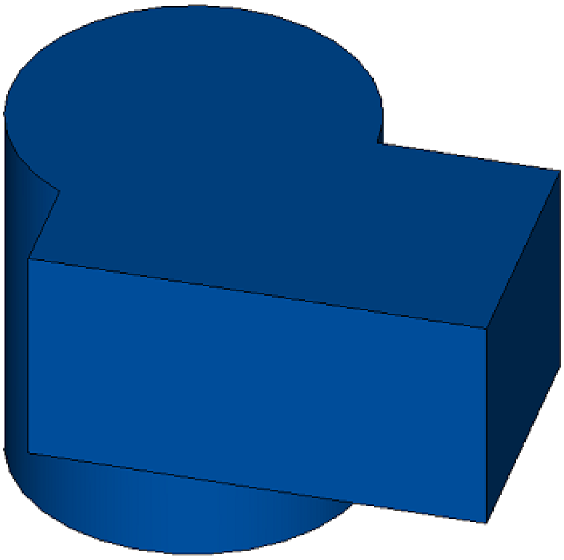


Fig. M.2.1.1.

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

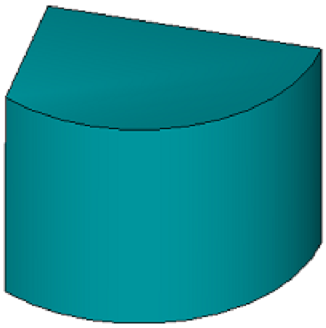


oType = bo_Union

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** bodies shown in Figure M.2.1.1.

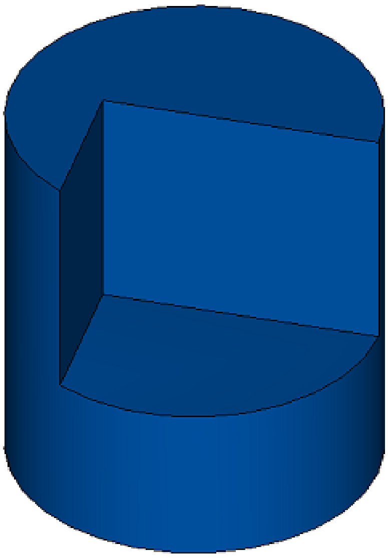


oType = bo_Intersect

solid1 & solid2

Fig. M.2.1.3.

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

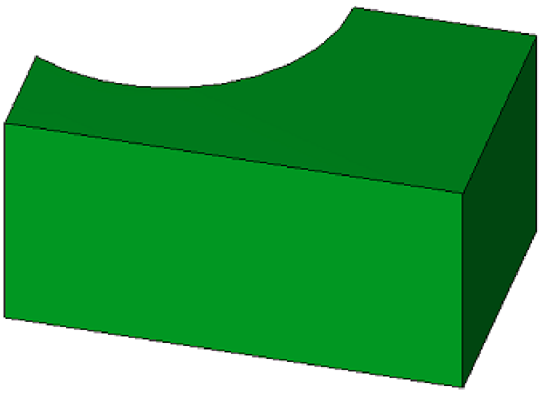


oType = bo_Difference

solid1 - solid2

Fig. M.2.1.4.

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



oType = bo_Difference

solid2 - solid1

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 bodies shown in Figure M.2.1.6.

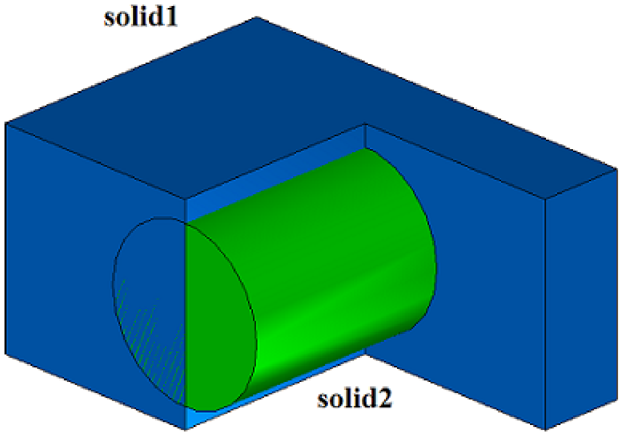


Fig. M.2.1.6.

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

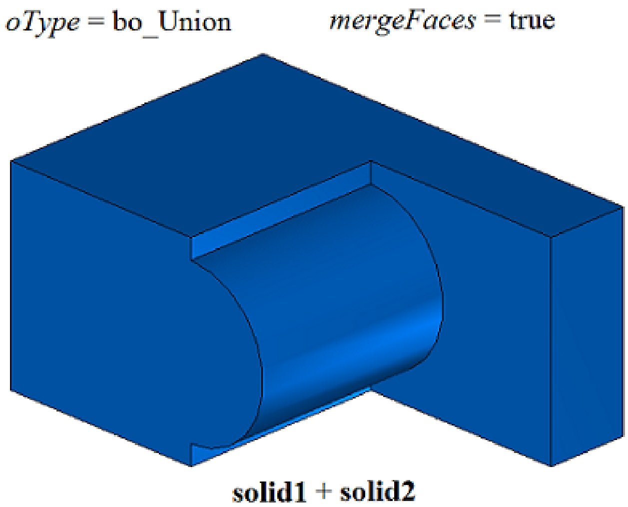


Fig. M.2.1.7.

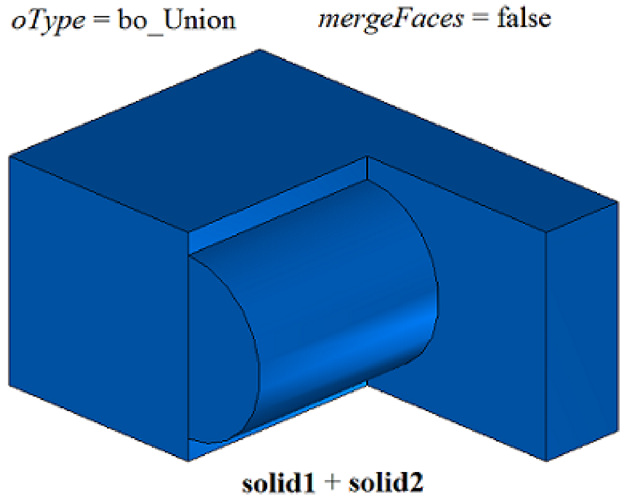


Fig. M.2.1.8.

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

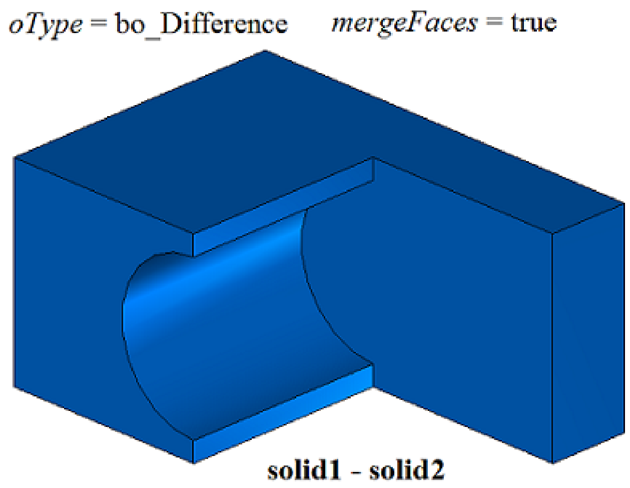


Fig. M.2.1.9.

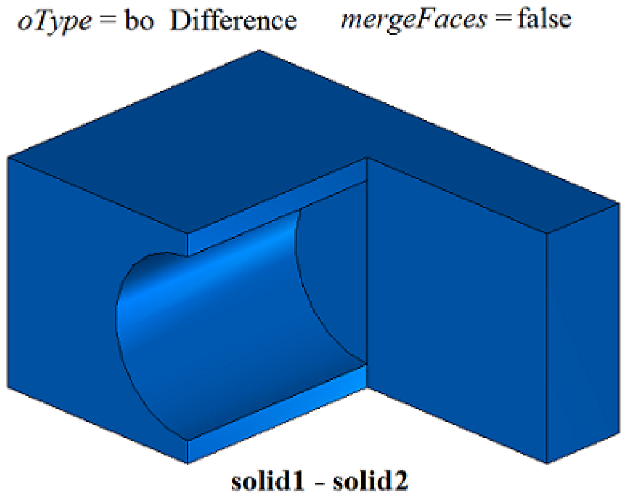


Fig. M.2.1.10.

BooleanResult and **BooleanSolid** methods add MbBooleanSolid constructor in the log of newly constructed body 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 bodies using New ->Body -> By Gluing to Body -> a Body, New ->Body -> By cutting from Body ->Body, New ->Body -> Intersecting with Body ->Body menu commands.

M.2.2. Boolean Operation on Non-Closed Bodies

The BooleanShell function has been deprecated. Instead, you can use the BooleanResult function with the MbBooleanResultParams parameter.

M.2.3. Boolean Operation on Extrusion Body

The 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 body and executes Boolean operation on the given body using the constructed body.

Input parameters of the method are as follows:

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

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

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 body by extruding **sweptData** curves according to given *params* parameters in **direction** and **BooleanSolid** method that executes *oType* Boolean operation on **solid** body that was constructed on the previous step. **ExtrusionSolid** method is described in item M.1.3. Constructing an Extrusion B, and **BooleanSolid** method is described in item M.2.1. Boolean Operation on. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original body to **result** constructed body.

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** body and the extruded body; if *oType*=bo_Intersect, then the method intersects **solid** body and the extruded body; if *oType*=bo_Difference, then the method subtracts the extruded body from **solid** body. names and snames parameters provide naming of the faces for newly constructed body.

If the body 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 body 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** body, a generating curve included in **sweptData** data and extrusion

direction.

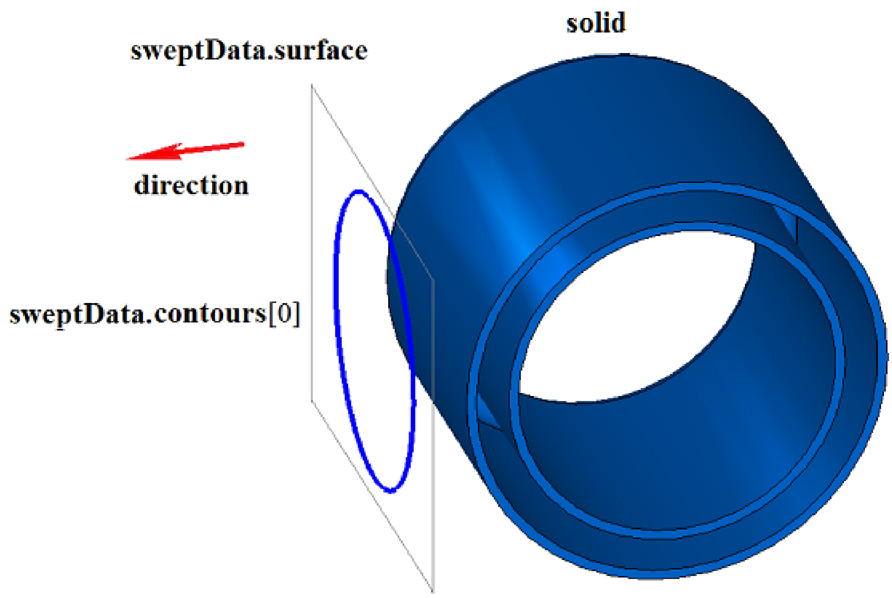


Fig. M.2.3.1.

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

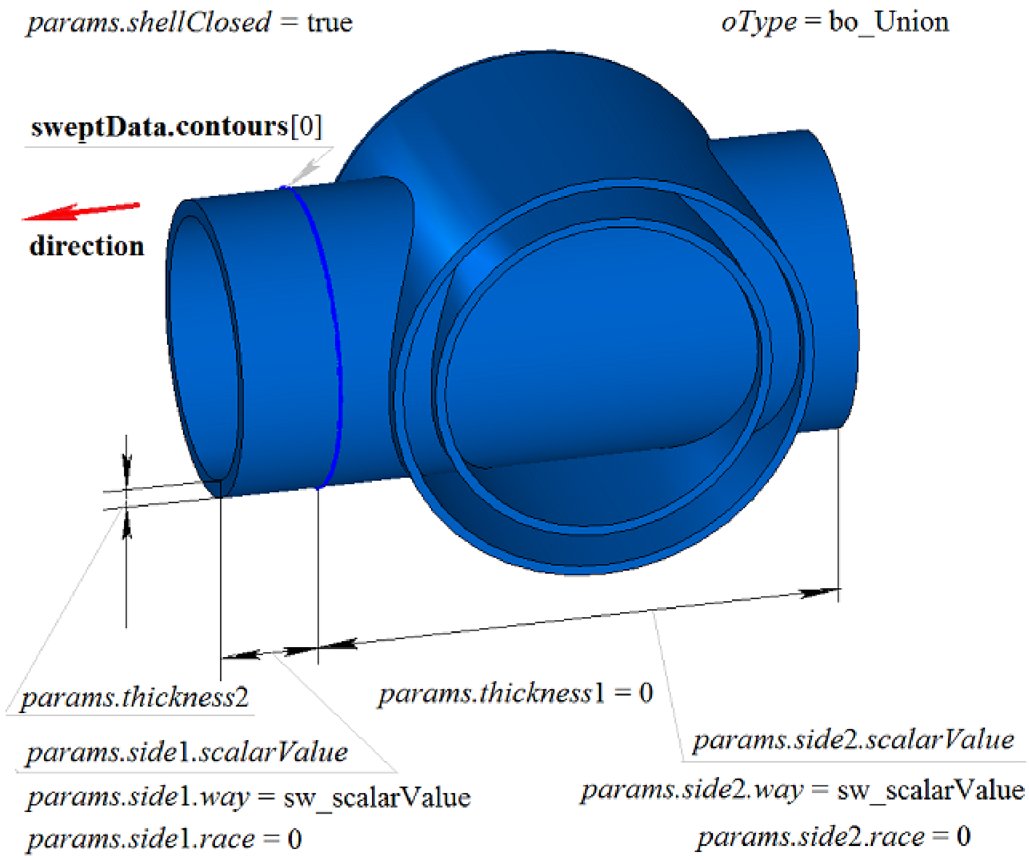


Fig. M.2.3.2.

In Fig. M.2.3.3, you can see the result of Boolean operation that merges **solid** body and a body 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).

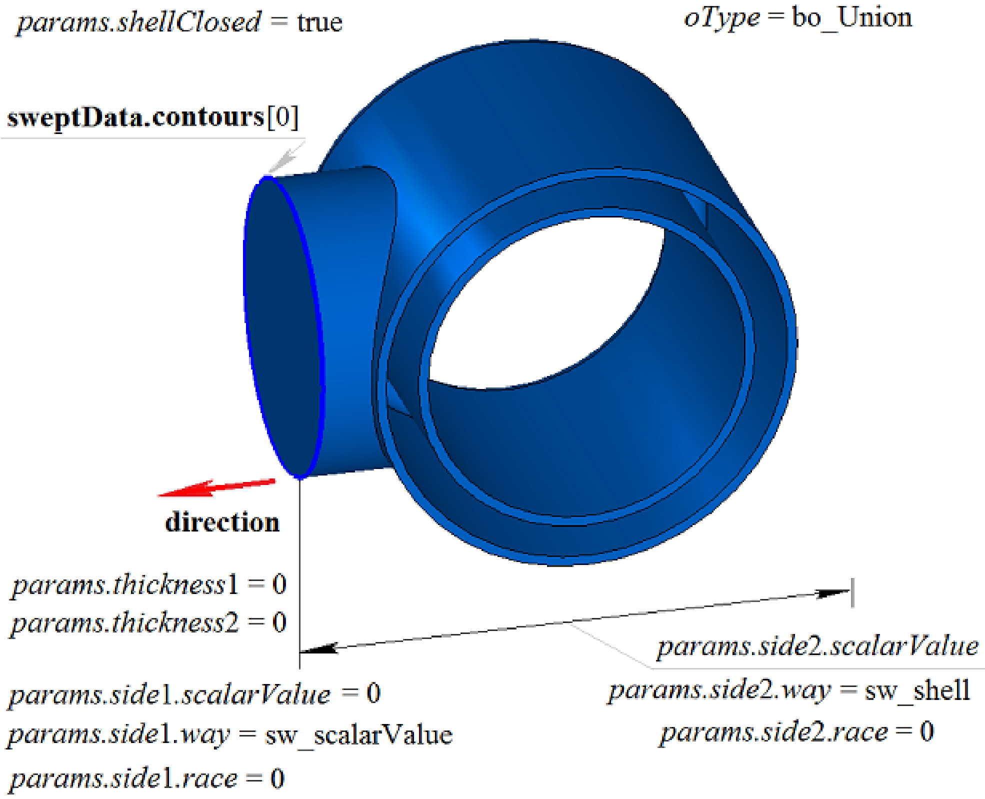


Fig. M.2.3.3.

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

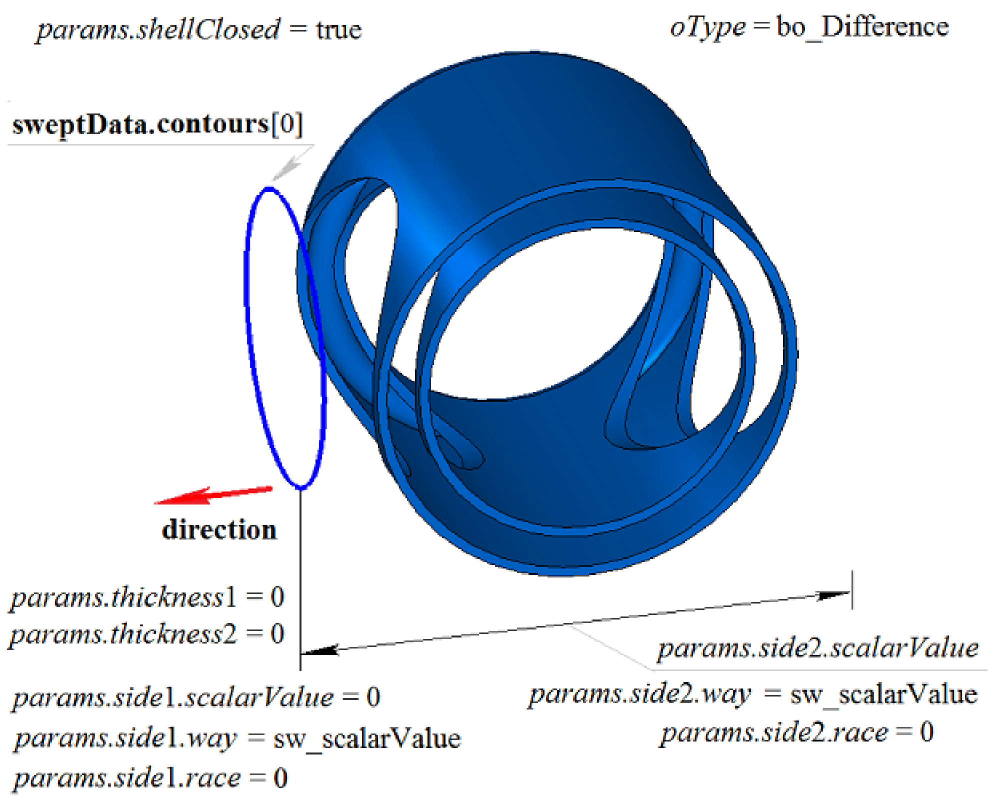


Fig. M.2.3.4.

In Fig. M.2.3.5, you can see the result of Boolean operation that subtracts a body received by extrusion of **sweptData** generating curve from **solid** body 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).

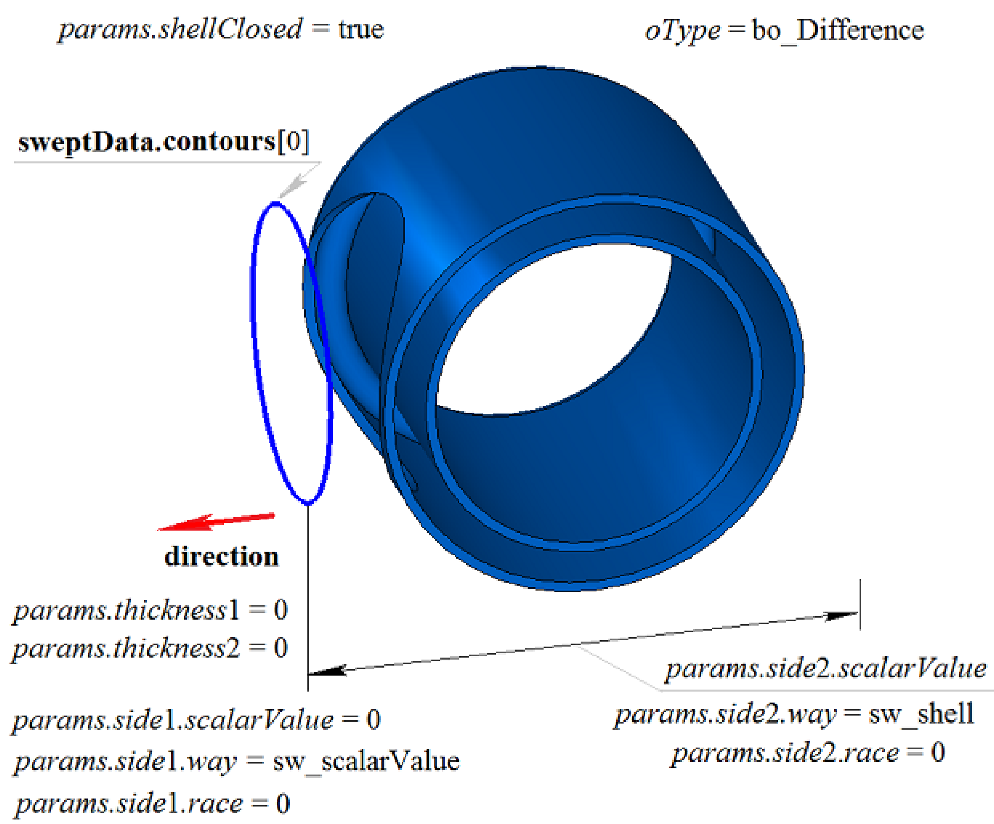


Fig. M.2.3.5.

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

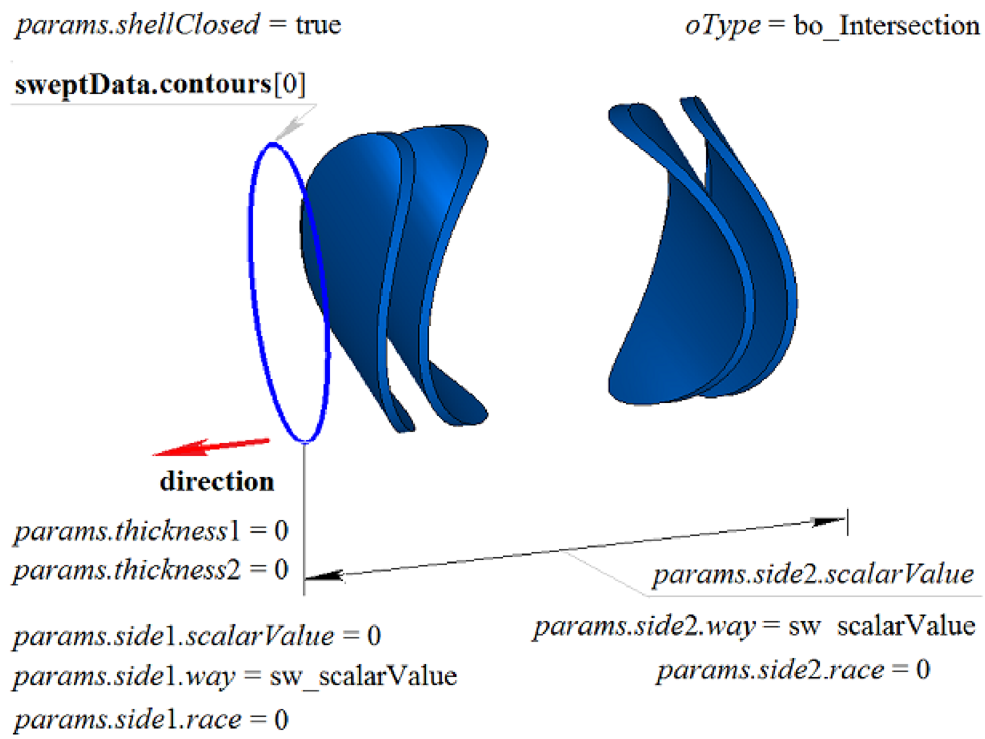


Fig. M.2.3.6.

In Fig. M.2.3.7, you can see the result of the Boolean operation that intersects **solid** body and a body 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.

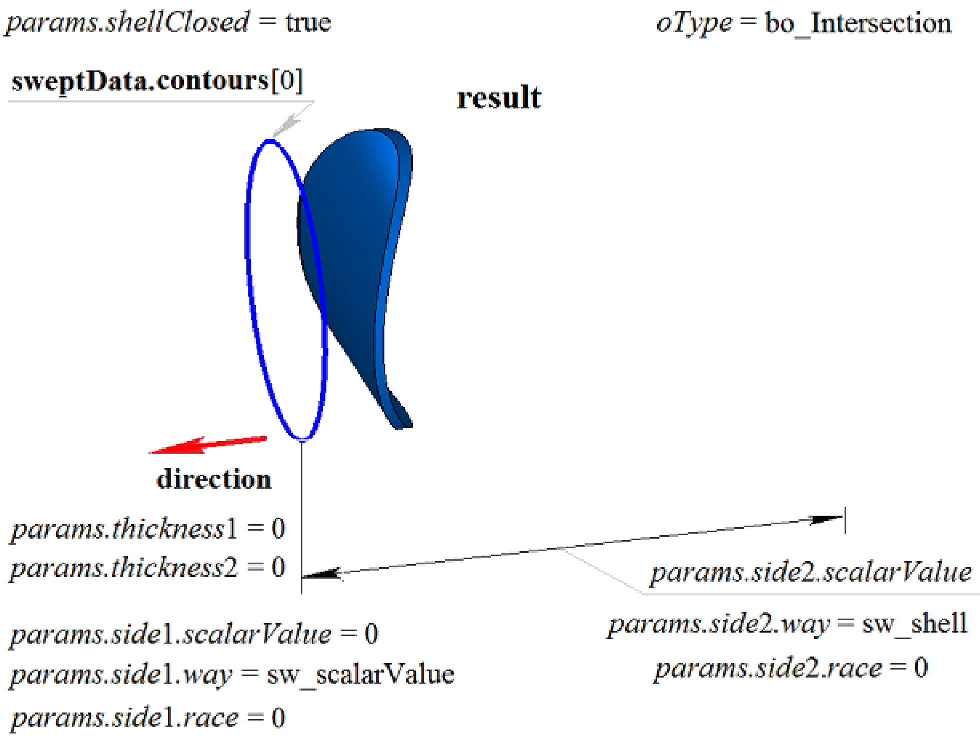


Fig. M.2.3.7.

ExtrusionResult method adds MbExtrusionSolid constructor in the log of newly constructed body 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 body recieved by extruding a body using New ->Body -> By Gluing to Body -> By Extruding Curve, New ->Body -> Cut from Body -> By Extruding Curve, New ->Body -> Intersection with Other Body -> By Extruding Curve menu commands.

M.2.4. Boolean Operation on Revolution Body

The 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 body and executes a Boolean operation of determined body with constructed body.

Input parameters of the method are as follows:

- **solid** is a body given for Boolean operation,
- *sameShell* is copying version for the given body,
- **sweptData** contains data on generating curves for construction of extruded body,

- **axis** is a rotation axis,
- *params* are construction parameters,
- *oType* is Boolean operation type: bo_Union means merging of the bodies,
  bo_Intersect means intersection of the bodies,
  bo_Difference means subtraction of the bodies,
- names is operation namer,
- snames are namers of rotation body faces.

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

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

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** body and a revolution body; if *oType*=bo_Intersect, then the method intersects **solid** body and a revolution body; if *oType*=bo_Difference, then the method subtracts a revolution body from **solid** body. names and snames parameters provide naming of the faces for newly constructed body.

When a body 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 body 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** body, generating curve included in **sweptData** data and **axis** rotation axis.

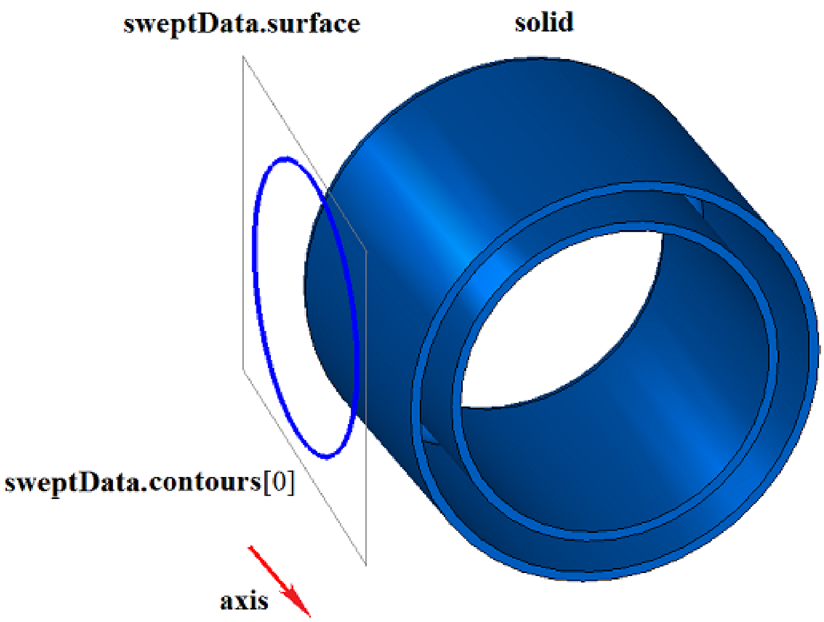


Fig. M.2.4.1.

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

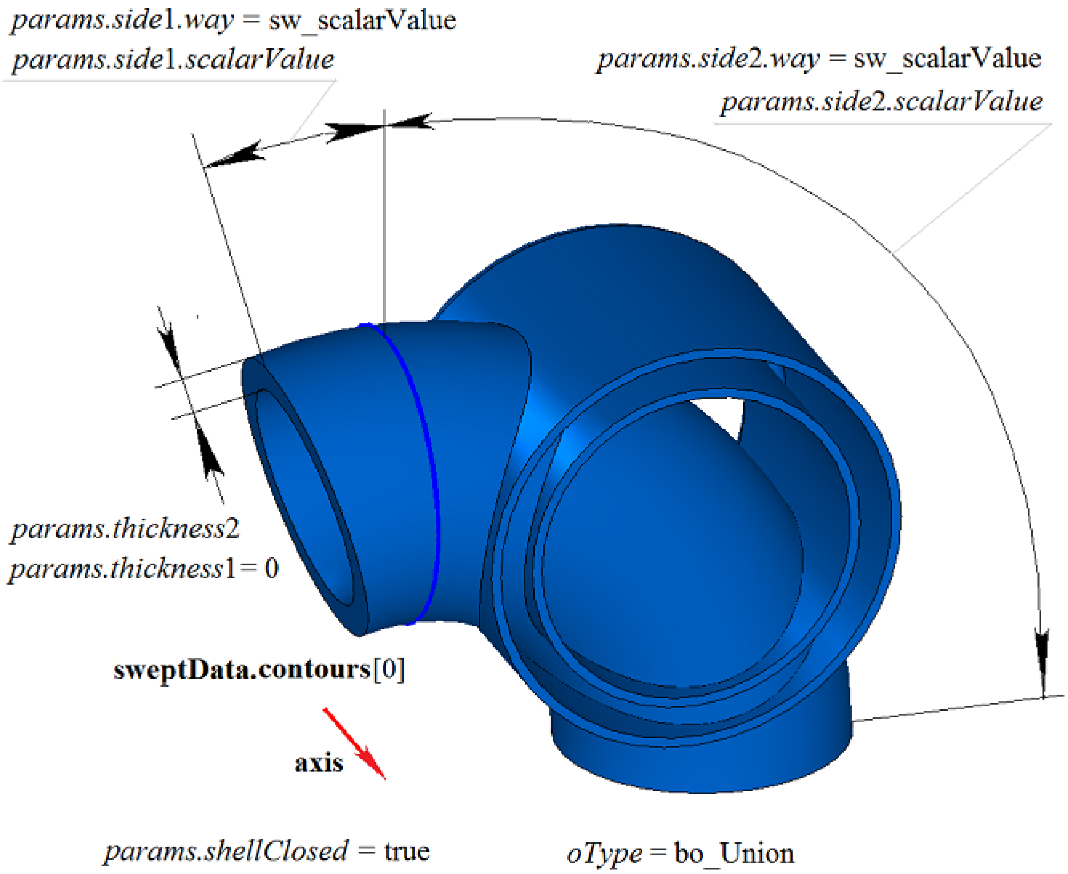


Fig. M.2.4.2.

In Fig. M.2.4.3, you can see the result of Boolean operation that merges **solid** body and a thin-walled body 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.side2.way*=sw_surface).

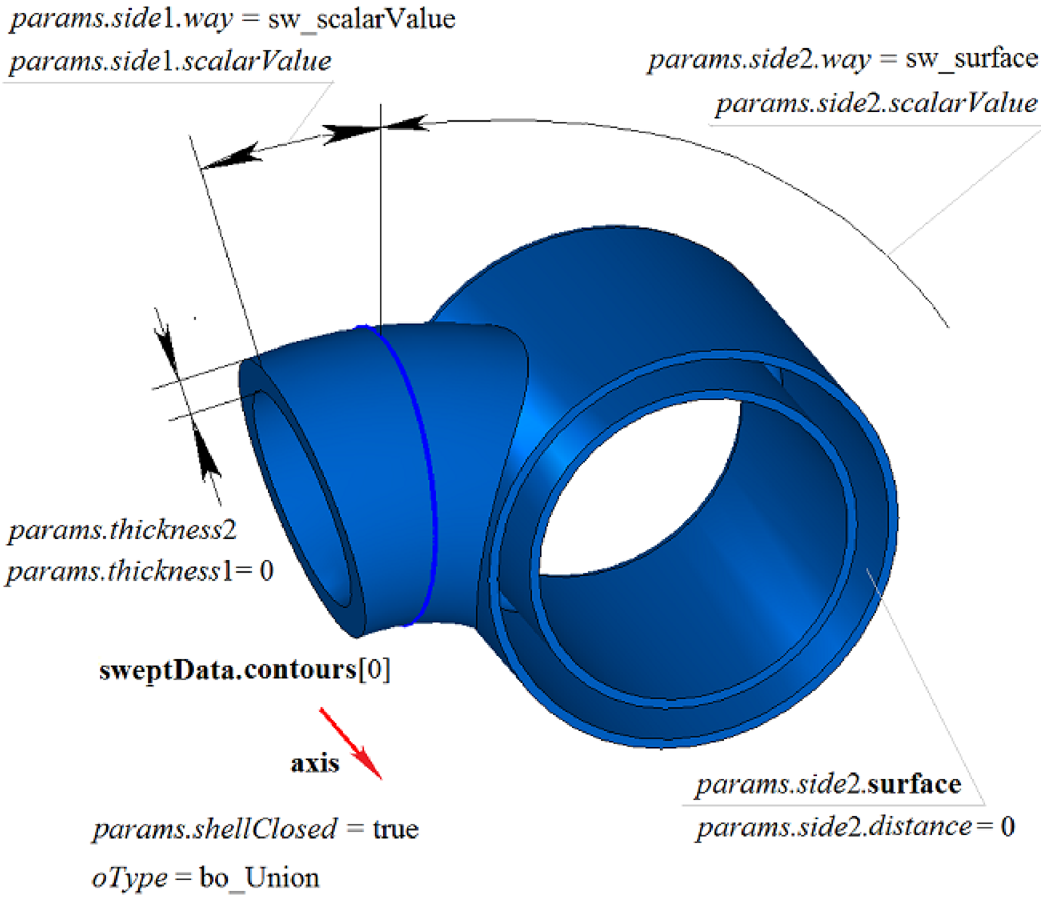


Fig. M.2.4.3.

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

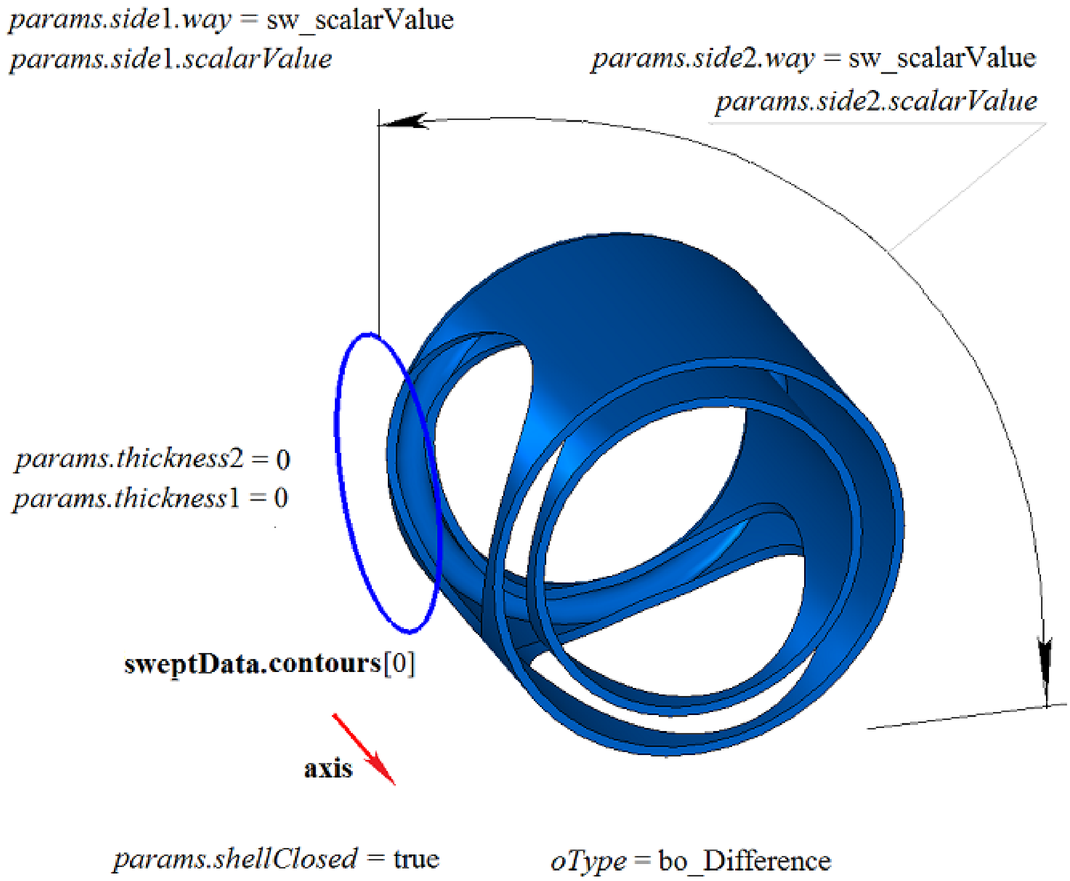


Fig. M.2.4.4.

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

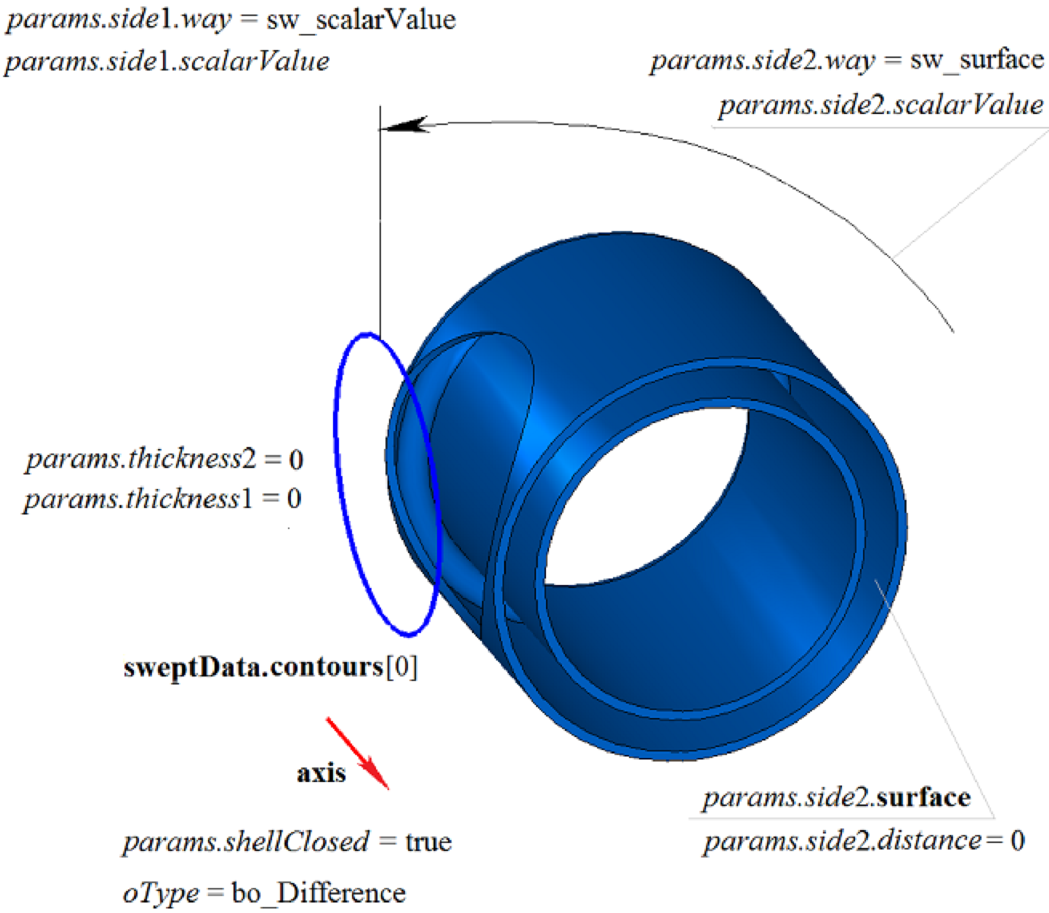


Fig. M.2.4.5.

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

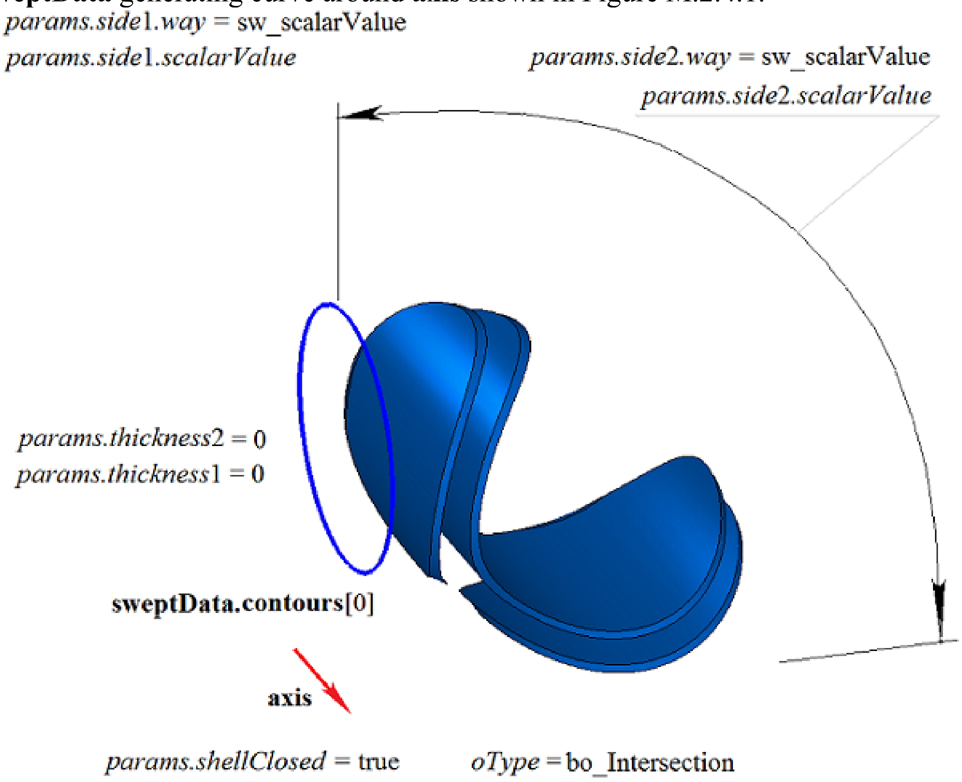


Fig. M.2.4.6.

In Fig. M.2.4.7, you can see the result of Boolean operation that intersects **solid** body and a body 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).

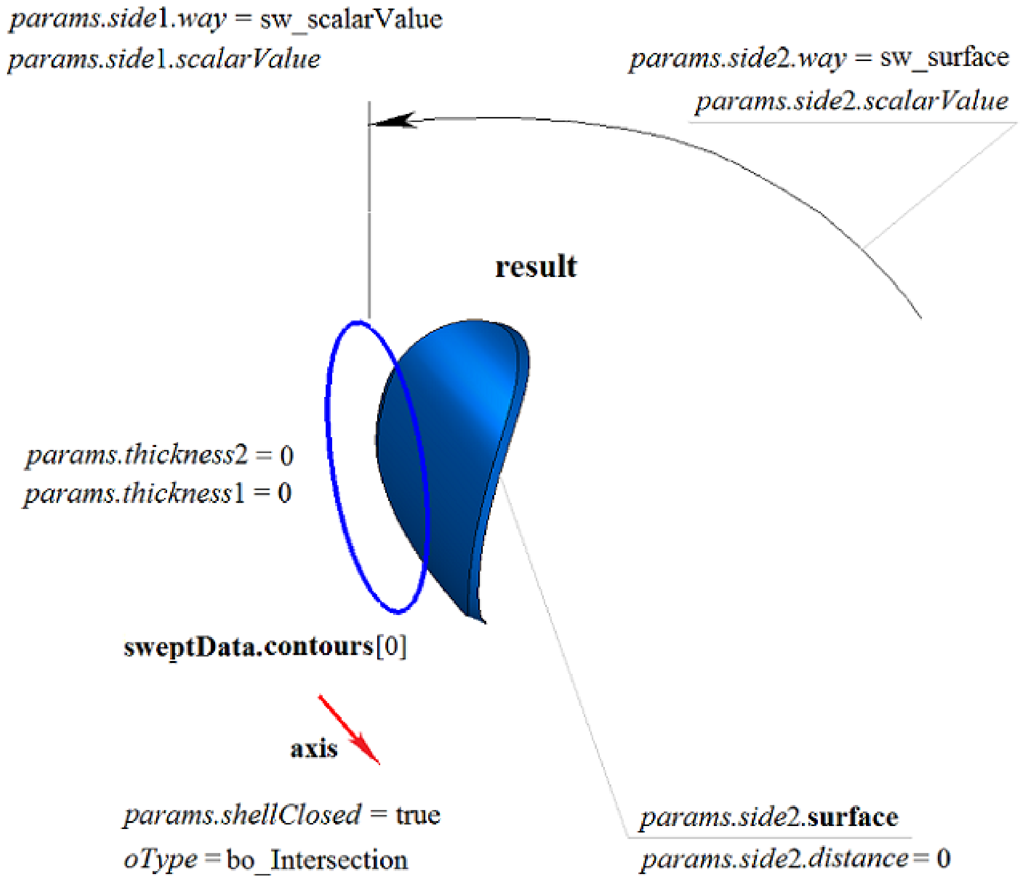


Fig. M.2.4.7.

RevolutionResult method adds MbRevolutionSolid constructor in the log of newly constructed body 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 body using New ->Body -> Attach to Other Body -> Curve Rotation, New ->Body -> Cut from Other Body -> Curve Rotation, New ->Body -> Intersection with Other Body -> Curve Rotation menu commands.

M.2.5. Boolean Operation on Swept Body

The 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 body and executes a Boolean operation on newly given body with newly constructed

body.

Input parameters of the method are as follows:

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

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

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

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** body and the swept body; if *oType*=bo_Intersect, then the method intersects **solid** body and the swept body; if *oType*=bo_Difference, then the method subtracts the swept body from **solid** body. names, cnames and snames parameters provide face naming for newly constructed body.

When **EvolutionResult** method constructs bodies 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 body 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** body, a generating curve included in **sweptData** data and **spine** guiding curve.

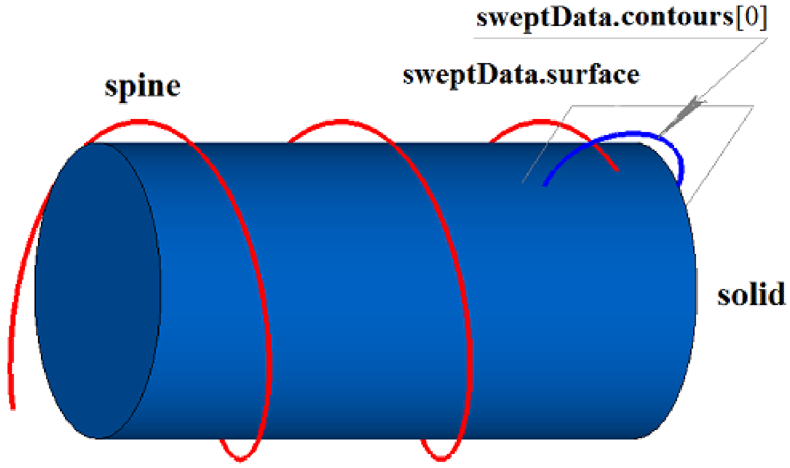


Fig. M.2.5.1.

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

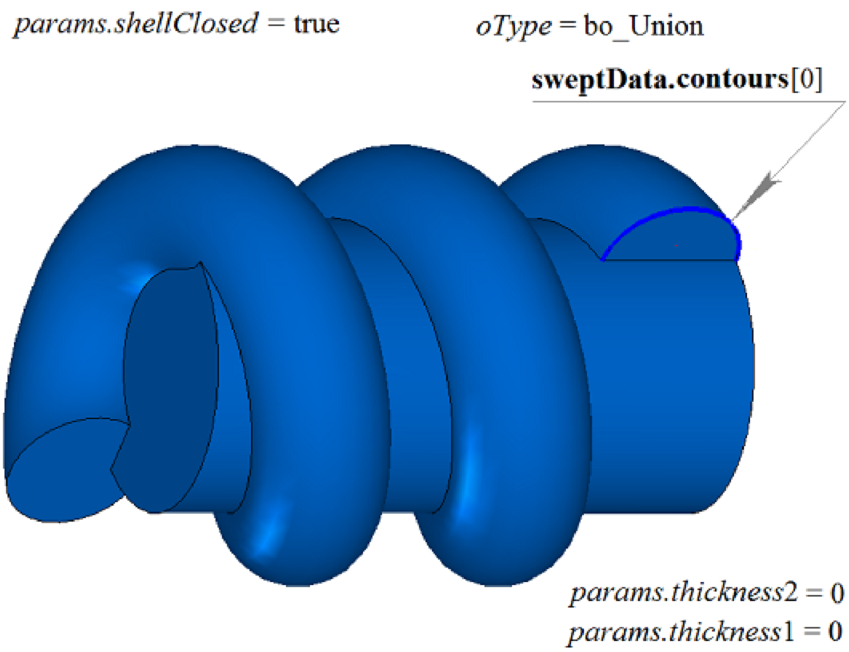


Fig. M.2.5.2.

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

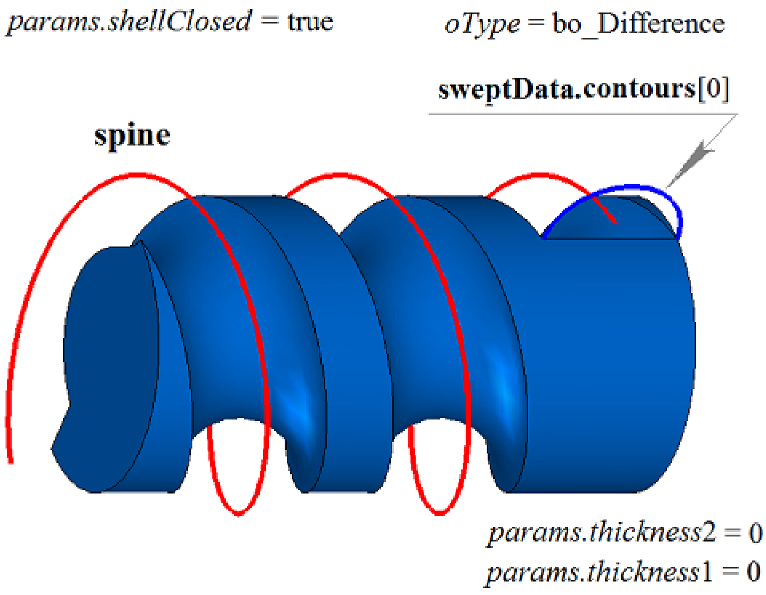


Fig. M.2.5.3.

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

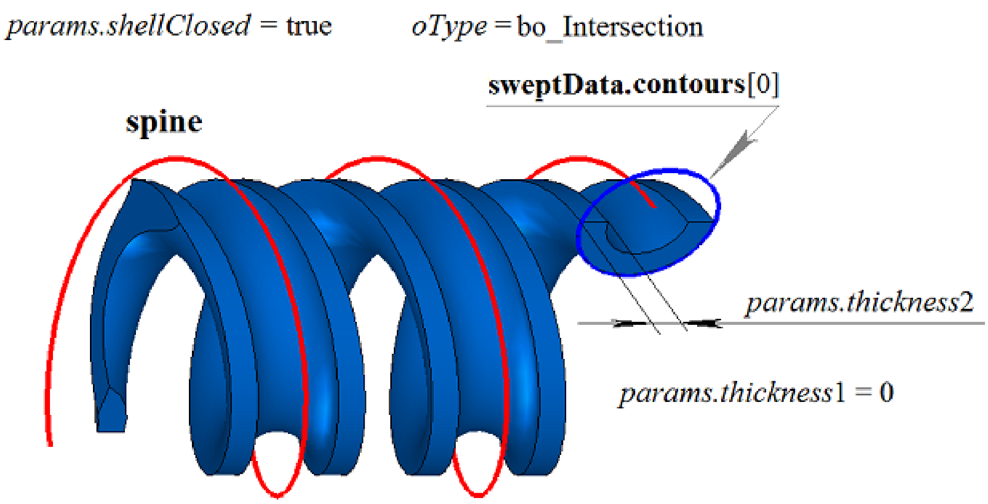


Fig. M.2.5.4.

EvolutionResult method adds MbEvolutionSolid constructor to the log of newly constructed body 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 body using New ->Body -> Attach to Other Body -> By Moving Curve, New ->Body -> Cut from Other Body -> By Moving a Curve, New ->Body -> By Intersection with Other Body -> By Moving a Curve menu commands.

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

The 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 body based on flat sections and executes Boolean operation on the specified body with newly constructed body.

Input parameters of the method are as follows:

- **solid** is a body given for Boolean operation,
- *sameShell* is copying version for the given body,
- **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 bodies,
  bo_Intersect means intersection of the bodies,
  bo_Difference means subtraction of the bodies,
- **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 body.

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

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** body and the swept body; if *oType*=bo_Intersect, then the method intersects **solid** body and the swept body; if *oType*=bo_Difference, then the method subtracts the swept body from **solid** body. names, cnames and snames parameters provide face naming for newly constructed body.

When **LoftedResult** method constructs bodies based on flat sections, it provides the same possibilities as **LoftedSolid** method: constructed body may be built as non-closed (Figure M.1.6.3) or as closed one (Figure M.1.6.4); the body may have various shapes near the ends (Figure M.1.6.5 and Figure M.1.6.6); the body may completely fill the closed curves (Figure M.1.6.3) or it may have a thin wall (Figure M.1.6.6); body 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** body and closed guiding curves.

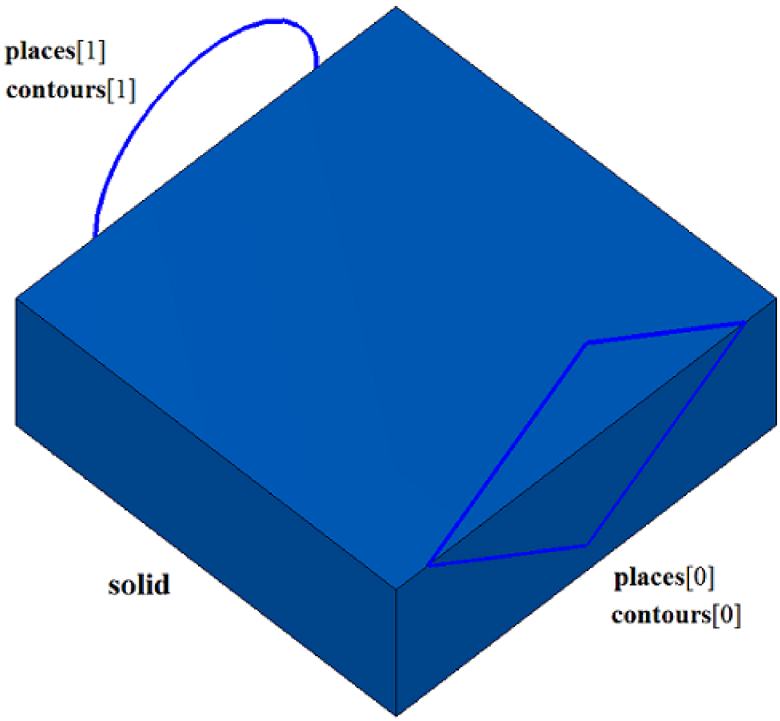


Fig. M.2.6.1.

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

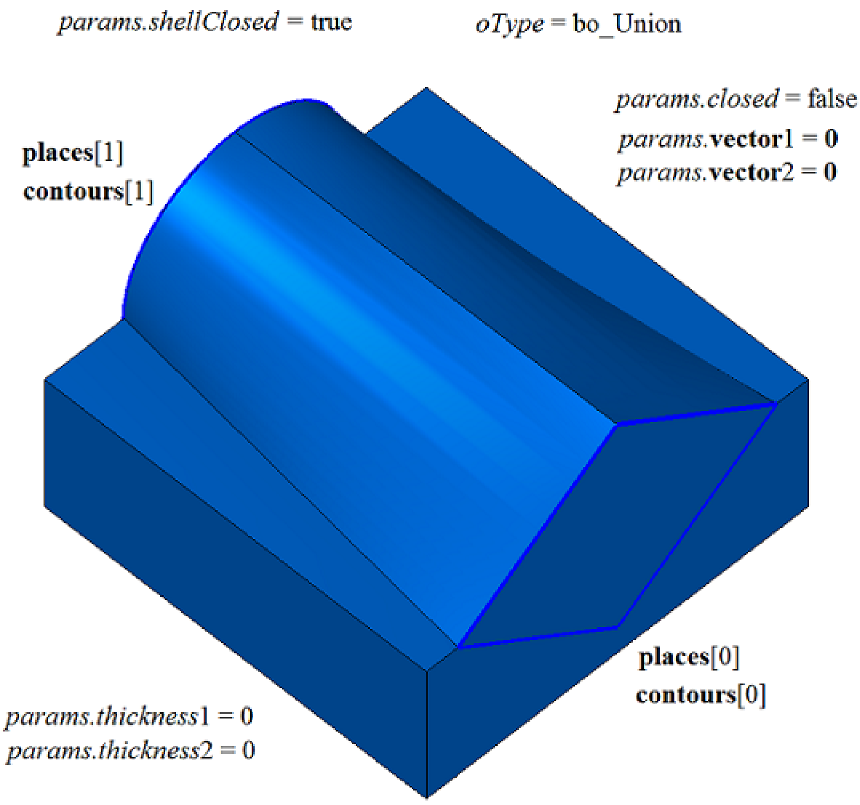


Fig. M.2.6.2.

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

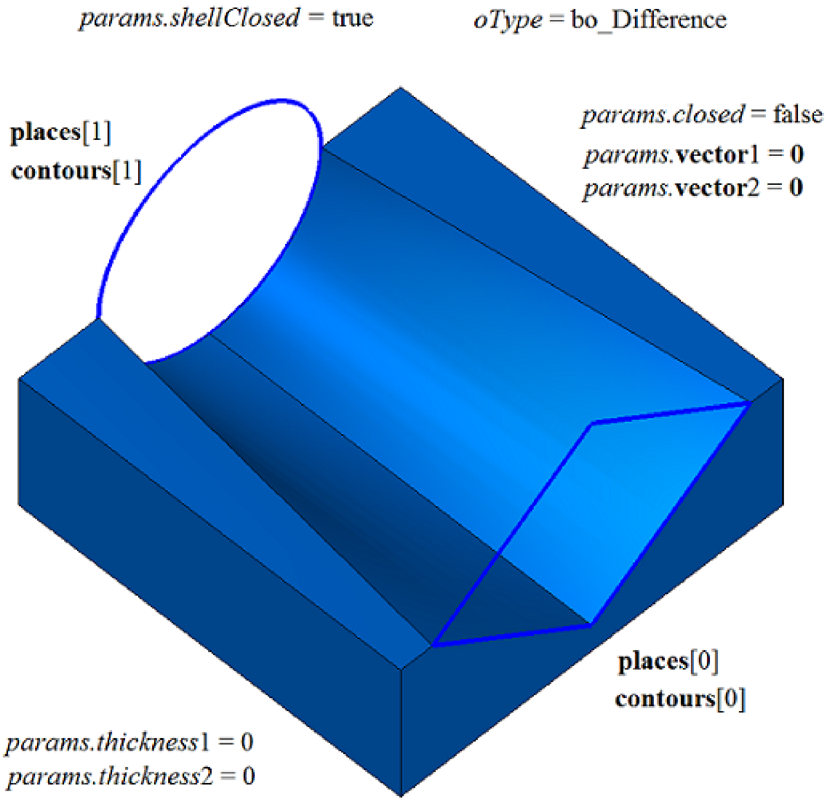


Fig. M.2.6.3.

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

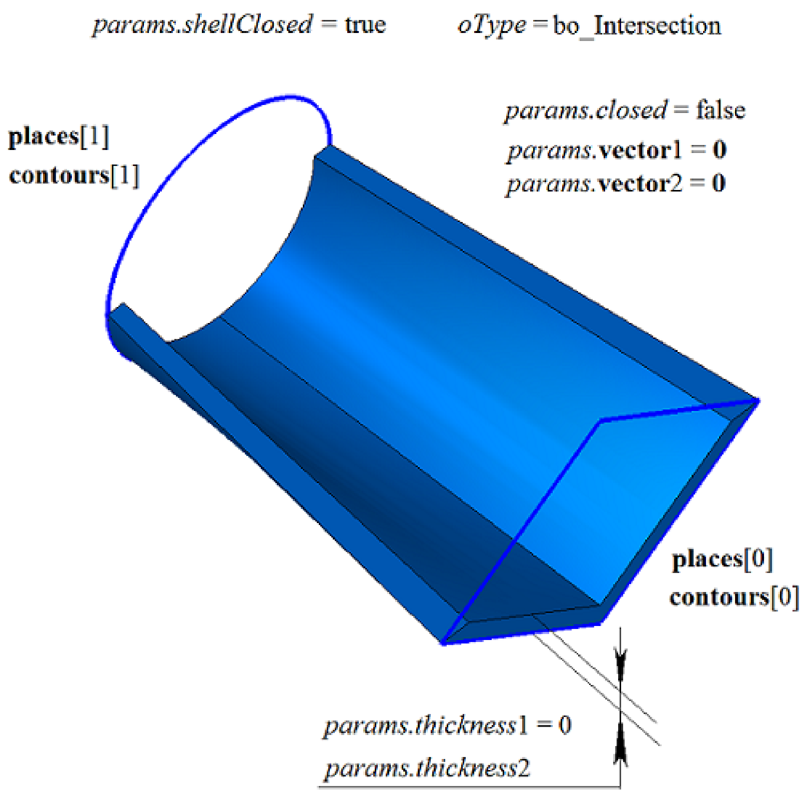


Fig. M.2.6.4.

LoftedResult method adds MbLoftedSolid constructor in the log of newly constructed body 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 body constructed based on flat sections using New ->Body -> Attach to Other Body -> By Sections, New ->Body -> Attach to Other Body -> By Sections and Generating Curve, New ->Body -> Cut from Other Body -> By Sections, New ->Body -> Cut from Other Body -> By Sections and Generating Curve, New ->Body -> Intersection with Other Body -> By Sections, New ->Body -> Intersection with Other Body -> By Sections menu commands.

M.2.7. Cutting a Body by a Surface

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

Input parameters of the method are as follows:

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

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

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 body with a non-closed shell. To execute the operation, the cutting surface should fully intersect the original body. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original body to **result** constructed body.

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 body 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 body will be kept. names parameter is used to name the faces of the constructed body. *closed* parameter defines whether **solid** original body is closed or non-closed.

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

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

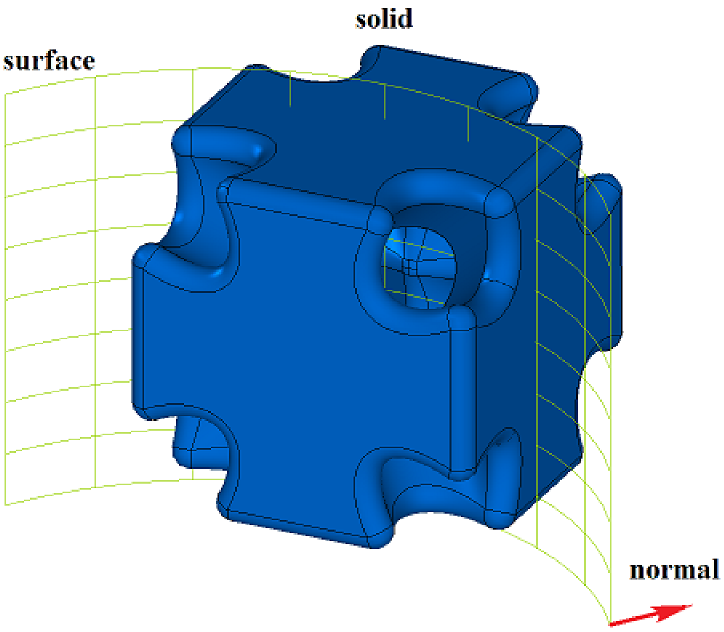


Fig. M.2.7.1.

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

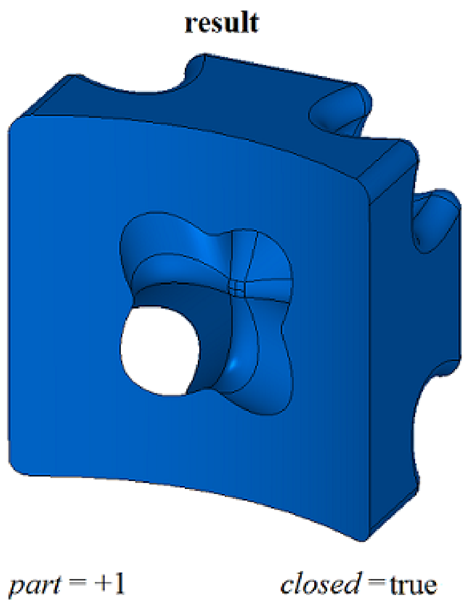


Fig. M.2.7.2.

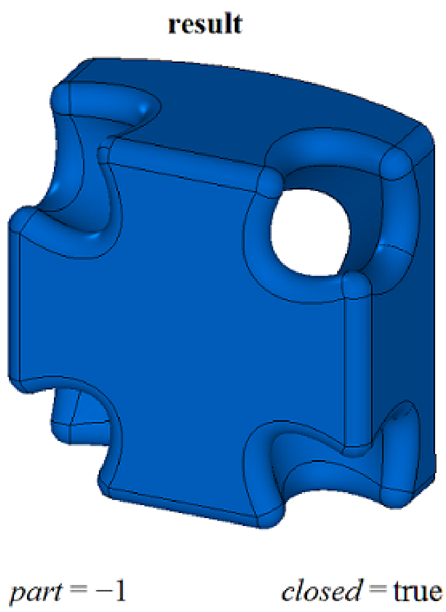


Fig. M.2.7.3.

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

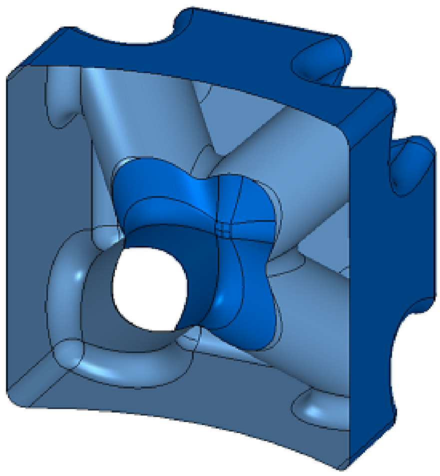


Fig. M.2.7.4.

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

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

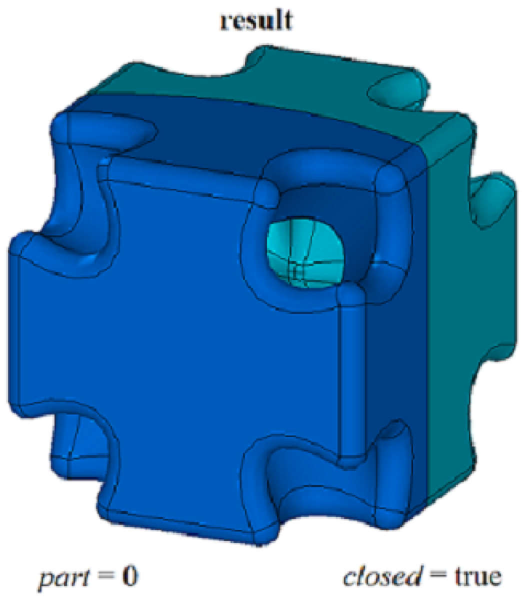


Fig. M.2.7.5.

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

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

M.2.8. Cutting a Body by a Flat Contour

The 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 body (constructed by extruding a flat contour) by a surface that intersects the body.

Input parameters of the method are as follows:

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

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

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

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 body 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 body 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 body. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original body to **result** constructed body.

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 body to be kept: If *part*=+1, then the part of the body to the right of the contour is to be kept; if *part*=–1, then the part of the body 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 body. *closed* parameter defines whether **solid** original body is closed or non-closed.

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

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

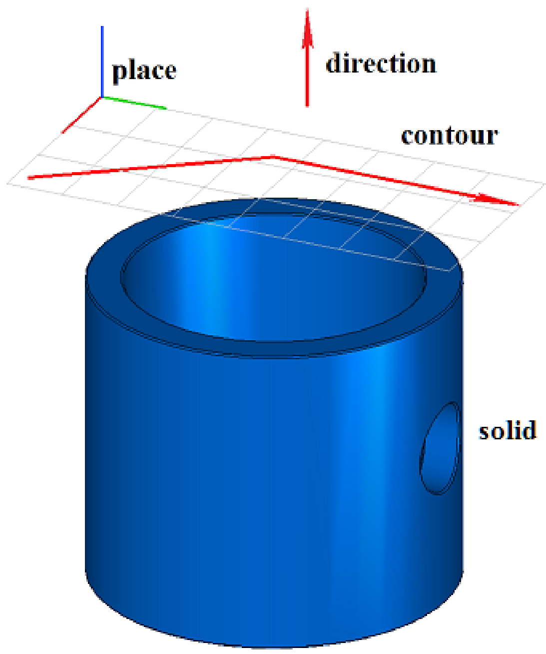
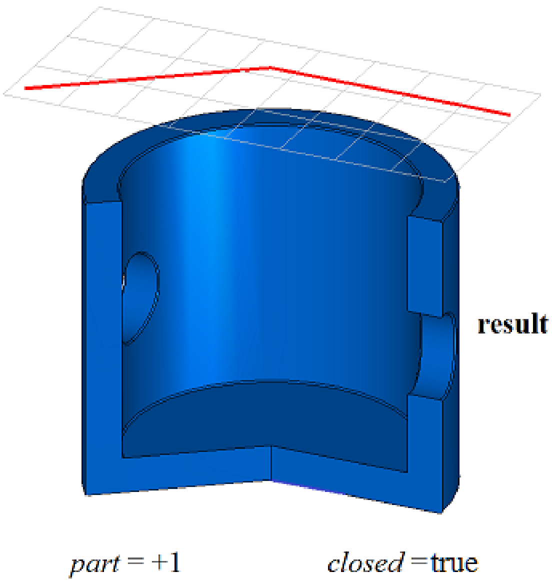


Fig. M.2.8.1.

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



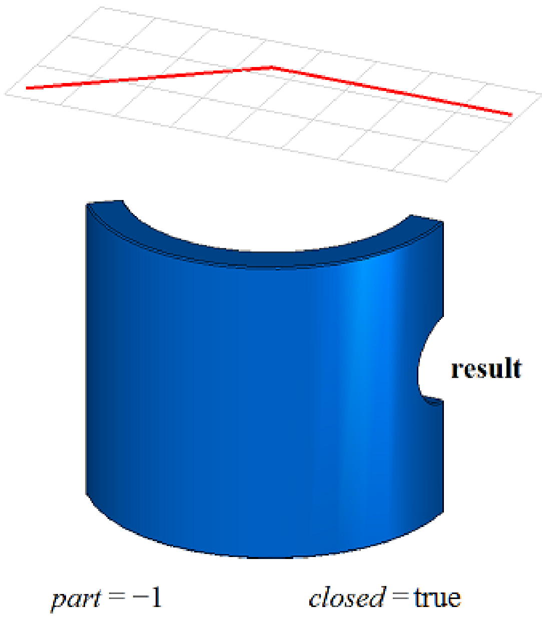


Fig. M.2.8.2. *Fig. M.2.8.3.*

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

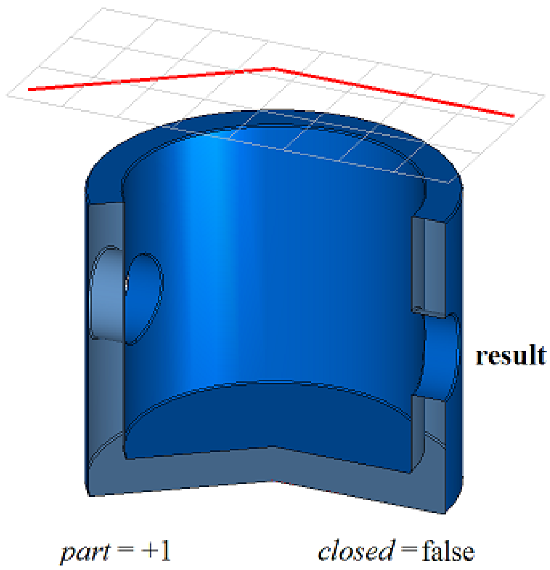


Fig. M.2.8.4.

If *part* = 0, then method constructs **result** body which contains all cutted parts of initial body. Method **DetachParts** or **CreateParts** can to detach part of **result** body. Methods **DetachParts** and **CreateParts** are described in item M.2.19. Divide a Body 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 body if *part*=0. Method has the same parameters besides *part*. In Fig. M.2.8.5, you can see **result** constructed bodies if *closed*=true.

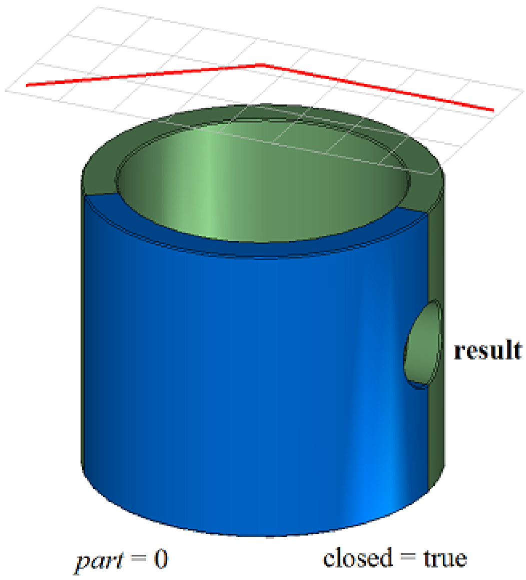


Fig. M.2.8.5.

SolidCutting methods adds MbCuttingSolid constructor in the log of newly constructed body 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 body with a surface using New ->Body -> Based on Body -> Cut with Curve and New -> Shell -> Based on Shell -> Cut with Curve menu commands.

M.2.9. Constructing a Symmetrical Body

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

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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

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 body and **place** symmetry plane.

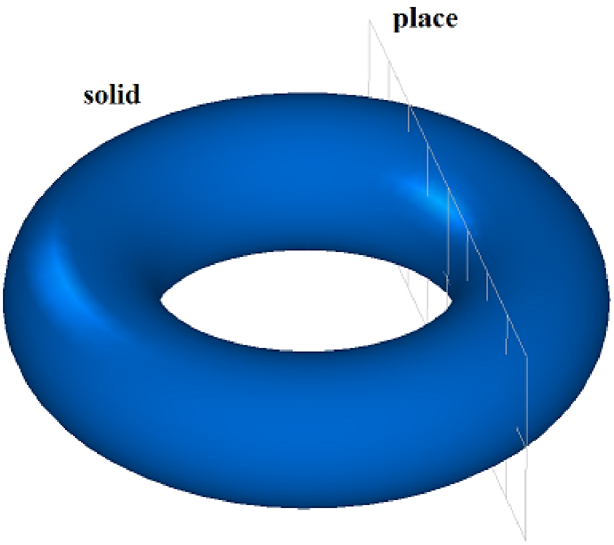


Fig. M.2.9.1.

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

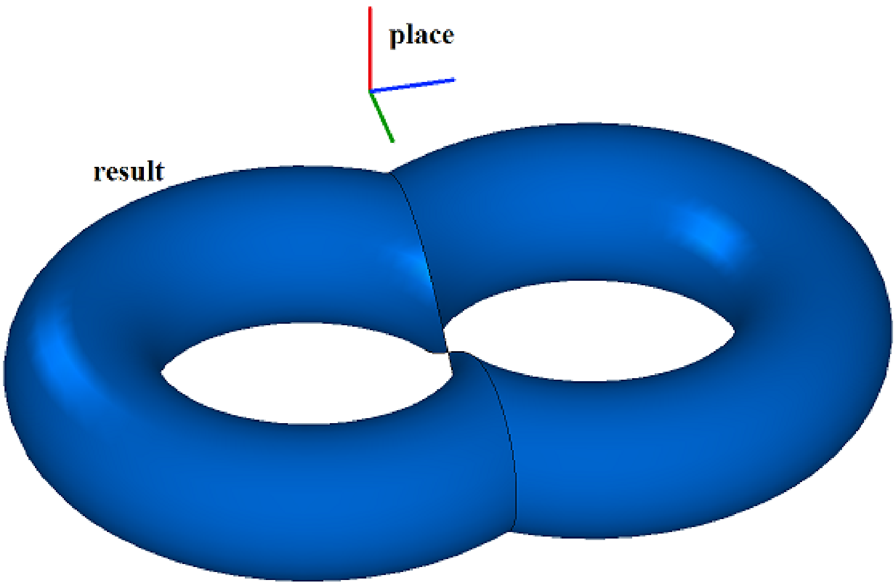


Fig. M.2.9.2.

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

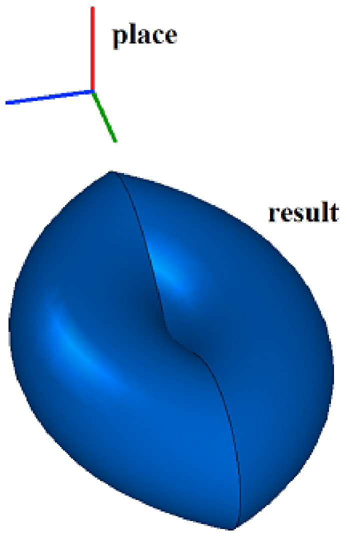


Fig. M.2.9.3.

If **solid** original body 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 body.

SymmetrySolid method adds MbSymmetrySolid constructor in the log of newly constructed body 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 body using New ->Body -> Based on Body -> Symmetrical menu command.

M.2.10. Rounding-off Body Edges

The 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 body.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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

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** body. **bounds** parameter contains faces of **solid** body 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.

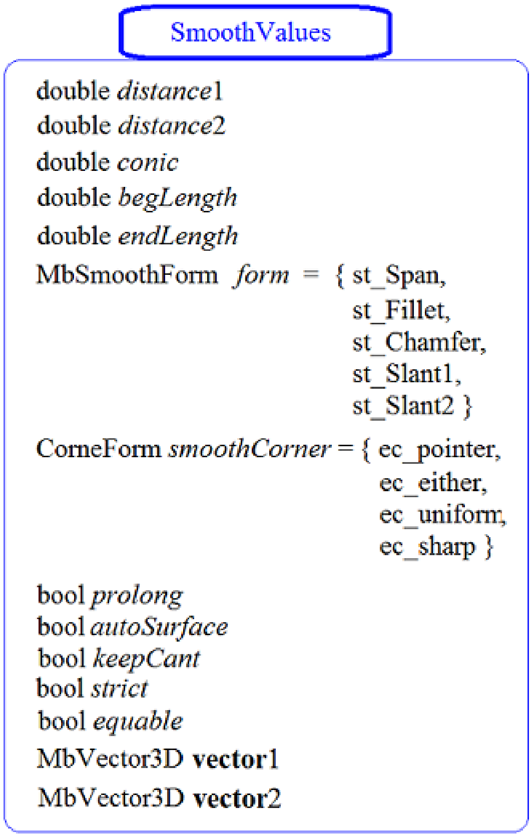


Fig. M.2.10.1.

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.

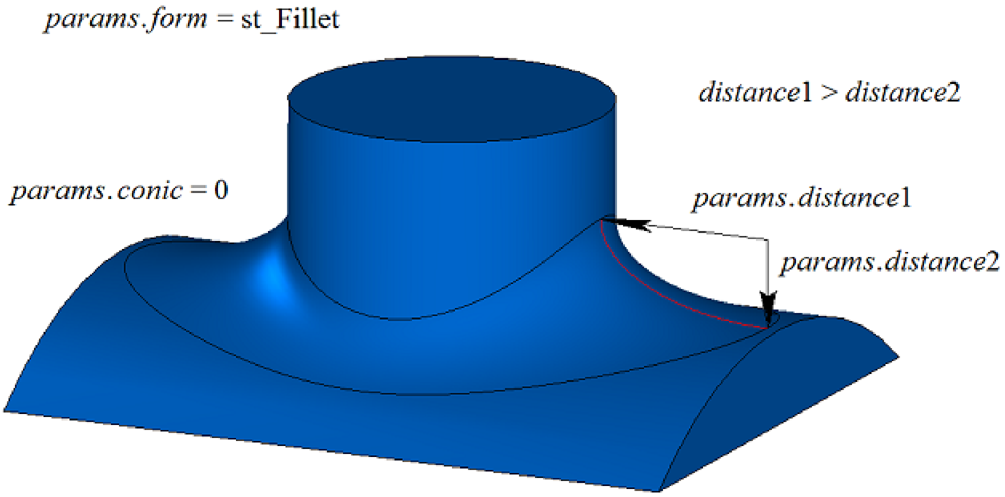


Fig. M.2.10.2.

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.

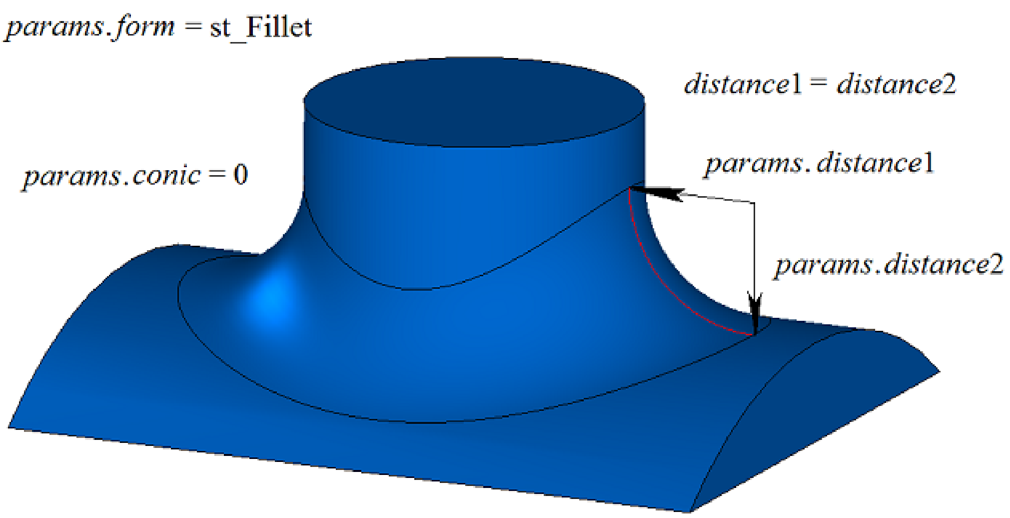


Fig. M.2.10.3.

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.

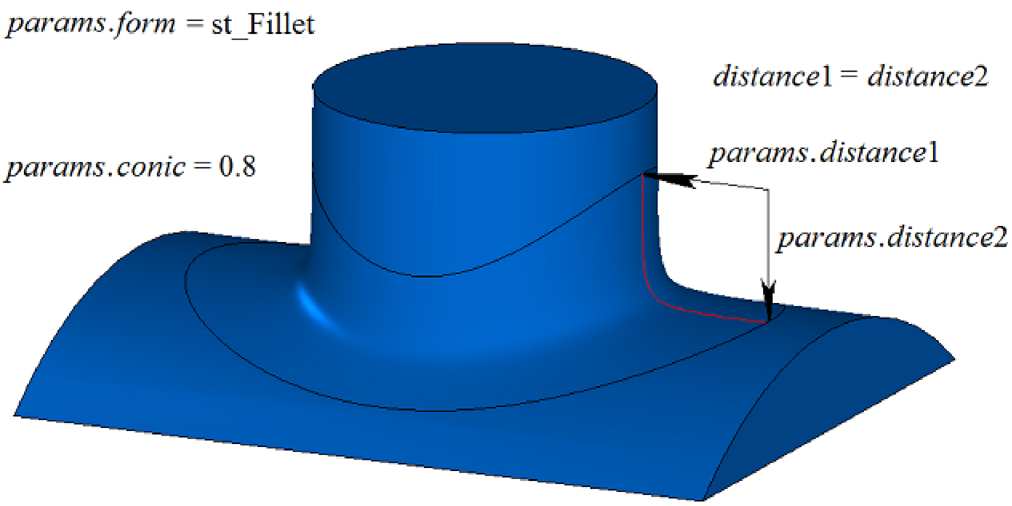


Fig. M.2.10.4.

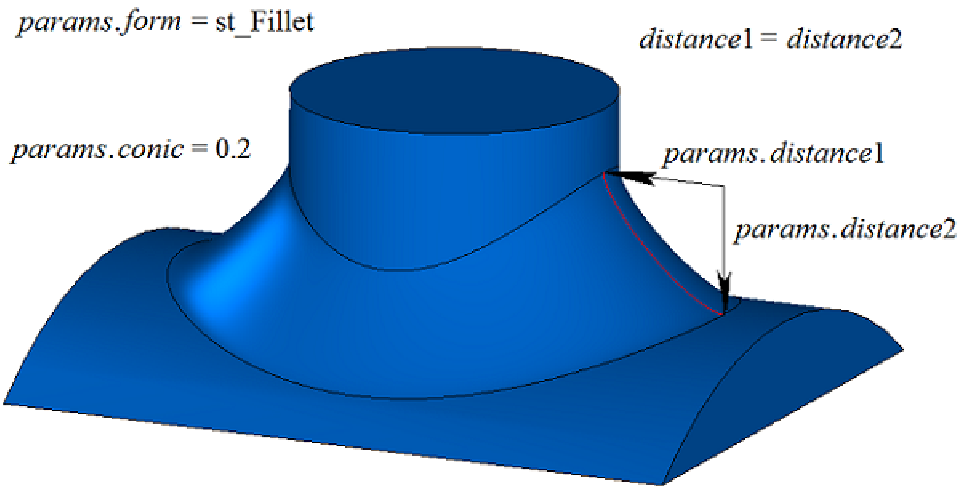


Fig. M.2.10.5.

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.

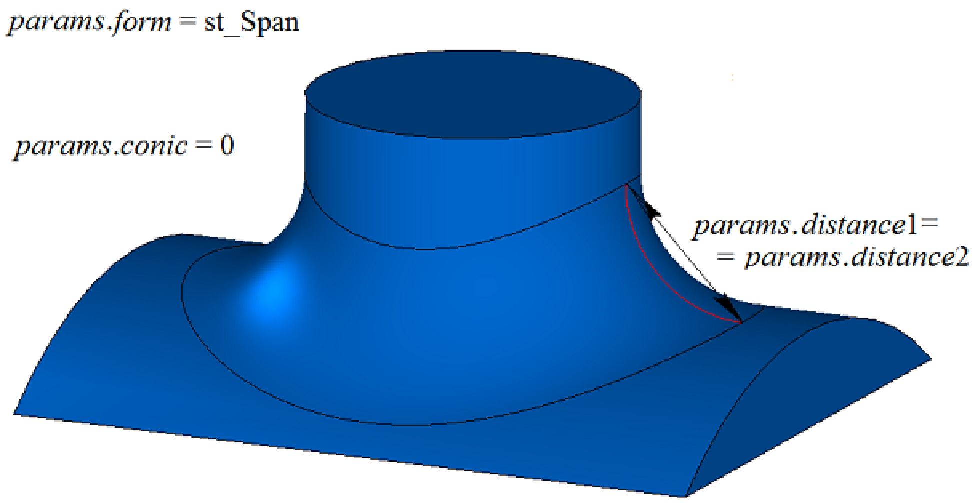


Fig. M.2.10.6.

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.

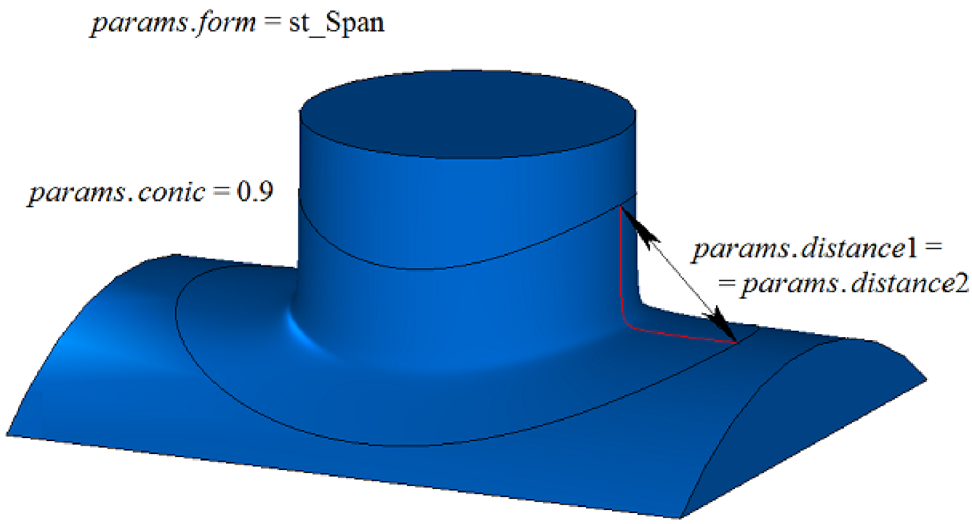


Fig. M.2.10.7.

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.

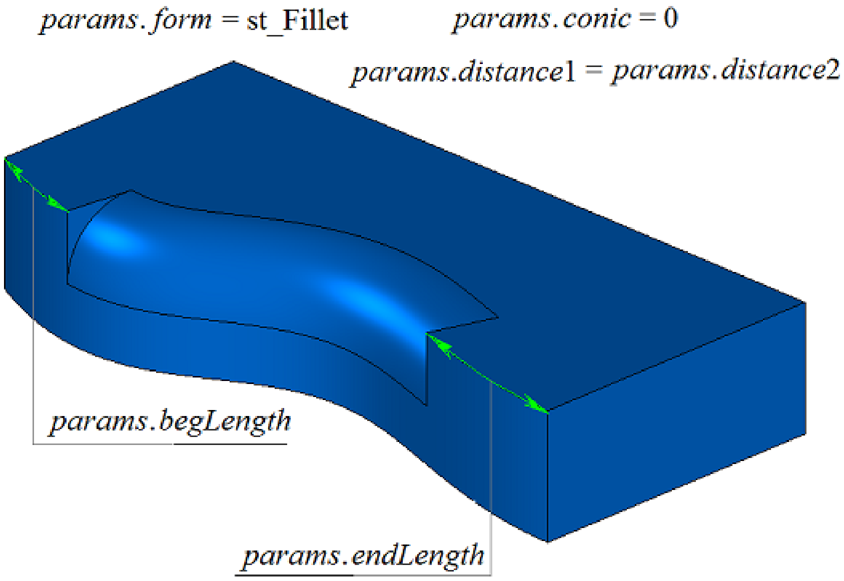


Fig. M.2.10.8.

Let's look at the body 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.

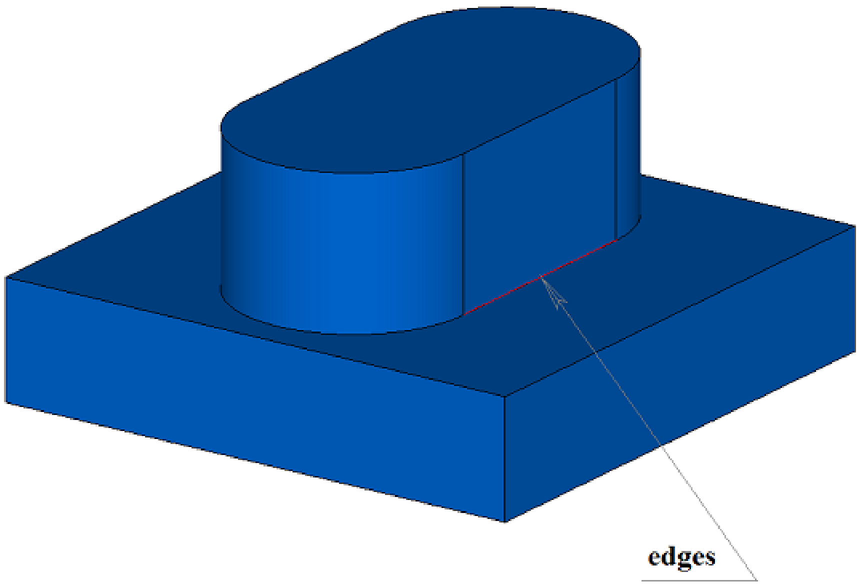


Fig. M.2.10.9.

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).

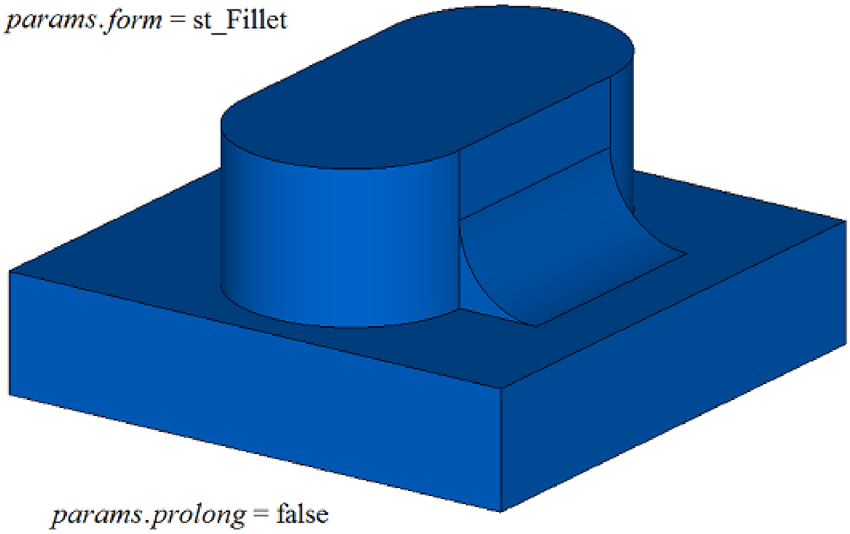


Fig. M.2.10.10.

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).

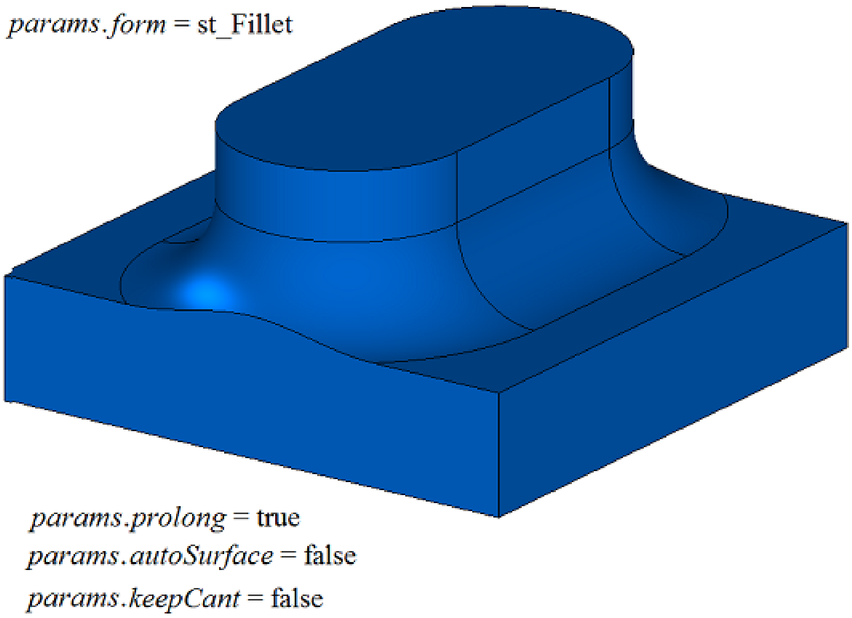


Fig. M.2.10.11.

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.

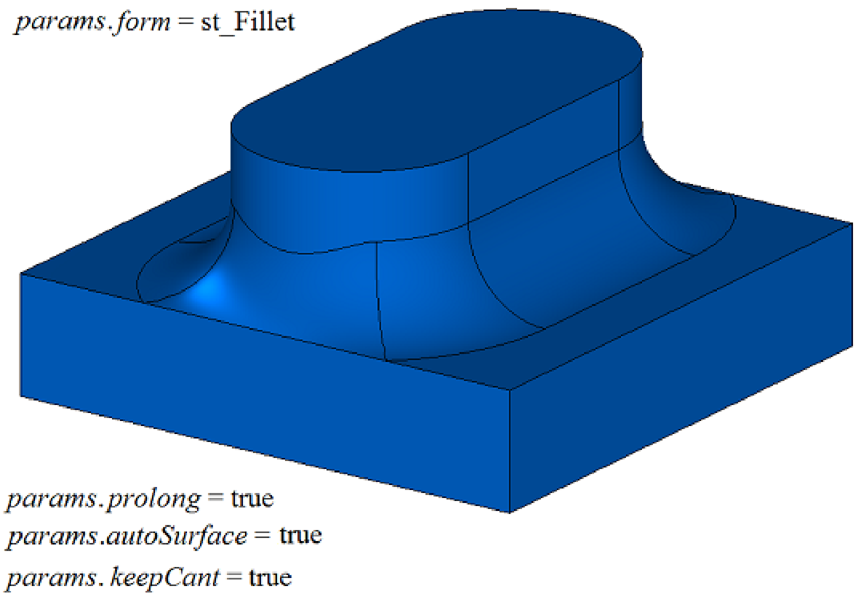


Fig. M.2.10.12.

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.

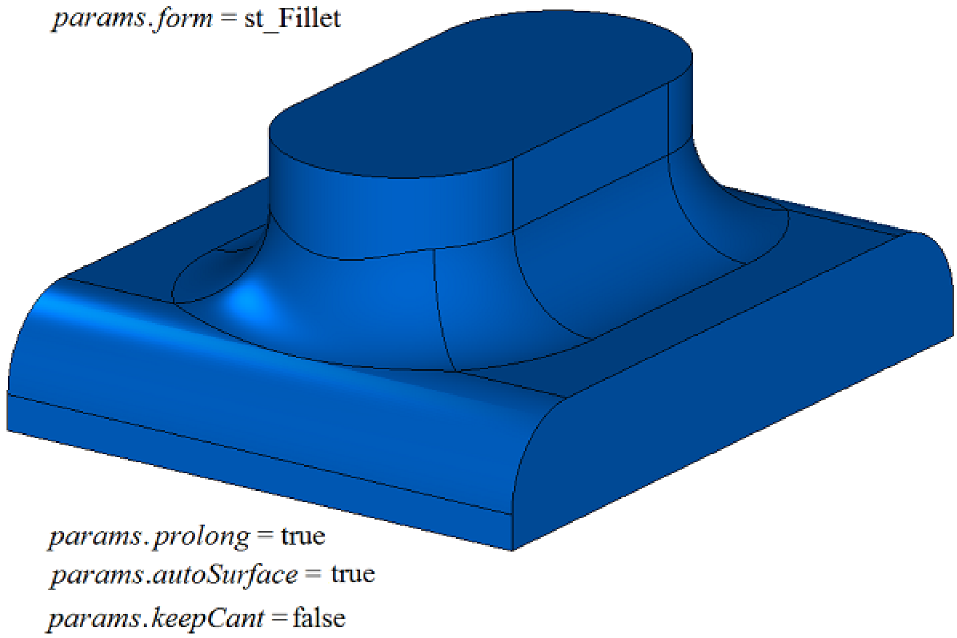


Fig. M.2.10.13.

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.

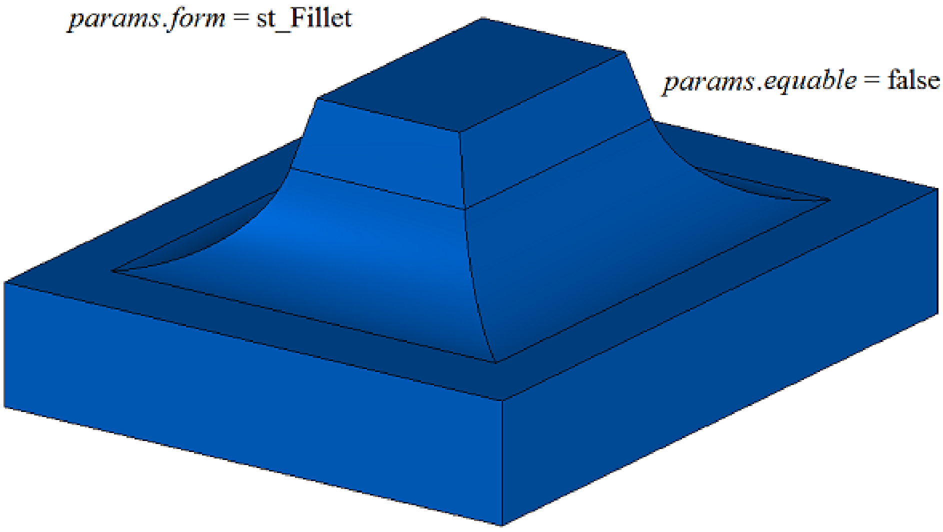


Fig. M.2.10.14.

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.

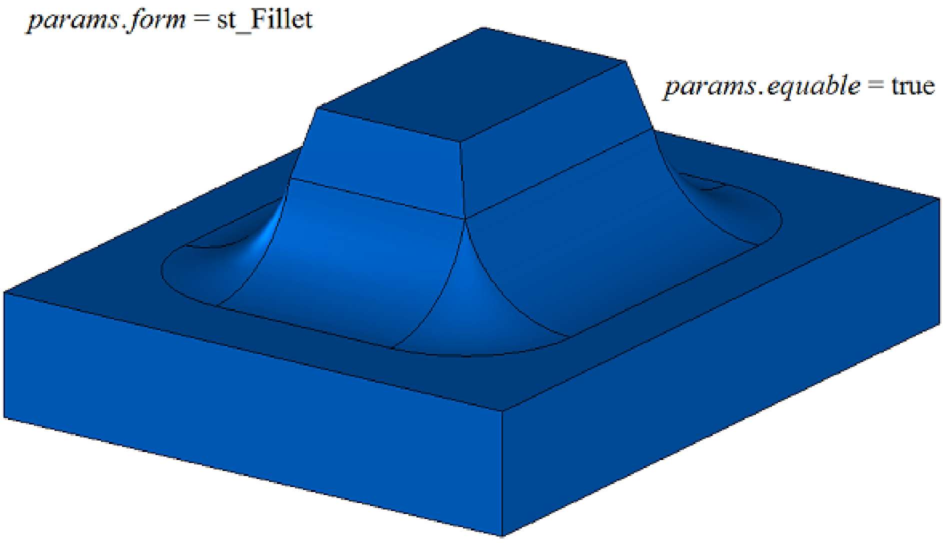


Fig. M.2.10.15.

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.

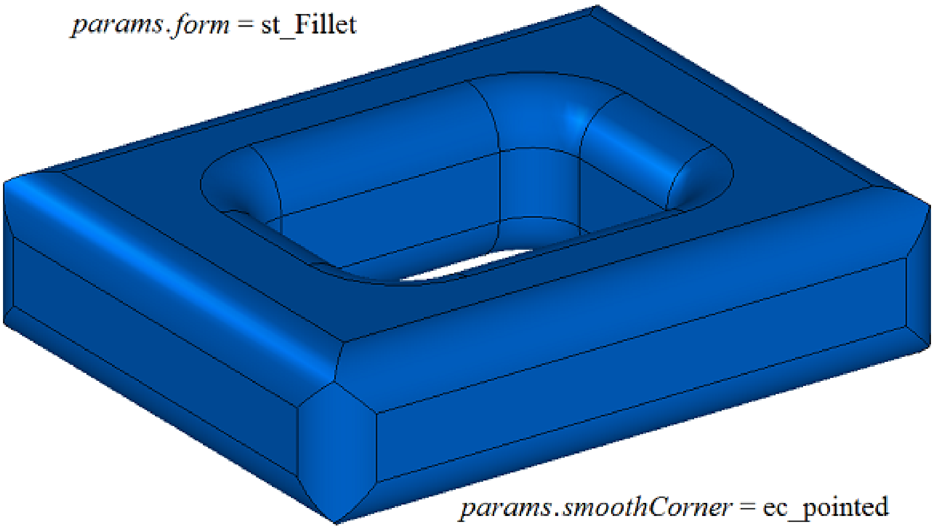


Fig. M.2.10.16.

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.

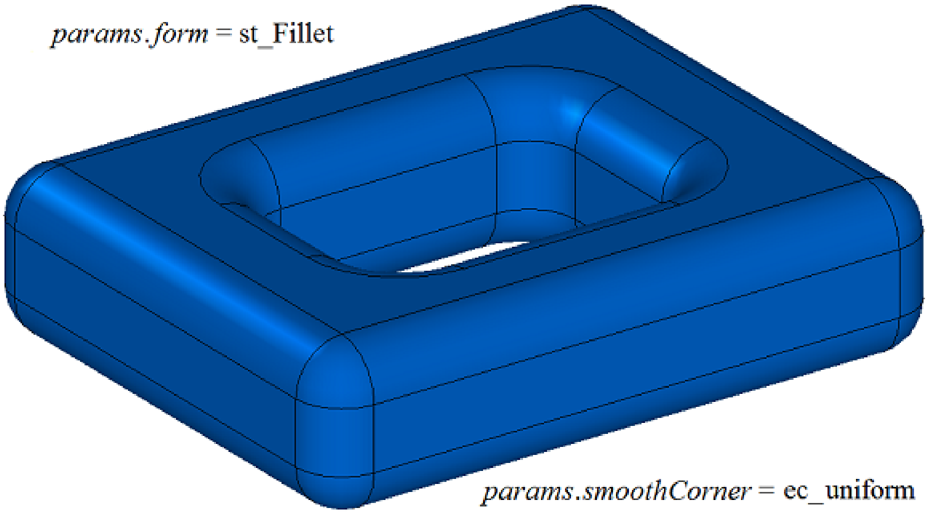


Fig. M.2.10.17.

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.

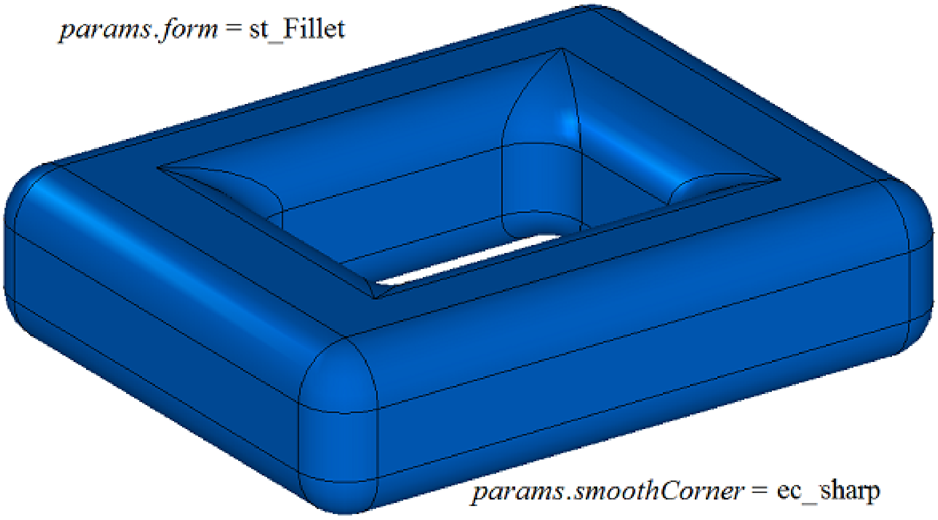


Fig. M.2.10.18

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** body 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.

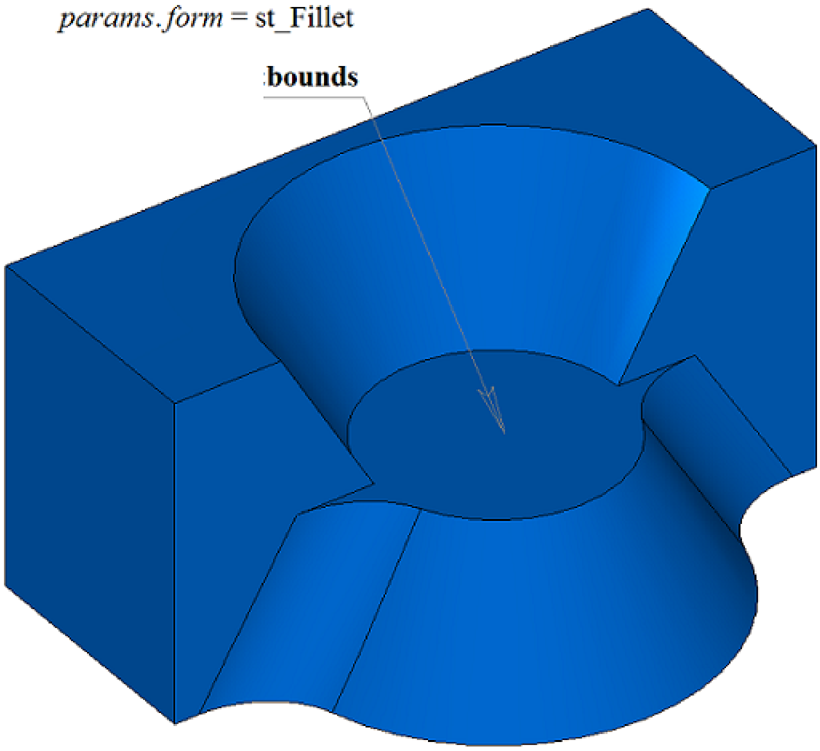


Fig. M.2.10.19

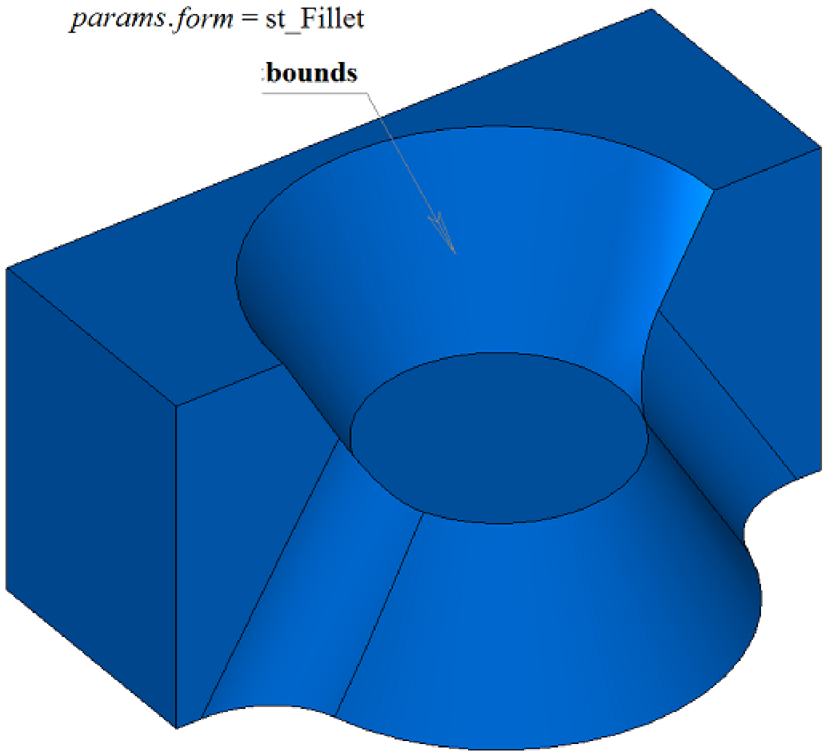


Fig. M.2.10.20

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 body.

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.

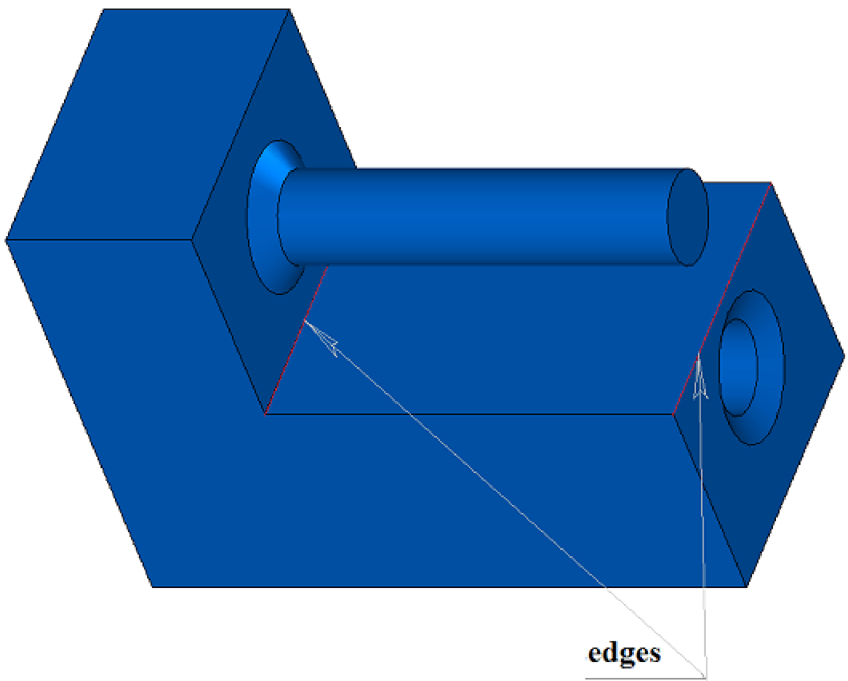


Fig. M.2.10.21

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

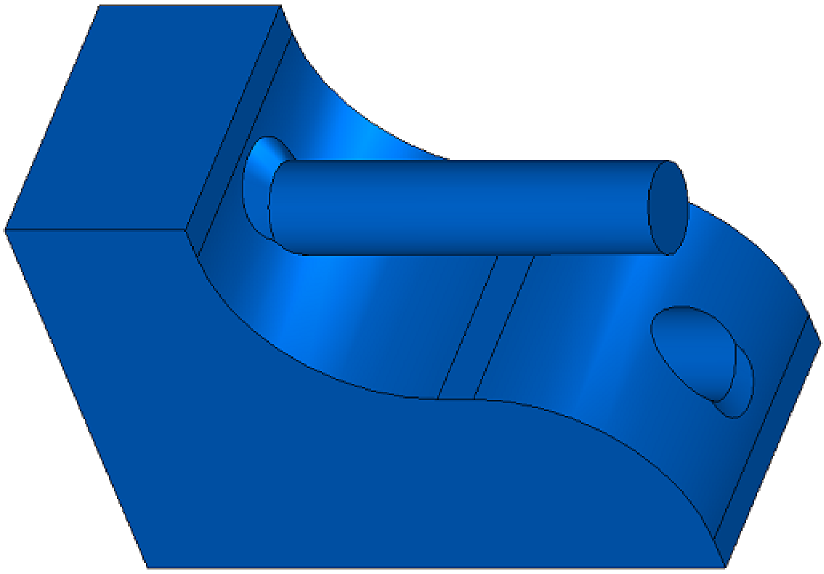


Fig. M.2.10.22

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

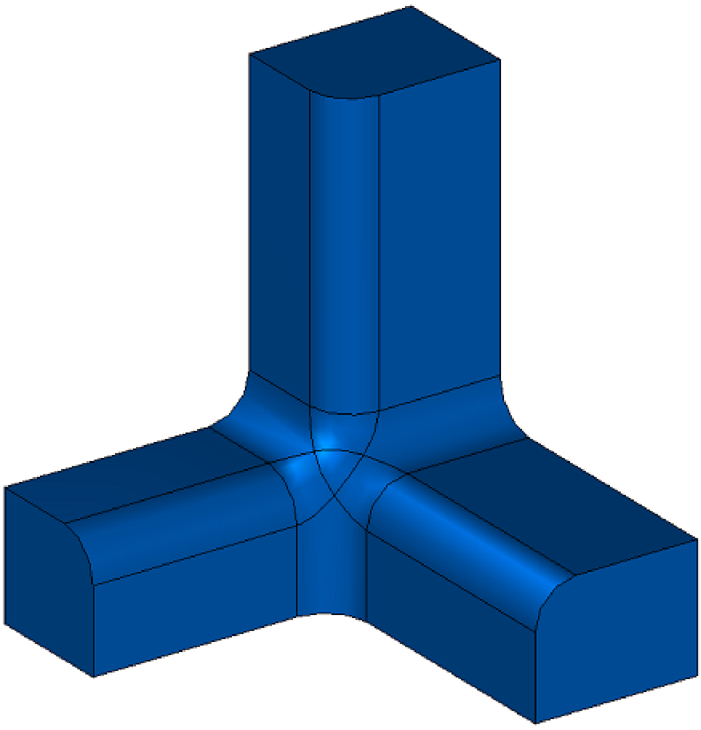


Fig. M.2.10.23

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 body for the body shown in Figure M.2.10.24 was constructed by subtracting four cylinders with axes coinciding with cube diagonals from a cube.

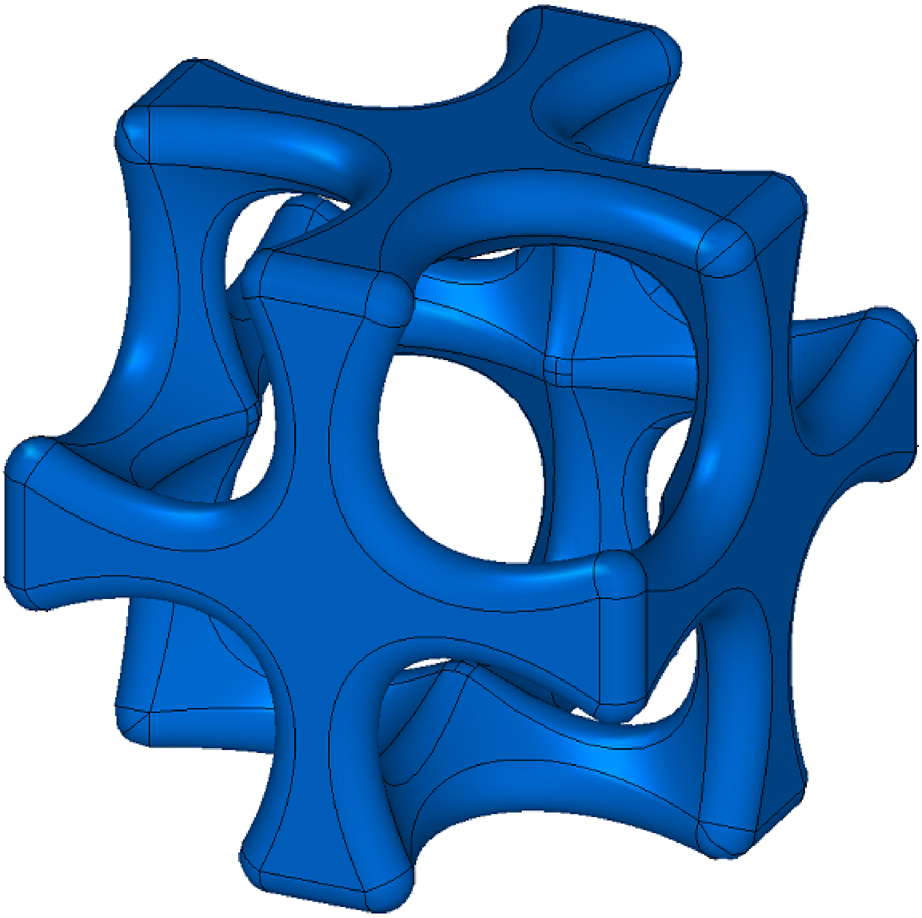


Fig. M.2.10.24

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

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

When in rounding methods, the parameters *distance*1 and *distance*2 are not equal and several edges are specified for processing, it is difficult to understand what corresponds to what. In this situation, it is suggested to use auxiliary methods: **SmoothPhantom**(...), **SmoothSequence**(...), **SmoothPositionData**(...), declared in the file action_phantom.h. The **SmoothPhantom**(...) method builds simplified surfaces to simulate future fillets. The **SmoothSequence**(...) method builds a series of edges to which, if desired, smoothly joining edges that can be processed together can be added. The **SmoothPositionData**(...) method calculates three points for the processed edges that are used for the phantom dimensions of the radii, legs, and corners of the operation. Usually, with the help of these methods, a phantom is drawn to understand what will happen as a result of the operation, and it is possible to change the parameters if necessary.

Before the construction of fillets processed edges are sorted, if necessary, or at the request added to them smoothly mating edges and of the edges are smooth sequence abutting edges. *distance*1 and *distance*2 parameters attached to the first and second surfaces, respectively, of the MbSurfaceIntersectionCurve curve of the first edge in the sequence of smoothly mating edges. Surfaces can be obtained intersection curve methods **GetCurveOneSurface**() and **GetCurveTwoSurface**(). According *distance*1 and *distance*2 first edge defined parameters for the other edges of each sequence. So, if the first edge pointer **edge**->**GetIntersectionCurve**().**GetSurfaceOne**() is a pointer &**edge**->**GetFacePlus**()->**GetSurface**().**GetSurface**(), then *distance*1 will correspond to the radius to the face **facePlus**, and *distance*2 will correspond to the radius to the face **faceMinus** of the first edge of sequence.

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

The 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 body using a variable radius.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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

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** body 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** body 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.

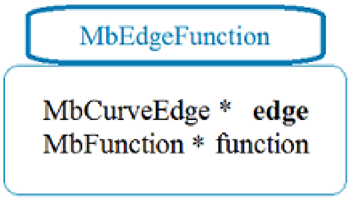


Fig. M.2.11.1.

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.

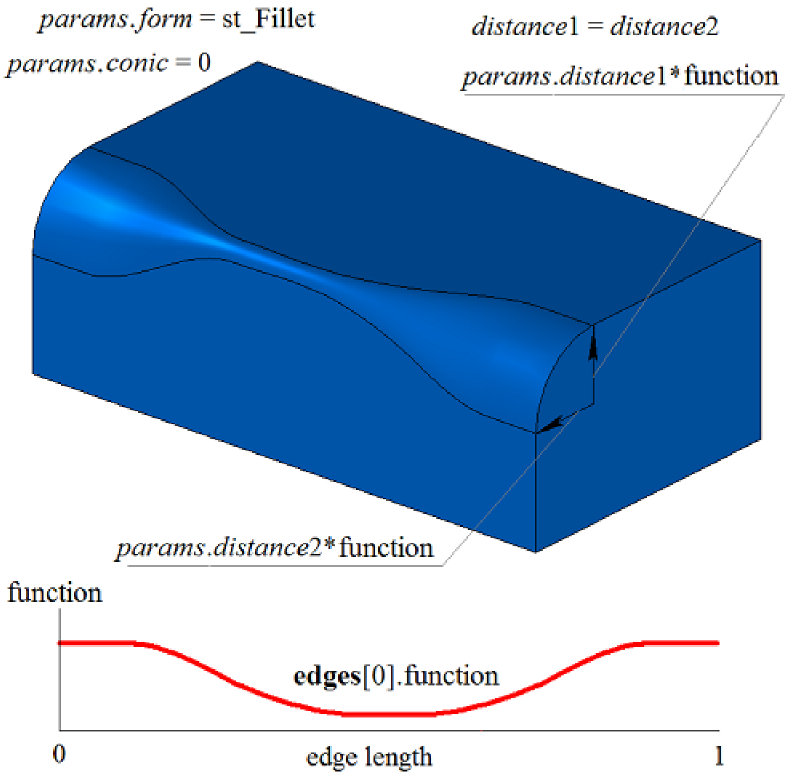


Fig. M.2.11.2.

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

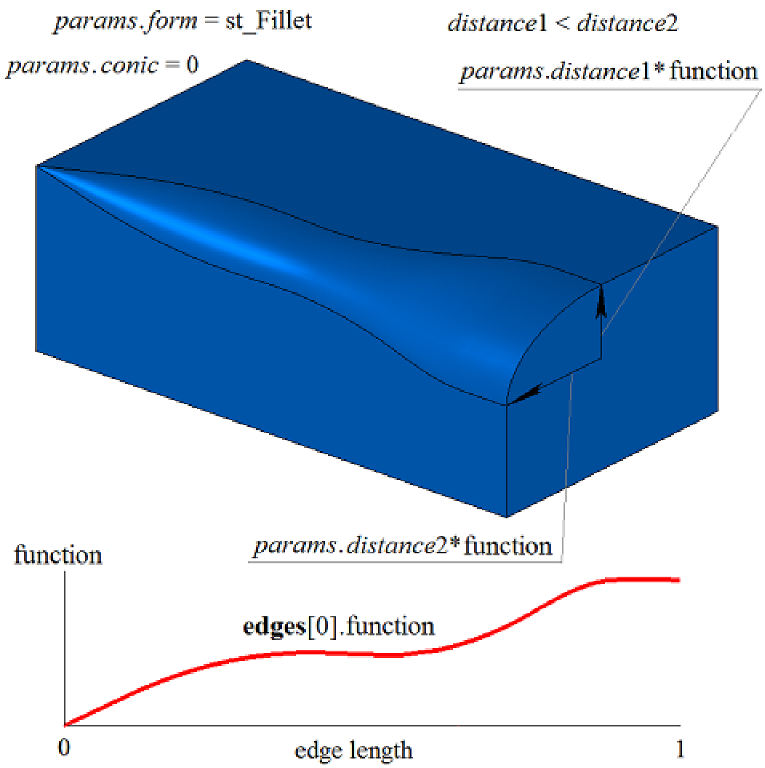


Fig. M.2.11.3.

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.

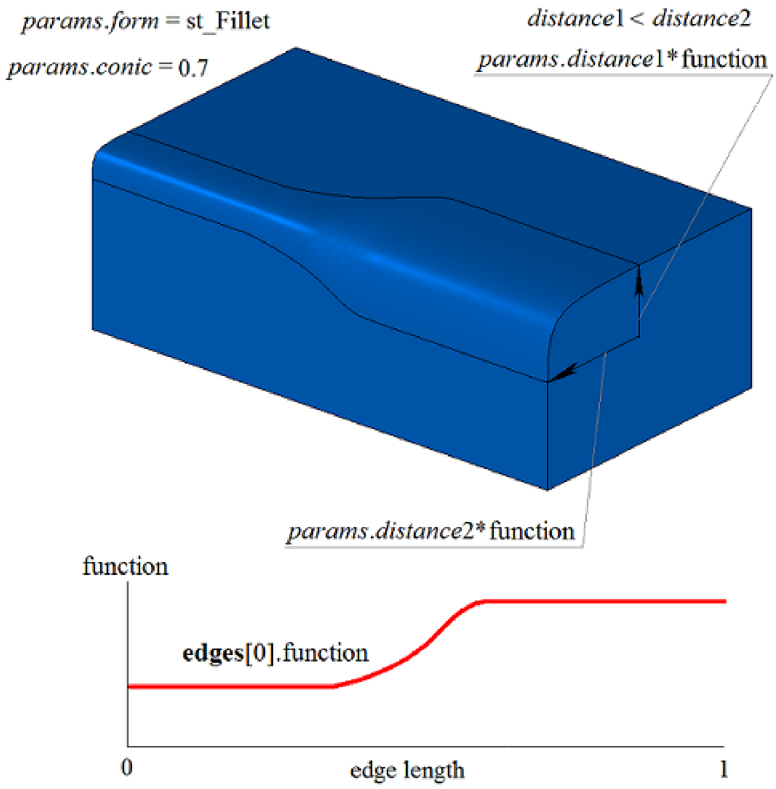


Fig. M.2.11.4.

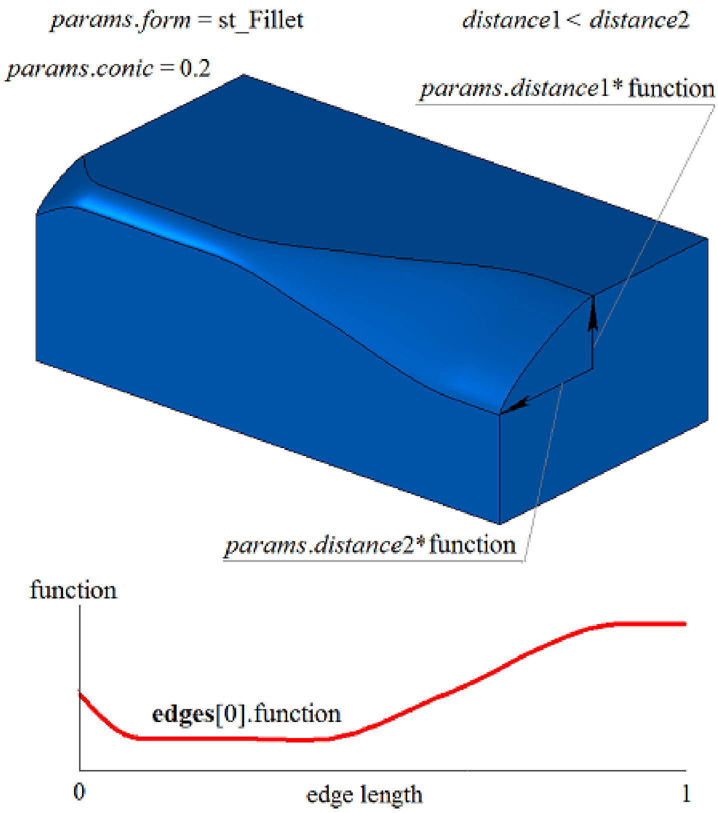


Fig. M.2.11.5.

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.

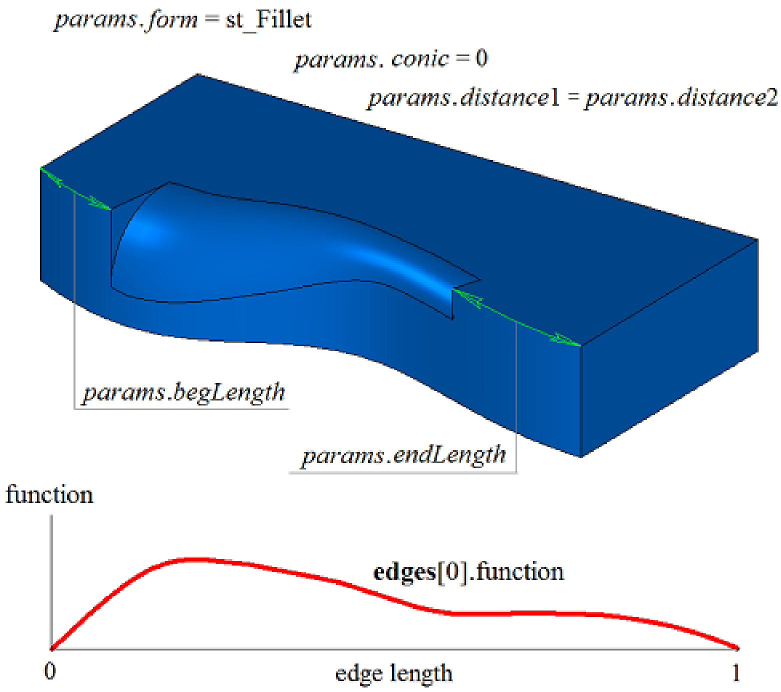


Fig. M.2.11.6.

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 body, edges that should be rounded-off, and radius change method for specified edges.

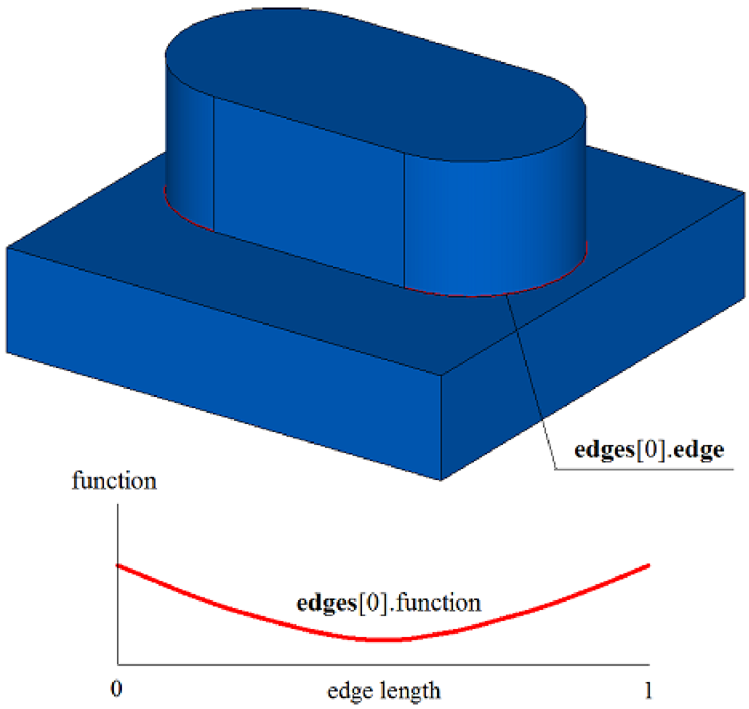


Fig. M.2.11.7.

In Fig. M.2.11.8, you can see the result of operation for the method for the original body 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.

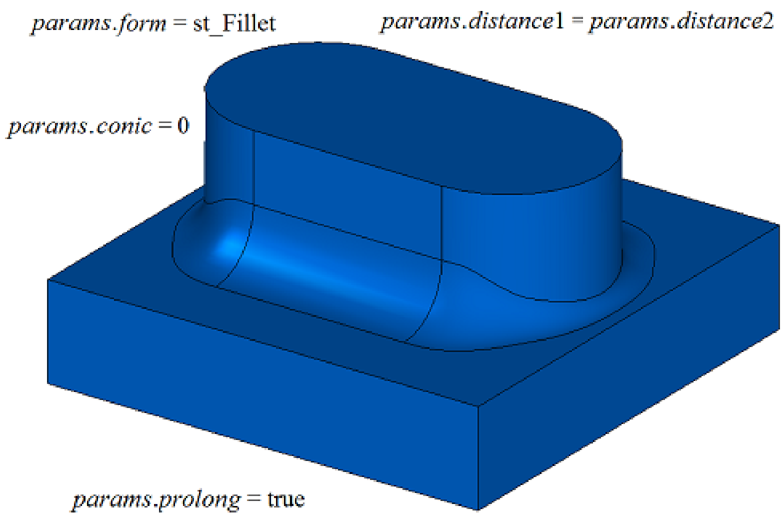


Fig. M.2.11.8.

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** body 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 body.

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

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

M.2.12. Constructing a Body with Edge Chamfers

The 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 body.

Input parameters of the method are as follows:

- **solid** is the original body,
- sameShell is original body 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 body.

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

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

The method is declared in action_solid.h file.

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

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

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** body. 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.

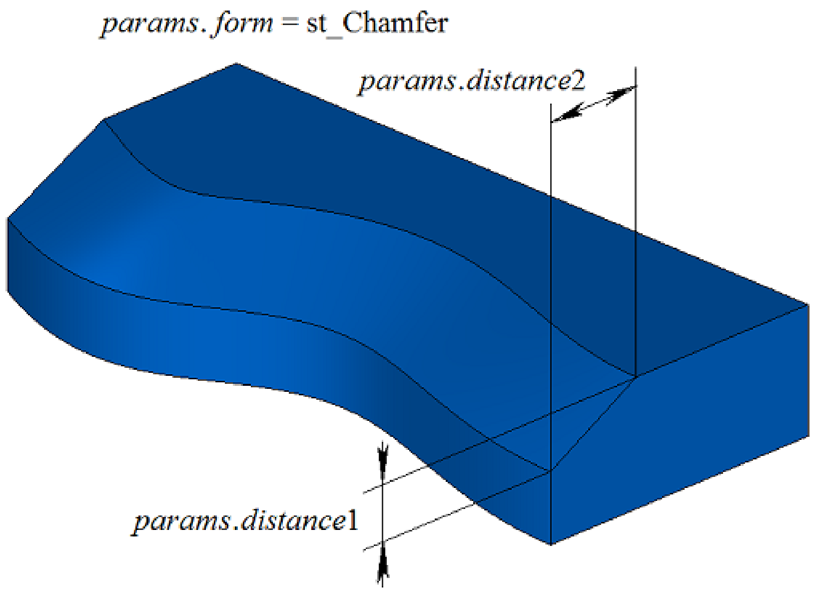


Fig. M.2.12.1.

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.

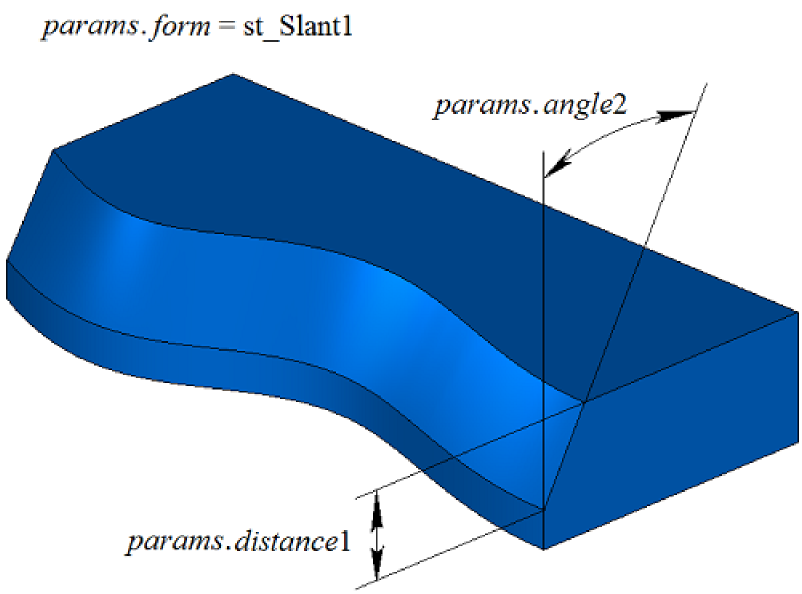


Fig. M.2.12.2.

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.

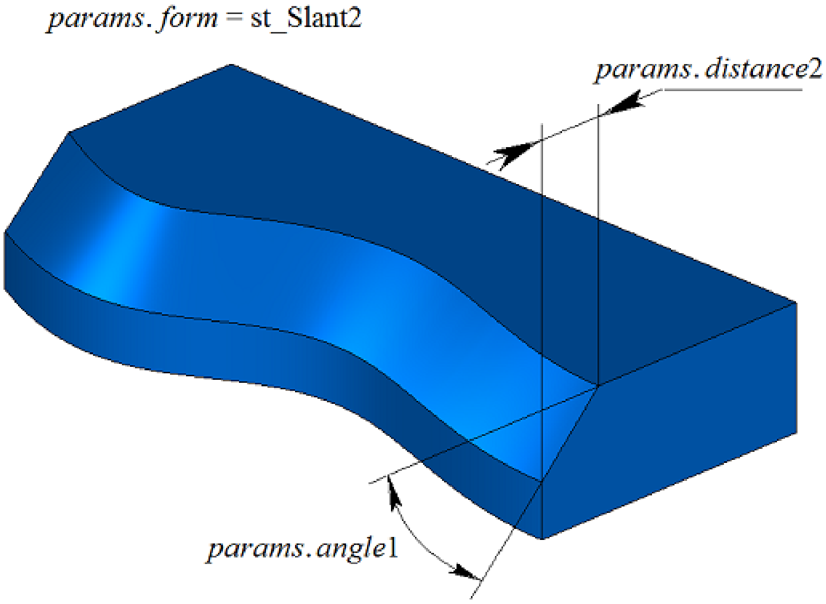


Fig. M.2.12.3.

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.

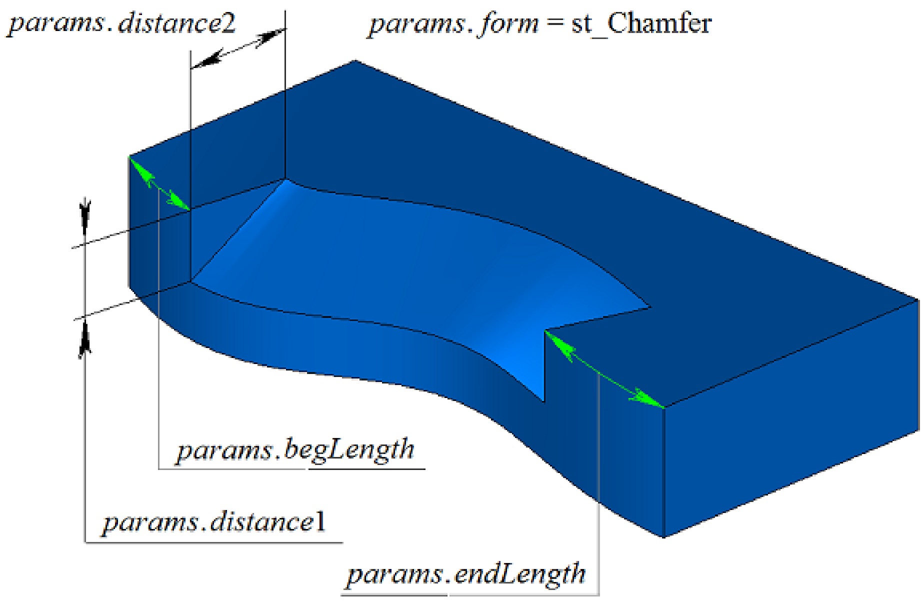


Fig. M.2.12.4.

Let's take the body 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.

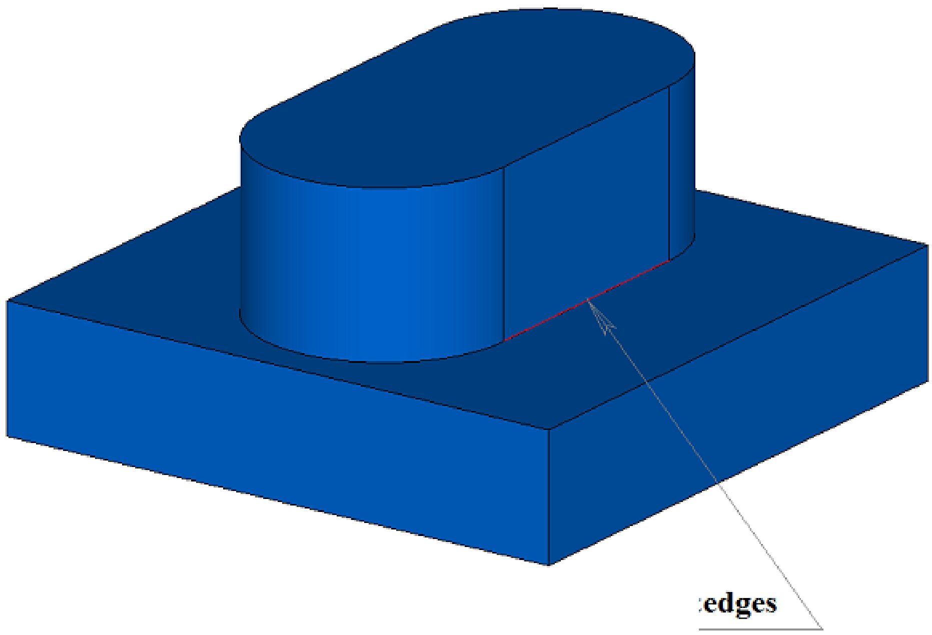


Fig. M.2.12.5.

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.

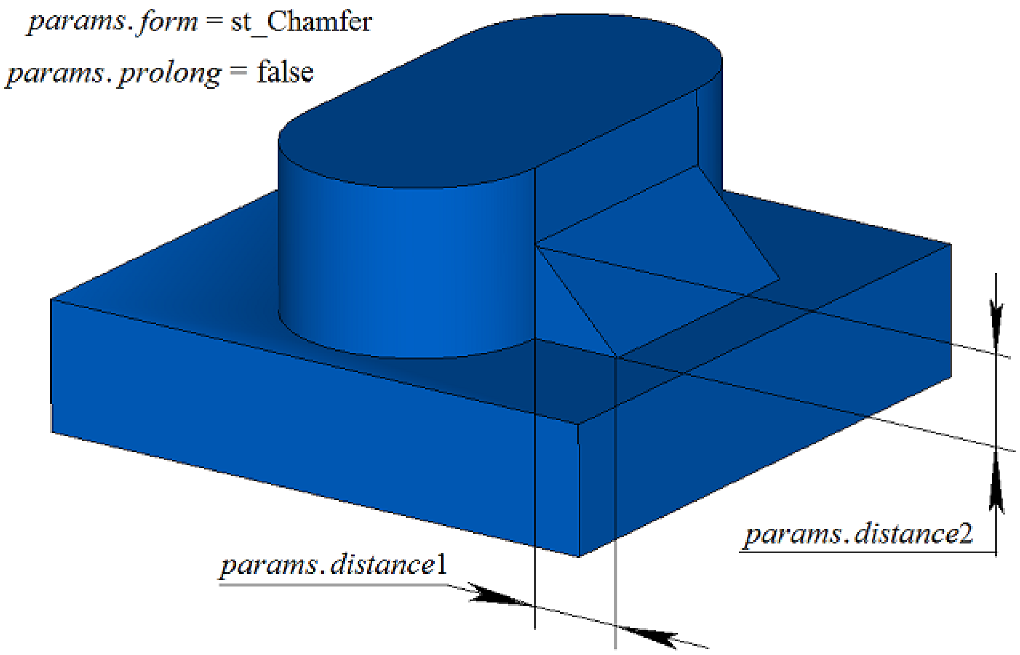


Fig. M.2.12.6.

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.

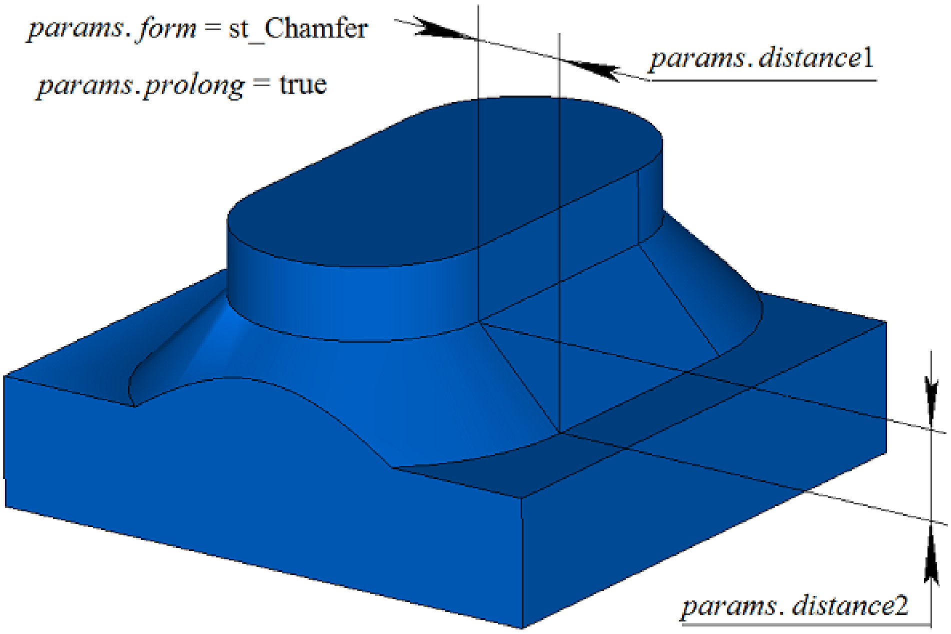


Fig. M.2.12.7.

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 body shown in Figure M.2.12.8 and show how to use *smoothCorner* parameter to construct chamfers at all edges of the body.

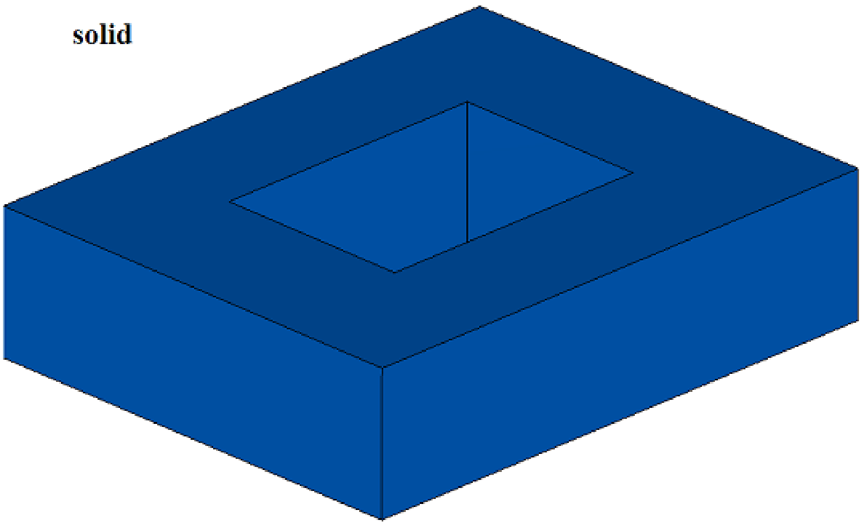


Fig. M.2.12.8.

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

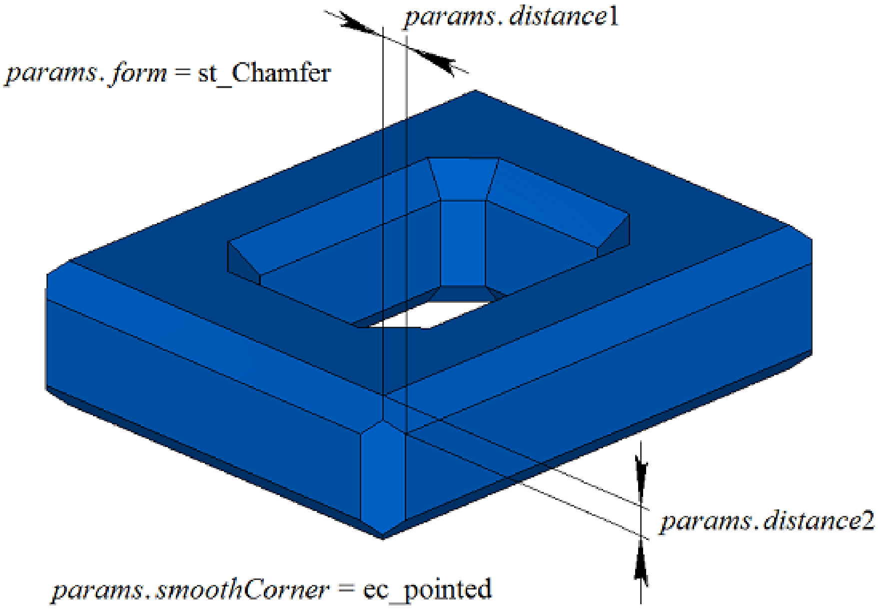


Fig. M.2.12.9.

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.

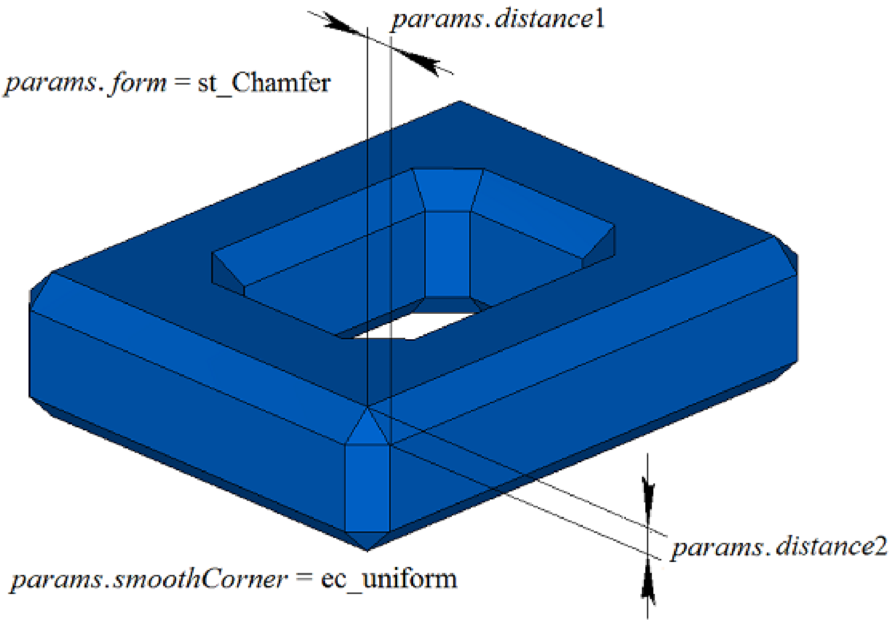
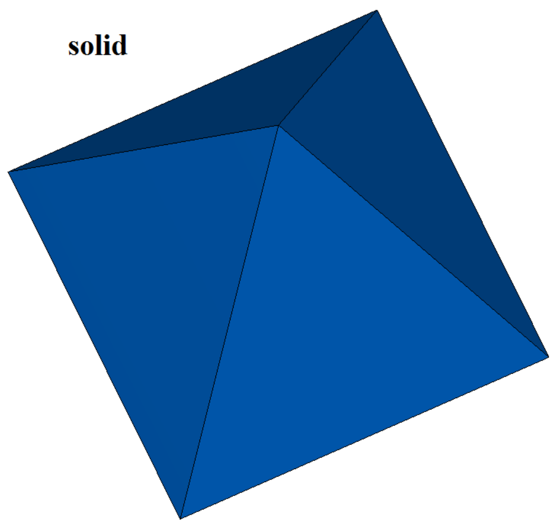


Fig. M.2.12.10.

Let's take a pyramidal body 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 body construction for symmetrical configuration of edges and symmetrical chamfer is shown in Figure M.2.12.12.



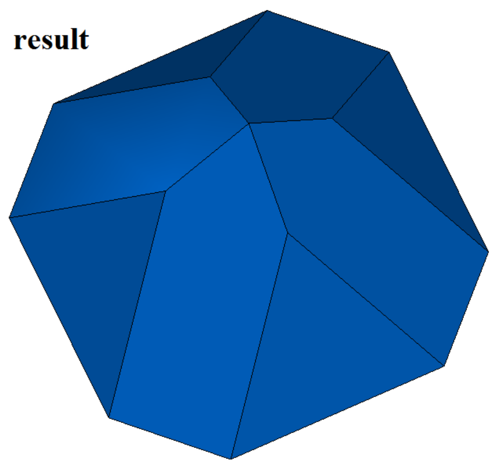


Fig. M.2.12.11. *Fig. M.2.12.12.*

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.

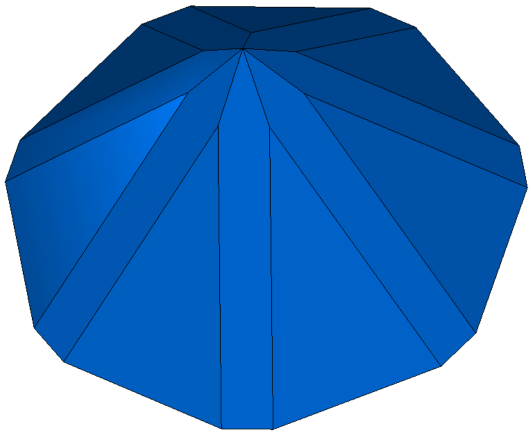


Fig. M.2.12.13.

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

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

When in methods of constructing a chamfer, the parameters *distance*1 and *distance*2 are not equal and several edges are specified for processing, it is difficult to understand what corresponds to what. In this situation, it is suggested to use auxiliary methods: **SmoothPhantom**(...), **SmoothSequence**(...), **SmoothPositionData**(...), declared in the file action_phantom.h. The **SmoothPhantom**(...) method builds simplified surfaces to simulate future fillets. The **SmoothSequence**(...) method builds a series of edges to which, if desired, smoothly joining edges that can be processed together can be added. The **SmoothPositionData**(...) method calculates three points for the processed edges that are used for the phantom dimensions of the radii, legs, and corners of the operation. Usually, with the help of these methods, a phantom is drawn to understand what will happen as a result of the operation, and it is possible to change the parameters if necessary.

Before the construction chamfers processed edges are sorted, if necessary, or at the request added to them smoothly mating edges and of the edges are smooth sequence abutting edges. *distance*1 and *distance*2 parameters attached to the first and second surfaces, respectively, of the MbSurfaceIntersectionCurve curve of the first edge in the sequence of smoothly mating edges. Surfaces can be obtained intersection curve methods **GetCurveOneSurface**() and **GetCurveTwoSurface**(). According *distance*1 and *distance*2 first edge defined parameters for the other edges of each sequence. So, if the first edge pointer **edge**->**GetIntersectionCurve**().**GetSurfaceOne**() is a pointer &**edge**->**GetFacePlus**()->**GetSurface**().**GetSurface**(), then the *distance*1 will correspond to the chamfer to the face **faceMinus** , and *distance*2 will correspond to the chamfer to the face **facePlus** of the first edge of sequence

M.2.13. Constructing a Thin-Wall Body

The method
MbResultType
ThinSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
RPArray<MbFace> & **outFaces**,
SweptValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
constructs a thin-wall body by excluding specified faces from the original body.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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 body, it also "sets a predetermined thickness" for remaining faces. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original body to **result** constructed body. *params* parameter contains data on wall thickness in remaining faces, as well as data on closure of constructed **result** body. 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 body will be constructed. names parameter is used to name the faces of the constructed body. To execute the operation, **outFaces** to be deleted should not have smooth edges attached to remaining edges of the original body 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 body and **outFaces** faces that are deleted.

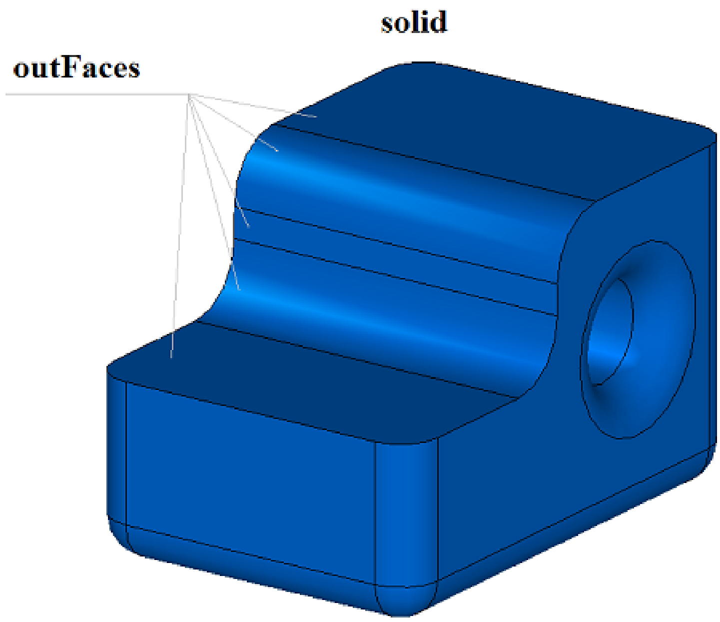


Fig. M.2.13.1.

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

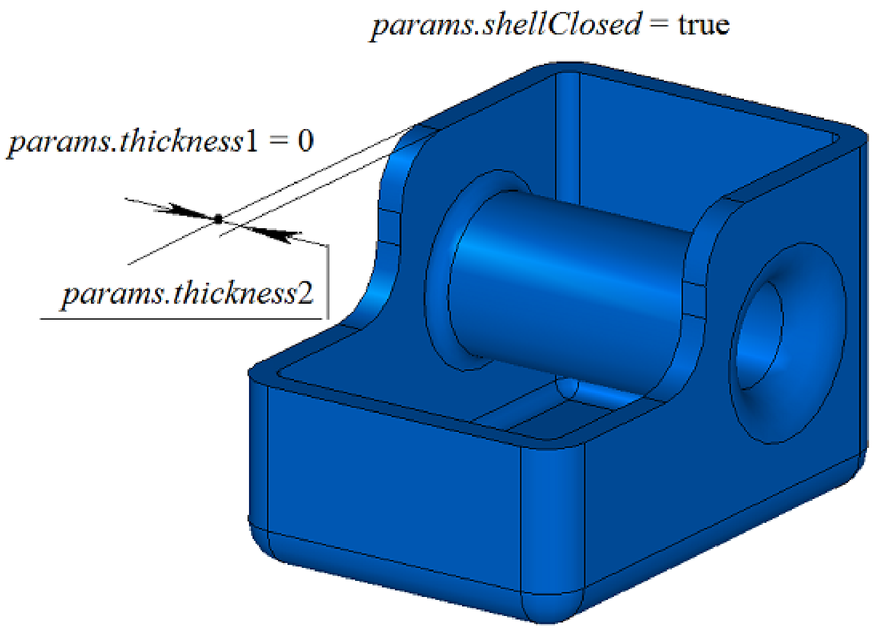


Fig. M.2.13.2.

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

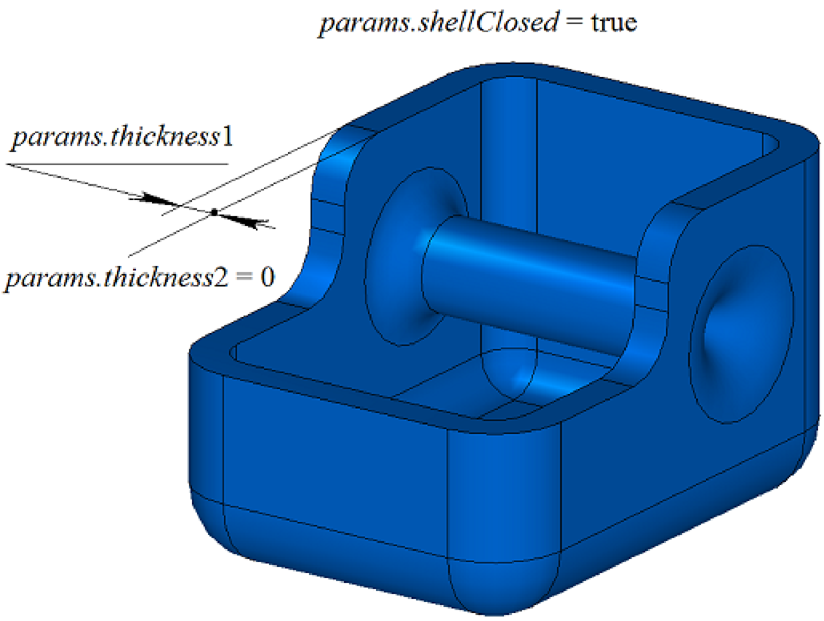


Fig. M.2.13.3.

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

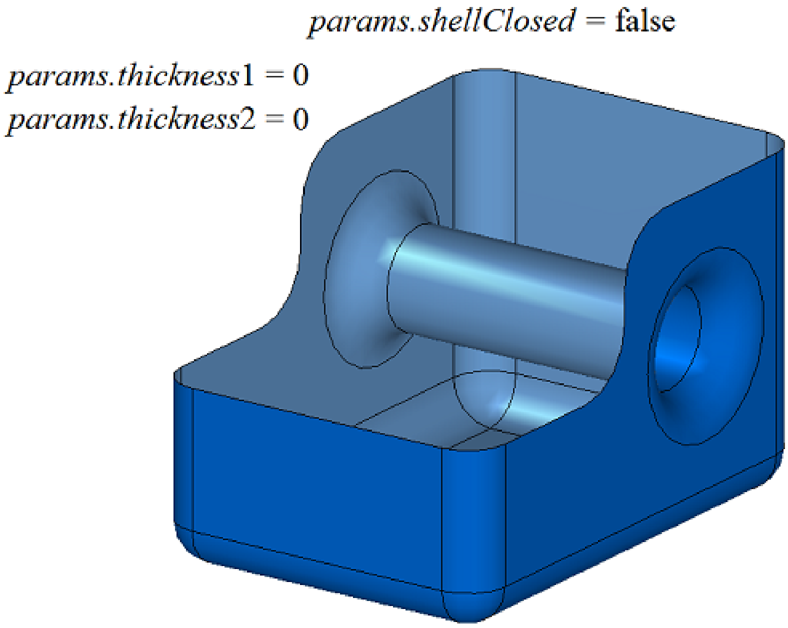


Fig. M.2.13.4.

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

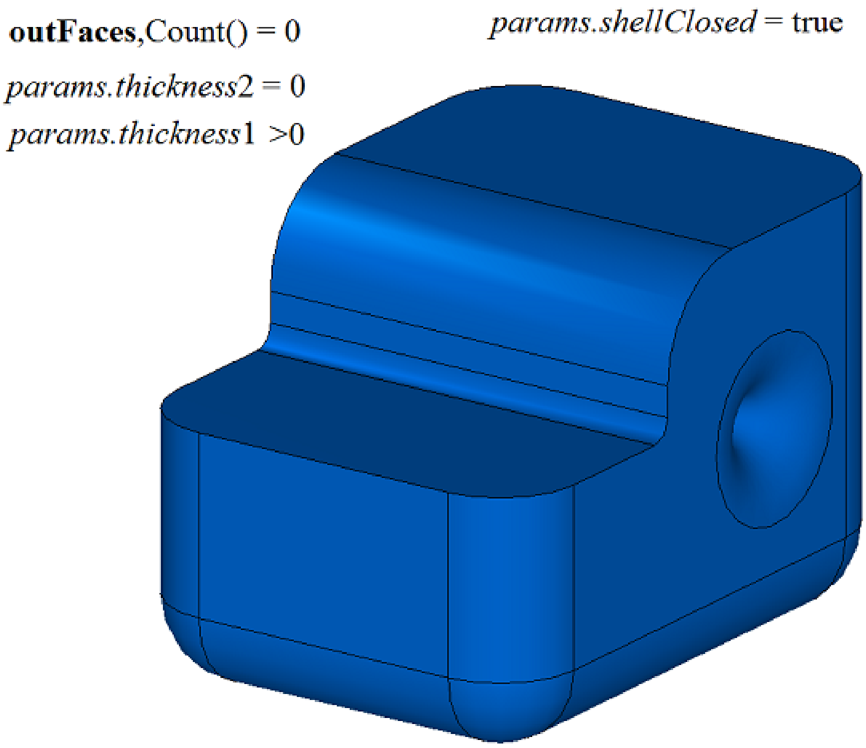


Fig. M.2.13.5.

ThinSolid method adds MbShellSolid constructor to the log of the newly constructed body 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 body using New ->Body -> By Processing Faces -> Uniform Thickening menu command.

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

The 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 body by excluding specified faces and setting various thickness of remaining faces in the original body.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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 body, 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 body to **result** constructed body. **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** body, 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 body. To execute the operation, **outFaces** to be deleted should not have smooth edges attached to remaining edges of the original body 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 body, **outFaces** faces that should be deleted and **offFaces** for which individual *offDistances* thickness values were set.

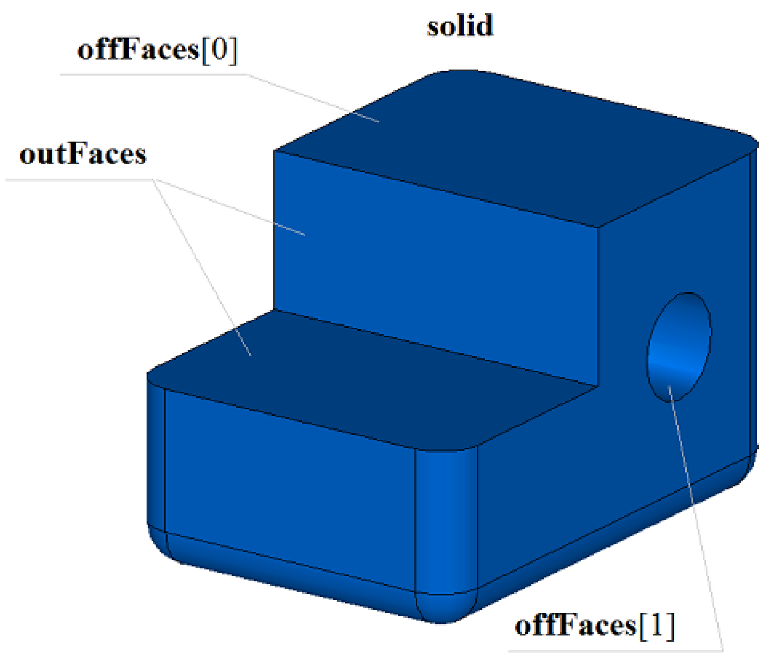


Fig. M.2.14.1.

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

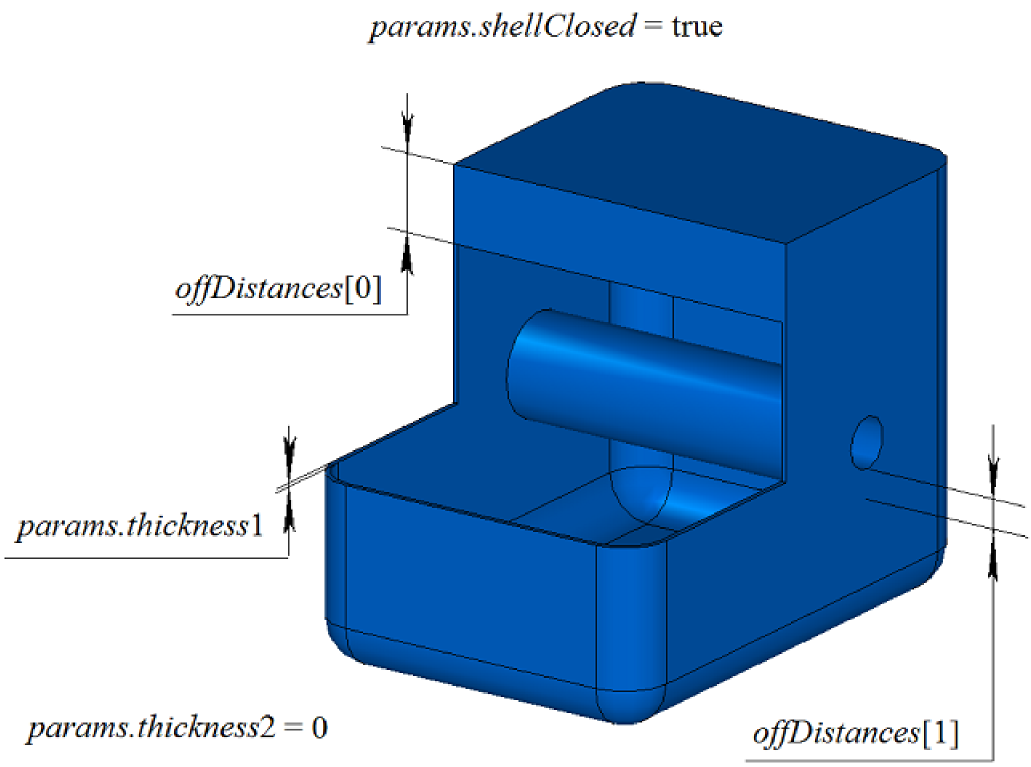


Fig. M.2.14.2.

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

ThinSolid method adds MbShellSolid constructor to the log of the newly constructed body 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 body using New ->Body -> By Processing Faces -> With Uneven Thickness menu command.

M.2.15. Constructing Bodies by Thickening the Surface

The method
MbResultType
ThinSolid (const MbSurface & **surface**,
bool *faceSense*,
SweptValues & *params*,
const MbSNameMaker & names,
SimpleName name,
MbSolid *& **result**)
constructs a body 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 body,
- *params* are construction parameters,
- names is face namer,
- name is operation name.

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

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 body 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** body. 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 body.

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

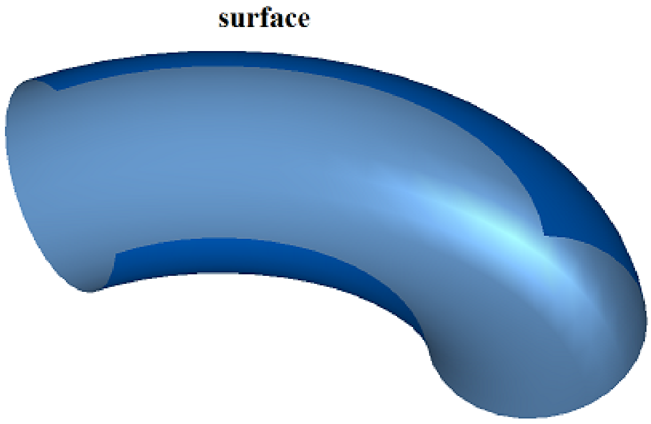


Fig. M.2.15.1.

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

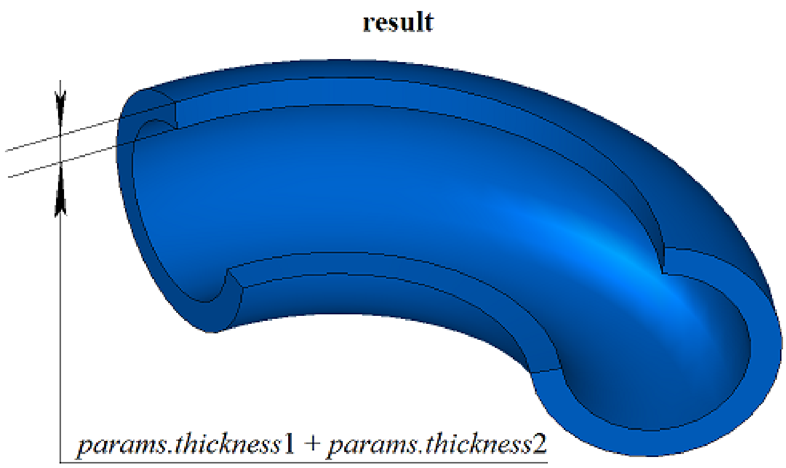


Fig. M.2.15.2.

ThinSolid method adds MbShellSolid constructor to the log of the newly constructed body 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 body using New ->Body -> Based on Surface -> Thickening menu command.

M.2.16. Constructing a Mirror Body

The method

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

Input parameters of the method are as follows:

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

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

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 body in relation to XY plane of specified **place** local coordinate system. names parameter is used to name the faces of the constructed body.

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

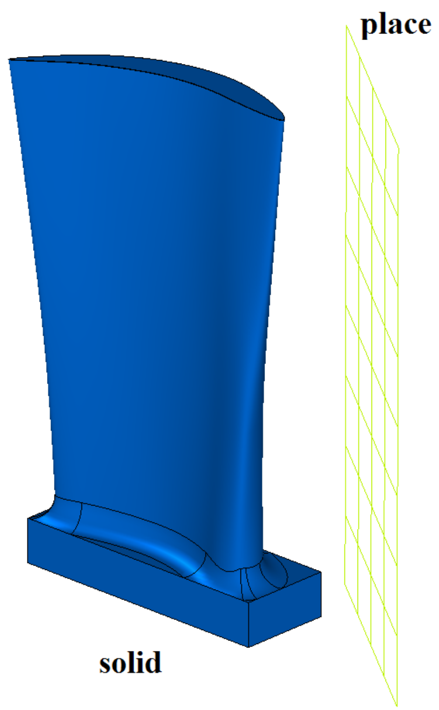


Fig. M.2.16.1.

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

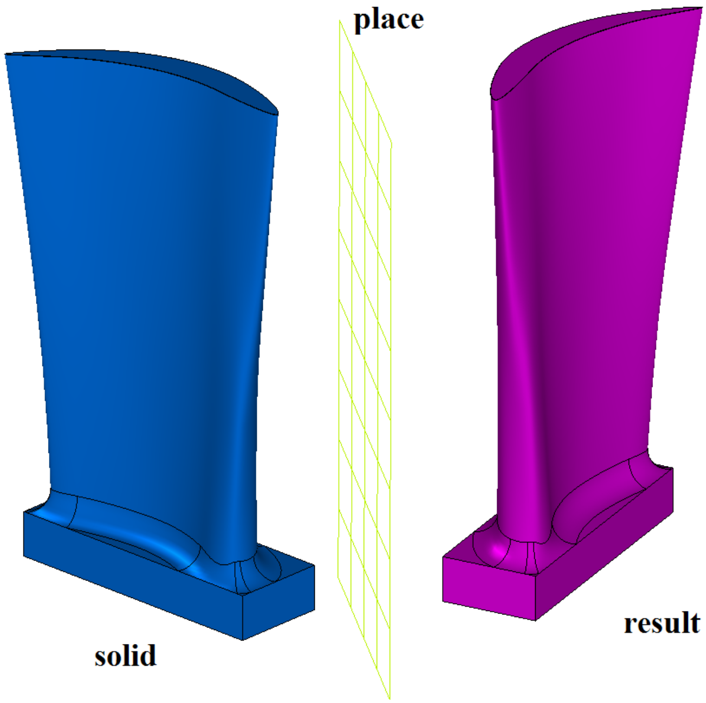


Fig. M.2.16.2.

MirrorSolid method adds MbSymmetrySolid constructor in a log of the newly constructed body 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 body using New ->Body -> Based on Body -> Symmetrical menu command.

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

The 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 bodies and executes a determined Boolean operation on the original body if it was specified.

Input parameters of the method are as follows:

- **solid** is the original body (it may be equal to zero),
- *sameShell* is a version of original body copying method,
- **solids** is the set of bodies,
- *sameShells* is copying method for the bodies in the set,
- *oType* is Boolean operation type: bo_Union means merging of the bodies,

bo_Intersect means intersection of the bodies,
bo_Difference means subtraction of the bodies,

- *checkIntersect* is a flag used to check intersection for a set of bodies (false means "no check"),
- *mergeFaces* indicates whether similar faces should be merged,
- names is face namer,
- *isArray* is regularity flag for the set of bodies.

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

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** body and **solids** set of bodies; if *oType*=bo_Intersect, then the method intersects **solid** body and **solids** set of bodies; if *oType*=bo_Difference, then the method subtracts **solids** set of bodies from **solid** body.

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

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

mergeFaces parameter permits to merge similar faces in **result** body 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 bodies. If *isArray*==true, then the bodies of the set are located in nodes of rectangular or circular grid, and positions of the bodies are specified in face names.

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

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

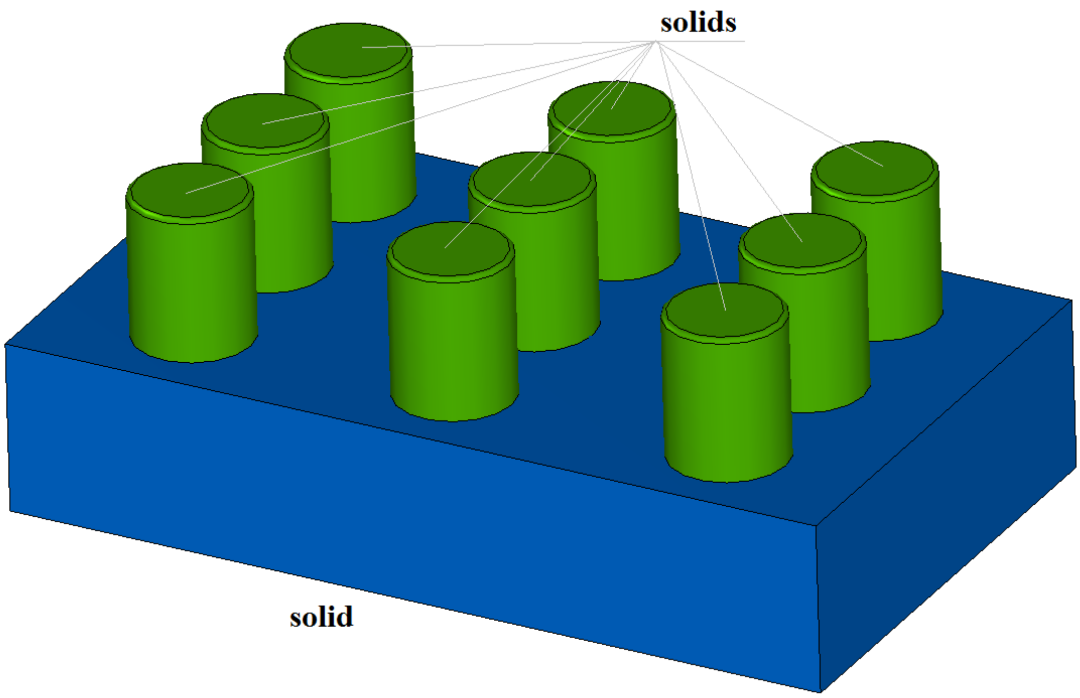


Fig. M.2.17.1.

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

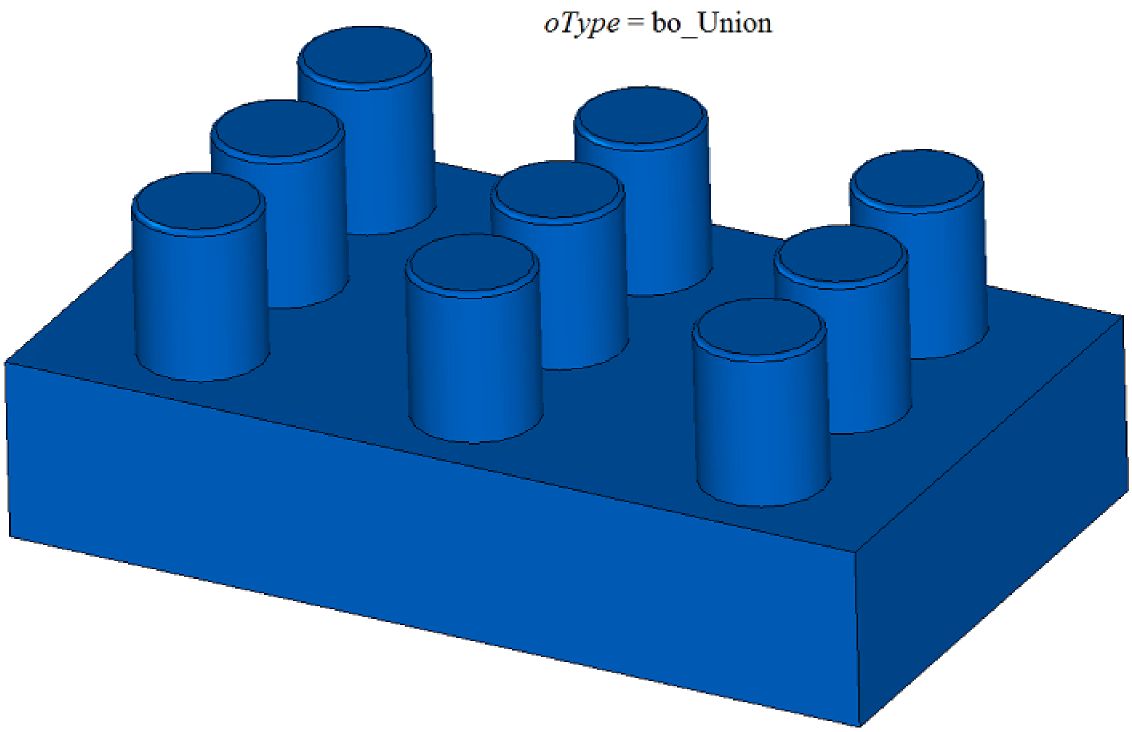


Fig. M.2.17.2.

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

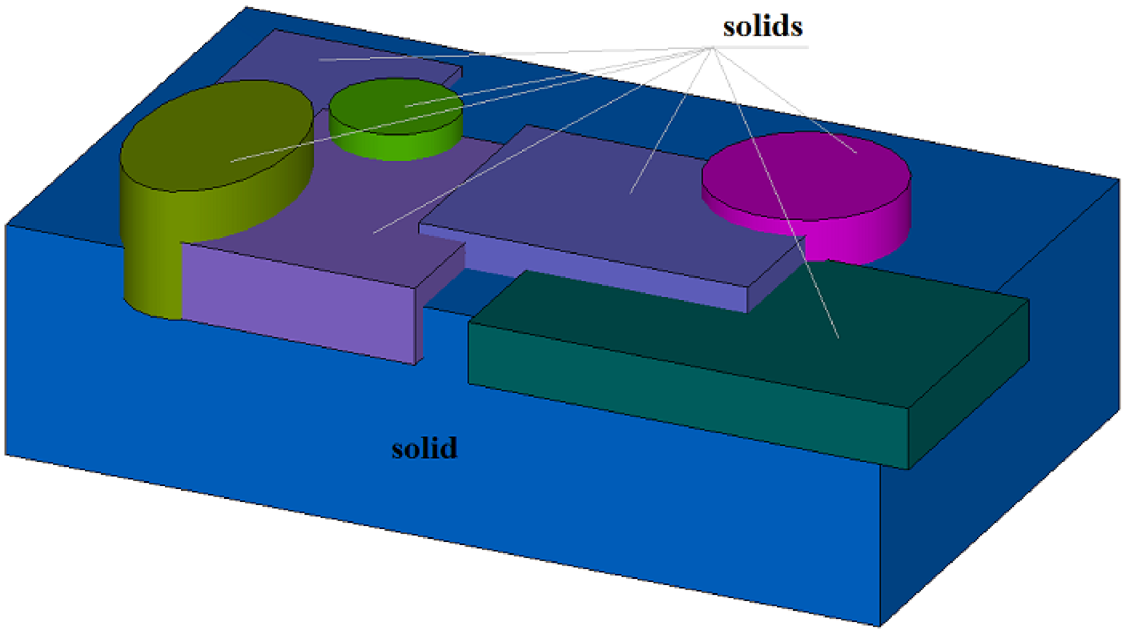


Fig. M.2.17.3.

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

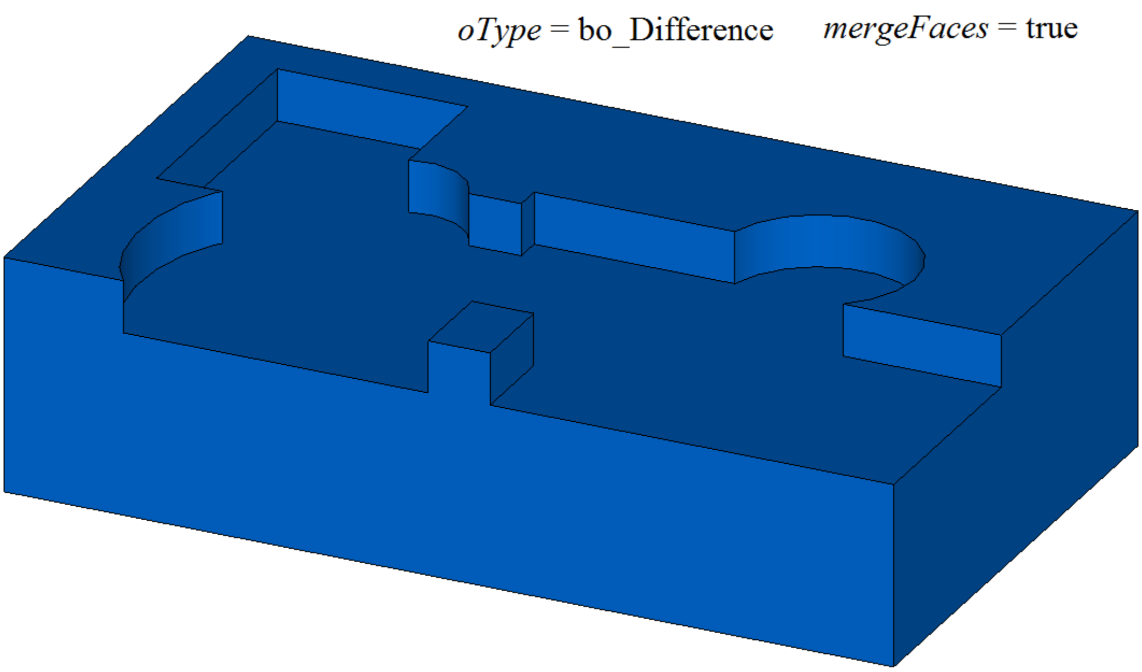


Fig. M.2.17.4.

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

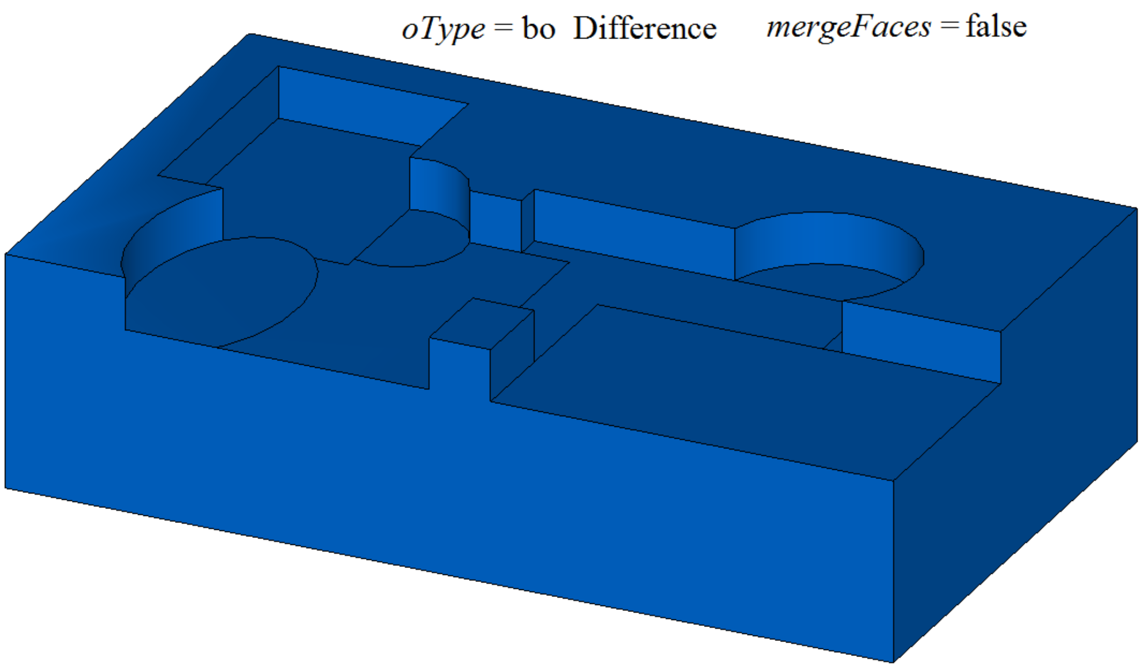


Fig. M.2.17.5.

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

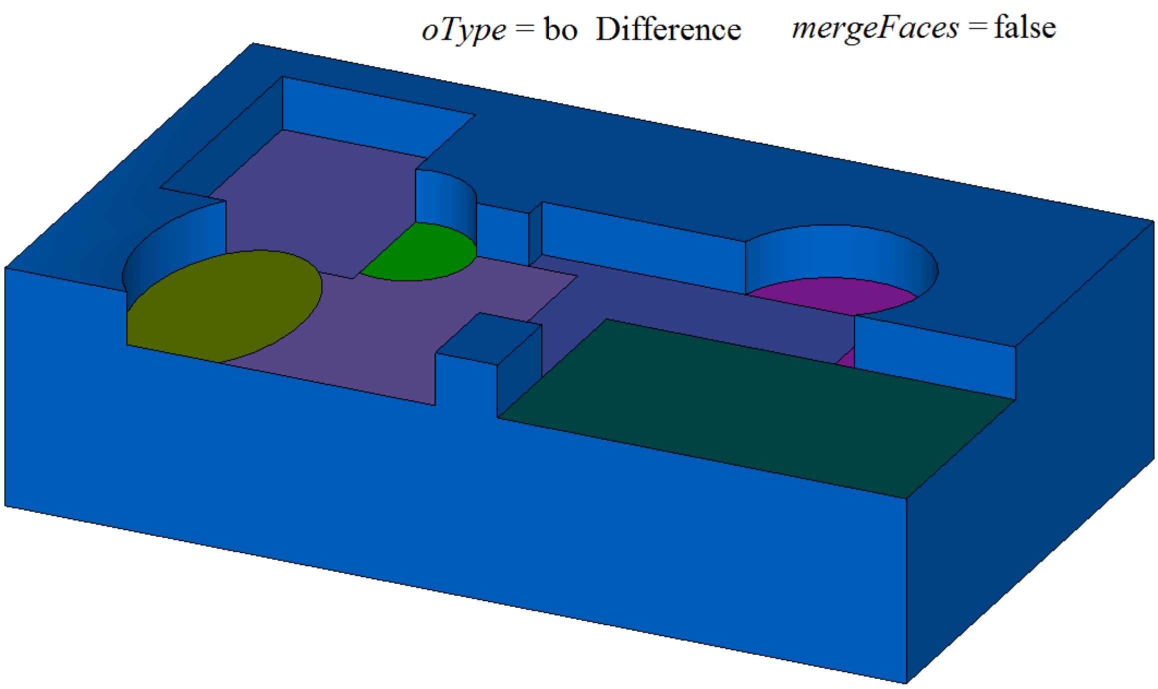


Fig. M.2.17.6.

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

file.

test.exe test application executes body Boolean operations on a set of bodies using New ->Body -> Attach to Other Body -> In a Set of Bodies, New ->Body -> Cut from Other Body -> In a Set of Bodies, New ->Body -> Intersection with Other Body -> In a Set of Bodies menu commands.

M.2.18. Merging a Set of Bodies

The method

MbResultType
UnionSolid (RPArray<MbSolid> & **solids**,
MbeCopyMode *sameShells*,
bool *checkIntersect*,
const MbSNameMaker & names,
bool *isArray*,
MbSolid*& **result**,
RPArray<MbSolid> * **notGluedSolids** = NULL)

merges bodies of the specified set.

Input parameters of the method are as follows:

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

Results of the method are **result** constructed body and **notGluedSolids** set of bodies 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 bodies. The method merges **solids** bodies and constructs **result** body; the bodies may not intersect with each other. *sameShells* parameter controls transfer of faces, edges and vertices from **solids** set of bodies to **result** resulting body. *checkIntersect* and *isArray* parameters control construction of the temporary body for **solids** set of bodies. names parameter is used to name the faces of the constructed body.

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

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

notGluedSolids parameter contains bodies 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 bodies.

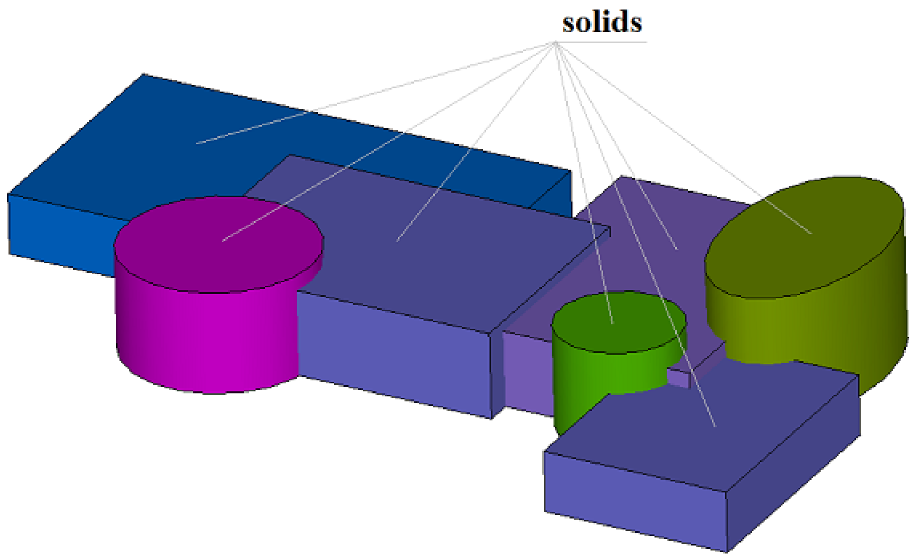


Fig. M.2.18.1.

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

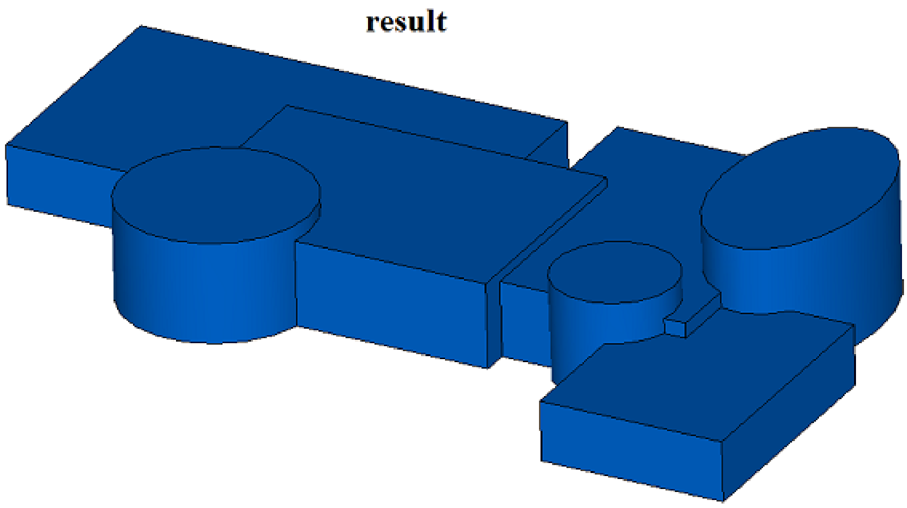


Fig. M.2.18.2.

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** body using all faces of **solids** bodies. So original bodies and a newly constructed body have the same faces.

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

file.

test.exe test application executes body Boolean operations on a set of bodies using New ->Body -> Based on Body -> Set of Bodies menu command.

M.2.19. Divide a Body to Disconnected Parts

The method

unsigned int
DetachParts (MbSolid & **solid**,
RPArray<MbSolid> & **parts**,
bool *sort*,
const MbSNameMaker & names)

divides a body to disconnected parts.

Input parameters of the method are as follows:

- **solid** is the original body,
- 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 body 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** body from **solid2** body 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** body that consists of several topologically disconnected parts to individual bodies. One part stays in **solid** original body, and all other parts are sent to received **parts** container.

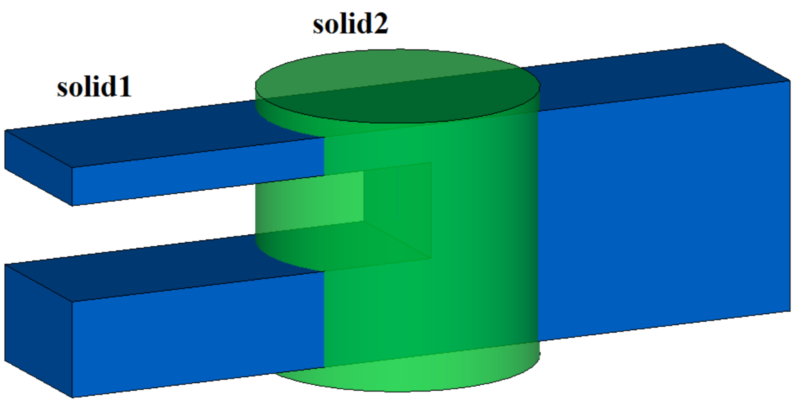


Fig. M.2.19.1.

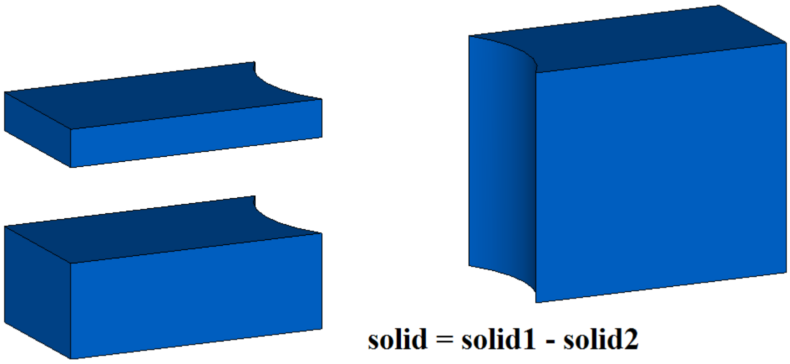


Fig. M.2.19.2.

If sorting flag sort==true, then the part with largest dimensions will remain in the original body, 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 body, and separated parts will be sorted by the number of initial face in the original body.

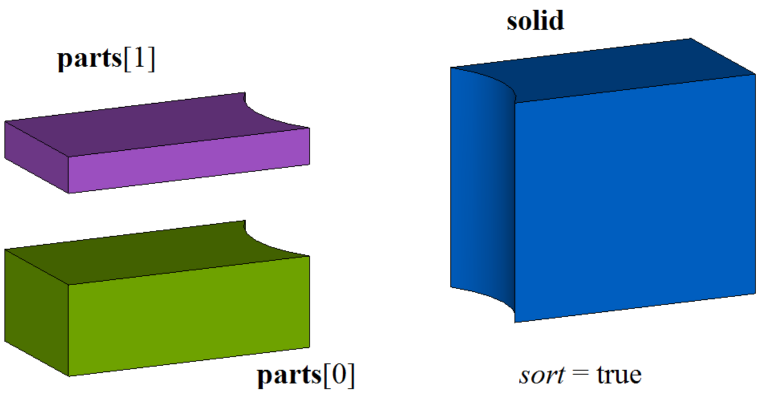


Fig. M.2.19.3.

names parameter provides naming of faces in the created body 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 body and adds all topologically disconnected parts of the original body to **parts** bodies as shown in Figure M.2.19.4.

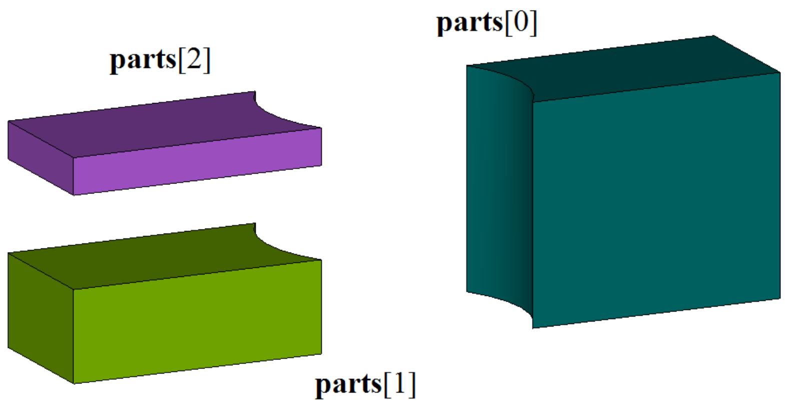


Fig. M.2.19.4.

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

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

test.exe test application executes body Boolean operations on a set of bodies using Modify ->Body or Shell->Divide Parts menu command.

M.2.20. Separation of Disconnected Parts

The method

MbResultType
ShellPart (const MbSolid & **solid**,
size_t *id*,
const MbPath & path,
const MbSNameMaker & names,
MbPartSolidIndices & partIndices,
MbSolid * & **result**)

creates a separate body from a specified part of the original body that falls apart.

Input parameters of the method are as follows:

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

Output parameters of this method are **result** constructed body and indices of body 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 body from the specified part of original body. The original body should consist of separate parts. In Fig. M.2.20.1, you can see the result of Boolean operation that subtracts bodies shown in Figure M.2.19.1. The resulting body consists of several topologically separated parts. This method permits to create a body keeping only one of topologically separated parts of the original body.

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

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

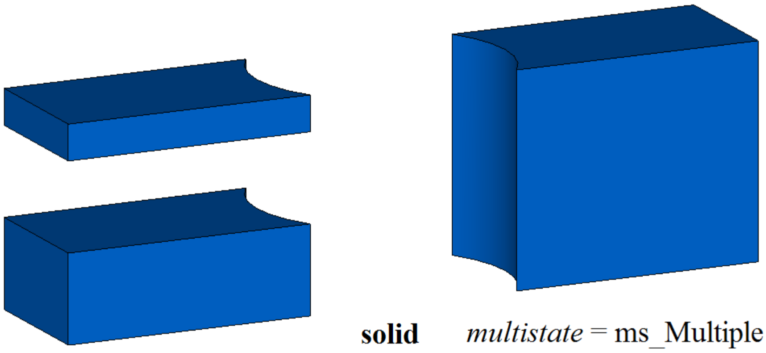


Fig. M.2.20.1.

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

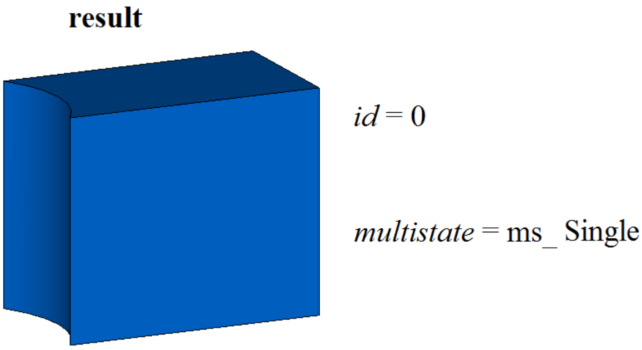


Fig. M.2.20.2.

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

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

test.exe test application executes body Boolean operations on a set of bodies using New ->Body -> Based on Body -> Part of Bodies Set menu command.

M.2.21. Splitting Body Faces

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

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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

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 body, **faces** that should be split and **items**[0] cutting surface.

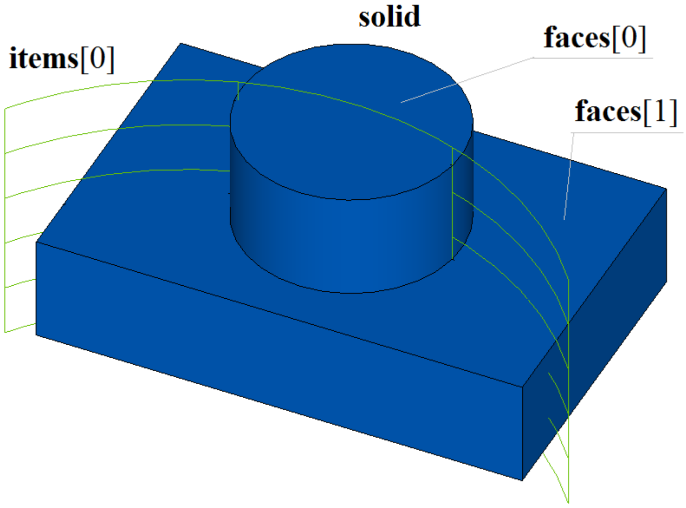


Fig. M.2.21.1.

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

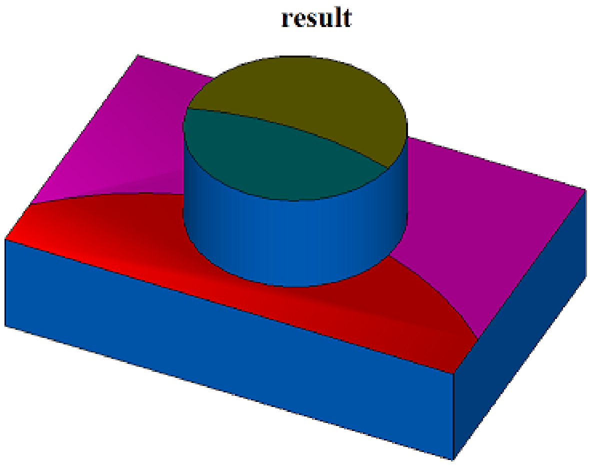


Fig. M.2.21.2.

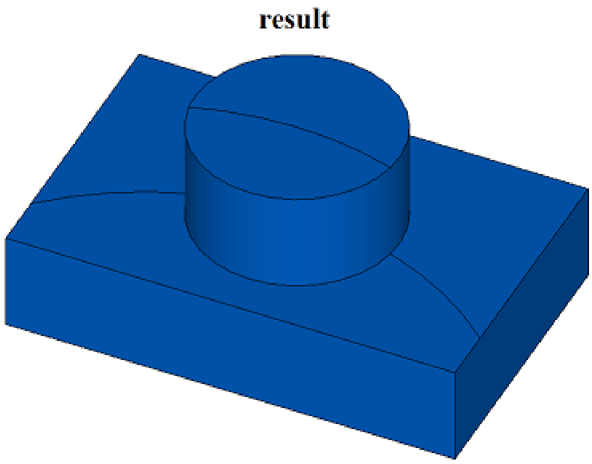


Fig. M.2.21.3.

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** body 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 body.

SplitSolid methods add MbSplitShell constructor in the log of the newly constructed body 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 body using New ->Body -> By Processing Faces -> By Splitting Face menu command.

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

The 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 body.

Input parameters of the method are as follows:

- **solid** is the original body (it may be equal to zero),
- *sameShell* is body 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 body.

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

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.

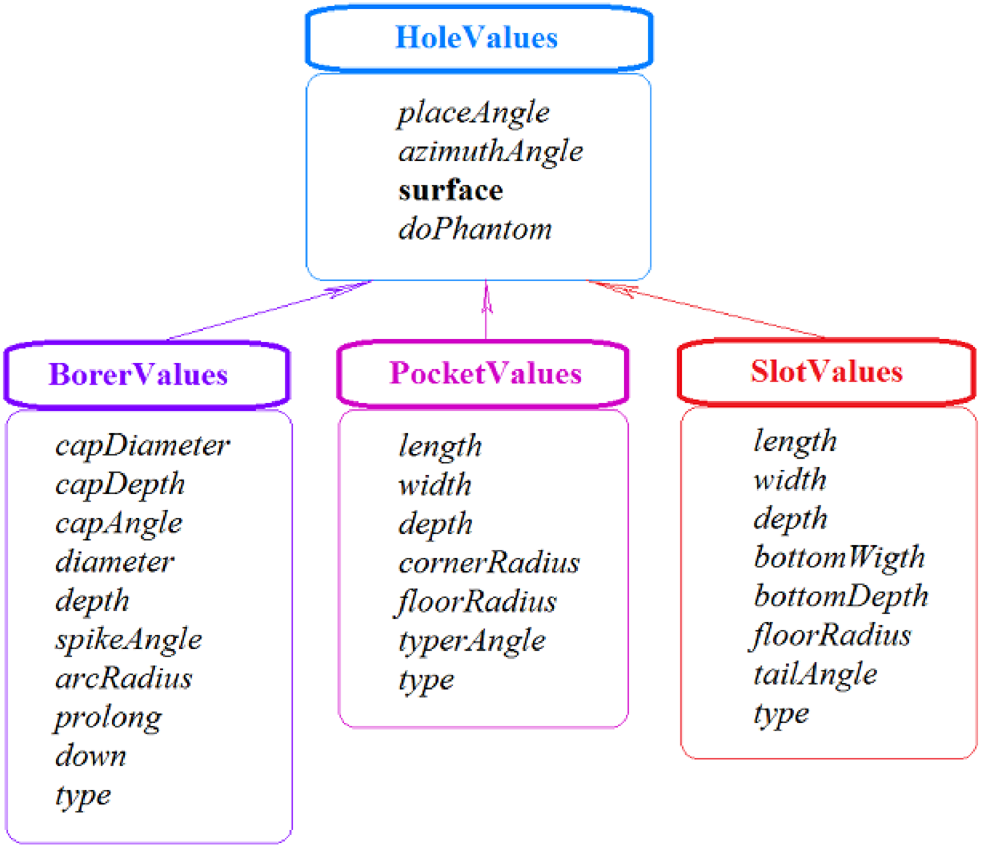
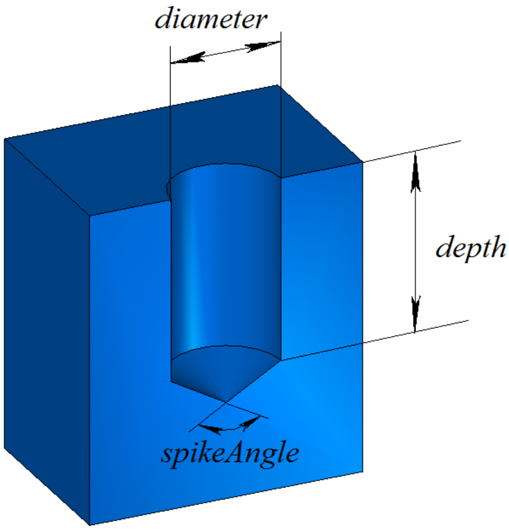


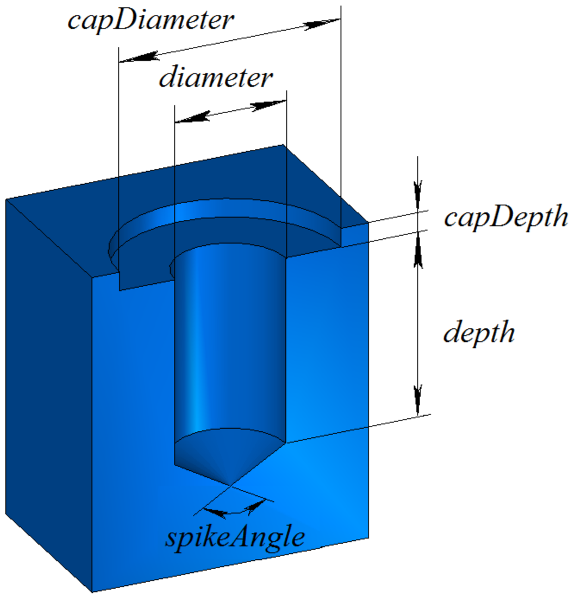
Fig. M.2.22.1.

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.



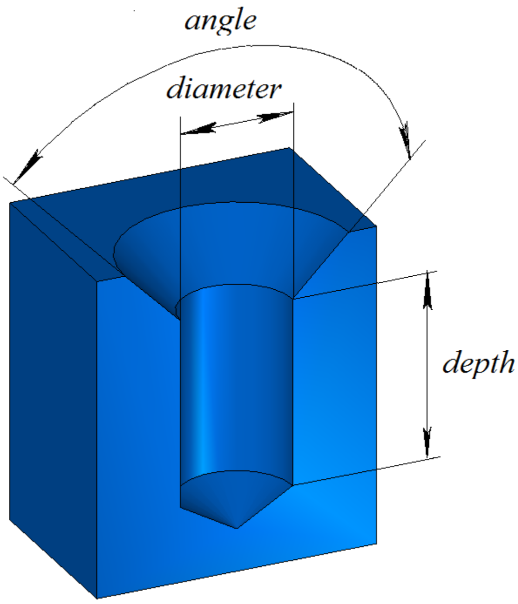
BorerValues::*type* = bt_SimpleCylinder

Fig. M.2.22.2.



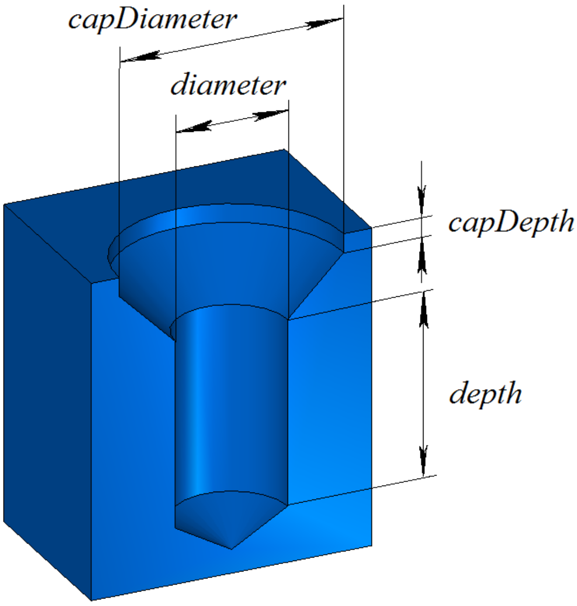
BorerValues::*type* = bt_TwofoldCylinder

Fig. M.2.22.3.



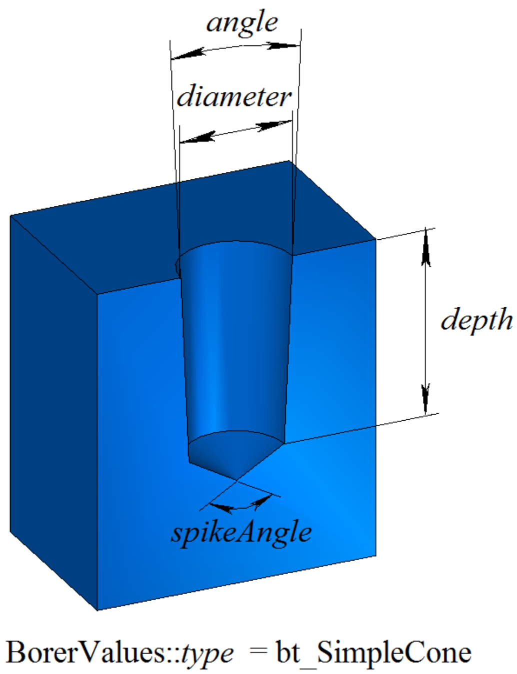
BorerValues::*type* = bt_ChamferCylinder

Fig. M.2.22.4.



BorerValues::*type* = bt_ComplexCylinder

Fig. M.2.22.5.



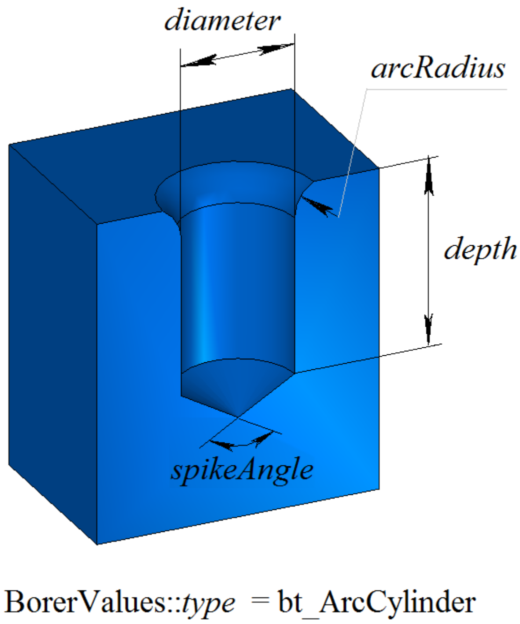


Fig. M.2.22.6. *Fig. M.2.22.7.*

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 body with a rectangular pocket without a slope of side faces.

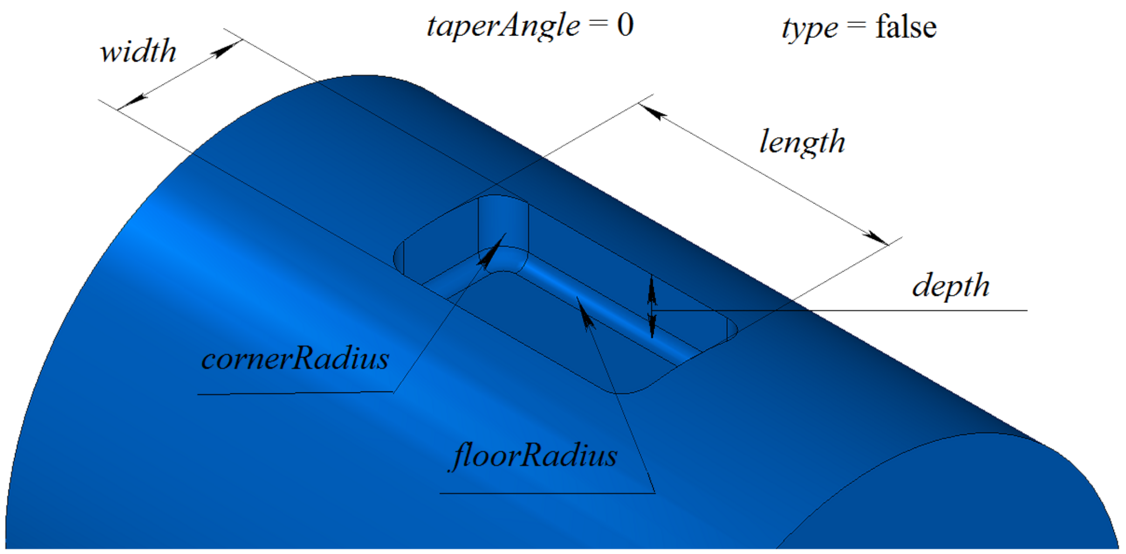


Fig. M.2.22.8.

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 body 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 body using New ->Body -> Based on Body -> With a Hole menu command.

M.2.23. Constructing a Body with an Enforcement Rib

The 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 body with an enforcement rib.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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 body. Contour segment with specified number defines a slope vector.

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

RibValues

double *angle*1
double *angle*2
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 body to **result** constructed body. names parameter is used to name the faces of the constructed body.

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 body, **place** local coordinate system and **contour** in XY plane of the latter.

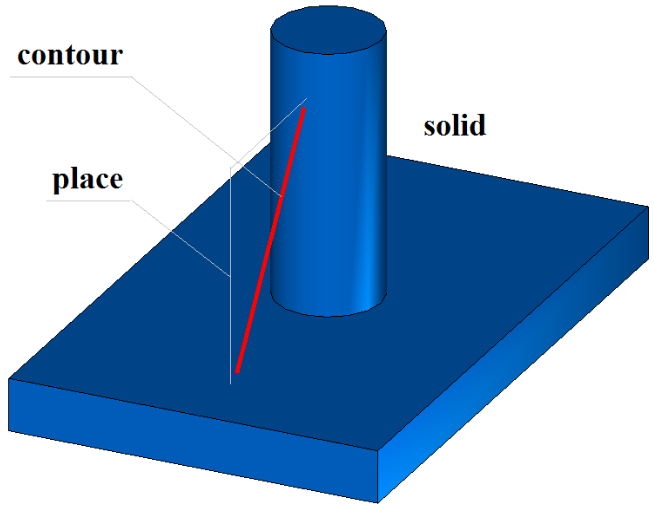


Fig. M.2.23.2.

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

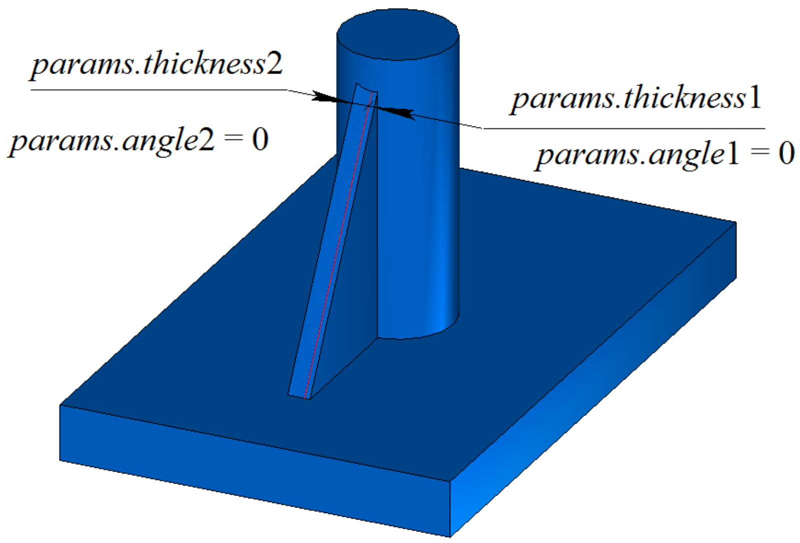


Fig. M.2.23.3.

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

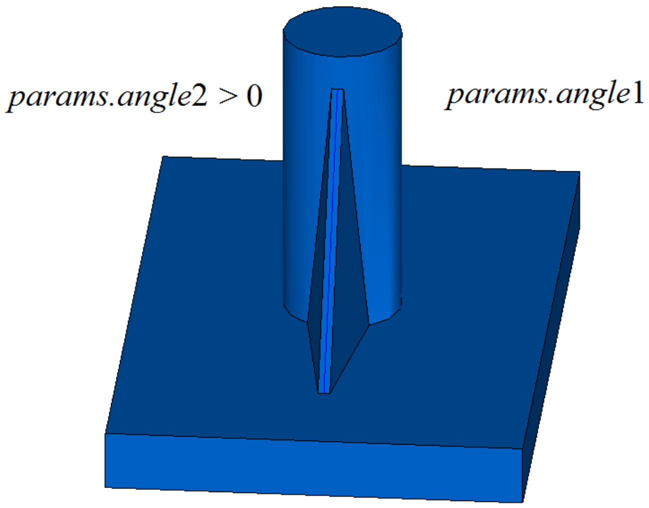


Fig. M.2.23.4.

RibSolid method adds MbRibSolid constructor in the log of newly constructed body 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 body using New ->Body -> Based on Body -> With Enforcement Rib menu command.

M.2.24. Sloping Body Faces

The 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 body with specified faces of the body sloped from neutral isometric plane at a predetermined angle.

Input parameters of the method are as follows:

- **solid** is the original body,
- *sameShell* is a version of original body 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 body.

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 body with faces sloped relative to their position in **solid** original body. *params* parameter defines construction parameters. *sameShell* parameter controls transfer of faces, edges and vertices from **solid** original body to **result** constructed body. XY plane in **place** local coordinate system defines the plane, in relation to which body 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 body **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 body.

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 body, **place** local coordinate system, in relation to XY plane of the latter sloping is executed and **faces** that should be sloped.

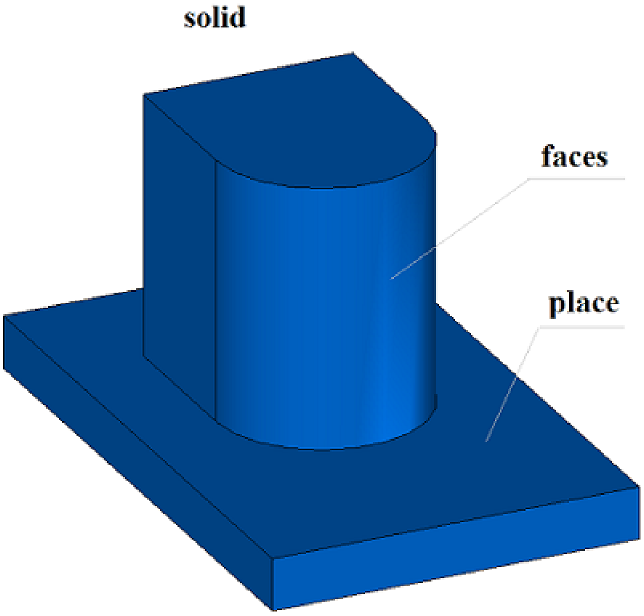


Fig. M.2.24.1.

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

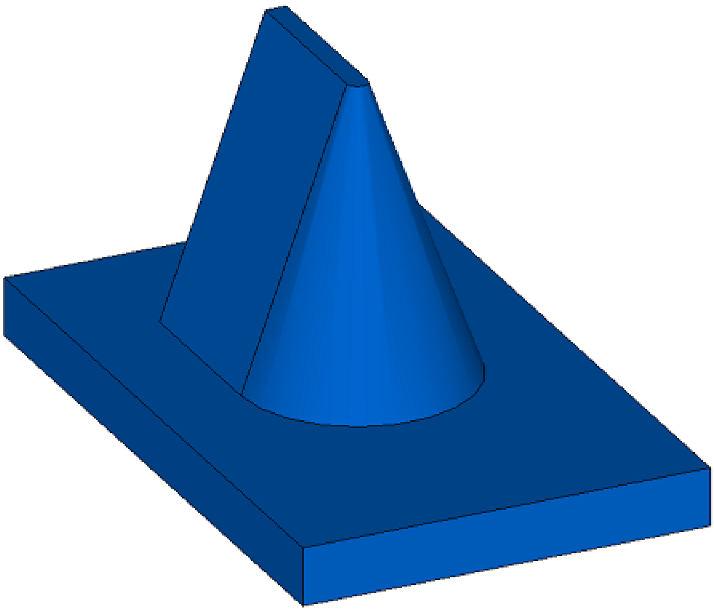


Fig. M.2.24.2.

DraftSolid method adds MbDraftSolid constructor in the log of the newly constructed body 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 body using New ->Body -> By Processing Faces -> By Sloping Faces menu command.

M.2.25. Multiplication of Bodies

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

Input parameters of the method are as follows:

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

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

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 body. *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.

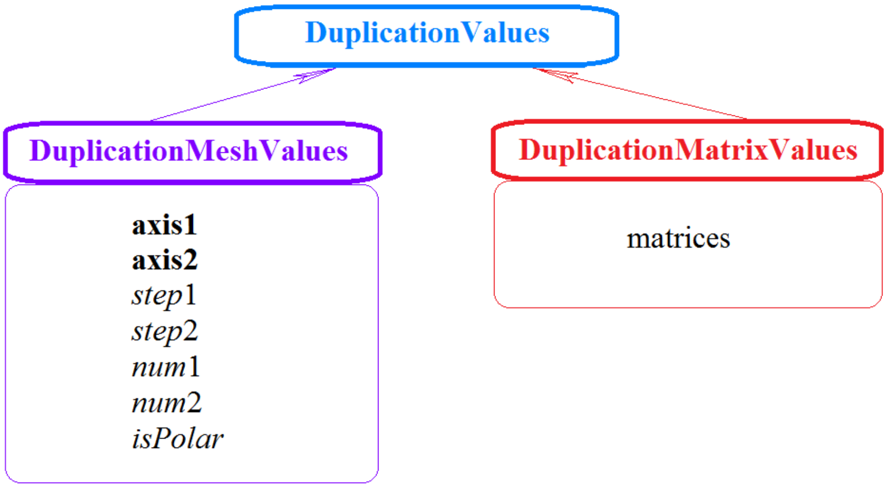


Fig. M.2.25.1.

DuplicationMeshValues parameters should be used to make copies of the body 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 body and its copies are located in the nodes of 2D grid having *parameters.***axis1** and *parameters.***axis2**. The original body is the reference point. Along *parameters.***axis1**, *parameters.num*1 copies of the body are located with *parameters.step*1; along *parameters.***axis2** axis, *parameters.num*2 copies of the body are located with *parameters.step*2, including the original body. If *parameters.isPolar*=true, than newly constructed copies of the body are located in nodes of a polar grid. The original body 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 body are located with *parameters.step*1 along radial directions; *parameters.num*2 copies of the body are located on each circle with angular *parameters.step*2 .

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

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

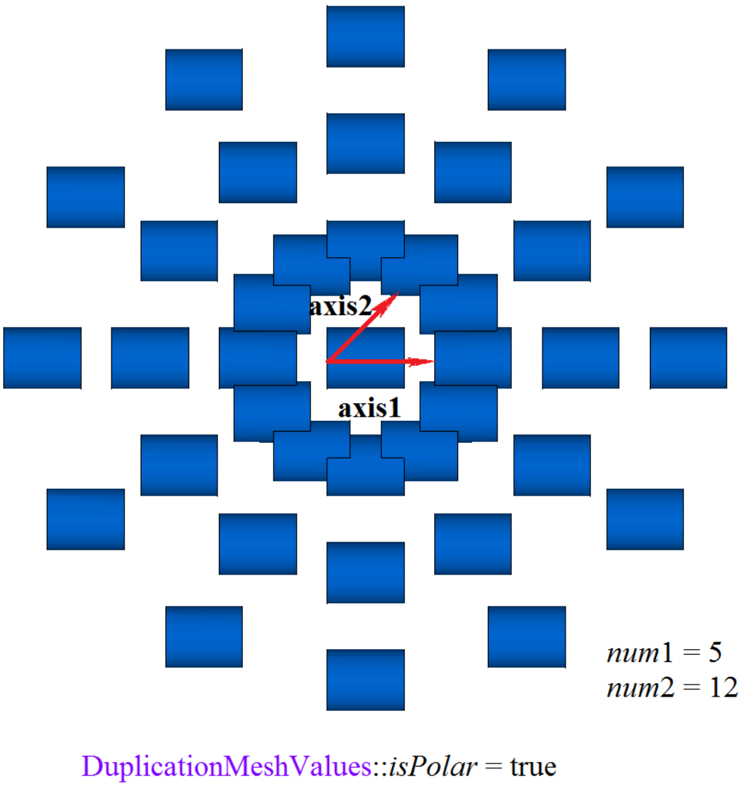


Fig. M.2.25.2.

DuplicationSolid method adds MbDuplicationSolid constructor in the log of newly constructed body 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 body using New ->Body -> Based on Body -> By Grid Multiplication and New ->Body -> Based on Body -> By Matrix Multiplication menu commands.

M.2.26. Dividing a shell into parts using a given set of edges

The method
```
MbResultType DivideShell( MbSolid              & solid,
                          MbeCopyMode            sameShell,
                    const MbDivideShellParams & params,
                          c3d::SolidSPtr       & resSolid );
```

divides a shell of the solid using a given set of edges.
Input parameters of the method are as follows:

- **solid -** the original body;
- **sameShell -** copying method for the body;
- **params -** the object with parameters of operation.

Method output parameter:

- **resSolid -** result constructed body.

The method returns the result code of the operation.

The `MbDivideShellParams` class contains the input data to perform the shell split operation. When creating an object of the `MbDivideShellParams` class, it can be immediately initialized with a vector of edges along which the division will be performed.

`MbDivideShellParams` class constructor

```
MbDivideShellParams( const c3d::EdgesSPtrVector & edges,
                     const MbSNameMaker         & operNames );
```
where,

- **edges** - the vector of edges along which the division will be made;
- **operNames** - namer.

As a result of the successful operation of the function, the **solid** input body will be split and all received non-closed bodies will be placed in the shell of the returned **resSolid** body.

M.2.27. Spliting the solid into separate parts

The method
```
size_t CreateParts( const MbSolid &          solid,
                          RPArray<MbSolid> & parts,
                    const MbSNameMaker &     names );
```
divides the original body into separate parts if the body contains topologically unrelated elements.

Input parameters of the method are as follows:

- **solid -** the original body;
- **operNames** - namer.

Method output parameter:

- **parts** - parts of the body.

The method returns the number of created parts.

M.3. TWO-DIMENSIONAL CURVE CONSTRUCTION METHODS

Two-dimensional curves are used to describe domain of definition for a surface, to work with curves on surfaces, to construct curves as intersection of surfaces, and to construct mating surfaces. In order to construct solid bodies, two-dimensional curves are used as input parameters as sketch elements. Furthermore, two-dimensional curves are used as elements of flat projections in geometric models. All two-dimensional curves are inheritors of MbCurve class, they are described in Chapter O.3. TWO-DIMENSIONAL CURVES. A curve can be constructed by direct call of corresponding constructor or using the methods described in this section.

M.3.1. Constructing a Two-Dimensional Straight Line/Segment

The method
MbResultType
Line (const MbCartPoint & ***point*1**,
constMbCartPoint & ***point*2**,
MbCurve *& ***result***)
constructs a two-dimensional straight line based on two non-matching points.

Input parameters of the method are as follows:

- ***point*1** is the first point that lies on the straight line,
- ***point*2** is the second point, that lies on the straight line,

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

***point*1** parameter defines the start point of the straight line that corresponds to zero parameter value. The vector starting in ***point*1** and ending in ***point*2** defines the direction of the line, Fig. M.3.1.1. A derivative of straight line has unit length.

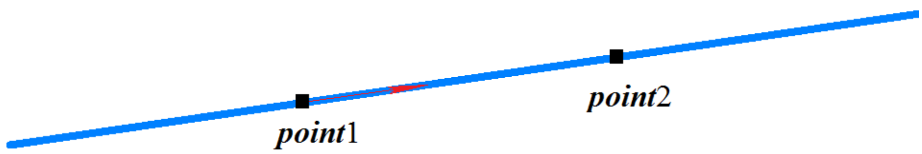


Fig. M.3.1.1.

Straight lines are described in Item O.3.2. MbLine Two-Dimensional Straight Line.

The method
MbResultType
Segment (const & ***point*1**,
const MbCartPoint& ***point*2**,
MbCartPoint *& ***result***)
constructs a two-dimensional straight line segment based on two non-matching points.

Input parameters of the method are as follows:

- ***point*1** is the segment starting point,
- ***point*2** is the segment end point.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

***point*1** parameter defines the starting point of the line segment, it corresponds to zero value of curve parameter. ***point*2** parameter defines the end point of the line segment, it corresponds to curve parameter equal to one, see Fig. M.3.1.2.

Fig. M.3.1.2.

Line segments are described in Item O.3.3. MbLineSegment Two-Dimensional Straight Line Segment.

M.3.2. Constructing a Two-Dimensional Circle, Ellipse and their Arcs

The method
MbResultType
Arc (const MbCartPoint & ***centre***,
const SArray<MbCartPoint> & ***points***,
bool *closed*,
double *angle*,
double & *a*,
double & *b*,
MbCurve *& ***result***)
constructs a two-dimensional elliptic arc. In a special case, the method constructs a two-dimensional circular arc.

Input parameters of the method are as follows:

- ***centre*** is a center of the ellipse,
- ***points*** is a set of points, it can be empty,
- *closed* is a flag that defines whether the curve is cyclically closed,
- *angle* is an angle that defines the size of the elliptic arc,
- *a* is the length of the first semi-axis of the ellipse (it is calculated if ***points*** set is non-empty),
- *b* is the length of the second semi-axis (calculated if ***points*** set is non-empty).

The output parameters of the method are semi-axes of the ellipse and constructed **result** curve.

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

The method is declared in action_curve.h file.

The curve can be constructed based either on specified points or on scalar parameters. ***centre*** parameter defines the central point of the ellipse. ***points*** set can be empty, but in this case *a* and *b* ellipse semi-axis lengths should be non-empty.

If ***points*** set contains two elements, then ***centre***, ***points***[0], ***points***[1] points determine the plane, in which ***axisX*** and ***axisY*** of the local ellipse coordinate system are located: ***axisX*** of local ellipse coordinate system is directed from the center to ***points***[0]; ***axisY*** of the local ellipse coordinate system is orthogonal to ***axisX*** and directed from the center to ***points***[1] point. The distance between points ***centre*** and ***points***[0] defines the length of the first ellipse semi-axis *a*, and the distance between ***centre*** point and the projection of ***points***[1] to ***axisY*** defines the length of the second ellipse semi-axis *b*, see Fig. M.3.2.1.

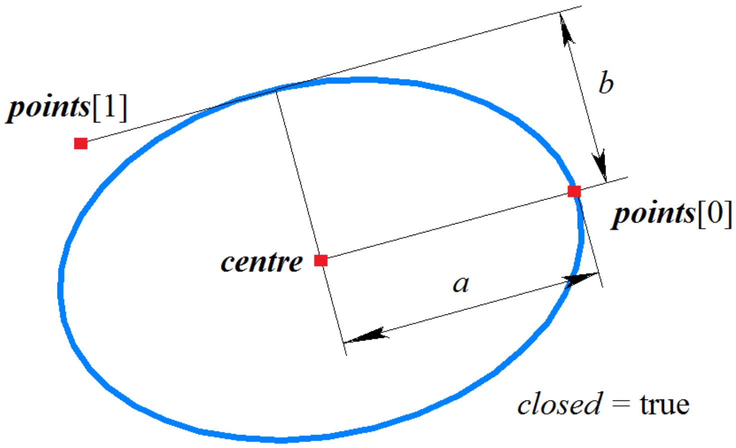


Fig. M.3.2.1.

If ***points*** set contains one element, a circle with a radius equal to the distance between ***centre*** and ***points***[0] points will be constructed. ***axisX*** of the local coordinate system of the circle will be directed from center to ***points***[0] point, and ***axisY*** of the local coordinate system will be orthogonal to ***axisX***.

If ***points*** set contains three elements, an ellipse with a center in ***centre*** point, will contain the following points: ***points***[0], ***points***[1] and ***points***[2]. The positions of the axes of the local ellipse coordinate system and the length of ellipse semi-axes will be calculated based on ***centre***, ***points***[0], ***points***[1] and ***points***[2] points, see Fig. M.3.2.2.

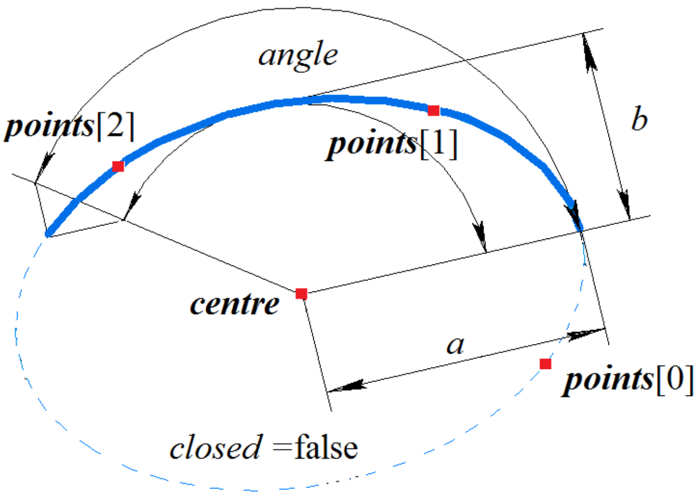


Fig. M.3.2.2.

Elliptical arcs are described in Item O.3.4. MbArc Two-Dimensional Elliptical Arc.

If ***points*** set is empty, then the lengths of ellipse semi-axes *a* and *b* are used as input parameters, and ***axisX*** and ***axisY*** axes of the local ellipse coordinate system match with global ellipse coordinate axes.

closed parameter defines whether the curve is cyclically closed. If *closed*=false, then *angle* parameter, which determines opening angle of the elliptical arc in parametric units, must be non-zero.

In a special case when *a*=*b*, this method constructs a circle (if *closed*=true or *angle*=0) or a circular arc (if *closed*= false, and *angle*>0, and *angle*<2π).

M.3.3. Constructing Two-Dimensional Curves Based on Control Points

The method
MbResultType

SplineCurve (const SArray<MbCartPoint> & ***points***,
bool *closed*,
MbePlaneType *curveType*,
MbCurve *& ***result***)

constructs a two-dimensional curve of the required type based on a specified set of control points.

Input parameters of the method are as follows:

- ***points*** is a set of control points,
- *closed* is a flag that defines whether the curve is cyclically closed,
- *curveType* is a curve type.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

points parameter contains control points of the curve. *closed* parameter defines whether the constructed curve is cyclically closed. *curveType* parameter defines curve type that determines curve shape. A various number of control points is required to construct created curves of different types. Table M.3.3.1 shows the number of control points from ***points*** set required to create *curveType* curve.

Table M.3.3.1.

curveType	**Curve type**	**Number of control points**
pt_LineSegment	Straight line segment	2 points
pt_Arc	Circular arc	3 points
pt_Polyline	Polygonal line	2 or more points
pt_Nurbs	NURBS curve	2 or more points
pt_Hermit	Hermite spline	2 or more points
pt_Bezier	Bezier curve	2 or more points
pt_CubicSpline	Cubic Spline	2 or more points

A line segment that starts in ***points***[0] and ends in ***points***[1] will be constructed if *curveType*=pt_LineSegment. Line segments are described in Item O.3.3. MbLineSegment Two-Dimensional Straight Line Segment.

A circular arc that starts in ***points***[0], passes through ***points***[1] and ends in ***points***[2] will be constructed if *curveType*=pt_Arc and *closed*=false, see M.3.3.1. If *closed*=true, then a circle containing ***points***[0], ***points***[1], and ***points***[2], will be constructed. Circular arc and an elliptical arc are described in Item O.3.4. MbArc Two-Dimensional Elliptical Arc.

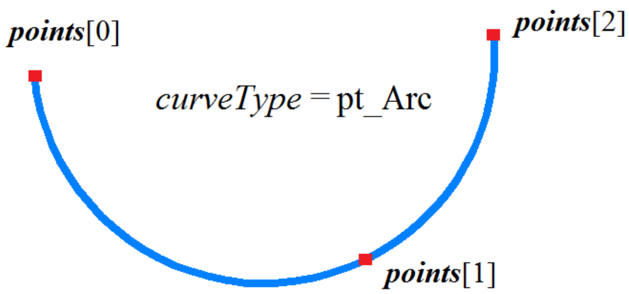


Fig. M.3.3.1.

If *curveType*=pt_Polyline, then a polyline containing ***points***[0], ***points***[1], ..., ***points***[*n*] will be constructed, see Fig. M.3.3.2. If *closed*=true, then a cyclically closed polyline containing a segment between ***points***[0] and ***points***[*n*] will be constructed. Polylines are described in Item O.3.5. MbPolyline Two-Dimensional Polyline.

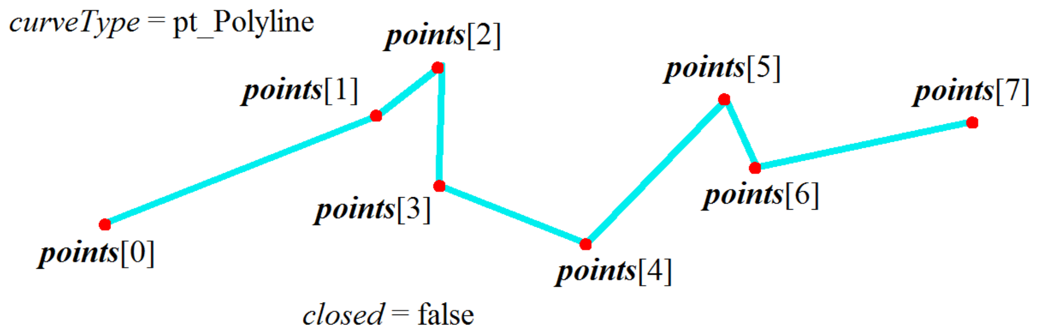


Fig. M.3.3.2.

If *curveType*=pt_Nurbs, then a fourth-order Non-Uniform Rational *B*-Spline will be constructed, see Fig. M.3.3.3. spline control points will be determined based on the condition that the spline contains ***points***[0], ***points***[1],..., ***points***[*n*]. If *closed*=true, then a cyclically closed spline will be constructed. NURBS curves are described in Item O.3.5. MbPolyline Two-Dimensional Polyline.

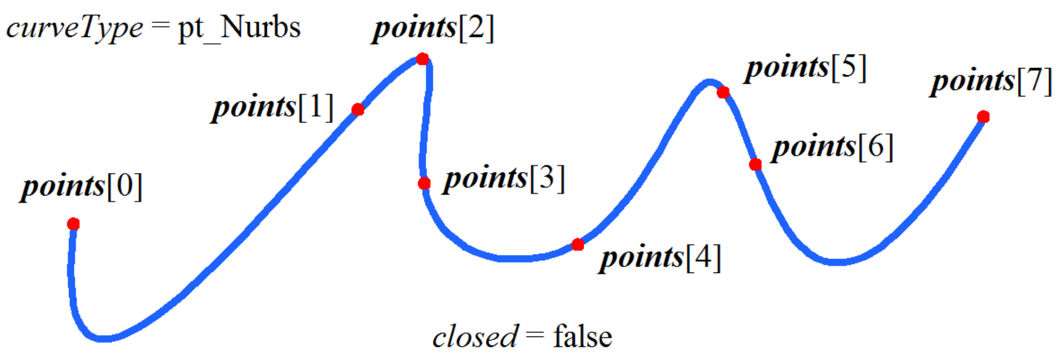


Fig. M.3.3.3.

If *curveType*=pt_Hermit, then a compound curve containing smoothly joined third-order Hermit splines will be constructed. Each third-order Hermit spline will connect adjacent ***points***[*i*–1] and ***points***[*i*], see Fig. M.3.3.4. If *closed*=true, then a cyclically closed curve containing an Hermit spline between ***points***[0] and ***points***[*n*] will be constructed. Compound third-order Hermit splines are described in Item O.3.7. MbHermit Two-Dimensional Hermite Curve.

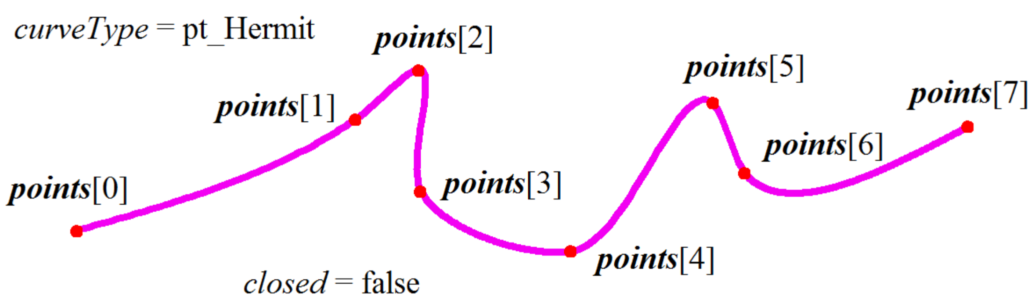


Fig. M.3.3.4.

If *curveType*=pt_Bezier, then a compound curve containing smoothly joined third-order Bezier splines will be constructed. Each third-order Bezier spline will connect adjacent ***points***[*i*–1] and ***points***[*i*], see Fig. M.3.3.5. If *closed*=true, then a cyclically closed curve containing a Bezier spline between ***points***[0] and ***points***[*n*] will be constructed. Compound third-order Bezier splines are described in ItemO.3.8. MbBezier

[Two-Dimensional Bezier Composite Curve](#).

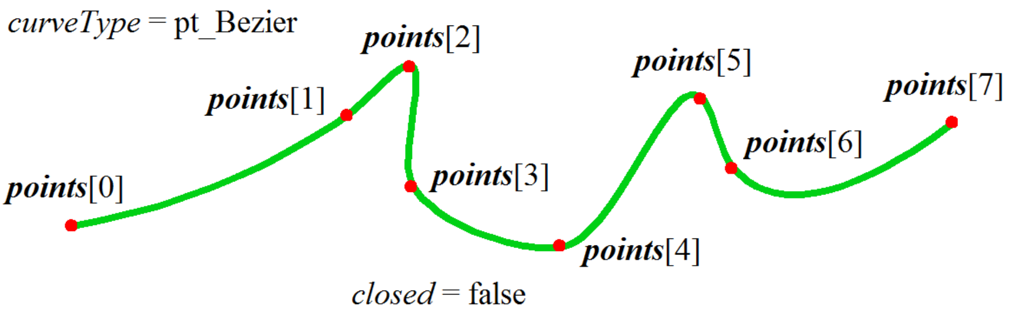


Fig. M.3.3.5.

If *curveType*=pt_CubicSpline, then a cubic spline containing ***points***[0], ***points***[1], ..., ***points***[*n*] will be constructed, see Fig. M.3.3.6. If *closed*=true, then a cyclically closed curve containing a segment between ***points***[0] and ***points***[*n*] will be constructed. Cubic splines are described in Item [O.3.9. MbCubicSpline Two-Dimensional Cubic Spline](#).

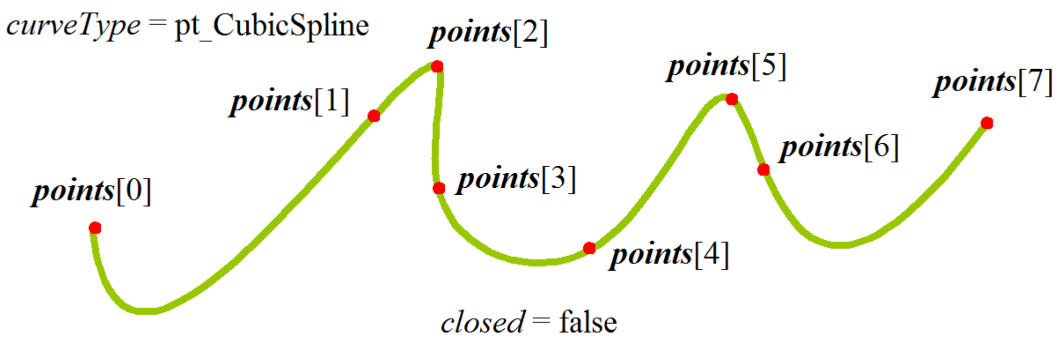


Fig. M.3.3.6.

For the purpose of comparison, Fig. M.3.3.7 shows a NURBS curve, a compound Hermite spline, a compound Bezier spline, and a cubic spline constructed based on the same control points (***points***[0], ***points***[1], ..., ***points***[*n*]) and the curves have the same order.

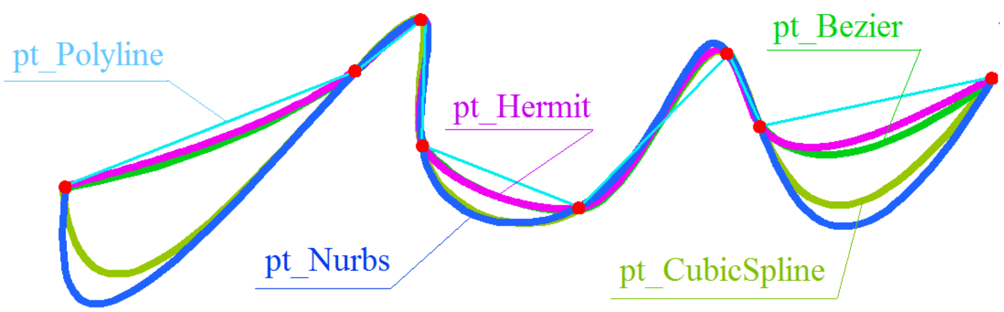


Fig. M.3.3.7.

One can see that the curves have different shapes.

M.3.4. Constructing Two-Dimensional NURBS Curve

The method
MbResultType
NurbsCurve (const SArray<MbCartPoint> & ***points***,
const SArray<double> & *weights*,
size_t degree,
const SArray<double> & *knots*,
bool *closed*,
MbCurve *& ***result***)
constructs a two-dimensional NURBS curve based on a given set of control points.

Input parameters of the method are as follows:

- ***points*** is a set of control points,
- *weights* is a set of weights of control points,
- degree is a curve (*B*-spline) order,
- *knots* is a set of parametric knots (knot vector),
- *closed* is a flag that defines whether the curve is cyclically closed.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

This method constructs a NURBS curve (a curve based on *B*-splines), such curves are described in Item O.3.6. MbNurbs Two-Dimensional NURBS-Curve. *weights* set should correspond to the set of control points named points. Degree of curve named degree should not exceed the number of control points. *closed* parameter defines whether the constructed curve is cyclically closed. *knots* vector is a non-declining sequence of real numbers that defines definition domain of curve parameter and curve shape. If *closed*=false, then knot vector should contain the number of elements equal to the number of control points plus degree of curve. If NURBS curve should pass through terminal control points, then the first degree values of *knots* vector elements should be equal, and the last degree values of *knots* vector elements should also should be equal. If *closed*=true, then node vector should contain the number of elements equal to the number of control points plus doubled degree of the curve minus one. Fig. M.3.4.1. shows closed fourth-degree NURBS curve.

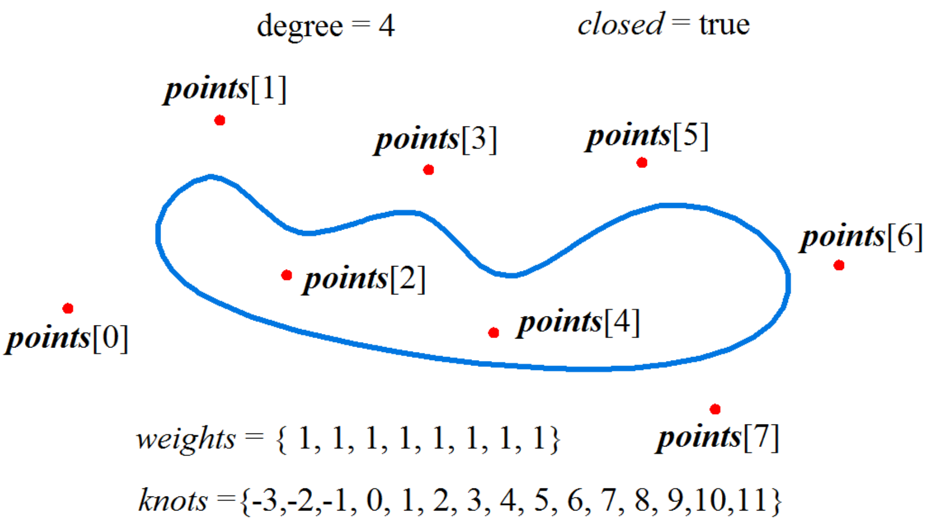


Fig. M.3.4.1.

Fig. M.3.4.2. shows non-closed fourth-degree NURBS curve.

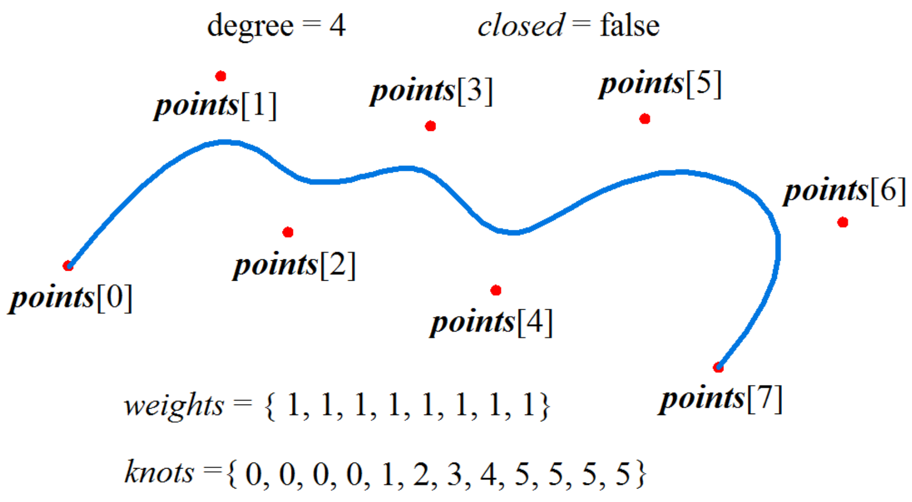


Fig. M.3.4.2.

Fig. M.3.4.3. shows three closed NURBS curves having different orders constructed based on the same control points.

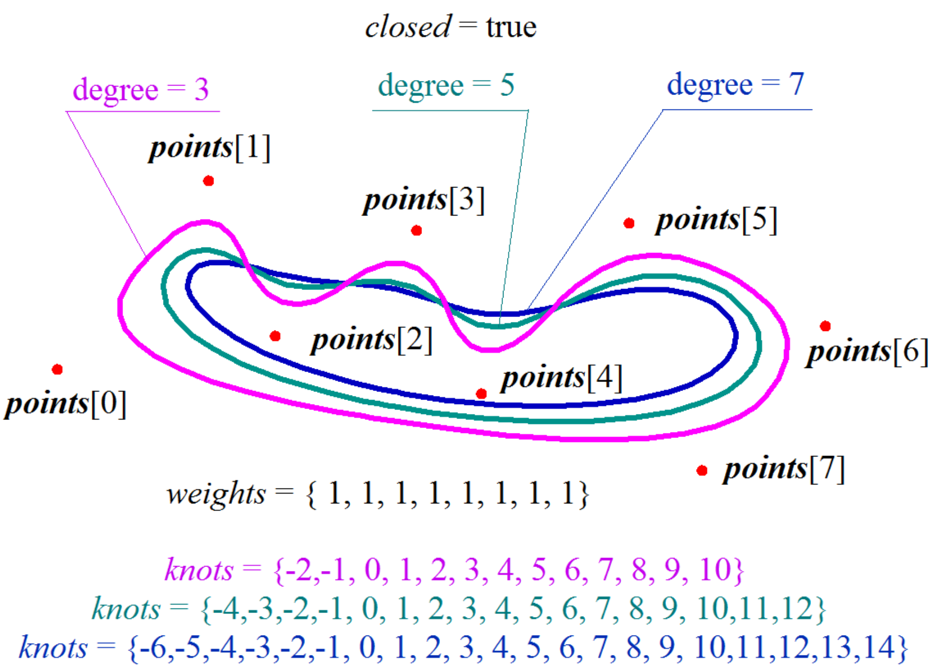


Fig. M.3.4.3.

Fig. M.3.4.4 shows three non-closed NURBS curves having various orders constructed based on the same control points.

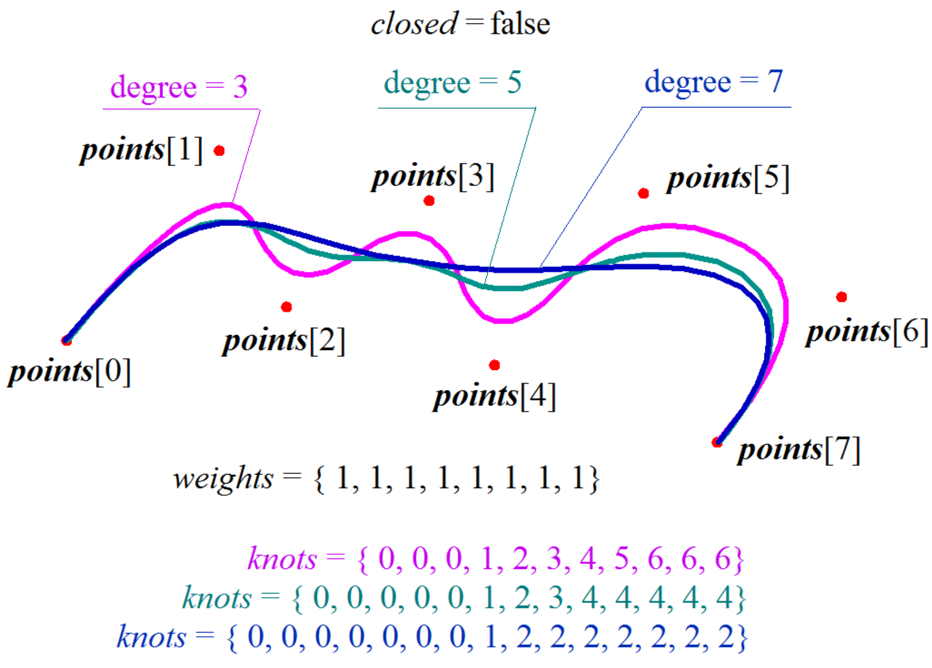


Fig. M.3.4.4.

Fig. M.3.4.5 shows three non-closed fourth-order NURBS curves with various control point weights.

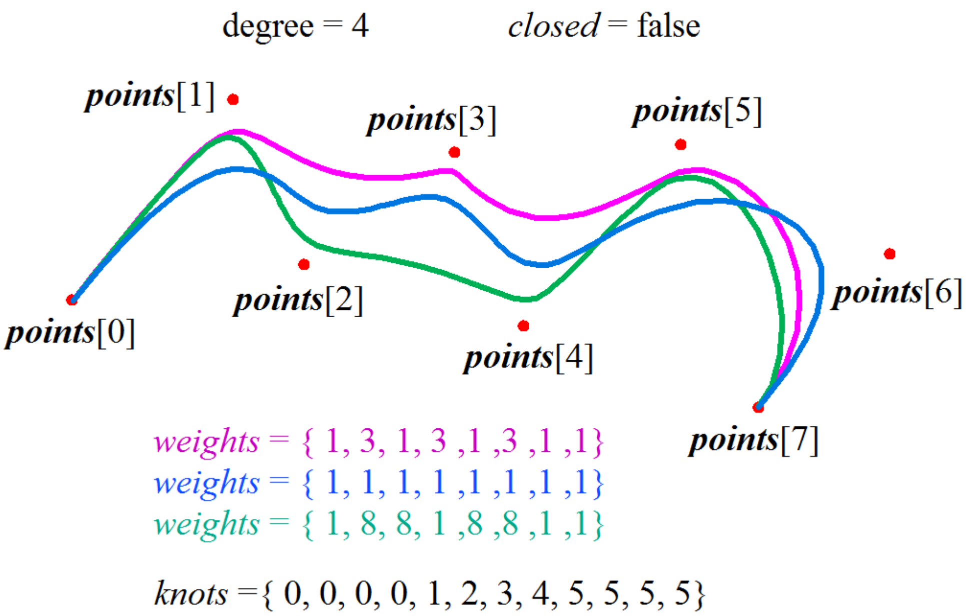


Fig. M.3.4.5.

Fig. M.3.4.6 shows a non-closed third-order NURBS curve, the shape of which matches the circular arc. The distance between ***points***[0] and ***points***[1] control points is equal to the distance between ***points***[1] and ***points***[2] control points, and *weights*[1] weight of the middle curve control point is equal to the weight of the terminal control points multiplied by half-angle cosine of the arc.

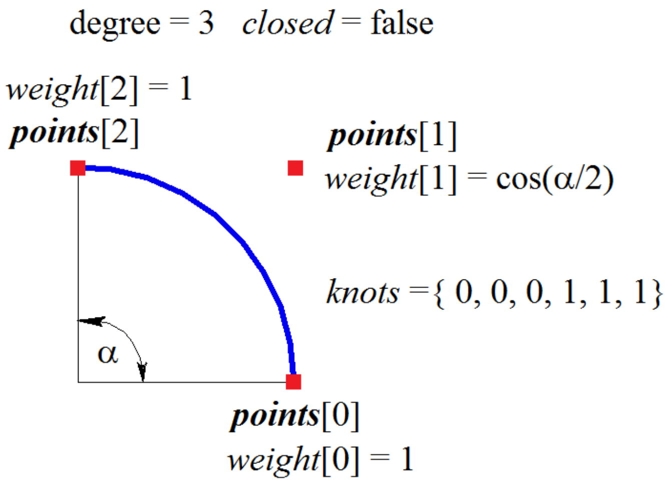


Fig. M.3.4.6.

The method
MbResultType
NurbsCopy (const MbCurve & ***curve***,
MbCurve *& ***result***)
constructs a NURBS copy of selected two-dimensional curve.

curve original curve is an input parameter of the method.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

This method constructs a NURBS curve with a shape that is a copy of the original curve. NURBS curves exactly copies the shape of the original curve for the majority of curve types. If it is impossible to exactly reproduce the shape of the original curve, then NURBS copy approximates the original curve with an error less than 0.0001.

M.3.5. Constructing Convex Equilateral Two-Dimensional Polyline

The method
MbResultType
RegularPolygon (const MbCartPoint & ***centre***,
const MbCartPoint & ***point***,
size_t *vertexCount*,
bool *describe*,
MbCurve *& ***result***)
constructs a closed two-dimensional polyline that is a regular polyline inscribed in a designated circle or circumscribed around such a circle.

Input parameters of the method are as follows:

- ***centre*** is a center of the circle circumscribed around an equilateral polygon or or inscribed in it,
- ***point*** is a point on the circle,
- *vertexCount* is a number of vertexes in a polygon,
- *describe* is a flag that indicates whether the circle is inscribed or circumscribed.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

centre and ***point*** parameters define the circle, in which the polygon with *vertexCount* vertexes will be inscribed (if *describe*=false) or around which the polygon with *vertexCount* vertexes will be circumscribed (if *describe*=true). The center of the circle will be located in ***centre*** point, and the circle will go through ***point***. vertexCount is a parameter that defines the number of vertexes in a regular polygon. The constructed curve will be a closed polyline, polylines are described in Item O.3.5. MbPolyline Two-Dimensional Polyline. Fig. M.3.5.1 shows a regular polygon inscribed into a circle, and Fig. M.3.5.2 shows a regular polygon circumscribed around a circle.

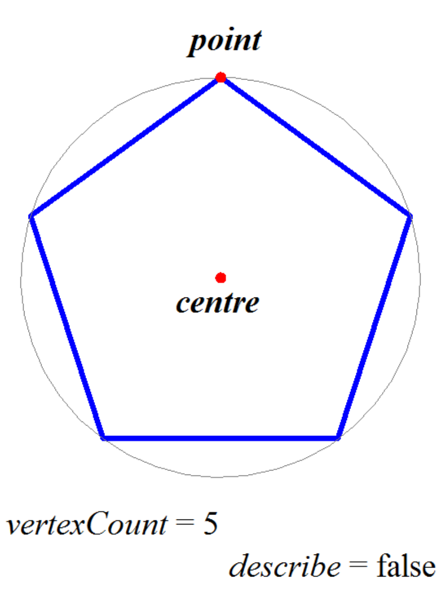


Fig. M.3.5.1.

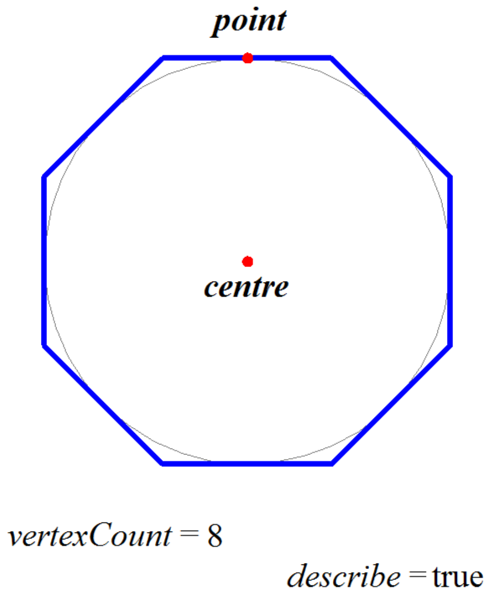


Fig. M.3.5.1.

If *vertexCount*<=1, then this method constructs a circle with a center in ***centre*** point passing through ***point***. If *vertexCount*=2, then this method constructs a rectangle with opposite sides parallel to the axes of global coordinates and opposite vertexes located in ***centre*** and ***point*** (see Fig. M.3.5.3).

Fig. M.3.5.3.

M.3.6. Constructing Two-Dimensional Cosine Wave

The method
MbResultType
Cosinusoid (const MbCartPoint & ***point***0,
const MbCartPoint & ***point***1,
const MbCartPoint & ***point***2,
double *phase*,
double *waveLength*,

[MbCurve](#) *& ***result***)
constructs a cosine wave (two-dimensional harmonic curve).

Input parameters of the method are as follows:

- ***point*0** is an origin of local coordinate system,
- ***point*1** is a point located on X axis of the local coordinate system,
- ***point*2** is a point defining Y axis of the local coordinate system,
- *phase* is phase shift of the harmonic curve,
- *waveLength* is a wavelength of the harmonic curve.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

point*0**, ***point*1** and ***point*2** points define the local coordinate system of the cosine wave. ***point*0** will be the origin of the local coordinate system of the cosine wave, ***axisX of the local coordinate system will be directed from ***point*0** to ***point*1**, ***axisY*** of the local coordinate system will be orthogonal to ***axisX*** and directed from ***point*0** to ***point*2**. ***point*0**, ***point*1** and ***point*2** points should not lie along a single stright line. Two-dimensional cosine waves are described in Item. The amplitude of the cosine wave equals to the length of ***point*2** projection to ***axisY***. *waveLength* is a wavelength of the cosine. The number of cosine waves is defined by the distance between ***point*0** and ***point*1**. *phase* defines the phase shift of the cosine wave, see Fig. M.3.6.1.

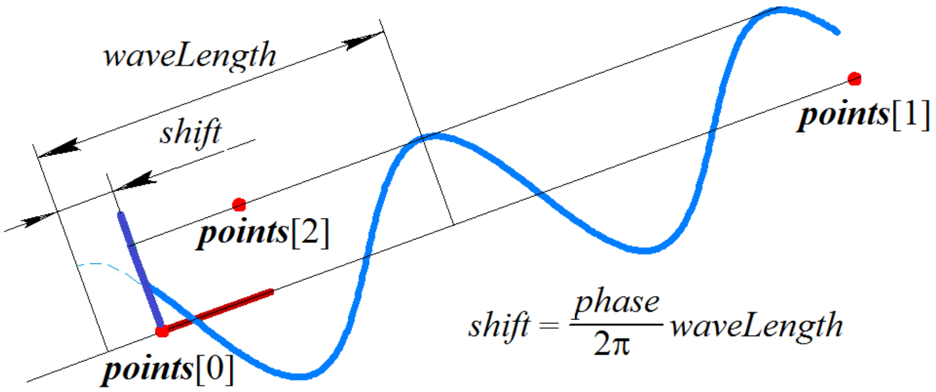


Fig. M.3.6.1.

M.3.7. Constructing Two-Dimensional Compound Curve

The method
MbResultType
CreateContour ([MbCurve](#) & ***curve***,
[MbContour](#) *& ***result***)
constructs a compound curve based on an original curve.

curve original curve is an input parameter of the method.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

The method constructs ***result*** compound curve based on the original ***curve***. If the original ***curve*** is also compound, then ***result*** curve will include the components of the original curve. Two-dimensional compound curves are described in Item [O.3.17. MbContour Two-Dimensional Contour](#).

The method
MbResultType

AddCurveToContour (MbCurve & ***curve***,
MbContour & ***contour***,
bool *toEnd*)
modifies a compund curve by adding another curve.

Input parameters of the method are as follows:

- ***curve*** is an added curve,
- ***contour*** is a modified compound curve,
- *toEnd* is a flag indicating the point where the curve is added.

The output parameter of the method is ***contour***, the modified curve.

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

The method is declared in action_curve.h file.

The method modifies compound curve ***contour*** by adding ***curve*** at the beginning or at the end of the compound curve. If *toEnd*=true, then ***curve*** will be added to the end of ***contour*** compound line, if *toEnd*=false, then ***curve*** will be added to the beginning of ***contour*** compound line. Joint points of modified curve and added curve should be the same, see Fig. M.3.7.1.

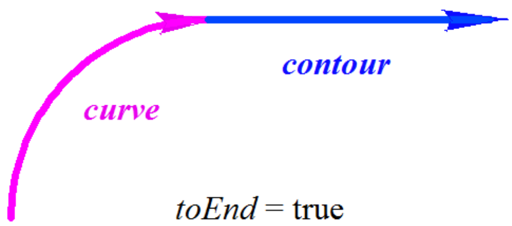


Fig. M.3.7.1.

If added ***curve*** is also compound, then ***contour*** curve will include components of the added curve.

M.3.8. Constructing Surface and Plane Intersection Curves

The method
void
SurfaceSection (const MbSurface & **surface**,
const MbPlacement3D & **place**,
RPArray<MbCurve & ***result***)
constructs surface and plane intersection curves.

Input parameters of the method are as follows:

- **surface** is a surface,
- **place** is a local coordinate system of the surface.

The output parameter of the method is ***result***, a set of constructed curves.

The method returns no value.

The method is declared in action_curve.h file.

The method constructs an intersection of **surface** with XY plane of **place** local coordinate system. ***result*** curves are constructed in XY plane of **place** local coordinate system. Fig. M.3.8.1 shows an example of constructing two-dimensional curves formed by an intersection of a torus surface and XY plane of the local coordinate system.

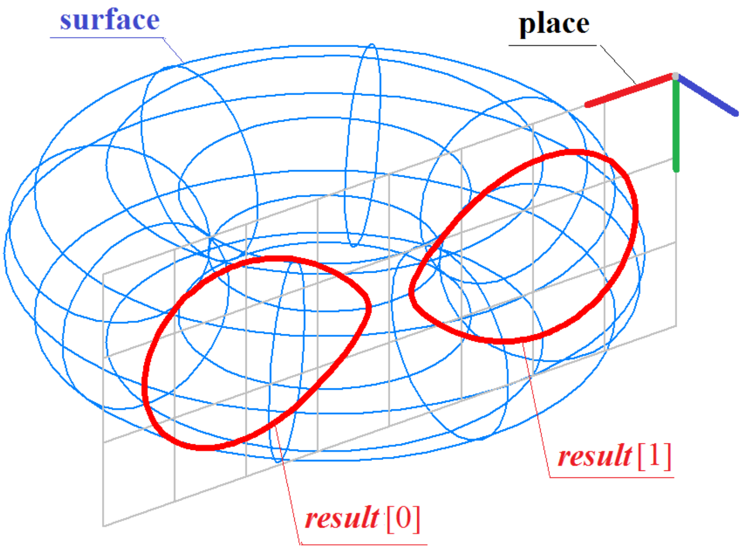


Fig. M.3.8.1.

M.3.9. Constructing Two-Dimensional Face Edge Curve

The method
MbResultType
FaceBoundSegment (const MbFace & **face**,
size_t loopIndex,
size_t edgeIndex,
const MbSurface & **surface**,
VERSION version,
MbCurve *& ***result***)
projects a face edge onto a surface.

Input parameters of the method are as follows:

- **face** is the face itself,
- loopIndex is a cycle index in the face, where the projected edge is located,
- edgeIndex is an edge index in a cycle,
- **surface** is a projection surface,
- version is a construction version.

The output parameter of the method is a ***result*** constructed body.

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

The method is declared in action_curve.h file.

The method projects **face** edge on **surface**. Ace edge is defined by loopIndex and edge index in a cycle. ***result*** curve is constructed in parameter space of **surface**. Fig. M.3.9.1 shows an example of projecting a face edge on a selected surface.

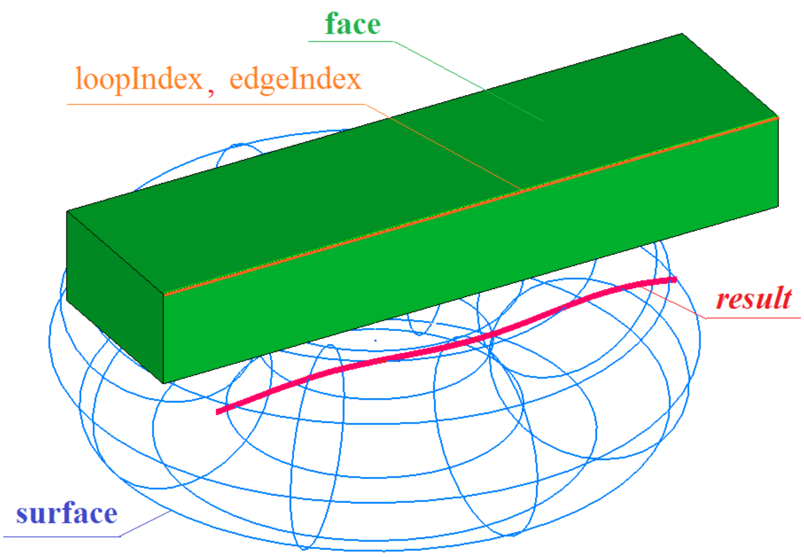


Fig. M.3.9.1.

M.3.10. Projecting a Curve on a Surface

The method
MbResultType
SurfaceBoundContour (const MbSurface & **surface**,
const MbCurve3D& **curve**,
VERSION version,
MbContour *& ***result***)
constructs a two-dimensional curve in parameter space of a surface for 3D curve lying on the surface.

Input parameters of the method are as follows:

- **surface** is the surface,
- **curve** is the 3D curve,
- version is a construction version.

The output parameter of the method is ***result***, the constructed compound curve.

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

The method is declared in action_curve.h file.

The method projects the 3D **curve** on **surface**. The 3D curve should lie on **surface**. ***result*** curve is constructed in parameter space of **surface**. Fig. M.3.10.1 shows an example of projecting a 3D curve on a given surface.

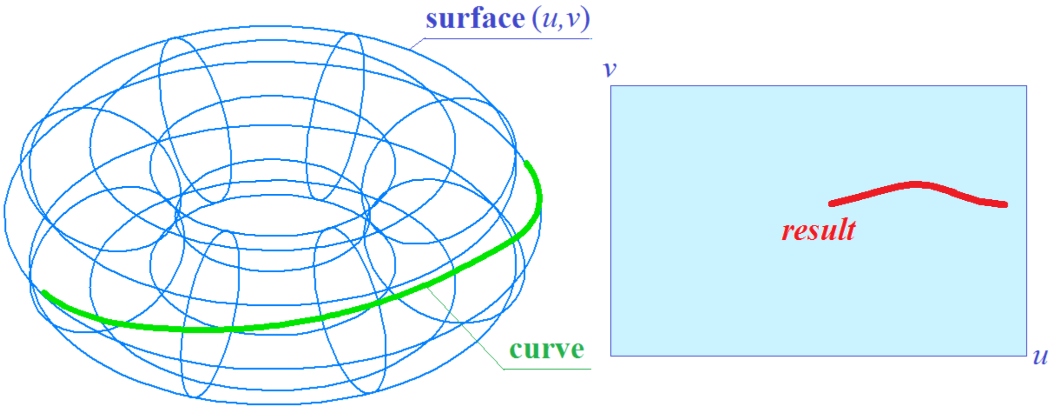


Fig. M.3.10.1.

M.4. CURVE CONSTRUCTION METHODS

Curves are used as building blocks to construct surfaces. Curves are used to join surfaces with each other. Curves form a basis to construct faces used in solid body model and wireframe model. In some cases curves act as reference objects to position other elements of a geometric model. All curves are the MbCurve3D class inheritors. They are described in Chapter O.4. CURVES. The curves can be constructed by directly calling the corresponding constructors or using the methods described in this item.

M.4.1. Constructing a Line and a Segment

The method
MbResultType
Line (const MbCartPoint3D & **point**1,
const MbCartPoint3D & **point**2,
MbCurve3D *& **result**)
constructs a line based on two non-coincident points.

The method input parameters are:

- **point**1 is the first point that lies on the line,
- **point**2 is the second point that lies on the line.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

The parameter **point**1 defines the start point of the straight line corresponding to the zero parameter value. The vector beginning in the **point**1 and ending in the **point**2 defines the line direction, see Fig. M.4.1.1. Line derivative has a unit length.

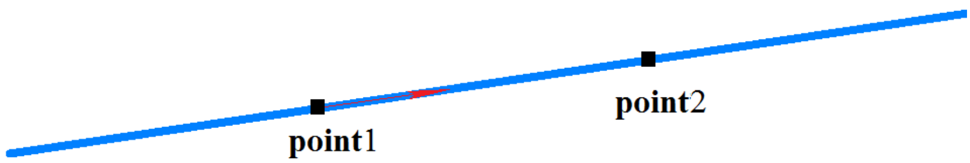


Fig. M.4.1.1.

Lines are described in Item O.4.2. MbLine3D Straight Line.

The method
MbResultType
Segment (const MbCartPoint3D & **point**1,
const MbCartPoint3D & **point**2,
MbCurve3D *& **result**)
constructs a line segment based on two non-coincident points.

The method input parameters are:

- **point**1 is the segment starting point;
- **point**2 is the segment end point.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

The parameter **point**1 defines the starting point of the line segment corresponding to the zero curve parameter. The parameter **point**2 defines the end point of the line segment corresponding to the curve parameter equal to one, see Fig. M.4.1.2.

Fig. M.4.1.2.

Line segments are described in Item O.4.3. MbLineSegment3D Straight Line Segment.

M.4.2. Constructing a Circle, an Ellipse And Their Arcs

The method
MbResultType
Arc (const MbCartPoint3D & **centre**,
const SArray<MbCartPoint3D> & **points**,
bool *closed*,
double *angle*,
double & *a*,
double & *b*,
MbCurve3D *& **result**)
constructs an elliptic arc. In a special case, the method constructs a circular arc.

The method input parameters are:

- **centre** is the ellipse center,
- **points** is a set of points that can be empty,
- *closed* is a flag that defines whether the curve is cyclically closed,
- *angle* is an angle that defines the size of the elliptic arc,
- *a* is the first ellipse semi-axis length (calculated if the **points** set is non-empty),
- *b* is the second semi-axis length (calculated if the **points** set is non-empty).

The method output parameters are ellipse semi-axes and the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

The curve can be constructed based on either specified points or scalar parameters. The parameter **centre** defines ellipse central point. The **points** set can be empty, but in this case the *a* and *b* ellipse semi-axis lengths should be non-empty.

If the **points** set contains two elements, then the **centre**, the **points**[0], and the **points**[1] points determine the plane, in which the **axisX** and the **axisY** of the local ellipse coordinate system are located: the **axisX** of the local ellipse coordinate system is directed from the center to the **points**[0], the **axisY** of the local ellipse coordinate system is orthogonal to the **axisX** and it is directed from the center to the **points**[1]. The distance between the **centre** and **points**[0] defines the length of the first ellipse semi-axis *a*, and the distance between the **centre** point and the projection of the **points**[1] to the **axisY** defines the length of the second ellipse semi-axis *b*, see Fig. M.4.2.1.

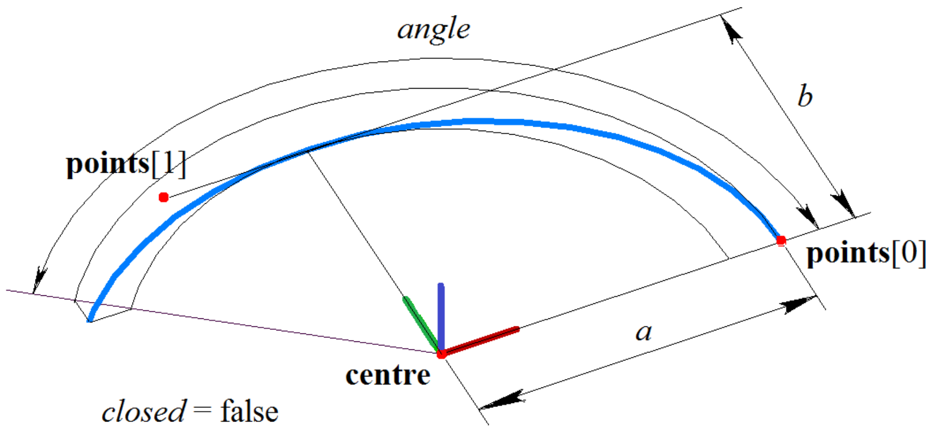


Fig. M.4.2.1.

Elliptical arcs are described in Item O.4.4. MbArc3D Elliptical Arc.

If the **points** set is empty, then the lengths of ellipse semi-axes (*a* and *b*) are used as input parameters, and the **axisX**, **axisY**, and **axisZ** of the local ellipse coordinate system coincide with the global ellipse coordinate axes.

The parameter *closed* defines whether the curve is cyclically closed. If *closed*=false, then the parameter *angle*, which determines the elliptical arc opening angle in parametric units, should be non-zero.

In a special case when $a = b$, this method constructs a circle (if *closed*=true or *angle*=0) or a circular arc (if *closed*=false, and *angle* >0, and *angle* < 2π).

M.4.3. Constructing Curves Based on Control Points

The method

MbResultType

SplineCurve (const SArray<MbCartPoint3D> & **points**,
bool *closed,*
MbeSpaceType *curveType*,
MbCurve3D *& **result**)

constructs a curve of the required type based on the specified set of control points.

The method input parameters are:

- **points** is a set of control points,
- *closed* is a flag that defines whether the curve is cyclically closed,
- *curveType* is a curve type.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

The parameter **points** contains the curve control points. The parameter *closed* defines whether the constructed curve is cyclically closed. The parameter *curveType* sets the curve type determining the curve shape. Various numbers of control points is required to construct curves of different types. The Table M.4.3.1 shows the number of control points from the given **points** set required to create the *curveType* type curve.

Table M.4.3.1.

curveType	**Curve type**	**Number of control points**
pt_LineSegment3D	line segment	2 points
pt_Arc3D	Circular arc	3 points
pt_Polyline3D	Polygonal line	2 or more points
pt_Nurbs3D	NURBS curve	2 or more points
pt_Hermit3D	Hermite spline	2 or more points
pt_Bezier3D	Bezier curve	2 or more points
pt_CubicSpline3D	Cubic Spline	2 or more points

A line segment starting in **points**[0] and ending in **points**[1] will be constructed if *curveType3D*=st_LineSegment. Line segments are described in Item O.4.3. MbLineSegment3D Straight Line Segment.

A circular arc starting in **points**[0], passing through **points**[1] and ending in **points**[2] will be constructed if *curveType*=pt_Arc and *closed*=false, see Fig. M.4.3.1. If *closed*=true, then a circle containing the **points**[0], the **points**[1], and the **points**[2], will be constructed. Circular and elliptical arcs are described in Item O.4.4. MbArc3D Elliptical Arc.

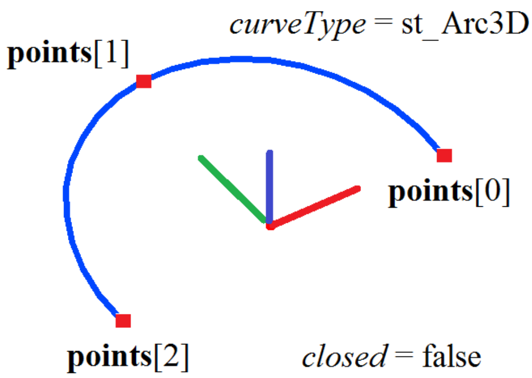


Fig. M.4.3.1.

If *curveType*=st_Polyline3D, then a polyline containing **points**[0], **points**[1],..., **points**[*n*] will be constructed, see Fig. M.4.3.2. If *closed*=true, then a cyclically closed polyline containing the segment between the **points**[0] and the **points**[*n*] will be constructed. Polylines are described in Item O.4.5. MbPolyline3D Polyline.

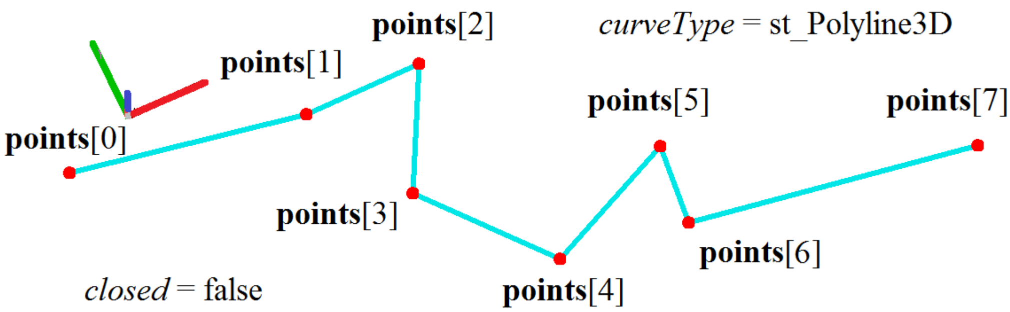


Fig. M.4.3.2.

If *curveType*=st_Nurbs3D, then a fourth-order Non-Uniform Rational *B*-Spline (NURBS) will be constructed, see Fig. M.4.3.3. Spline control points will be determined based on the condition that the spline contains **points**[0], **points**[1],..., **points**[*n*]. If *closed*=true, then a cyclically closed spline will be constructed. NURBS curves are described in Item O.4.6. MbNurbs3D NURBS-Curve.

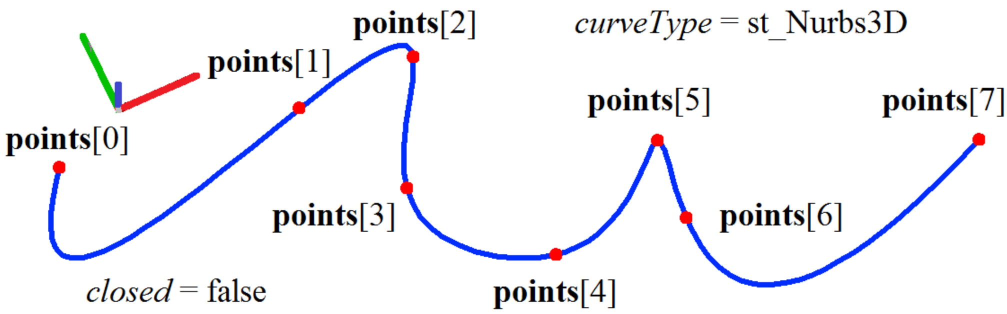


Fig. M.4.3.3.

If *curveType*=st_Hermit3D, then a compound curve containing smoothly joined third-order Hermit splines will be constructed. Each third-order Hermit spline will join the adjacent **points**[*i*–1] and **points**[*i*], see Fig. M.4.3.4. If *closed*=true, then a cyclically closed curve containing a Hermit spline between the **points**[0] and the **points**[*n*] will be constructed. Composite third-order Hermit splines are described in Item O.4.7. MbHermit3D Hermite Curve.

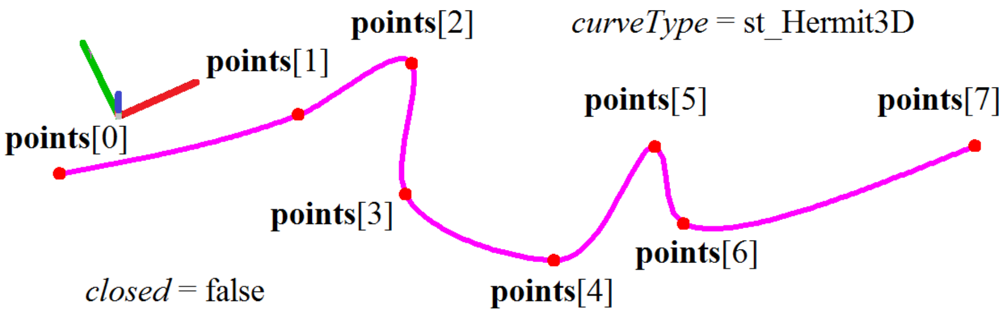


Fig. M.4.3.4.

If *curveType*=st_Bezier3D, then a compound curve containing smoothly joined third-order Bezier splines will be constructed. Each third-order Bezier spline will join the adjacent **points**[*i*–1] and **points**[*i*], see Fig. M.4.3.5. If *closed*=true, then a cyclically closed curve containing a Bezier spline between the **points**[0] and the **points**[*n*] will be constructed. Compound third-order Bezier splines are described in Item .

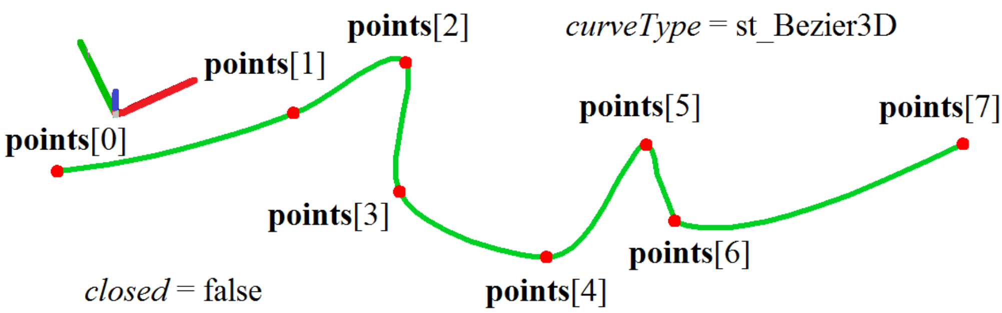


Fig. M.4.3.5.

If *curveType*=st_CubicSpline3D, then a cubic spline containing **points**[0], **points**[1],..., **points**[*n*] will be constructed, see Fig. M.4.3.6. If *closed*=true, then a cyclically closed curve containing the segment between the **points**[0] and the **points**[*n*] will be constructed. Cubic splines are described in Item O.4.9. MbCubicSpline3D Cubic Spline.

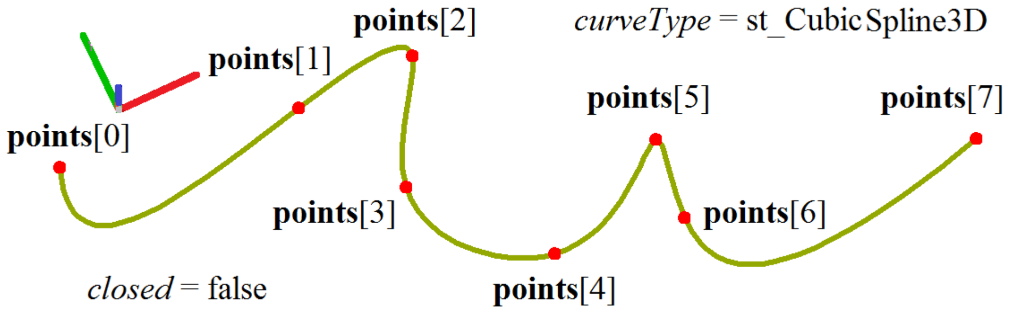


Fig. M.4.3.6.

For the purpose of comparison, Fig. M.4.3.7 shows a NURBS curve, a compound Hermite spline, a compound Bezier spline, and a cubic spline constructed based on the same control points (**points**[0], **points**[1], ..., **points**[*n*]) and having the same order.

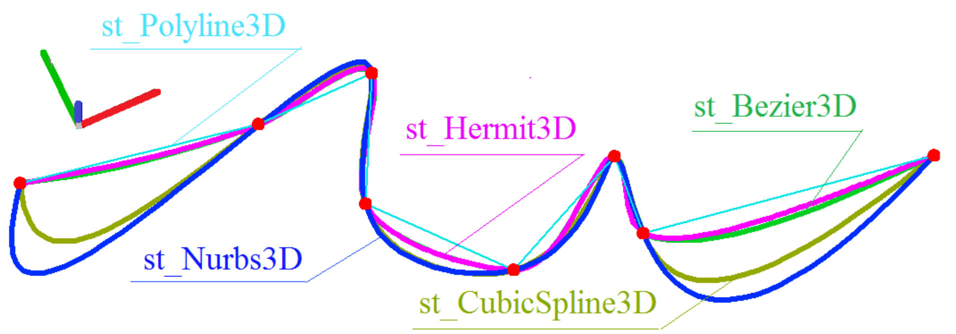


Fig. M.4.3.7.

You can see that the curves have various shapes.

M.4.4. NURBS Curve Construction

The method
MbResultType
NurbsCurve (const SArray<MbCartPoint3D> & **points**,
const SArray<double> & *weights*,
size_t degree,
const SArray<double> & *knots*,
bool *closed*,
MbCurve3D *& **result**)
constructs a NURBS curve based on a given set of control points.

The method input parameters are:

- **points** is a set of control points,
- *weights* is a set of control point weights,
- degree is a curve (*B*-spline) order,
- *knots* is a set of parametric knots (knot vector),
- *closed* is a flag that defines whether the curve is cyclically closed.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

This method constructs a NURBS curve (a curve based on *B*-splines), NURBS curves are described in

Item O.4.6. MbNurbs3D NURBS-Curve. The *weights* set should match the control point set (points). The curve order named "degree" should not exceed the number of control points. The parameter *closed* defines whether the constructed curve is cyclically closed. The *knots* vector is a non-declining sequence of real numbers, which determines the curve parameter domain of definition and the curve shape. If *closed*=false, then the knot vector should contain the number of elements equal to the sum of the number of control points and the curve order. If you want the NURBS curve to pass through the terminal control points, then the first degree elements of *knots* vector should be equal to each other, and the last degree elements of the *knots* vector should also be equal to each other. If *closed*=true, then the knot vector should contain the number of elements equal to the number of control points plus doubled curve order and minus one. Fig. M.4.4.1 shows a closed fourth-order NURBS curve with equal distances between knots.

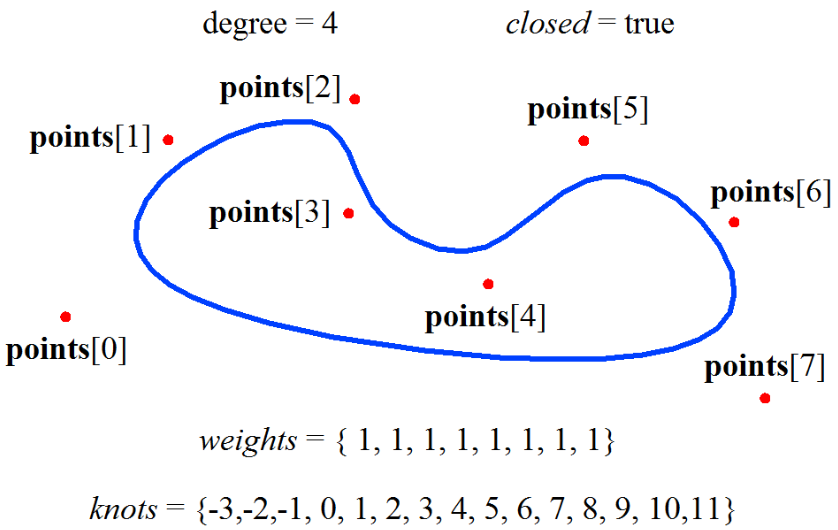


Fig. M.4.4.1.

Fig. M.4.4.2 shows two non-closed fourth-order NURBS curves. The first curve has equal distances between knots and coincides with the curve part shown in Fig. M.4.4.1. The second curve has equal first degree knot vector elements and equal last degree knot vector elements.

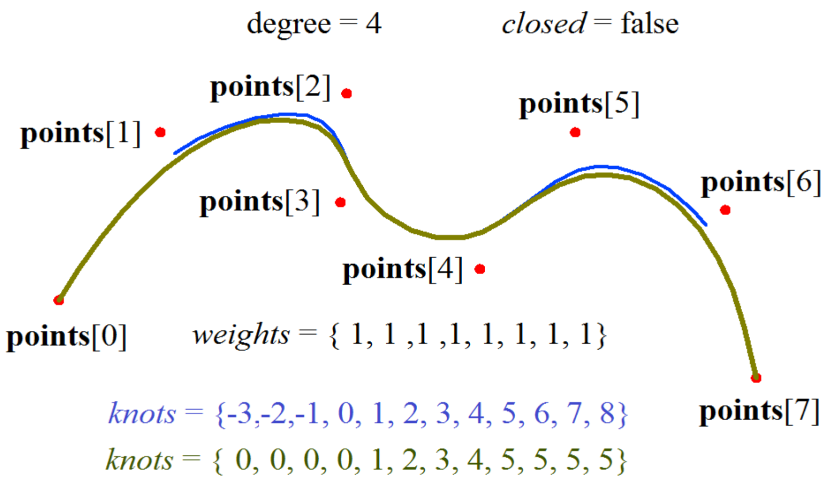


Fig. M.4.4.2.

Fig. M.4.4.3 shows three non-closed fourth-order NURBS curves with various weights (*weights*[2] and *weights*[4]) of the control points (**points**[2] и **points**[4]).

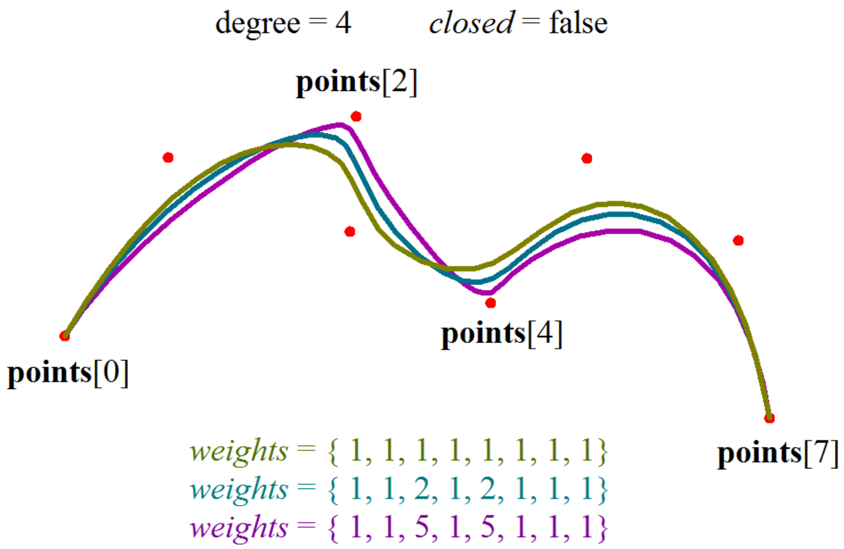


Fig. M.4.4.3.

Fig. M.4.4.4 shows three non-closed NURBS curves of various order constructed based on the same control points.

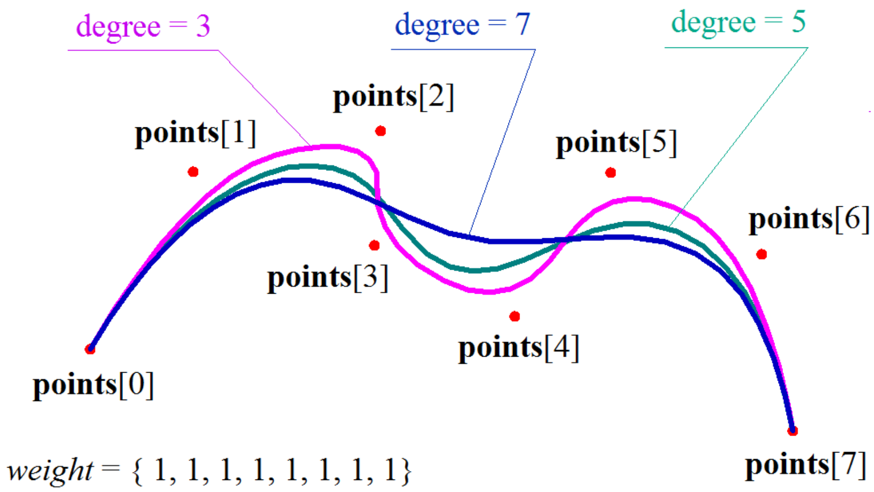


Fig. M.4.4.4.

Fig. M.4.4.5 shows a non-closed third-order NURBS curve, the shape of which coincides with a circular arc. The distance between the **points**[0] and the **points**[1] control points is equal to the distance between the **points**[1] and the **points**[2] control points, and the *weights*[1] of the middle curve control point is equal to the weight of the terminal control points multiplied by the arc half-angle cosine.

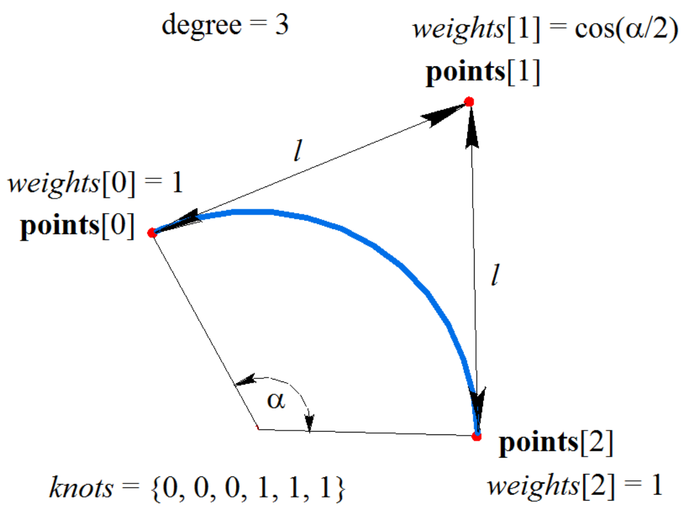


Fig. M.4.4.5.

The method
MbResultType
NurbsCopy (const MbCurve3D & **curve**,
MbCurve3D *& **result**)
constructs a NURBS copy of the given curve.

The original **curve** is the input parameter of the method.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

This method constructs a NURBS curve with a shape that copies the original curve shape. NURBS copies completely coincide with the original curves for the majority of curve types. If it is impossible to accurately reproduce the original curve shape, then the NURBS copy approximates the original curve with an error less than 0.0001.

M.4.5. Convex Equilateral Polyline Construction

The method
MbResultType
RegularPolygon (const MbCartPoint3D & **centre**,
const MbCartPoint3D & **point**,
const MbVector3D & **axisZ**,
size_t *vertexCount*,
bool *describe*,
MbCurve3D *& **result**)
constructs a closed polyline that is a regular polygon inscribed in the given circle or that circumscribes the circle.

The method input parameters are:

- **centre** is the center of the circle that is circumscribed around an equilateral polygon or is inscribed in it,
- **point** is a point on the circle,
- **axisZ** is a vector perpendicular to the circle plane,
- *vertexCount* is the number of polygon vertexes,
- *describe* is a flag that indicates whether the circle is inscribed or circumscribed.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

The **centre**, **point** and **axisZ** parameters define the circle, in which the polygon with *vertexCount* vertexes will be inscribed (if *describe*=false) or around which the polygon with *vertexCount* vertexes will be circumscribed (if *describe*=true). The circle center will be placed in the **centre** point, the circle will contain the **point**, and the circle axis will be parallel to the **axisZ** vector. *vertexCount* is the parameter that defines the number of vertexes in a regular polygon. The constructed curve will be a closed polyline described in Item O.4.5. MbPolyline3D Polyline. Fig. M.4.5.1 shows a regular polygon inscribed in a circle, and Fig. M.4.5.2 shows a regular polygon that circumscribes a circle.

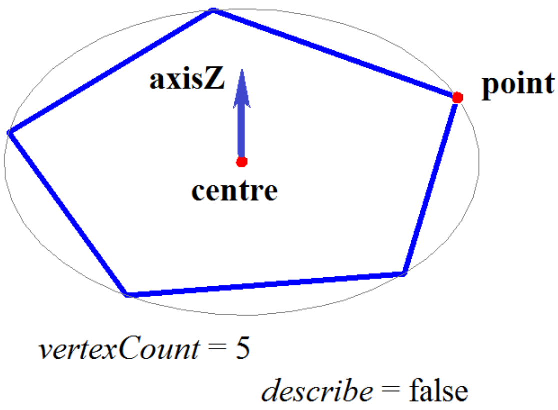


Fig. M.4.5.1.

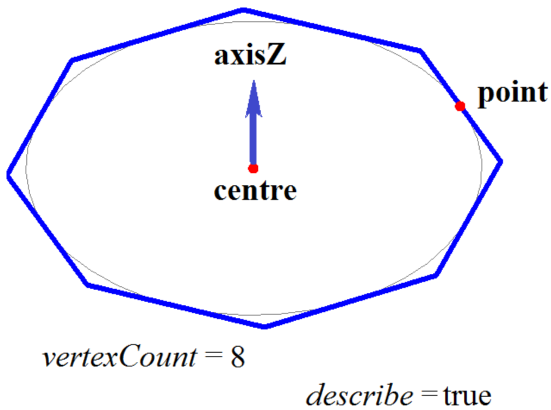


Fig. M.4.5.2.

If *vertexCount*<=1, then this method constructs a circle with a center in **centre** point passing through the **point**. If *vertexCount*=2, then this method constructs a polyline with one section, starting in **centre** and ending in **point**.

M.4.6. Spiral Construction

The method

MbResultType

SpiralCurve (const MbPlacement3D & **place**,
double *radius*,
double *step*,

MbCurve & ***lawCurve***,
bool *spiralAxis*,
MbCurve3D *& **result**)
constructs a spiral with a variable radius or a spiral with a curved axis.
The method input parameters are:

- **place** is the local coordinate system of the spiral,
- *radius* is the spiral radius,
- *step* is the spiral step,
- ***lawCurve*** is a shape-forming two-dimensional curve,
- *spiralAxis* is a shape forming mode.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

If *spiralAxis*=false, then the ***lawCurve*** defines how the spiral radius changes. In this particular case the spiral axis coincides with the **axisZ** of the local coordinate system (**place**), the ***lawCurve*** will be placed into the ZX plane of the local spiral coordinate system and it gives the law how the spiral radius changes. The first coordinate (x) of each two-dimensional point of the ***lawCurve*** will be plotted along the **axisZ** of the local spiral coordinate system, the second coordinate (y) of each two-dimensional point of the ***lawCurve*** will be plotted along the **axisX** of the local spiral coordinate system, see Fig. M.4.6.1.

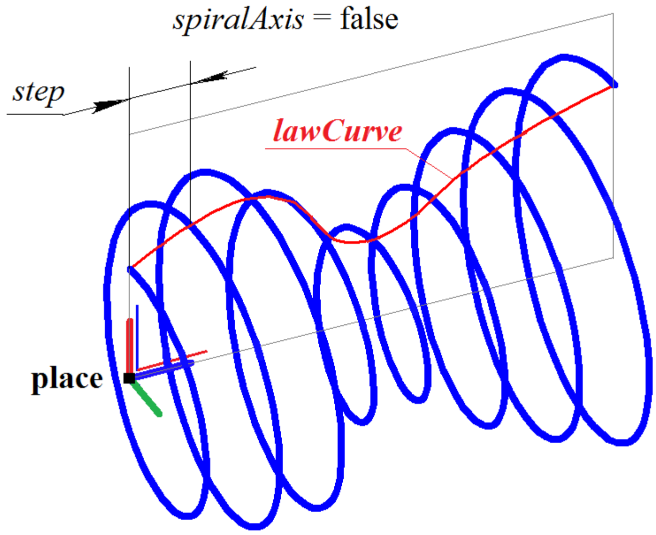


Fig. M.4.6.1.

The second coordinate (y) of the two-dimensional ***lawCurve*** will define the spiral radius as a function of the first coordinate (x) of the curve. The second coordinate (y) of each point of the two-dimensional ***lawCurve*** should be positive as the ***lawCurve*** should not cross the **axisZ**. Spirals with a variable radius are described in Item O.4.15. MbCurveSpiral Variable Radius Spiral. The *step* parameter gives the spiral step, and the *radius* parameter is not used.

If *spiralAxis*=true, then the ***lawCurve*** defines the spiral axis. In this case, the ***lawCurve*** will lie in the ZX plane of the local coordinate system and it will circumscribe about the spiral axis. The first coordinate (x) of each two-dimensional point of the ***lawCurve*** will be plotted along the **axisZ** of the local spiral coordinate system, the second coordinate (y) of each two-dimensional point of the ***lawCurve*** will be plotted along the **axisX** of the local spiral coordinate system, see Fig. M.4.6.2.

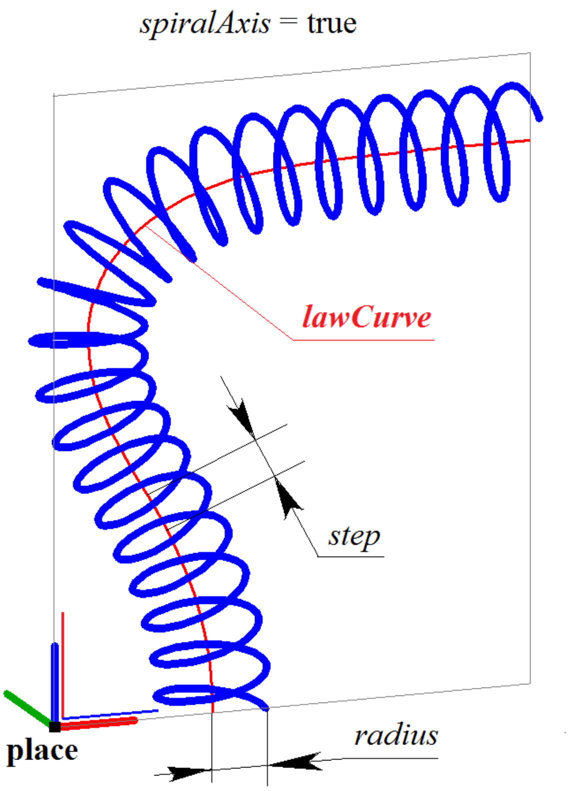


Fig. M.4.6.2.

To avoid spiral self-intersections, the radius of curvature of two-dimensional ***lawCurve*** should exceed the spiral *radius* in each point. Spirals with curvilinear axes are described in Item O.4.16. MbCrookedSpiral Spiral with Curved Planar Axis. The *step* parameter defines the spiral step along its axis.

The method
MbResultType
SpiralCurve (const MbCartPoint3D & **point**0,
const MbCartPoint3D & **point**1,
const MbCartPoint3D & **point**2,
double *radius*,
double *step*,
double *angle*,
MbCurve * ***lawCurve***,
bool *spiralAxis*,
MbCurve3D *& **result**)
constructs a conical spiral, or a spiral with variable radius, or a spiral with a curvilinear axis.

The method input parameters are:

- **point**0 is the origin of the spiral local coordinate system,
- **point**1 is a point located on the Z axis of the spiral local coordinate system,
- **point**2 is a point defining the X axis of the spiral local coordinate system,
- *radius* is the spiral radius,
- *step* is the spiral step,
- *angle* is the angle of the conical spiral,
- ***lawCurve*** is a shape-forming two-dimensional curve, this parameter can be zero,
- *spiralAxis* is a shape forming mode.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

Point0, **point**1 and **point**2 define the local coordinate system of the spiral. The **point**0 will be the origin of the spiral local coordinate system, **axisZ** of the local coordinate system will be directed from **point**0 to **point**1, **axisX** of the local coordinate system will be orthogonal to **axisZ** and directed from **point**0 to the **point**2. The **point**0, the **point**1 and the **point**2 should not coincide or lie along a line.

If ***lawCurve***=0, then the method will construct a conical spiral with **axisZ** of the local coordinate system, *step* as the spiral step, and *angle* as the cone angle. The radius in the beginning of the spiral will be equal to *radius*, see Fig. M.4.6.3.

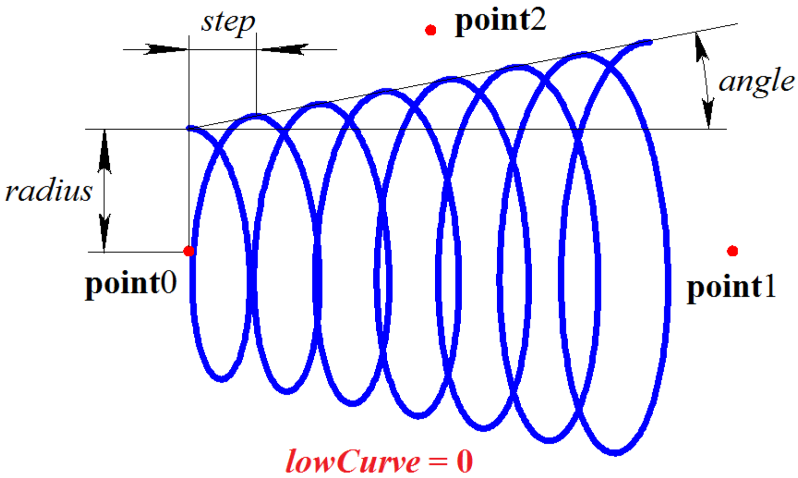


Fig. M.4.6.3.

If *angle*=0, then a cylindrical spiral will be constructed. *spiralAxis* will not be used.

If ***lawCurve*** is not equal to zero and *spiralAxis*=false, then the method will use the procedure described above to construct a spiral with a variable radius in the local coordinate system defined by the **point**0, **point**1 and **point**2, shown in Fig. M.4.6.1. The ***lawCurve*** parameter defines how the spiral radius changes.

If ***lawCurve*** is not equal to zero and *spiralAxis*=true, then the method will use the procedure described above to construct a spiral with a curved axis in the local coordinate system defined by the **point**0, **point**1 and **point**2, as shown in Fig. M.4.6.2. ***lawCurve*** parameter defines the spiral axis.

M.4.7. Compound Curve Construction

The method
MbResultType
CreateContour (MbCurve3D & **curve**,
MbContour3D *& **result**)
constructs a compound curve based on one original curve.

The original **curve** is the input parameter of the method.

The output parameter of the method is the **result** constructed curve.

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

The method is declared in the action_curve3d.h file.

The method constructs the **result** compound curve based on the original **curve**. If the original **curve** is also compound, then the **result** curve will include the components of the original curve. Compound curves are described in Item O.4.18. MbContour3D Contour.

The method
MbResultType
AddCurveToContour (MbCurve3D & **curve**,
MbCurve3D & **contour**,

bool *toEnd*)
modifies the compound curve by adding other curve.

The method input parameters are:

- **curve** is an added curve,
- **contour** is a modified compound curve,
- *toEnd* is a flag indicating the place where the curve is added.

The output parameter of the method is the **contour**, the modified curve.

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

The method is declared in the action_curve.h file.

The method modifies the compound curve **contour** by adding the **curve** at the beginning or at the end of the compound curve. If toEnd=true, then the **curve** will be added to the end of **contour** compound curve, if toEnd=false, then the **curve** will be added to the beginning of **contour** compound curve. Modified and added curve joint points should coincide, see Fig. M.4.7.1.

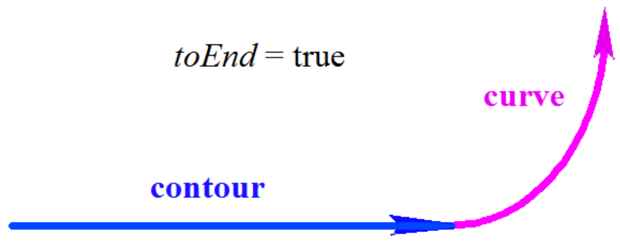


Fig. M.4.7.1.

If the added **curve** is also compound, then the **contour** curve will include the components of the added curve.

The method
MbResultType
CreateContours (RPArray<MbCurve3D> & **curves**,
double *epsilon*,
RPArray<MbContour3D> & **result**,
bool *onlySmoothConnected* = false)
constructs compound curves from a set of original curves.

The method input parameters are:

- **curves** is a set of original curves,
- *epsilon* is a matching radius for joining the original curves,
- *onlySmoothConnected* is a flag that indicates using only smooth joining of original curves.

The output parameter of the method is **result**, the constructed curve set.

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

The method is declared in the action_curve3d.h file.

The method creates compound curves on the basis of original **curves** and adds them to the **result** set. If *onlySmoothConnected*=true, then all the constructed compound curves will be smooth, i.e. the curves will have similarly directed tangent lines in joining points, but the derivative by length in the joining points may change discontinuously. If any original curve in the **curves** set is compound, then the components of such original curve will be added to the resulting compound curve.

M.4.8. Wireframe Construction

The method

MbResultType
WireFrame (const MbCurve3D & **curve**,
const MbName & name,
SimpleName mainName,
MbWireFrame *& **result**)
creates a wireframe based on a curve.

The method input parameters are:

- **curve** is the original curve,
- name is a curve name,
- mainName is the main wireframe name.

The output parameter of the method is the **result** constructed wireframe.

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

The method is declared in the action_curve3d.h file.

The method creates an MbEdge, an edge that serves as **result** wireframe element, on the basis of the **curve**. MbWireFrame wireframe is described in Item O.8.3. MbWireFrame Wireframe.

The method
MbResultType
WireFrame (const RPArray<MbCurve3D> & **curves**,
const RPArray<MbName> & names,
SimpleName mainName,
MbWireFrame *& **result**)
creates a wireframe based on a curve set.

The method input parameters are:

- **curves** is a set of curves,
- names is a set of curve names,
- mainName is the main wireframe name.

The output parameter of the method is the **result** constructed wireframe.

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

The method is declared in the action_curve3d.h file.

The method creates an edge (MbEdge) on the basis of each curve of the **curve** set. The **result** wireframe is then created from these edges. The wireframe edges are joined in vertexes, each edge and vertex have their own attributes. The wireframe provides information about the edges joined in shared vertexes, the wireframe also has attributes and a construction log. MbWireFrame wireframe is described in Item O.8.3. MbWireFrame Wireframe.

M.4.9. Curve Projection onto a Surface

The method
MbResultType
CurveProjection (const MbSurface & **surface**,
const MbCurve3D & **curve**,
MbVector3D * **direction**,
bool *createExact*,
bool *truncateByBounds*,
RPArray<MbCurve3D> & **result**,
VERSION version = Math::DefaultMathVersion());
projects a curve onto the selected surface using the normal projection mehod or parallel projection method.

The method input parameters are:

- **surface** is a surface, to which the curve is projected,
- **curve** is a projected curve,

- **direction** is a projection direction vector (this value can equal zero),
- *createExact* is a constructed curve accuracy flag,
- *truncateByBounds* is a flag that indicates whether the projections are truncated by surface boundaries,
- version is a construction version.

The output parameter of the method is **result**, the constructed curve set.

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

This method is declared in the action_surface_curve.h file.

If **direction**=0, then the method creates a normal projection of the **curve** onto the **surface**, wherein the vector constructed from the projection point to the corresponding point of the projected curve is directed normally to the surface.

If the **direction** vector is non-zero, then the method creates a parallel projection of the **curve** onto the **surface**, wherein the vector constructed from the projection point to the corresponding point of the projected curve of the projected curve is parallel to the **direction** vector.

The **result** curves lie on the MbSurfaceCurve surface described in Item O.4.20. MbSurfaceCurve Curve on Surface. If *createExact*=false, then in the general case the two-dimensional curves among the curves on the **result** surface will be spline curves passing through the finite number of points that are projections of particular points of the projected curve. If *createExact*=true, then in the general case the two-dimensional curves among the curves on the **result** surface will be MbProjCurve projection curves, which accurately describe the projections of spatial curves onto the surface. In special cases, the **result** curves are an accurate projection of the **curve**. If the **direction** vector is defined, then it is considered that *createExact*=false.

If *truncateByBounds*=false, then the **result** curves will be located inside the parametric rectangle, which covers the definition domain of **surface** parameters. If the definition domain of surface parameters does not coincide with its outline rectangle, then the **result** curves may go beyond the **surface** boundaries.

If *truncateByBounds*=true, then the **result** curves completely lie within the **surface**. Fig. M.4.9.1 shows a normal curve projection onto a surface.

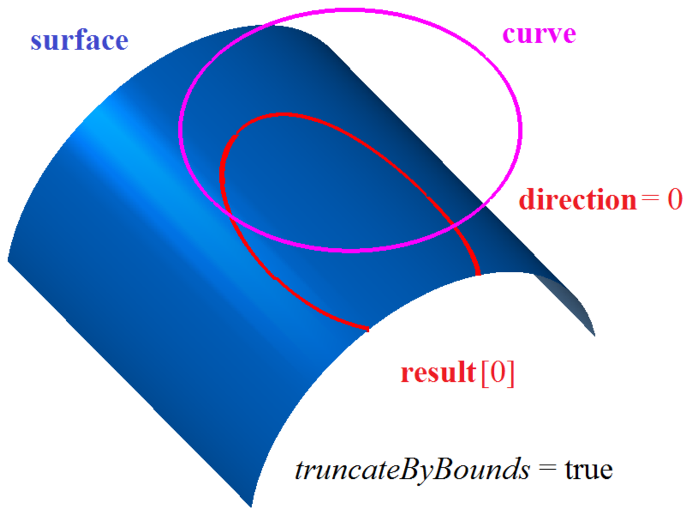


Fig. M.4.9.1.

If *truncateByBounds*=false, then the **result** curves may go beyond the **surface** boundaries. Fig. M.4.9.2 shows a parallel curve projection onto a surface.

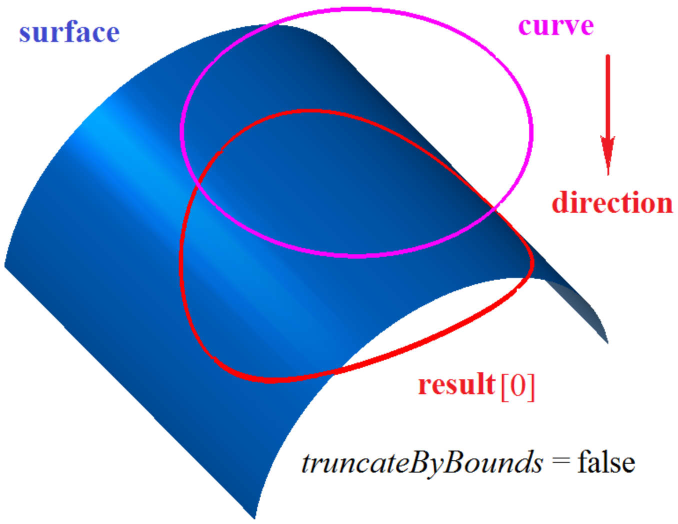


Fig. M.4.9.2.

M.4.10. Construction of Surface Intersection Curves

The method
MbResultType
IntersectionCurve (const MbSurface & **surface**1,
const MbSurface & **surface**2,
const MbSNameMaker & names,
MbWireFrame *& **result**)
constructs the curves formed by intersection of two surfaces.

The method input parameters are:

- **surface**1 is the first surface,
- **surface**2 is the second surface,
- names is a namer for the construction result elements.

The output parameter of the method is the **result** constructed wireframe.

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

This method is declared in the action_surface_curve.h file.

Two surfaces can intersect so that the intersection would form several curves. This method constructs all intersection curves and converts them to a wireframe. The wireframe curves are MbSurfaceIntersectionCurve surface intersection curves described in Item O.4.24. MbSurfaceIntersectionCurve Surface Intersection Curve. Names parameter defines the names of edges of the constructed wireframe. Fig. M.4.10.1 shows the surface intersection result.

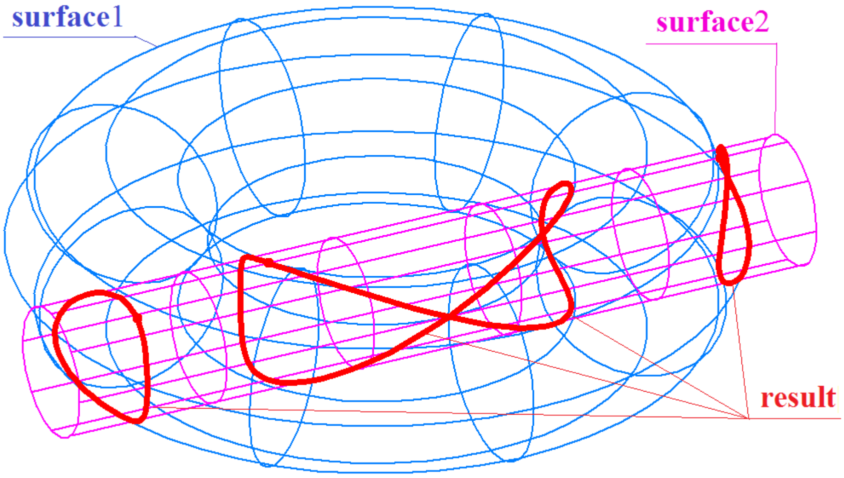


Fig. M.4.10.1.

M.4.11. Silhouette Curve Construction

The method

MbResultType
SilhouetteCurve (const MbFace & **face**,
const MbVector3D & **eye**,
bool *perspective*,
RPArray<MbCurve3D> & **result**)

constructs the silhouette curves of a face using parallel or perspective projection.

The method input parameters are:

- **face** is the face itself,
- **eye** is a view direction or a viewpoint vector,
- *perspective* is a perspective projection flag.

The output parameter of the method is **result**, the constructed curve set.

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

This method is declared in the action_surface_curve.h file.

Silhouette curves lie on the face and separate the face part visible from the viewpoint and the invisible part. There can be several silhouette curves for one face. The set of silhouette curves varies for different view directions and various viewpoints.

If *perspective*=false, then the **eye** vector determines the direction of view that is constant for all face points. If *perspective*=true, then the **eye** vector determines the viewpoint position instead of the view direction, because the view direction can vary for various face points. If *perspective*=false, then the silhouette curves are constructed for parallel face projections. If *perspective*=true, then the silhouette curves are constructed for perspective face projections.

In special cases silhouette curves may coincide with the face borders. Silhouette curves that coincide with the face borders are not constructed because they would coincide with the face edges.

Fig. M.4.11.1 shows an example of silhouette curves constructed for parallel face projection.

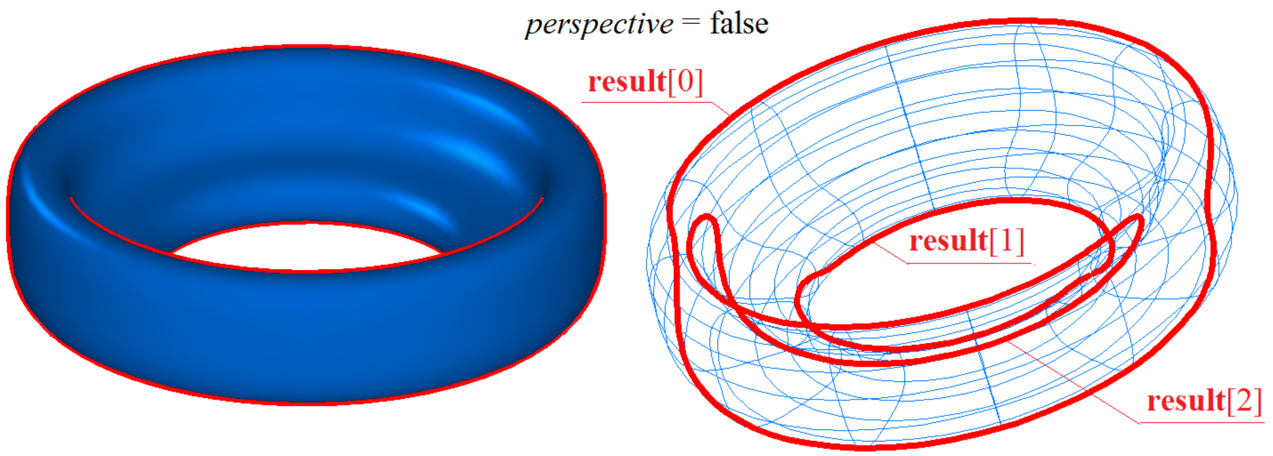


Fig. M.4.11.1.

Fig. M.4.11.2 shows an example of silhouette curves constructed using a perspective face projection with a viewpoint close to the face.

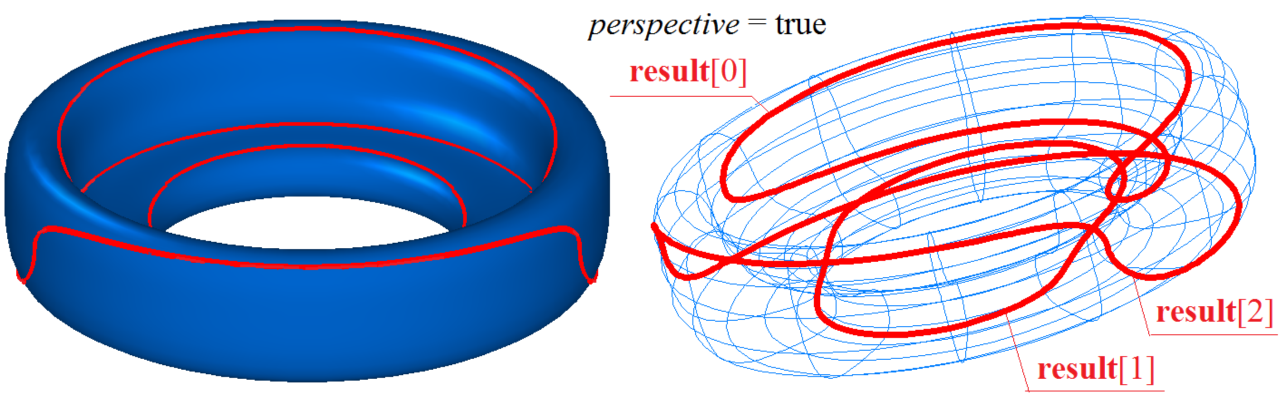


Fig. M.4.11.2.

Silhouette curves are described in Item O.4.21. MbSilhouetteCurve Silhouette Curve.

The method
MbResultType
SilhouetteCurve (const MbFace & **face**,
const MbAxis3D & **axis**,
RPArray<MbCurve3D> & **result**)
constructs silhouette curves of a face rotating along the selected axis.

The method input parameters are:

- **face** is the face itself,
- **axis** is a rotation axis.

The output parameter of the method is **result**, the constructed curve set.

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

This method is declared in the action_surface_curve.h file.

It is required to construct a set of silhouette face rotation curves to construct a turning section. When a face rotates along an axis, it may be considered that the view direction for the face points is orthogonal to the rotation axis and the segment joining the face point and the projection of this point onto the axis.

In special cases, the silhouette curves may coincide with the edge faces. Silhouette curves that coincide with the face borders are not constructed because they would coincide with the face edges.

Fig. M.4.11.3 shows an example of silhouette curves formed by rotating a face around an axis.

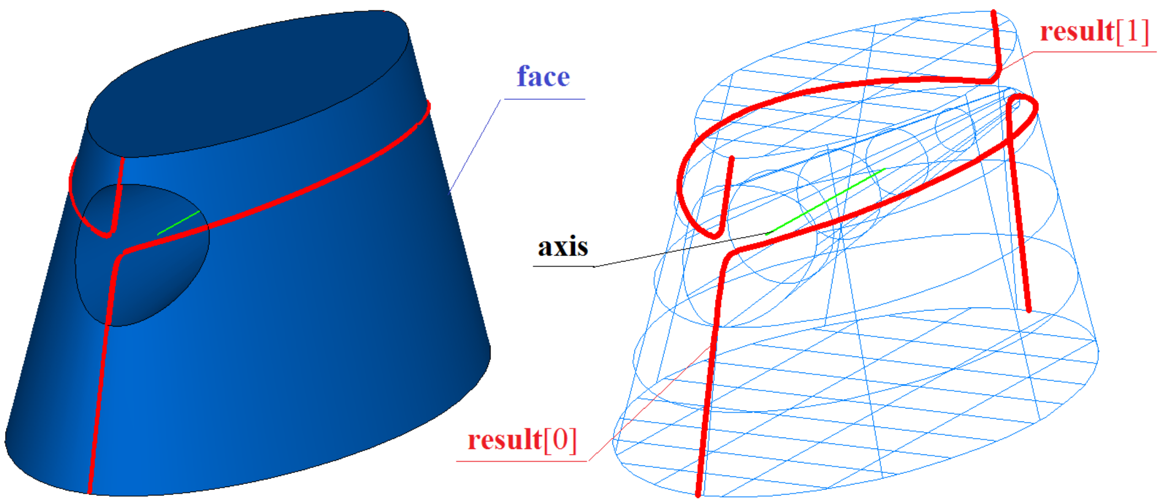


Fig. M.4.11.3.

M.4.12. Constructing a Curve Mating Curve

The method
MbResultType
FilletCurve (const MbCurve3D & **curve**1,
double & *t*1,
double & *w*1,
const MbCurve3D & **curve**2,
double & *t*2,
double & *w*2,
double & *radius*,
bool *sense*,
bool & *unchanged*,
const MbeConnectingType *type*,
const MbSNameMaker & names,
MbElementarySurface *& **surface**,
MbWireFrame*& **result**)
constructs a curve that smoothly joins two curves and lies on a cylindrical surface tangential to the mating curves.

The method input parameters are:

- **curve**1 is the first curve,
- *t*1 is the first curve mating point parameter,
- *w*1 is the first curve end point parameter,
- **curve**2 is the second curve,
- *t*2 is the second curve mating parameter,
- *w*2 is the second curve end point parameter,
- *radius* is a mating radius,
- *sense* is a direction of the constructed mating,
- *type* is a mating type,
- names is a constructed mating namer.

The output parameters of the method are as follows: **result** is the constructed wireframe, **surface** is a cylindrical surface tangential to the curves, *t*1 and *t*2 are the curve mating point parameters, *w*1 and *w*2 are the end point parameters of the mated curves, *radius* is the mating radius, *unchanged* is a flag indicating a special case of the constructed mating.

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

This method is declared in the action_surface_curve.h file.

The method provides five ways of mating two curves. If *type*=ft_Fillet, then the method constructs a curve on a cylindrical surface (a circle arc in a special case) tangential to the **curve**1 and **curve**2 in mating points. If *type*=ft_Fillet, then the *radius* can be both an input and an output parameter. If *radius* is an input parameter (more than zero), then the parameters *t*1 and *t*2 will be calculated, if *radius* is not specified (it is equal to zero), then *t*1 and *t*2 should be input parameters. This method will return the cylindric **surface**, on which the mating curve will be constructed, as an output parameter.

If **curve**1 and **curve**2 lie on the common surface, then they can be mated by a curve formed by intersection of the common surface with a cylindrical surface. In this case, *type*=ft_OnSurface. The *radius* parameter can set the radius of the cylindrical surface.

If *type*=ft_Spline, then the method constructs a NURBS curve mating **curve**1 in a point with *t*1 parameter and **curve**2 in a point with *t*2 parameter. The *radius* can define the tension of the NURBS curve in the mating points.

If *type*=ft_Double, then **curve**1 and **curve**2 will be mated by two arcs having *radius* radius. The mating points will be located on the curve ends. If necessary, it is possible to insert a line segment between the arcs.

If *type*=ft_Bridge, then the method mates **curve**1 and **curve**2 by MbBridgeCurve3D joining curve described in Item O.4.17. MbBridgeCurve3D Joining Curve. A joining curve is a cubic Hermite spline constructed based on two extreme points and curve derivatives in these points. In this case, the *radius* parameter is not used, rather *t*1 and *t*2 should be used as input parameter.

The parameter *t*1 defines the point of **curve**1 parametric domain, wherein the **curve**1 mates with the constructed curve. If **curve**1 is cyclically closed, then the parameter *w*1 should coincide with *t*1. For a non-closed curve, the parameter *w*1 combined with *t*1 defines the section of the **curve**1 that smoothly transforms into the constructed curve. The parameters *t*1 and *w*1 can be used to trim the **curve**1.

The parameter *t*2 defines the point of **curve**2 parametric domain, wherein the **curve**2 mates with the constructed curve. If **curve**2 is cyclically closed, then the parameter *w*2 should coincide with *t*2. For a non-closed curve, the parameter *w*2 combined with *t*2 defines the section of the **curve**2 that smoothly transforms into the constructed curve. The parameters *t*2 and *w*2 can be used to trim the **curve**2.

The parameter *radius* defines the radius of the mating curve (for the cases when *type* is not equal to ft_Spline and ft_Bridge values). If *radius* is a positive value, then it is used as an input parameter. In this case, the values of the *t*1 and *t*2 mating point parameters will be found. If *radius* is less than or equal to zero, then it is used as an output parameter. In this case, it is required to specify the *t*1 and *t*2 point parameter values of the mated curves.

The parameter *sense* defines the direction of the constructed curve. If *sense*=true, then the constructed curve will be directed from **curve**1 to **curve**2. If *sense*=false, then the constructed curve will be directed from **curve**2 to **curve**1.

The parameter *unchanged* informs about the changes in the constructed curve. If *unchanged*=true, then a circle arc having the set radius was constructed. In general case, the method returns value *unchanged*=false.

Names parameter defines the names of edges of the constructed wireframe.

Fig. M.4.12.1 shows an example of curve filleting for the *type*=ft_Fillet.

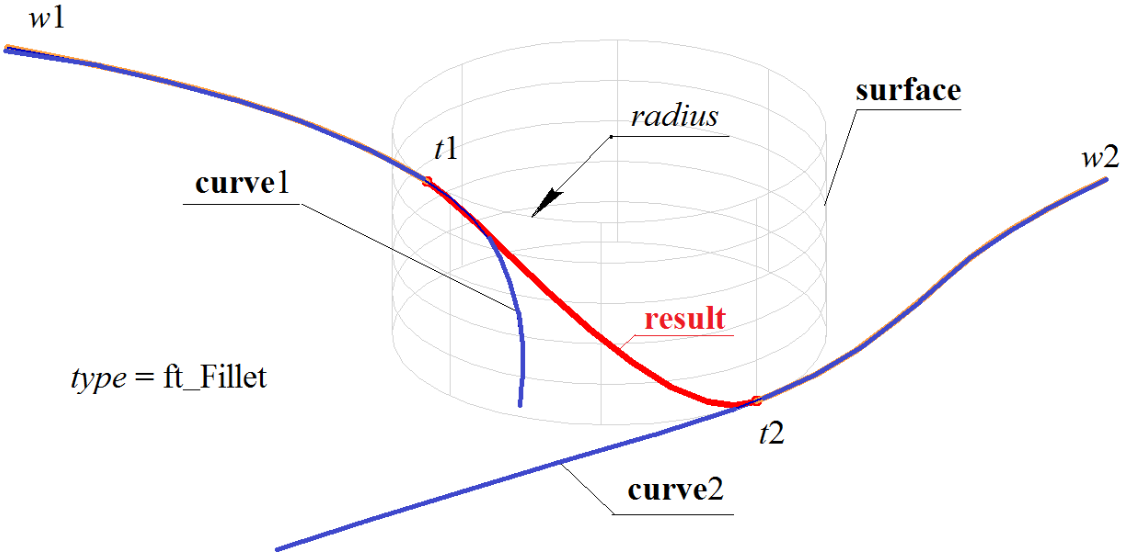


Fig. M.4.12.1.

Fig. M.4.12.2 shows an example of curve filleting for the *type*=ft_Double.

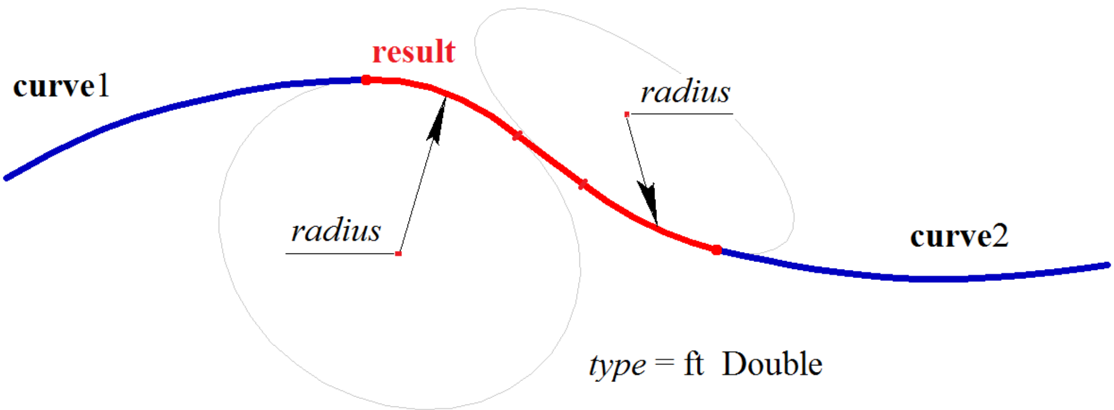


Fig. M.4.12.2.

Fig. M.4.12.3 shows an example of curve filleting for the *type*=ft_OnSurface.

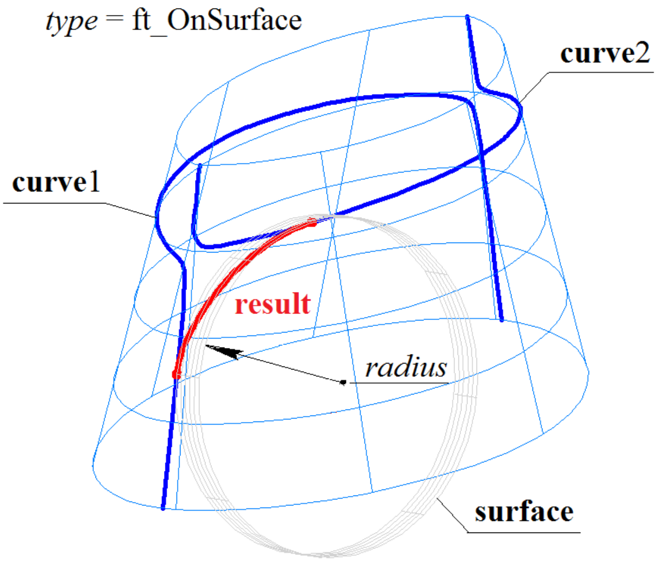


Fig. M.4.12.3.

The **FilletCurve** method adds the MbConnectingCurveCreator constructor containing all data required to construct the mating, to a log of a newly constructed wireframe. The MbConnectingCurveCreator constructor is declared in the cr_duplication_solid.h file.

M.5. SURFACE CONSTRUCTION METHODS

Surfaces describe the shape of simulated objects. Faces are constructed on the basis of surfaces that form solid bodies. In some cases surfaces act as reference objects to position other geometric model elements. All surface are MbSurface class inheritors, they are described in Chapter O.5. SURFACES. The surfaces can be constructed by directly calling the corresponding constructors or using the methods described in this item.

M.5.1. Elementary Surface Construction

The method
MbResultType
ElementarySurface (const MbCartPoint3D & **point**0,
const MbCartPoint3D & **point**1,
const MbCartPoint3D & **point**2,
MbeSpaceType *surfaceType*,
MbSurface *& **result**)
constructs an elementary surface.

The method input parameters are:

- **point**0 is a point that defines the origin of the local surface coordinate system,
- **point**1 is a point that defines the direction of local coordinate system axis and the surface radius,
- **point**2 is a point that defines the direction of local coordinate system axis,
- *surfaceType* is a type of the surface.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

The *surfaceType* parameter defines the created surface type. **point**0, **point**1 and **point**2 are the control points that define the local coordinate system and the elementary surface dimensions. **point**0 defines the local coordinate system origin. The Table M.5.1.1 shows surface type, local coordinate system axis and a value defined by **point**1 to create *surfaceType* surface.

Table M.5.1.1.

surfaceType	***surface* type**	**point1**	**defines the axis**
st_Plane	plane	**axisX**	
st_CylinderSurface	cylindrical surface	**axisZ**	height
st_ConeSurface	conical surface	**axisZ**	height
st_SphereSurface	spherical surface	**axisZ**	
st_TorusSurface	toroidal surface	**axisX**	larger radius

point1 defines the cylinder height, cone height, or larger torus radius. **point**2 defines the cylinder radius, cone radius and angle, sphere radius, and smaller torus radius.

When the surface is constructed, **point**0, **point**1 and **point**2 define the local coordinate system of the surface with the origin in **point**0. **axis**X of the local surface coordinate system is directed from **point**0 to **point**1, **axisY** of the local surface coordinate system is orthogonal to **axisX** and directed towards **point**2. The first parameter domain of definition is equal to doubled distance between **point**0 and **point**1. The first parameter domain of definition is equal to doubled distance between **point**0 and projection to **axisY** of **point**2, see Fig. M.5.1.1.

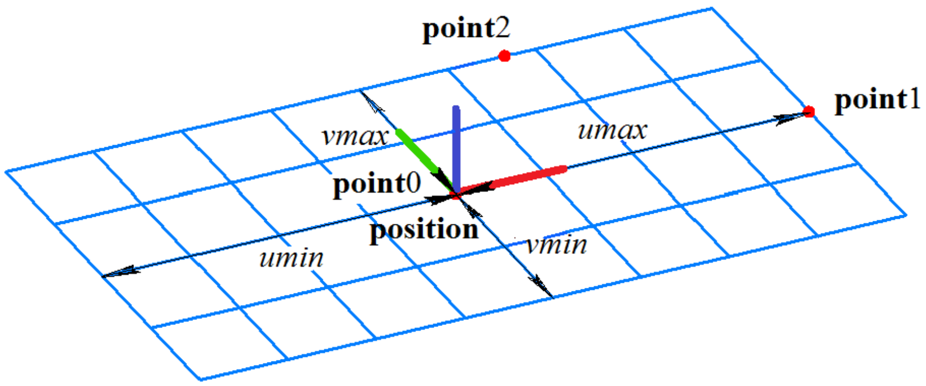


Fig. M.5.1.1.

When a cylindrical surface is constructed, **point**0 defines the center of the lower cylinder base, which serves as the local coordinate system origin. **point**1 defines the center of the upper cylinder base. **axisZ** of the local surface coordinate system is directed from **point**0 to **point**1. **point**2 along with other points defines the plane of **axisX** and **axisZ** of the local surface coordinate system. The distance between **point**0 and **point**1 defines the cylindric surface height, the distance from **axisZ** to **point**[2] defines the cylindric surface radius, see Fig. M.5.1.2.

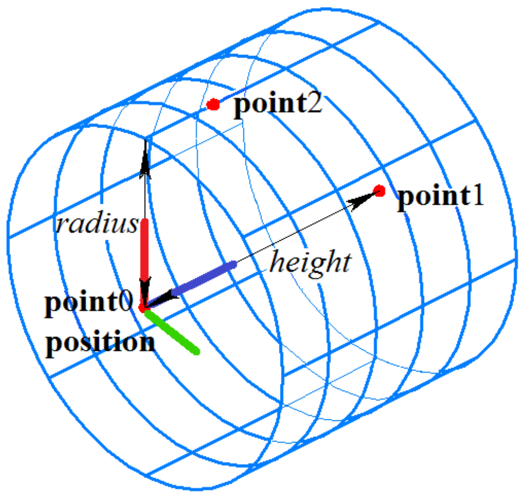


Fig. M.5.1.2.

Cylindrical surfaces are described in Item O.5.3. MbCylinderSurface Cylindrical Surface.

When a conical surface is constructed, **point**0 defines the cone vertex, which serves as the local coordinate system origin. **point**1 defines the cone base center and **axisZ** of the local coordinate system of the cone. **point**2 along with other points defines the plane of **axisX** and **axisZ** of the local surface coordinate system. The distance between **point**0 and **point**1 defines the height of the conical surface. **point**2 defines the cone angle if **point**2 lies on the conical surface, see Fig. M.5.1.3.

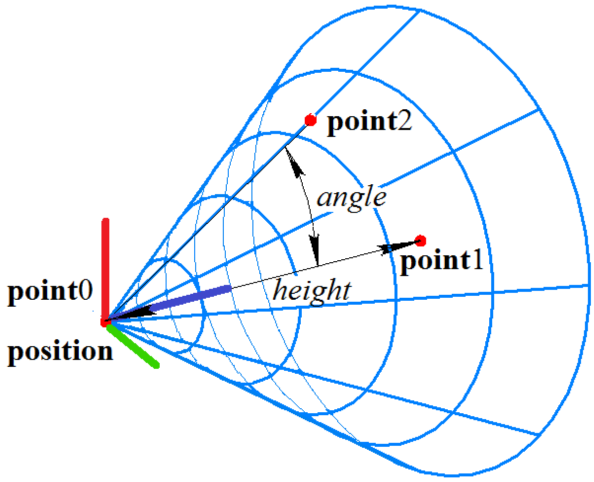


Fig. M.5.1.3.

Conical surfaces are described in Item O.5.4. MbConeSurface Conical Surface.

When a spherical surface is constructed, **point**0 defines the center of the sphere, which serves as the local coordinate system origin. **axisZ** of the local surface coordinate system is directed from **point**0 to **point**1. **point**2 along with other points defines the plane of **axisX** and **axisZ** of the local surface coordinate system. The distance between **point**0 and **point**2 defines the spherical surface radius, see Fig. M.5.1.4.

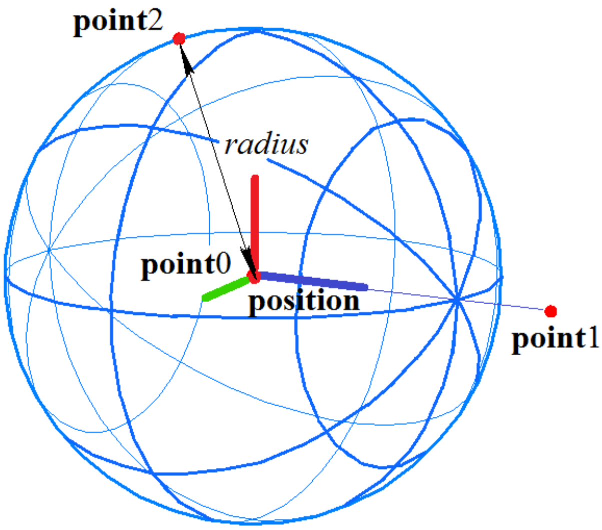


Fig. M.5.1.4.

Spherical surfaces are described in Item O.5.5. MbSphereSurface Spherical Surface.

When a toroidal surface is constructed, **point**0 defines the center of the torus, which serves as the local coordinate system origin. **axisX** of the surface local coordinate system is directed from **point**0 to **point**1. **point**2 along with previous points defines the plane of **axisX** and **axisZ** of the local surface coordinate system. The distance between **point**0 and **point**1 points defines the larger torus radius; the distance between **point**1 and **point**2 defines the smaller torus radius, see Fig. M.5.1.5.

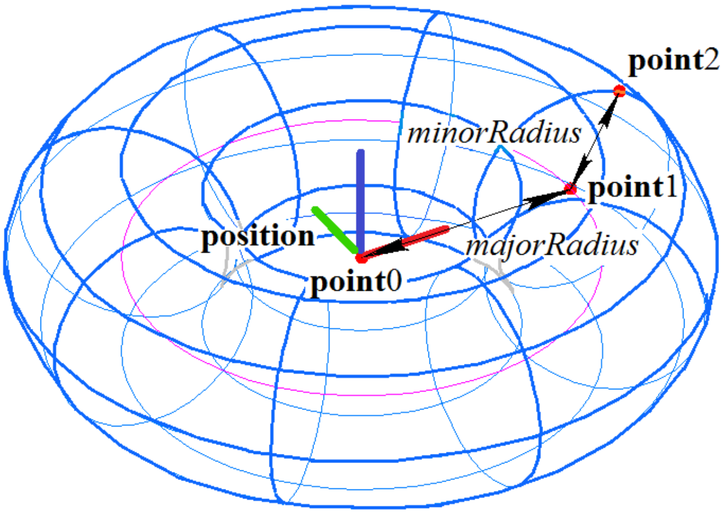


Fig. M.5.1.5.

Torus surfaces are described in Item O.5.6. MbTorusSurface Toroidal Surface.

The test.exe application constructs an elementary surface based on the given points using "Create->Surface->Elementary->" menu command.

M.5.2. NURBS Surface Construction

The method

MbResultType

SplineSurface (const MbCartPoint3D & **pUMinVMin**,
const MbCartPoint3D & **pUMaxVMin**,
const MbCartPoint3D & **pUMaxVMax**,
const MbCartPoint3D & **pUMinVMax**,
size_t uCount,
size_t vCount,
size_t uDegree,
size_t vDegree,
MbSurface *& **result**)

constructs a flat NURBS surface based on corner control points.

The method input parameters are:

- **pUMinVMin** is the lower left corner surface point,
- **pUMaxVMin** is the lower right corner surface point,
- **pUMaxVMax** is the upper right corner surface point,
- **pUMinVMax** is the upper right corner surface point,
- uCount is a number of control points along the first parameter (in horizontal direction),
- uCount is a number of control points along the second parameter (in vertical direction),
- uDegree is a B-spline degree along the first parameter,
- vDegree is a B-spline degree along the second parameter,

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

This method constructs a NURBS surface, the control points of which are located in the nodes of four-corner table with uCount columns and vCount lines. The surface *B*-spline degree along the first parameter is uDegree, and the surface *B*-spline degree along the second parameter is vDegree. The surface control points will be calculated based on the equidistant condition and coincidence of corner points with **pUMinVMin**,

pUMaxVMin, **pUMaxVMax**, **pUMinVMax**, see Fig. M.5.2.1.

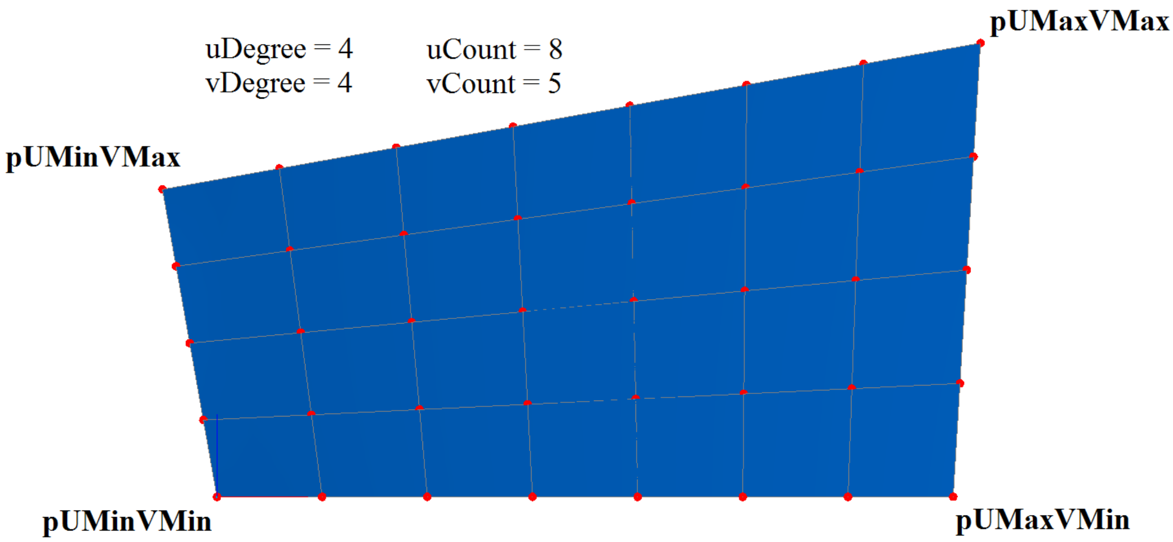


Fig. M.5.2.1.

The weights of all control points are equal to one. The constructed surface is designed to be further modified. The method parameters should satisfy the following inequalities: uCount≥uDegree and vCount≥vDegree. NURBS surfaces are described in Item O.5.22. MbSplineSurface NURBS Surface.

The method

MbResultType

SplineSurface (const SArray<MbCartPoint3D> & **points**,
const SArray<double> & *weights*,
size_t uCount,
size_t vCount,
size_t uDegree,
const SArray<double> & *uKnots*,
bool *uClosed*,
size_t vDegree,
const SArray<double> & *vKnots*,
bool *vClosed*,
MbSurface *& **result**)

constructs a NURBS surface based on control points and their weights.

The method input parameters are:

- **points** is a set of control points that can be represented as vCount lines that have uCount points in each line,
- *weights* is a set of control point weights consistent with the set of control points,
- uCount is a number of control points along the first parameter (in each line),
- uCount is a number of control points along the second parameter (number of lines),
- uDegree is a B-spline degree along the first parameter,
- *uKnots* is a knot vector of the first parameter,
- *uClosed* is a parameter that defines whether the surface is closed along the first parameter,
- vDegree is a B-spline degree along the second parameter,
- *vKnots* is a knot vector of the second parameter,
- *vClosed* is a parameter that defines whether the surface is closed along the second parameter,

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

This method constructs a NURBS surface with **points** as control points. The control points of the **points** set will be conditionally split into lines and columns. Each line will contain uCount control points from the **points** set, and the total number of lines will be equal to vCount. Therefore, the **points** set will contain uCount·vCount control points. The surface *B*-spline degree along the first parameter is uDegree, and the surface *B*-spline degree along the second parameter is vDegree. The method parameters should satisfy the following inequalities: uCount≥uDegree and vCount≥vDegree. NURBS surfaces are described in Item O.5.22. MbSplineSurface NURBS Surface.

The sets *weights*, *uKnots*, *vKnots* can be empty.

If the *weights* set is not empty, then it should be aligned with the **points** control point set.

If the *uKnots* and *vKnots* sets are not empty, then they should contain a specific number of elements: for *uClosed*=false, the number of elements in the *uKnots* vector should be equal to uCount+uDegree; for *uClosed*=true, the number of elements in the *uKnots* vector should be equal to uCount+2uDegree–1; for *vClosed*=false, the number of elements in the *vKnots* vector should be equal to vCount+vDegree; for *vClosed*=true, the number of elements in the *vKnots* vector should be equal to vCount+2vDegree–1. Fig. M.5.2.2 shows the positions of control points to construct a torus-shaped spline surface.

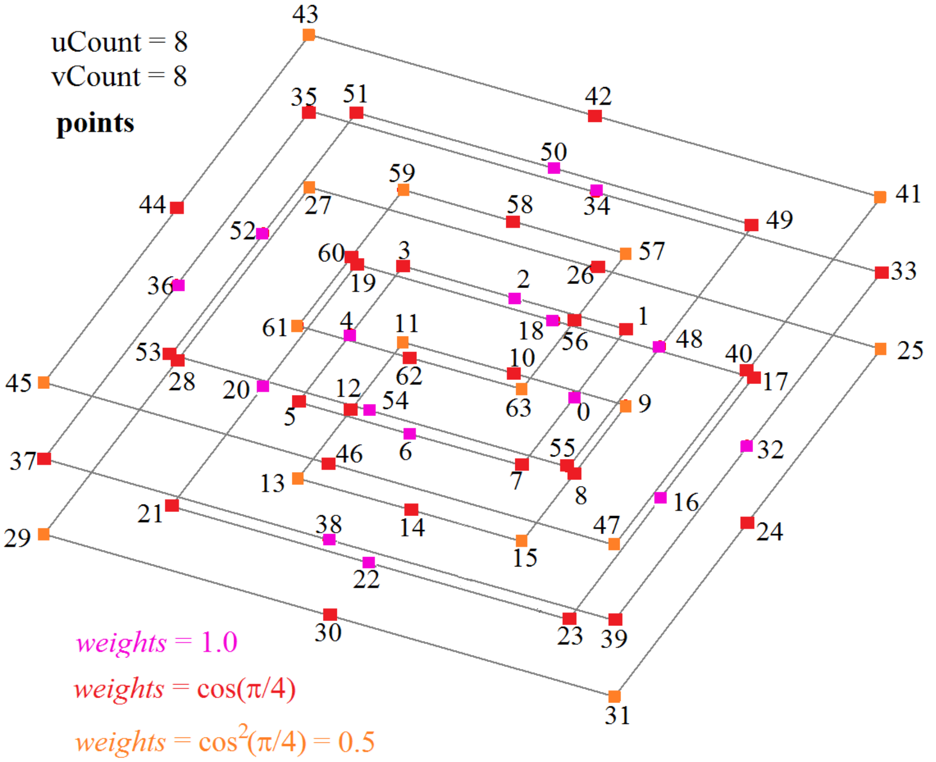


Fig. M.5.2.2.

Fig. M.5.2.3 shows a torus-shaped spline surface constructed based on the control points specified above.

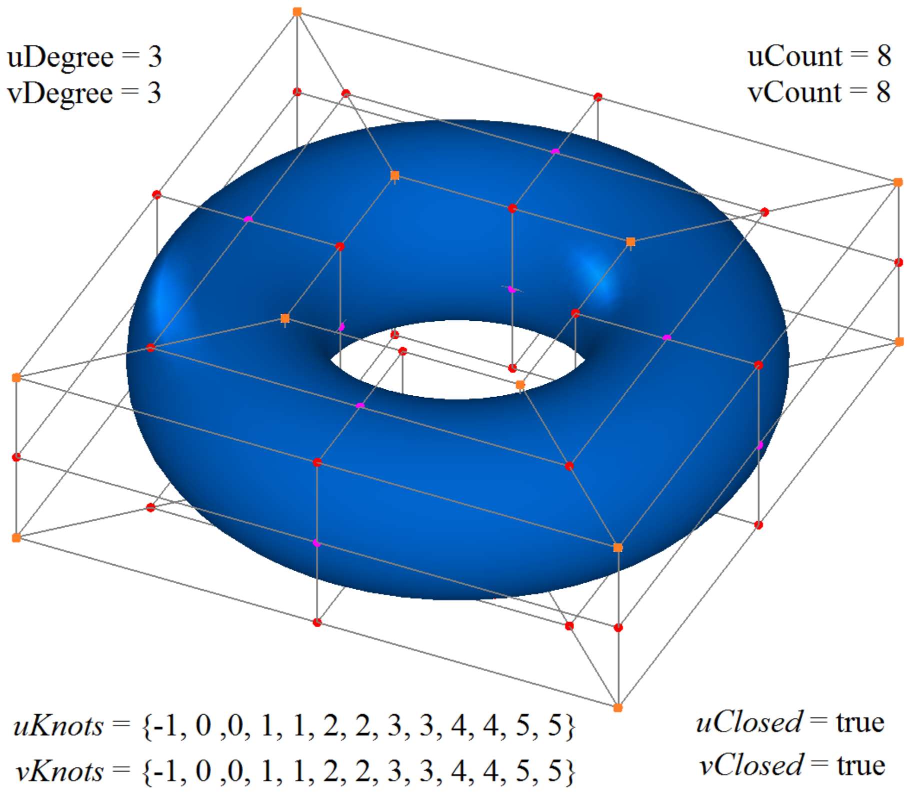


Fig. M.5.2.3.

Below you can find the C++ code that constructs a torus-shaped spline surface using this method (the torus center coincides with the global coordinate system origin, and the torus axis is directed along the global axis Z).

```
//------------------------------------------------------------------------------
void AddTorusPoints( double r, double z, SArray<MbCartPoint3D> & points, double w, SArray<double> & weights )
{
 MbCartPoint3D p( r, 0.0, z );
 MbVector3D toX( 1.0, 0.0, 0.0 ), toY( 0.0, 1.0, 0.0 );
 double wI( w ), wA = w * ::cos( M_PI_4 );
 points.Add( p );  weights.Add( wI );
 p.Add( toY, r );  points.Add( p );  weights.Add( wA );
 p.Add( toX,-r );  points.Add( p );  weights.Add( wI );
 p.Add( toX,-r );  points.Add( p );  weights.Add( wA );
 p.Add( toY,-r );  points.Add( p );  weights.Add( wI );
 p.Add( toY,-r );  points.Add( p );  weights.Add( wA );
 p.Add( toX, r );  points.Add( p );  weights.Add( wI );
 p.Add( toX, r );  points.Add( p );  weights.Add( wA );
}
//------------------------------------------------------------------------------
void GetTorusPoints( double majorR, double minorR, SArray<MbCartPoint3D> & points, SArray<double> & weights )
{
 double zLavel( 0.0 );
 double w0( 1.0 ), wA = ::cos( M_PI_4 );
 ::AddTorusPoints( majorR - minorR, zLavel, points, w0, weights );
 ::AddTorusPoints( majorR - minorR,-minorR, points, wA, weights );
 ::AddTorusPoints( majorR         ,-minorR, points, w0, weights );
 ::AddTorusPoints( majorR + minorR,-minorR, points, wA, weights );
 ::AddTorusPoints( majorR + minorR, zLavel, points, w0, weights );
 ::AddTorusPoints( majorR + minorR, minorR, points, wA, weights );
 ::AddTorusPoints( majorR         , minorR, points, w0, weights );
 ::AddTorusPoints( majorR - minorR, minorR, points, wA, weights );
}
//------------------------------------------------------------------------------
void GetTorusKnots( double tBeg, double tEnd, SArray<double> & knots )
{
 double dt = ( tEnd - tBeg ) / 4.0;
```

```
  double t = tBeg - dt;  knots.Add( t );
  t = tBeg;  knots.Add( t );  knots.Add( t );
  t += dt;  knots.Add( t );  knots.Add( t );
  t += dt;  knots.Add( t );  knots.Add( t );
  t += dt;  knots.Add( t );  knots.Add( t );
  t = tEnd;  knots.Add( t );  knots.Add( t );
  t += dt;  knots.Add( t );  knots.Add( t );
}
//-----------------------------------------------------------------------------
MbSurface * CreateTorusSurface( double majorR, double minorR )
{
  MbSurface * result( NULL );
  MbResultType res( rt_Error );
  if ( majorR > METRIC_NEAR && majorR > minorR - EPSILON ) {
    SArray<MbCartPoint3D> points( 64, 1 );
    SArray<double> weights( 64, 1 );
    SArray<double> uKnots( 13, 1 );
    SArray<double> vKnots( 13, 1 );
    size_t uDegree( 3 ), vDegree( 3 );
    size_t uCount( 8 ), vCount( 8 );
    bool uClosed( true ), vClosed( true );
    ::GetTorusPoints( majorR, minorR, points, weights );
    ::GetTorusKnots( 0.0,  4.0, uKnots );
    ::GetTorusKnots( -2.0, 2.0, vKnots );
    res = ::SplineSurface( points, weights, uCount, vCount, uDegree, uKnots, uClosed, vDegree, vKnots, vClosed, result );
  }
  return ( res == rt_Success ) ? result : NULL;
}
```

The method

MbResultType
NurbsSurface (const MbSurface & **surface**,
VERSION version,
MbSurface *& **result**)

constructs a NURBS copy of the given surface.

The method input parameters are:

- **surface** is an original surface,
- version is a construction version.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

This method constructs a NURBS surface with a shape that copies the original surface shape. NURBS surfaces are described in Item O.5.22. MbSplineSurface NURBS Surface. NURBS surfaces completely coincide with the original curves for the majority of surface types. If it is impossible to accurately reproduce the original surface shape, then the NURBS copy approximates the original surface with an error less than 0.0001.

If the original surface is an MbCurveBoundedSurface with arbitrary bounds described in Item O.5.27. MbCurveBoundedSurface Surface with Arbitrary Borders, then this method constructs a surface with arbitrary boundaries, wherein the NURBS copy of the base surface of the original surface will be used as a base surface.

M.5.3. Construction of Extrusion Surface

The method

MbResultType
ExtrusionSurface (MbCurve3D & **curve**,
const MbVector3D & **direction**,
MbSurface*& **result**)

constructs an extrusion surface.

The method input parameters are:

- **curve** is a generatrix curve,

- **direction** is a vector defining the extrusion direction and length.

The output parameter of the method is the **result** constructed surface.

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

Extrusion surfaces belong to the type of sliding surfaces which are constructed by moving a generatrix along a guiding curve. An extrusion surface is formed by moving the generatrix along the segment, the direction and length of which are defined by the **direction** vector. Extrusion surfaces are described in Item O.5.7. MbExtrusionSurface Extrusion Surface. Fig. M.5.3.1 shows a surface constructed by extruding the curve along the given vector.

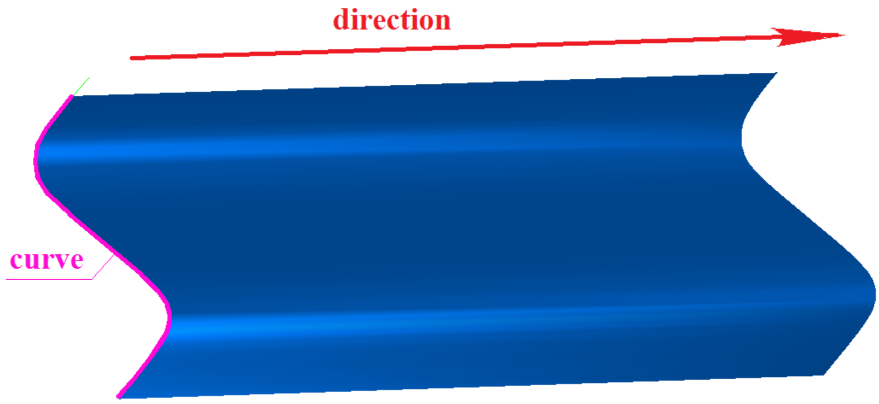


Fig. M.5.3.1.

M.5.4. Construction of Revolution Surface

The method

MbResultType
RevolutionSurface (MbCurve3D & **curve**,
const MbCartPoint3D & **origin**,
const MbVector3D & **axis**,
double *angle*,
MbSurface *& **result**)

constructs a revolution surface.

The method input parameters are:

- **curve** is a generatrix curve,
- **origin** is a point on the revolution axis,
- **axis** is a direction of the revolution axis,
- *angle* is a revolution angle.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

Revolution surfaces belong to the type of sliding surfaces which are constructed by moving a generatrix along a guiding curve. A revolution surface is formed by moving the generatrix along a circle arc, wherein the center is located in the **origin** point, the axis is parallel to the **axis** vector, and the opening angle is equal to *angle* value. Revolution surfaces are described in Item O.5.8. MbRevolutionSurface Revolution Surface. Fig. M.5.4.1 shows a surface constructed by rotation of a curve around an axis for the given angle.

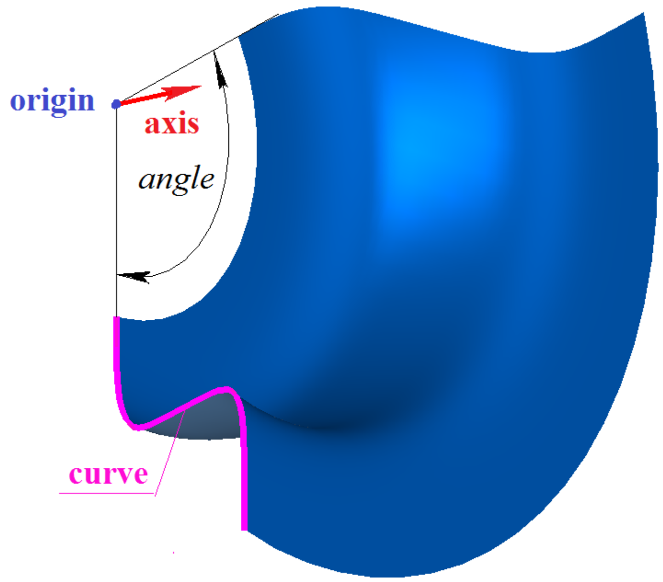


Fig. M.5.4.1.

M.5.5. Sweep Surface Construction

The method

MbResultType
ExpansionSurface (MbCurve3D & **curve**,
MbCurve3D & **spine**,
MbSurface *& **result**)

constructs a translation surface.

The method input parameters are:

- **curve** is a generatrix curve,
- **spine** is a guiding curve.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

Sweep surfaces belong to the type of sliding surfaces which are constructed by moving a generatrix along a guiding curve. Translation surfaces are constructed by plane-parallel motion of the generatrix along the guiding curve. Sliding surfaces are described in Item O.5.9. MbExpansionSurface Motion Surface. Fig. M.5.5.1 shows a surface constructed by plane-parallel movement of the generatrix along the guiding curve.

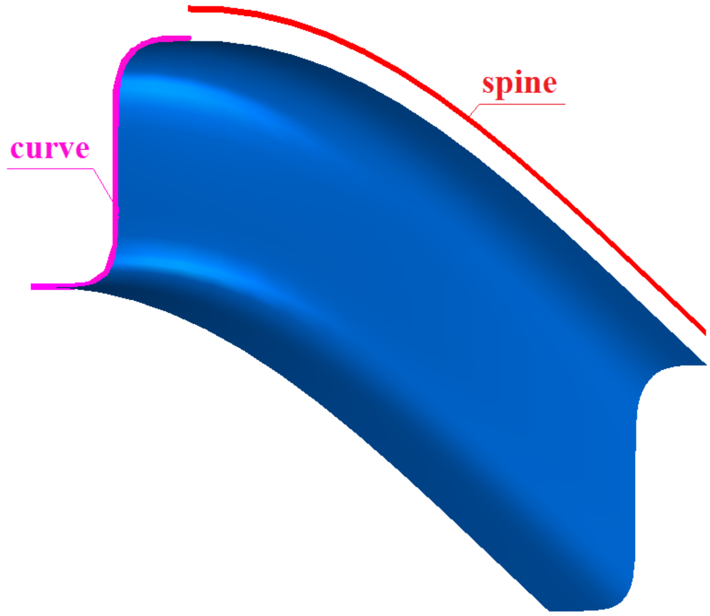


Fig. M.5.5.1.

The method
MbResultType
EvolutionSurface (MbCurve3D & **curve**,
MbCurve3D & **spine**,
MbSurface *& **result**)
constructs a kinematic surface.

The method input parameters are:

- **curve** is a generatrix curve,
- **spine** is a guiding curve.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

Kinematic surface is constructed by movement of **curve** generatrix along **spine** guiding curve. Kinematic surfaces are described in Item O.5.11. MbEvolutionSurface Swept Surface. Fig. M.5.5.2 shows a kinematic surface constructed by sweeping of a generatrix along a guiding curve.

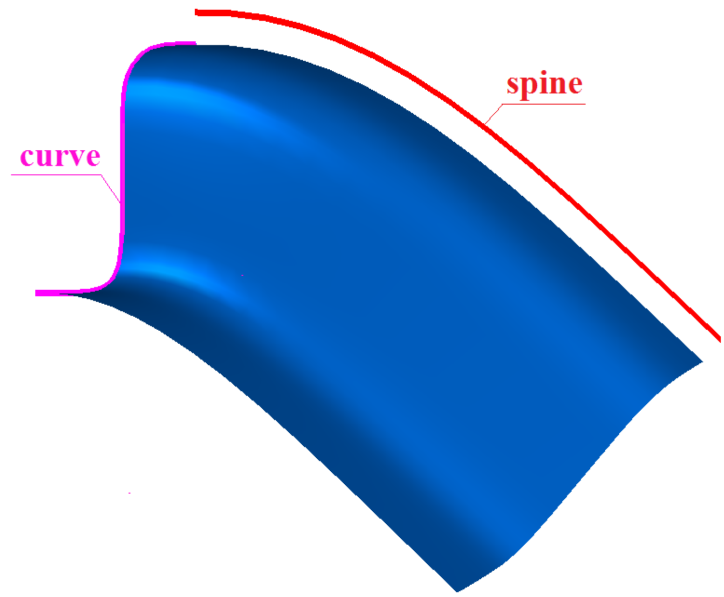


Fig. M.5.5.2.

The method
MbResultType
SpiralSurface (MbCurve3D & **curve**,
const MbSurface & **point**0,
const MbSurface & **point**1,
const MbSurface & **point**2,
double *step*,
MbSurface *& **result**)
constructs a spiral surface.

The method input parameters are:

- **curve** is a generatrix curve,
- **point**0 is the origin of the local coordinate system,
- **point**1 is a point located on the Z axis of the local coordinate system,
- **point**1 is a point located in the X axis direction of the local coordinate system,
- *step* is a spiral step.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

A spiral surface is a special case of the kinematic surface described above. It is formed by moving the generatrix **curve** along a cylindrical spiral. **point**0, **point**1 and **point**2 are the control points that define the local coordinate system and the spiral length. **point**0 defines the local coordinate system origin. **axisZ** of the local spiral coordinate system is directed from **point**0 to **point**1. **point**2 along with other points defines the plane of **axisX** and **axisZ** of the local spiral coordinate system. The distance between **point**0 and **point**1 defines the height of the spiral. The *step* parameter defines the spiral step. Spiral surfaces are described in Item O.5.10. MbSpiralSurface Spiral Surface. Fig. M.5.5.3 shows a spiral surface.

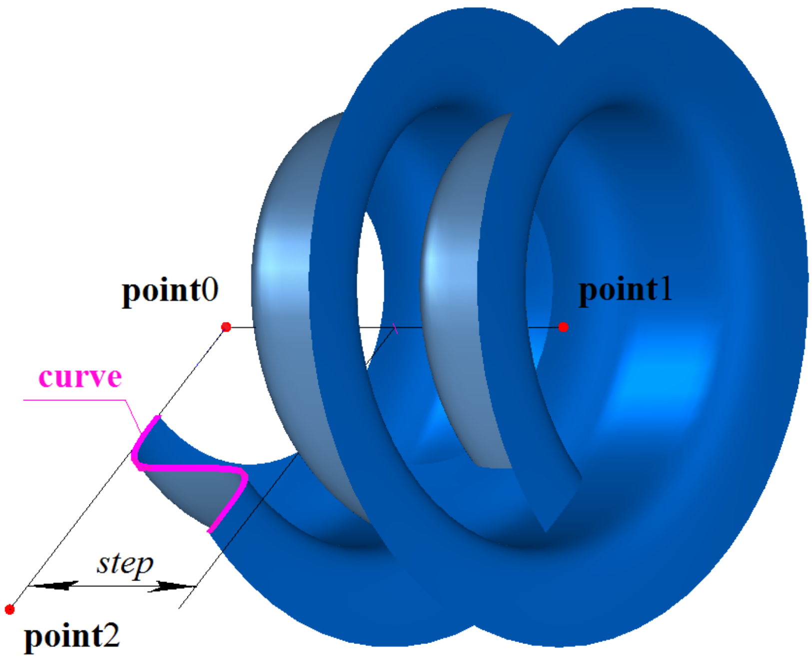


Fig. M.5.5.3.

M.5.6. Surface Construction Based on a Family of Curves

The method
MbResultType
LoftedSurface (const RPArray<MbCurve3D> & **curves**,
bool *closed*,
const MbVector3D & **begDirection**,
const MbVector3D & **endDirection**,
MbSurface *& **result**)
constructs a surface from a family of curves.

The method input parameters are:

- **curves** is a family of curves,
- *closed* is a parameter that defines whether the surface is closed along the second parameter,
- **begDirection** is a direction vector at the surface beginning,
- **endDirection** is a direction vector at the surface ending.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

The constructed surface contains the **curves** set. If all the curves of the family are cyclically closed, then the surface will be cyclically closed by the first parameter. When a surface is constructed, it is important to keep the curve directions in mind, because the surface may intersect itself if the curves have different directions. The *closed* parameter defines whether the surface is cyclically closed by the second parameter. If *closed*=false, then the **begDirection** and **endDirection** parameters define the surface direction on edge curves. If the length of **begDirection** and **endDirection** vectors is equal to zero, then the direction of the surface on edge curves will be determined from the condition that the second derivatives at the surface edges should be equal to zero. Surfaces on families of curves are described in Item O.5.15. MbLoftedSurface Surface Based on a Family of Curves. Fig. M.5.6.1 shows a surface cyclically closed by the second parameter that was constructed based on a family of curves.

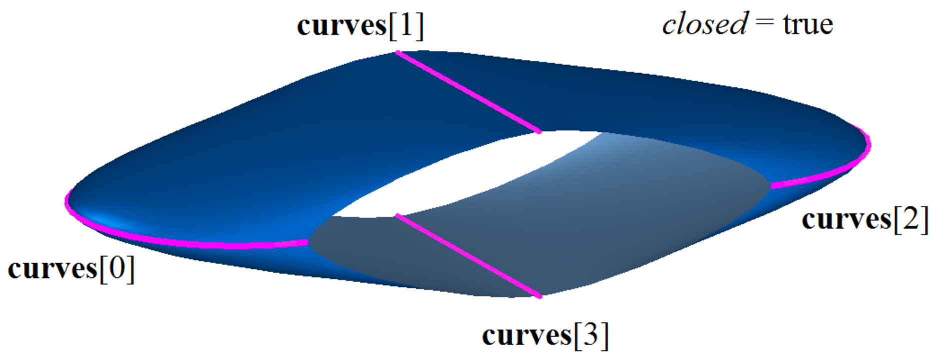


Fig. M.5.6.1.

Fig. M.5.6.2 shows a non-closed surface with specified edge directions constructed based on the family of curves.

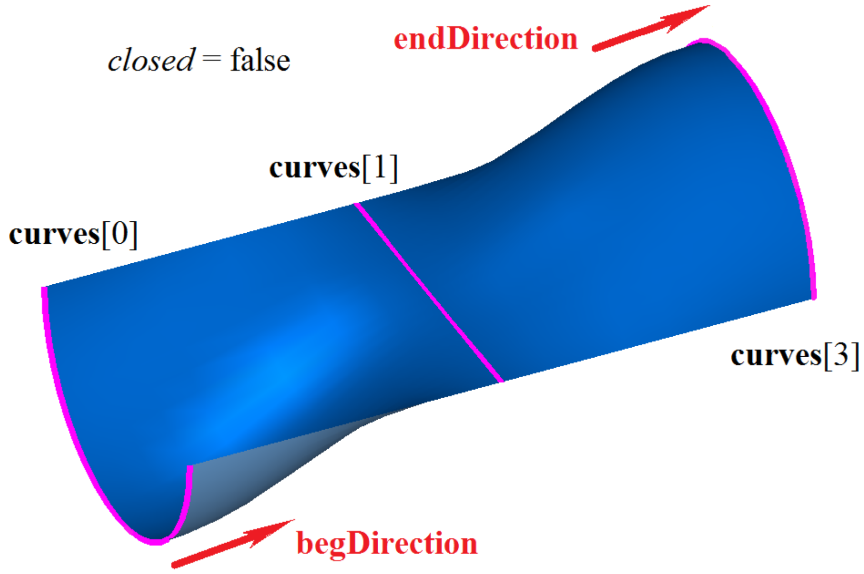


Fig. M.5.6.2.

The method
MbResultType
LoftedSurface (const RPArray<MbCurve3D> & **curves**,
MbCurve3D & **spine**,
MbSurface *& **result**)
constructs a surface based on a family of curves and a guiding curve.

The method input parameters are:

- **curves** is a family of curves,
- **spine** is a guiding curve.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

The constructed surface contains the **curves** set. The **spine** guiding curve defines the shape of the surface between the curves of the family. Surfaces on families of curves and guiding curves are described in Item O.5.16. MbElevationSurface Surface Based on a Family of Curves And a Guiding Curve. Fig. M.5.6.3 shows a surface constructed based on a family of curves and a guiding curve.

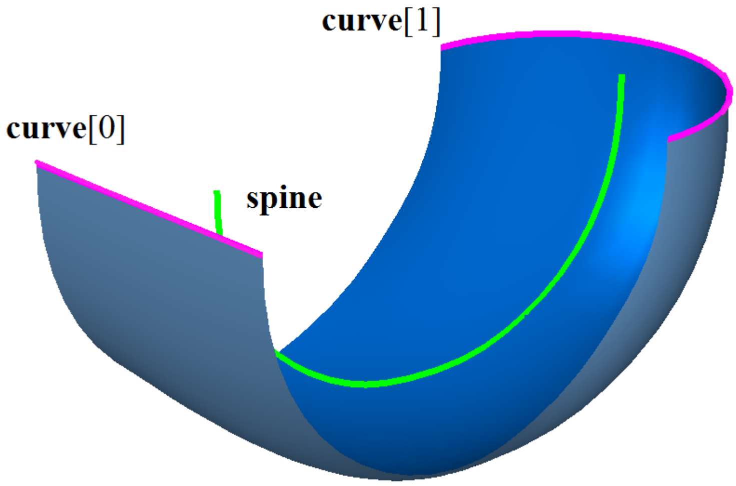


Fig. M.5.6.3.

M.5.7. Construction of Ruled Surfaces

The method
MbResultType
SectorSurface (MbCurve3D & **curve**,
const MbCartPoint3D & **point**,
MbSurface *& **result**)
constructs a sectorial surface based on a curve and a line.

The method input parameters are:

- **curve** is a curve,
- **point** is a point.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

The constructed surface contains the **curve** and the **point**. In the **point**, the surface has a pole, where the derivative of the surface radius vector with respect to the first parameter becomes zero. The **curve** should not be degenerated and it should not contain the **point**. Sectorial surfaces are described in Item O.5.13. MbSectorSurface Sectorial Surface. Fig. M.5.7.1 shows a sectorial surface.

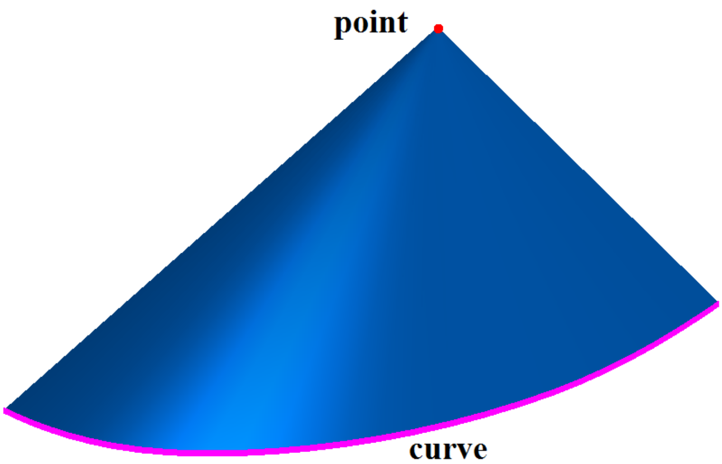


Fig. M.5.7.1.

The method
MbResultType
RuledSurface (MbCurve3D & **curve**1,
MbCurve3D & **curve**2,
MbSurface *& **result**)
constructs a ruled surface based on two curves.

The method input parameters are:

- **curve**1 is the first curve,
- **curve**2 is the second curve.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

The constructed surface contains **curve**1 and **curve**2. The surface will have no self-intersections if **curve**1 and **curve**2 have no intersections in the points not coinciding with the curve edges. Ruled surfaces are described in Item O.5.14. MbRuledSurface Ruled Surface. Fig. M.5.7.2 shows a ruled surface.

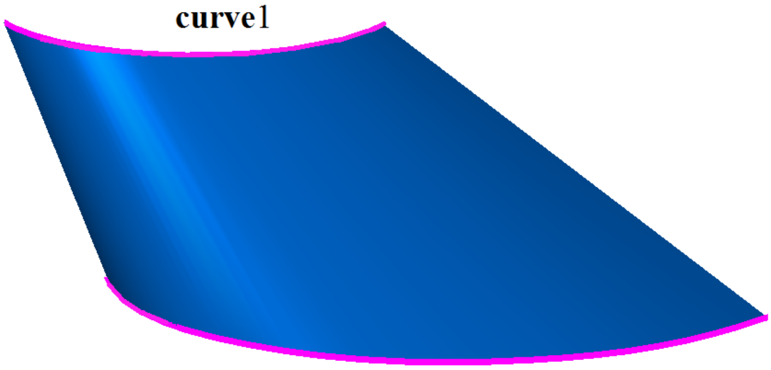


Fig. M.5.7.2.

M.5.8. Surface Construction Based on Three Curves

The method
MbResultType
CornerSurface (MbCurve3D & **curve**1,
MbCurve3D & **curve**2,
MbCurve3D & **curve**3,
MbSurface *& **result**)
constructs a surface based on three curves.

The method input parameters are:

- **curve**1 is the first curve,
- **curve**2 is the second curve,
- **curve**3 is the third curve.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

In the general case, the constructed surface does not contain **curve**1, **curve**2 and **curve**3. To construct the surface, the curves should have crossing points, where the segment joining the crossing points is orthogonal to the curves. The constructed surface will contain **curve**1, **curve**2 and **curve**3 if all these curves intersect with each other. The surface has a pole, where the derivative of the surface radius vector with respect to the first parameter becomes zero. Surfaces constructed based on three curves are described in Item O.5.17. MbCornerSurface Surface Based on Three Curves. Fig. M.5.8.1 shows a surface constructed based on three curves in two different aspects.

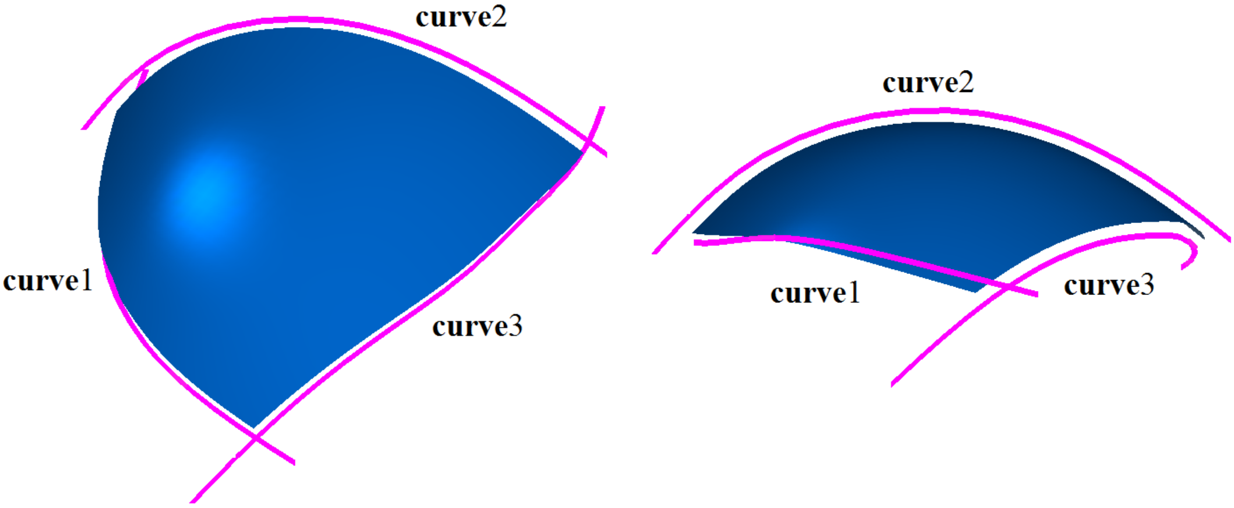


Fig. M.5.8.1.

M.5.9. Surface Construction Based on Four Curves

The method

MbResultType

CoverSurface (MbCurve3D & **curve**1,
MbCurve3D & **curve**2,
MbCurve3D & **curve**3,
MbCurve3D & **curve**4,
MbSurface *& **result**)

constructs a surface based on four curves.

The method input parameters are:

- **curve**1 is the first curve,
- **curve**2 is the second curve,
- **curve**3 is the third curve,
- **curve**4 is the fourth curve.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

In the general case, the constructed surface does not contain **curve**1, **curve**2, **curve**3 and **curve**4. To construct the surface, **curve**1 and **curve**2, **curve**3 and **curve**4, **curve**1 and **curve**4, **curve**2 and **curve**3 should have the crossing points, where the segment joining the crossing points is orthogonal to the curves. The constructed surface will contain **curve**1, **curve**2, **curve**3 and **curve**4 if all pairs of these curves intersect. Surfaces constructed based on four curves are described in Item O.5.18. MbCoverSurface Coons Surface. Fig. M.5.9.1 shows a surface constructed based on four curves in two different aspects.

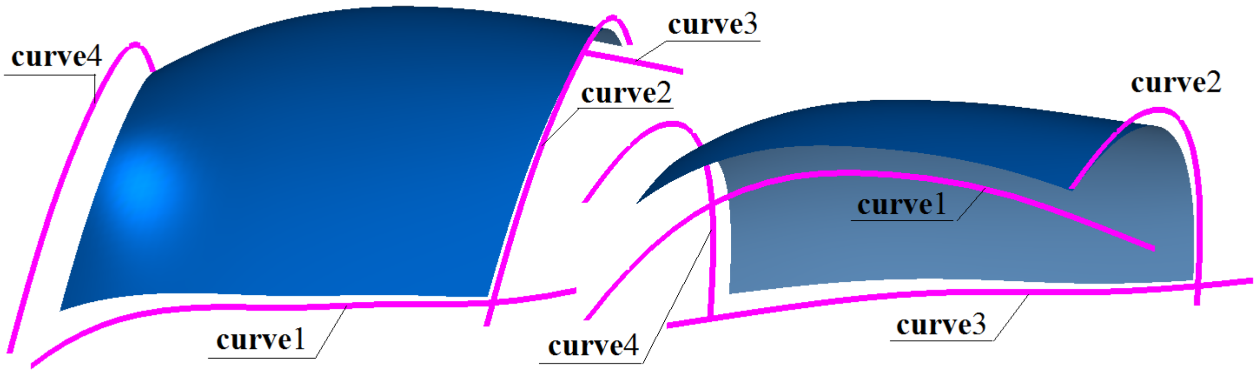


Fig. M.5.9.1.

M.5.10. Construction of Surface Based on a Curve Grid

The method
MbResultType
MeshSurface (const RPArray<MbCurve3D> & **uCurves**,
const RPArray<MbCurve3D> & **vCurves**,
MbSurface *& **result**)
constructs a surface based on two families of curves.

The method input parameters are:

- **uCurves** is the first set of curves (by the first parameter),
- **vCurves** is the second set of curves (by the second parameter),

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

To construct the surface, each curve of the **uCurves** set should intersect with all curves of the **vCurves** set, and, correspondingly, all curves of the **vCurves** set should intersect with all curves of the **uCurves** set. To avoid surface self-intersection, the adjacent curves of the **uCurves** set should have the same direction, as well as the adjacent curves of the **vCurves** set. The constructed surface will contain the curves of the **uCurves** set and the curves of the **vCurves** set. Surfaces constructed based on curve grid are described in Item O.5.20. MbMeshSurface Surface Based on a Network of Curves. Fig. M.5.10.1 shows two sets of curves, and Fig. M.5.10.2 shows a surface constructed based on these two curves.

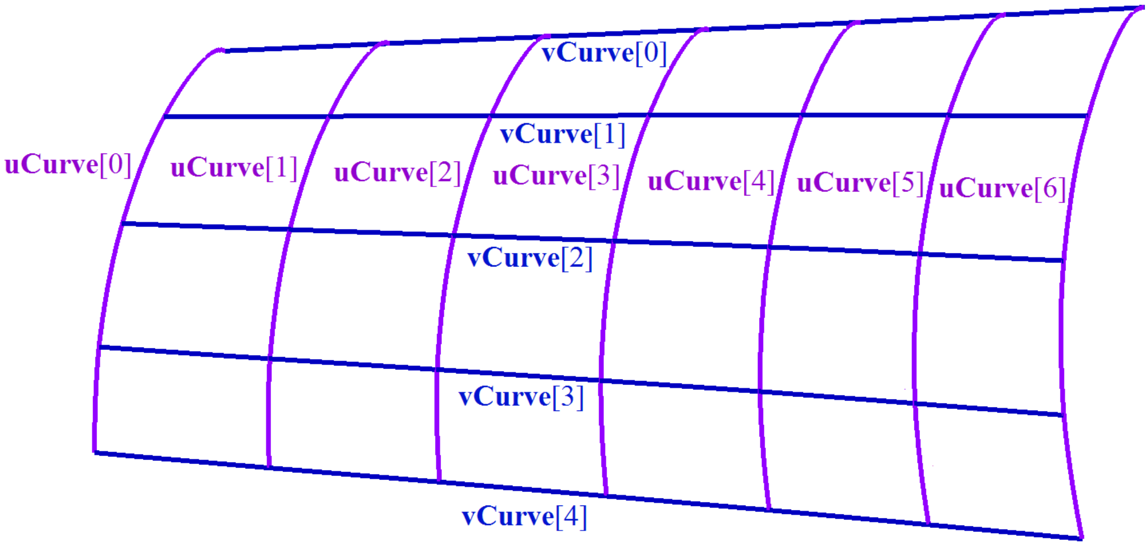


Fig. M.5.10.1.

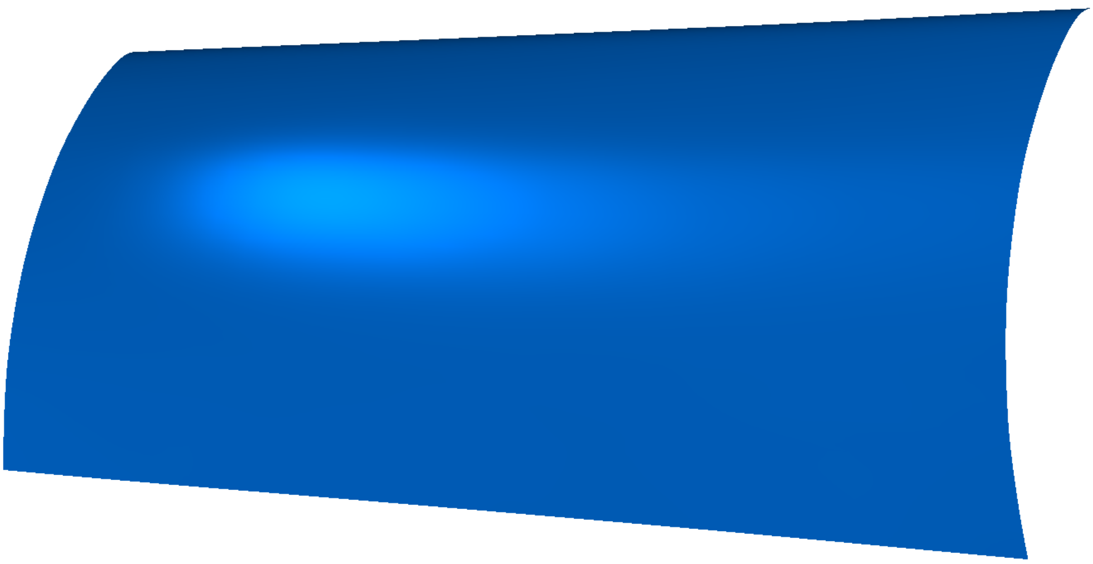


Fig. M.5.10.2.

M.5.11. Equidistant Surface Construction

The method
MbResultType
OffsetSurface (MbSurface & **surface**,
double *distance*,
MbSurface *& **result**)
constructs an equidistant surface.

The method input parameters are:

- **surface** is a base surface,
- *distance* is a signed equidistant value.

The output parameter of the method is the **result** constructed surface.

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

The method is declared in the action_surface.h file.

Surface construction is possible, if the **surface** has no points, where the surface bends in the direction of the equidistant and curve radius is less than *distance*. Each point of the constructed surface will be located at the *distance* from the local normal of **surface** with the same parameters. Equidistant surfaces are described in Item O.5.23. MbOffsetSurface Equidistant Surface. Fig. M.5.11.1 shows an equidistant surface.

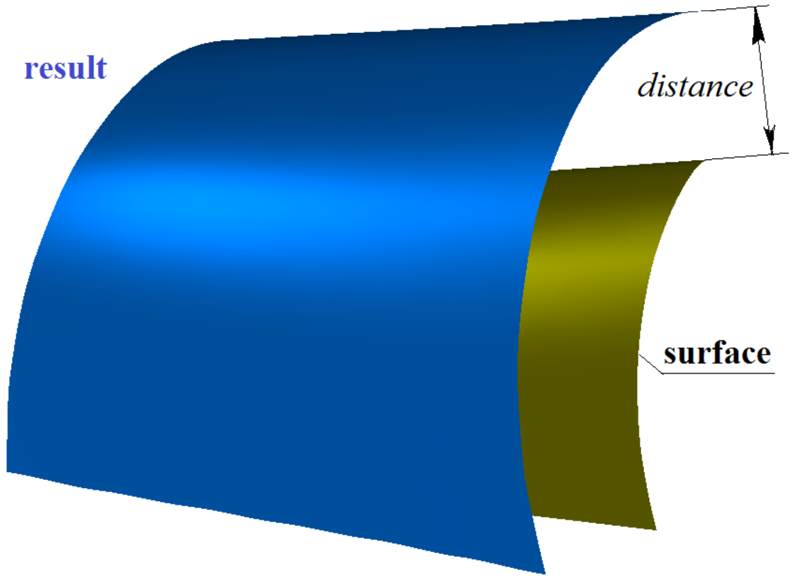


Fig. M.5.11.1.

The domain of definition of the constructed equidistant surface coincides with the domain of definition of **surface**. The domain of definition of equidistant surface can be extended.

The method
MbResultType
ExtendedSurface (MbSurface & **surface**,
double *uMin*,
double *uMax*,
double *vMin*,
double *vMax*,
MbSurface *& **result**)
constructs an extended surface.

The method input parameters are:

- **surface** is a base surface,
- *uMin* is the minimum value of the first surface parameter,
- *uMax* is the maximum value of the first surface parameter,
- *vMin* is the minimum value of the second surface parameter,
- *vMax* is the maximum value of the second surface parameter.

The output parameter of the method is the **result** constructed surface.

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

The surface can be constructed if it has no poles and other singular points on a boundary of surface domain of definition and its extension. The constructed surface coincides with **surface**, but it has other domains of definition of the parameters. A domain can be both reduced and extended. Beyond the domain of parameter definition, **surface** is extended tangentially. The extended surface is constructed as a zero-offset equidistant surface. Fig. M.5.11.2 shows an extended surface.

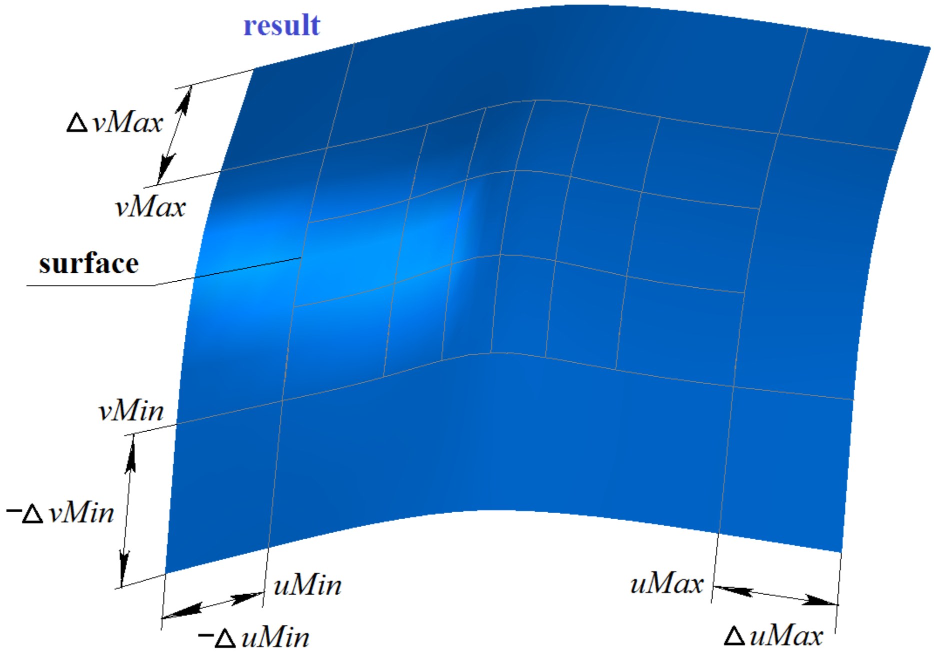


Fig. M.5.11.2.

M.5.12. Construction of a Surface With Arbitrary Borders

The method
MbResultType
BoundedSurface (MbSurface & **surface**,
const RPArray<MbCurve> & **bounds**,
MbSurface *& **result**)
constructs a surface with a given boundary.

The method input parameters are:

- **surface** is an original surface,
- **bounds** is a set of boundaries in the parametric space.

The output parameter of the method is the **result** constructed surface.

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

By default, the surface has rectangular domain of parameter definition. The original domain of parameter definition can be changed by describing the domain boundaries with two-dimensional closed curves **bounds**. If the **bounds** set contains more than one curve, then the first curve of the **bounds** set should describe the external boundary, and all other curves should lie within the first curve of the set. The curves of the **bounds** set can go beyond the original parameter domain of definition, if there are no singular points. If the set of boundaries is empty, then closed compound curve will be constructed based on the original domain of definition of the **surface** parameters. Surfaces with arbitrary boundaries are described in Item O.5.27. MbCurveBoundedSurface Surface with Arbitrary Borders. Fig. M.5.12.1 shows a surface with two closed curves on it, and Fig. M.5.12.2 shows a result of constructing a surface with boundaries shaped like these curves.

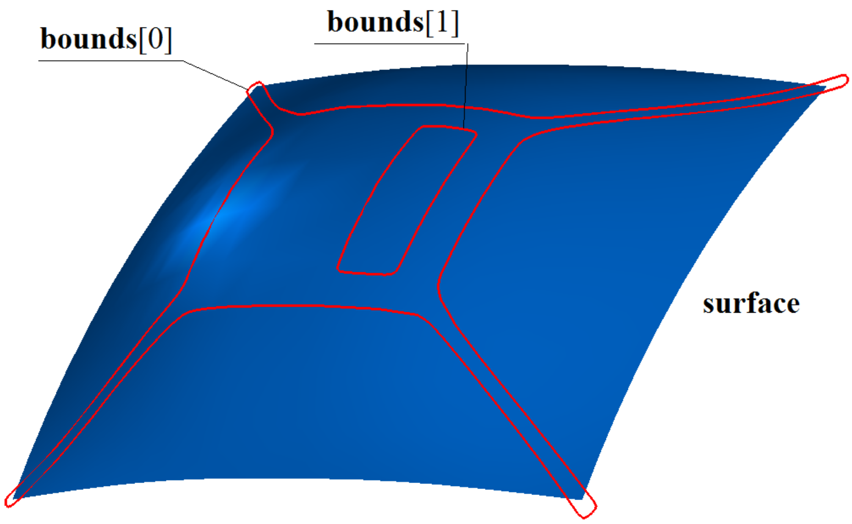


Fig. M.5.12.1.

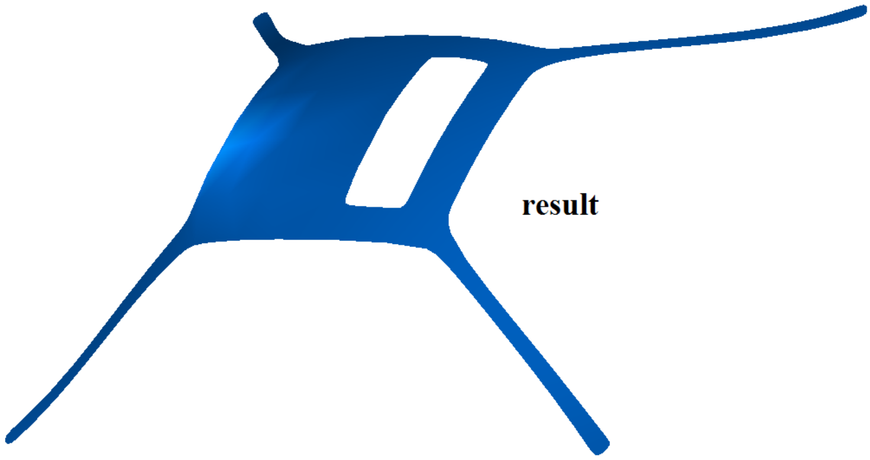


Fig. M.5.12.2.

M.6. DIRECT MODELING METHODS

Direct modeling modifies the geometric model by modifying its components. A geometric model at any stage can be used: it can be a template or a finalized model. Direct modeling methods can modify either all elements of a geometric model or its specific groups. For instance, a body face group can be moved relatively to other body faces, or it can be replaced with equidistant faces or deformable faces. A new body can be constructed from the selected group of faces. The selected fillets or characteristic features can be deleted from the geometric model, for instance, holes or raised portions.

M.6.1. Constructing a Transformed Body

The method
MbResultType
TransformedSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const TransformValues & *params*,
const MbSNameMaker & names,
MbSolid *& **result**)
transforms a copy of the original body using a given matrix.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copying option,
- *params* are transformation parameters,
- names is a face namer.

The output parameter of the method is the **result** constructed body.

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

This method copies the body and scales its copy based on a matrix having equal or different transformation scales for the global coordinate system axes.

The **solid** parameter contains an original body that should be edited. The *sameShell* parameter controls the transfer of unchanged faces, edges, and vertices from the original body **solid** to the constructed body **result**. The *sameShell* parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory* and *cm_Same*. The MbeCopyMode enumeration is declared in the mb_enum.h file and described in Item O.7.9. Copying a Set of Faces.

The *params* parameter contains the transformation matrix *params.***matrix** and the *params*.**fixedPoint**, *params*.*useFixed*, and *params*.*isotropy* values used to calculate the transformation matrix. The TransformValues transformation parameter is declared in the op_shell_parameter.h file.

The names parameter is used for operation versioning.

The transformation matrix is calculated based on the bounding box of the original body **solid** and the offset of one of the bounding box points. The following method is used for calculation:
bool
MbCube::**CalculateMatrix**(size_t *pIndex*,
const MbCartPoint3D & **point**,
const MbCartPoint3D & **fixedPoint**,
bool *useFixed*,
bool *isotropy*,
MbMatrix3D & **matrix**) const,
where *pIndex* is a bounding box point number; **point** is a point in space, with which a point of the bounding box with the *pIndex* number should coincide; **fixedPoint** is a fixed transformation point; *useFixed* is a flag that indicates whether the method uses a fixed point; *isotropy* is a flag that indicates whether the transformation scales are equal. Fig. M.6.1.1 shows how the points of the dimension cube of the **solid** body are numbered. The vertices of the dimension cube are numbered from 0 to 7, the midpoints of the bounding

box edges are numbered from 8 to 19, and the centers of the bounding box faces are numbered from 20 to 25.

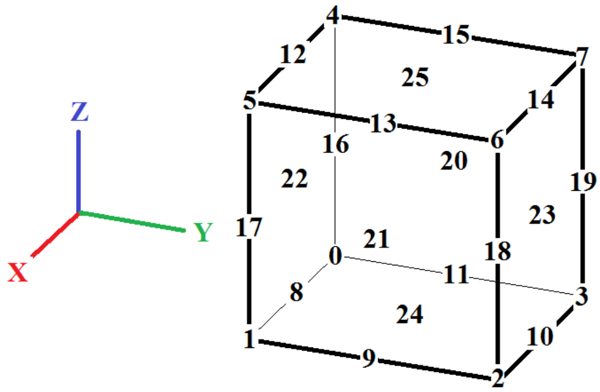


Fig. M.6.1.1.

The **matrix** transformation matrix is calculated based on the deformed cube received by aligning the bounding box point with the *pIndex* number and the **point** in space. If *useFixed*=true, then the **fixedPoint** point will be used as the fixed point for the transformation. If *useFixed*=false, then the bounding box point opposite to the point with the *pIndex* number will be used as the fixed point for the transformation. The following bounding box points are considered to be opposite: 0–6, 1–7, 2–4, 3–5, 8–14, 9–15, 10–12, 11–13, 16–18, 17–19, 20–21, 22–23, 24–25. If *isotropy*=true, then the transformation scales along all axes will be the same and proportional to the bounding box offset vector with the *pIndex* number. If *isotropy*=false, then the transformation scales will be proportional to the projections of bounding box offset vector point with the *pIndex* number on the bounding box sides. The *params*.**fixedPoint**, *params.useFixed*, *params.isotropy* and *params.***matrix** values are used as **fixedPoint**, *useFixed*, *isotropy* and **matrix** parameters of the **CalculateMatrix** method.

Fig. M.6.1.2 shows **solid** an original body, the points of the bounding box and the point in space **point**, with which the specified point of the bounding box should coincide, and Fig. M.6.1.3 shows the **result** transformed body constructed by this method if *useFixed*=false and *isotropy*=false.

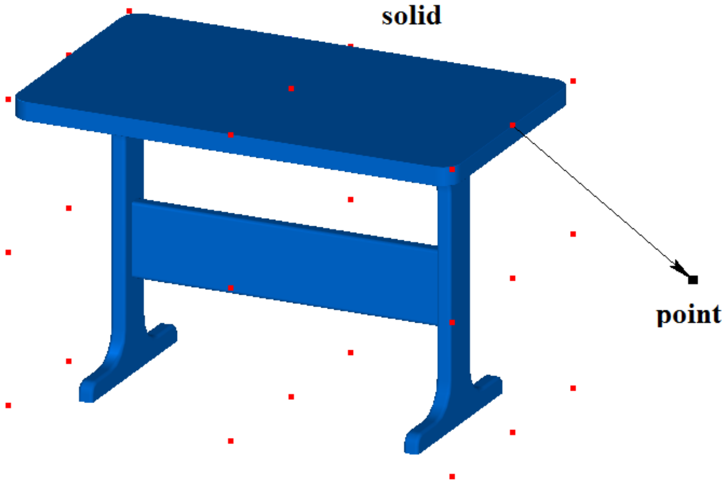


Fig. M.6.1.2.

$$params.\mathbf{matrix} = \begin{bmatrix} 1.35 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.55 & 0 \\ 20.0 & 0 & -35.0 & 1 \end{bmatrix}$$

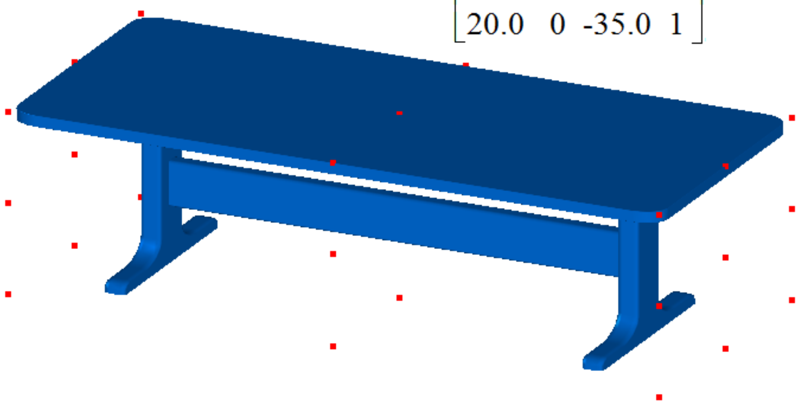


Fig. M.6.1.3.

Transform is a class method, it transforms a body based on the given matrix. This method is loaded for all geometric objects, but in this case the body won't contain any data about the actions performed with it.

The **TransformedSolid** method adds the MbSymmetrySolid constructor in the newly constructed body log containing all data required to execute the operation. The MbTransformedSolid constructor is declared in the cr_transformed_solid.h file.

The Test.exe test application constructs a transformed body using the "Create -> Body -> By direct editing -> Transformation of bounding box" menu command.

M.6.2. Constructing a Modified Body

The method
MbResultType
FaceModifiedSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const ModifyValues & *params*,
const RPArray<MbFace> & **faces**,
const MbSNameMaker & names,
MbSolid *& **result**)
modifies the specified faces of the original body copy using one of the indicated methods.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copying option,
- *params* are modification parameters,
- **faces** are modified body faces,
- names is a face namer.

The output parameter of the method is the **result** constructed body.

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

This method is declared in the action_direct.h file.

This method copies the body and performs one of the following actions: deleting the specified faces from the body copy, creating a new body based on the copies of the specified faces with the environment, moving the specified faces relative to the remaining faces in the body copy, replacing the specified faces in the body copy by equidistant faces, replacing the specified faces in the body copy by deformable faces, or deleting the specified fillet faces in the body copy.

The **solid** parameter contains the original body to be processed. The *sameShell* parameter controls the transfer of unchanged faces, edges, and vertices from the original body **solid** to the constructed body **result**. The *sameShell* parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory* and *cm_Same*. The MbeCopyMode enumeration is declared in the mb_enum.h file and described in Item O.7.9. Copying a Set of Faces.

The *params* parameter describes the modification method defined by the *params.way* type, and the modification value defined by the *params*.**direction** vector. The *params.way* modification type can take one of the following seven MbeModifyingType enumeration values: *dmt_Remove*, *dmt_Create*, *dmt_Action*, *dmt_Offset*, *dmt_Fillet*, *dmt_Supple*, *dmt_Purify*. The ModifyValues modification parameters and the MbeModifyingType enumeration are declared in op_shell_parameter.h file.

The **faces** parameter contains the set of **solid** body faces, which will be somehow modified by this method in the copy of the body.

The names parameter is used to name new faces, edges, vertexes, and for operation versioning.

If *params.way=dmt_Remove*, then this method removes **faces** from the copy of the **solid** body and uses the environment to process the places where the removed faces were located, keeping the body closed. If **faces** are connected with other faces by fillets, then the fillet faces are added to **faces**. If the **faces** set is empty, then this method removes the cylindric and flat faces, the radius of which is less or equal to the *params*.**direction** vector length, that permits to delete small through or blind holes without a need to enumerate them. Fig. M.6.2.1 shows the original body, and Fig. M.6.2.2 shows the constructed body formed by removing the specified faces of the original body.

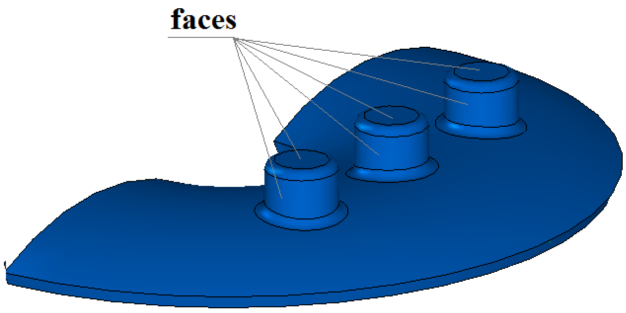


Fig. M.6.2.1.

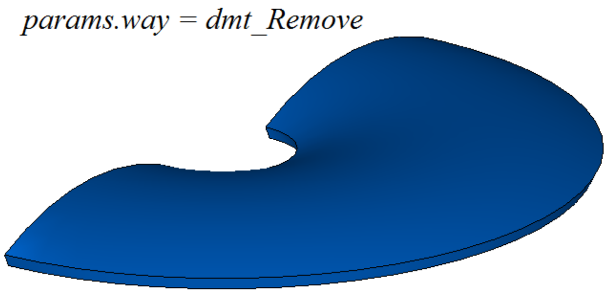


Fig. M.6.2.2.

If *params.way=dmt_Create*, then this method creates a new body from the copies of **faces.** If **faces** are connected by fillets with other faces, then the fillets are removed first, and after that a new body is constructed from **faces**. The new body will be closed, because it will contain **faces** and the faces constructed on the basis of the surfaces adjacent to **faces**. The additional faces are required to remove the border edges. Fig. M.6.2.3 shows the original body, and Fig. M.6.2.4 shows the newly constructed body based on the specified faces of the original body. The body shown on Fig. M.6.2.4 contains five faces: three faces from the **faces** set and two faces based on the cylindrical surface, on which two removed fillets were based.

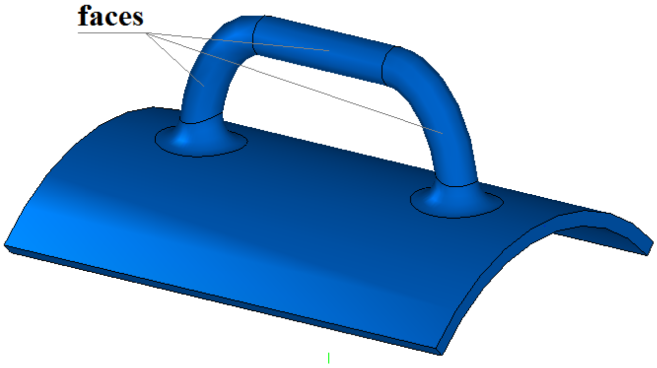


Fig. M.6.2.3.

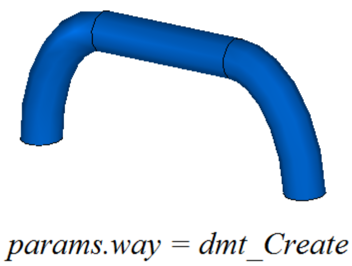


Fig. M.6.2.4.

If *params.way=dmt_Action*, then the constructed method moves the **faces** relative to other faces in the copy of the **solid** body. They are displaced in the direction of *params*.**direction** vector by its length. If all **faces** are connected with other faces by fillets, then the fillets are removed first and then **faces** are moved and after that the fillets are restored in new positions. Fig. M.6.2.5 and Fig. M.6.2.6 show the bodies constructed by moving the specified faces of the original bodies shown in Fig. M.6.2.1 and Fig. M.6.2.3 correspondingly.

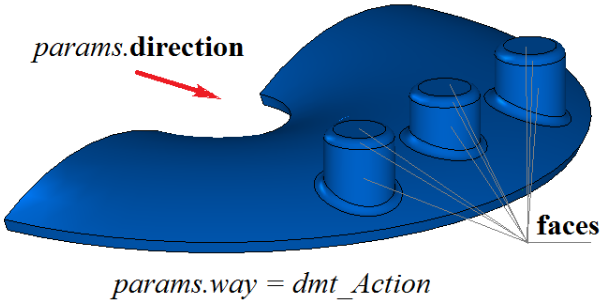


Fig. M.6.2.5.

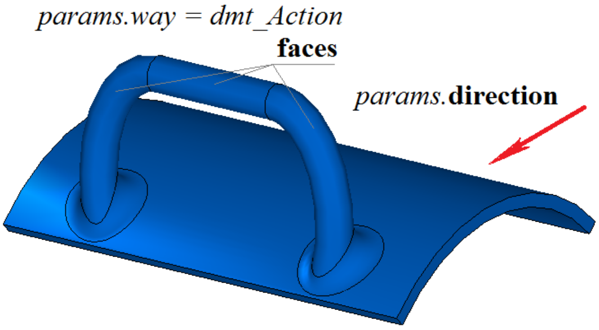


Fig. M.6.2.6.

If *params.way=dmt_Offset*, then this method replaces the **faces** in the **solid** body copy by equidistant faces. The equidistant faces are moved by the *params*.**direction** vector length. If the **faces** are connected with other faces by fillets, then the fillets are removed first, then the **faces** are replaced with equidistant faces, and after that the fillets are recovered in new places. Fig. M.6.2.7 and Fig. M.6.2.8 show bodies constructed by replacing the given faces with equidistant faces. The original bodies are shown in Fig. M.6.2.1 and Fig. M.6.2.3 correspondingly.

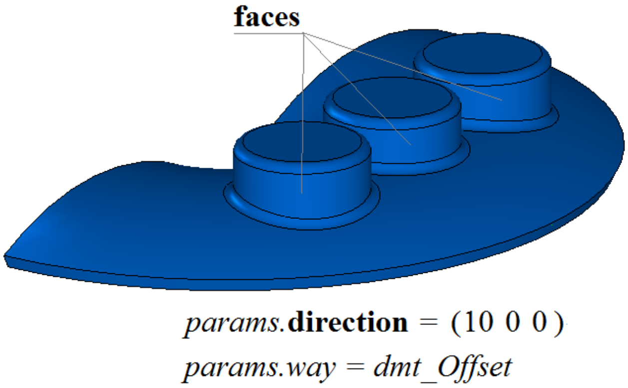


Fig. M.6.2.7.

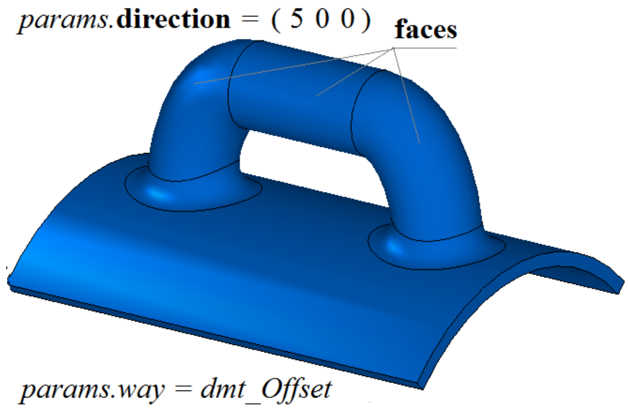


Fig. M.6.2.8.

If one of the *params*.**direction** vector components is negative, then the equidistant faces are displaced to a negative distance. Fig. M.6.2.9 and Fig. M.6.2.10 show bodies constructed by replacing the given faces by equidistant faces using faces with a negative distance. The original bodies are shown in Fig. M.6.2.5 and Fig. M.6.2.6 correspondingly.

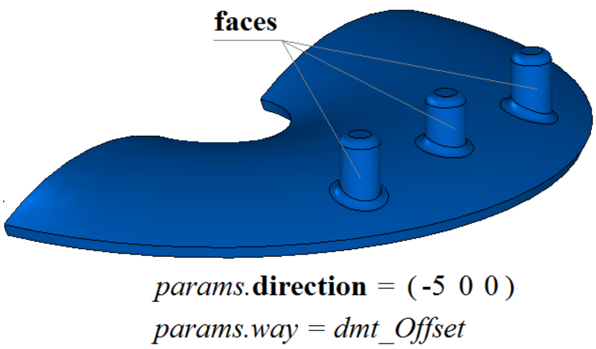


Fig. M.6.2.9.

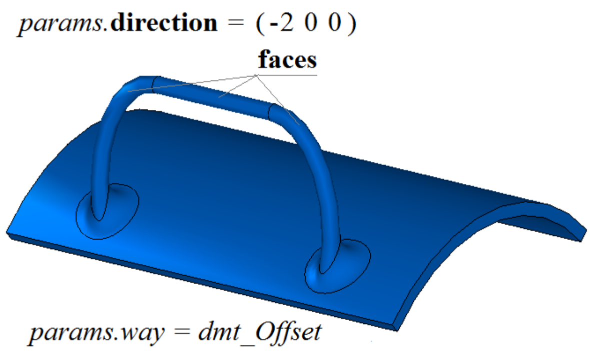


Fig. M.6.2.10.

If *params.way=dmt_Fillet*, then this method changes the fillet radii for the **faces** in the **solid** body copy. The fillet face radius is increased by the length of the *params*.**direction** vector. If one of *params*.**direction** vector components is negative, then the radius of the fillet faces is reduced by the length of the *params*.**direction** vector. Fig. M.6.2.11 and Fig. M.6.2.12 show the bodies constructed by changing the fillet radius for the specified faces. The original bodies are shown in Fig. M.6.2.1 and Fig. M.6.2.3 correspondingly.

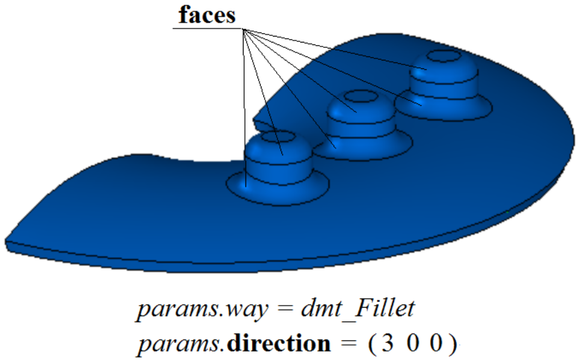


Fig. M.6.2.11.

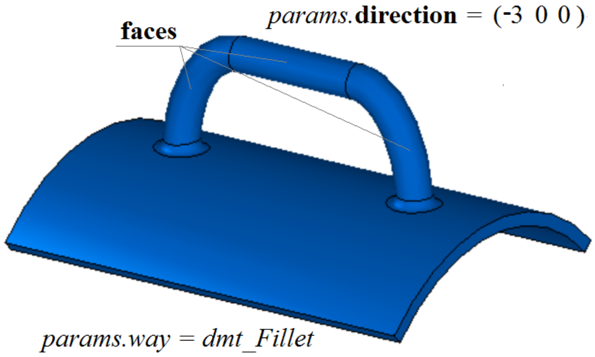


Fig. M.6.2.12.

If *params.way=dmt_Supple*, then this method replaces **faces** in the **solid** body copy by deformable faces in order to enable further editing of these faces. If **faces** are connected with other faces by fillets, then the fillets are removed first, then the **faces** are replaced with deformable faces, and after that the fillets are restored in new positions. Fig. M.6.2.13 shows the original body, and Fig. M.6.2.14 shows the body constructed by replacing a specified face of the original body with a deformable face, and also the control points of the deformable face. The deformable face is constructed based on a NURBS surface.

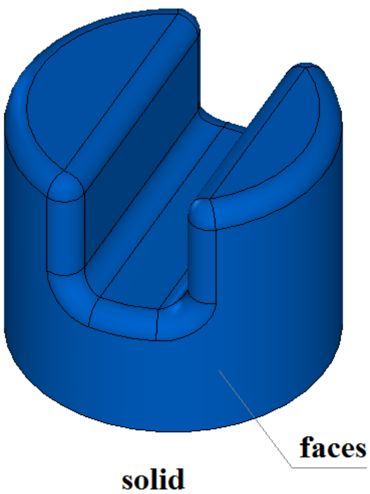


Fig. M.6.2.13.

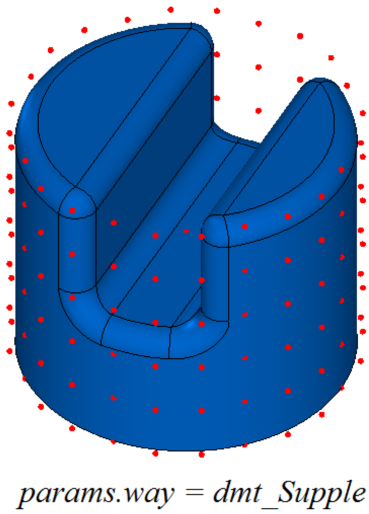
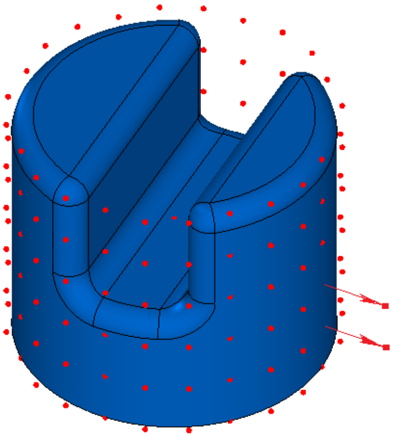
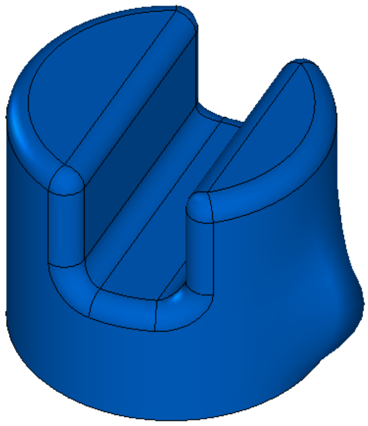


Fig. M.6.2.14.

Fig. M.6.2.15 shows a deformable body with displaced control points of the deformable face. Fig. M.6.2.16 shows the result of body deformation.



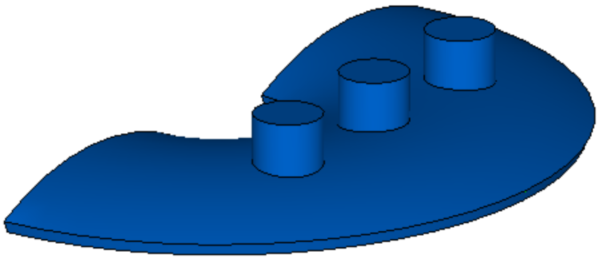
params.way = dmt_Supple



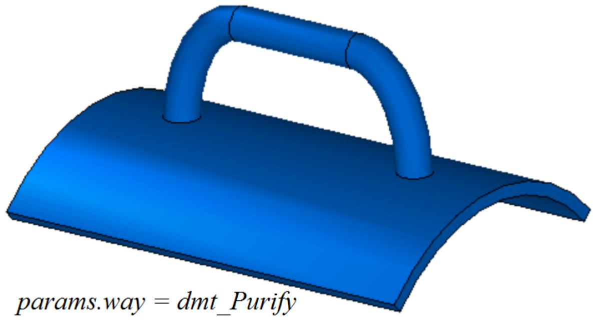
result

Fig. M.6.2.15. *Fig. M.6.2.16.*

If *params.way=dmt_Purify*, then this method removes fillet **faces** of in the **solid** body copy, and uses the environment to process the places where the removed faces were located, keeping the body closed. Fig. M.6.2.17 and Fig. M.6.2.18 show the bodies constructed by removing the specified fillet faces. The original bodies are shown in Fig. M.6.2.1 and Fig. M.6.2.3 correspondingly.



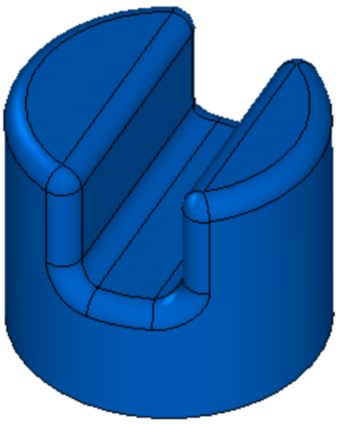
params.way = dmt_Purify



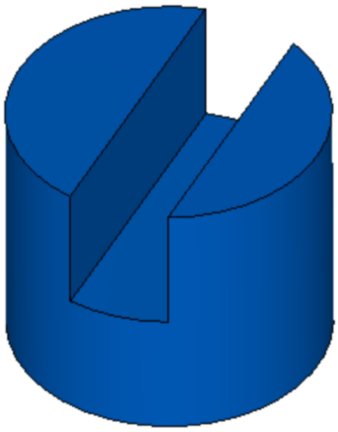
params.way = dmt_Purify

Fig. M.6.2.17. *Fig. M.6.2.18.*

If the **faces** set is empty, then this method removes fillet faces if the corresponding radius is less then or equal to the *params*.**direction** vector length. Fig. M.6.2.19 shows the original body, and M.6.2.20 shows the body constructed by removing all fillet faces, the radius of which is less then or equal to |*params*.**direction**|.



solid



params.way = dmt_Purify

Fig. M.6.2.19. *Fig. M.6.2.20.*

The **FaceModifiedSolid** method works if the topology of the modified body is not changed.

The **FaceModifiedSolid** method adds the MbSymmetrySolid constructor in a newly constructed body log that contains all data required to execute the operation. The MbFaceModifiedSolid constructor is declared in the cr_modified_solid.h file.

The test.exe test application creates a modified body using the following menu commands: "Create->Body->By direct editing->Removal of Faces", "Create->Body->By direct editing->Body creation by faces", "Create->Body->By direct editing->Translation of faces", "Create->Body->By direct editing->Replace by offset faces", "Create->Body->By direct editing->Fillets modification", "Create->Body->By direct editing->Deformation of faces", "Create->Body->By direct editing->Deletion of fillets".

M.6.3. Constructing a Deformable Body

Several operations are performed during deformable body construction: the surfaces of specified faces are replaced with deformable surfaces, the control points of the deformable surfaces are received, the control points are moved, and the deformable surfaces are initialized by new control points. The methods described below are used to perform the above operations.

The method

MbResultType

ModifiedNurbsItem (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const NurbsValues & *params*,
const RPArray<MbFace> & **faces**,
const MbSNameMaker & names,
MbSolid *& **result**)

replaces the specified faces of the body copy with deformable faces for further editing.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copying option,
- *params* are transformation parameters,
- **faces** are replaced body faces,
- names is a face namer.

The output parameter of the method is the **result** constructed body.

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

This method is declared in the action_direct.h file.

This method copies the **solid** body and prepares the body copy for further deformation of the specified faces of the original body.

The **solid** parameter contains an original body, its specified faces will be replaced by deformable faces based on NURBS surfaces. The *sameShell* parameter controls the transfer of unchanged faces, edges, and vertices from the original body **solid** to the constructed body **result**. The *sameShell* parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory* and *cm_Same*. The MbeCopyMode enumeration is declared in the mb_enum.h file and described in Item O.7.9. Copying a Set of Faces.

The *params* parameter contains MbNurbsParameters **uParameters** and MbNurbsParameters **vParameters** data required to construct NURBS surfaces, see Fig. M.6.3.1. MbNurbsParameters **uParameters** contains *degree*, the number of control points (pointsCount), the modified region (region), and the **knots** vector for the first parameter of the NURBS surfaces. MbNurbsParameters **vParameters** contains *degree*, the number of control points (pointsCount), the modified region (region), and the **knots** vector for the second parameter of the NURBS surfaces. The **uParameters**.*useApprox* and **vParameters**.*useApprox* permit to construct deformable surfaces, which approximate the original face surfaces in their initial state.

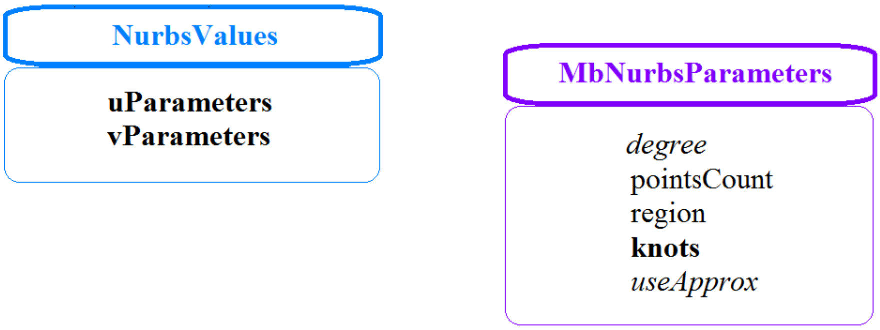


Fig. M.6.3.1.

The **faces** parameter contains the set of the **solid** body faces, which will be replaced by deformable faces based on NURBS surfaces in **result** constructed body.

The names parameter is used to name new faces, edges, vertexes, and for operation versioning.

The method

MbResultType
ModifiedNurbsItem (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const NurbsValues & *params*,
const MbFace & **face**,
const MbSNameMaker & names,
MbSolid *& **result**)

has the same name as the previous method and it replaces one face of the body with a deformable face for further editing. This method differs from the previous one by the fourth parameter, which contains one replaced **face** instead of a set of faces. This method is declared in the action_direct.h file.

The constructed **result** body can look like the original **solid** body, but unlike the original body it can provide NURBS surfaces or control points of NURBS surfaces for modification. The modified control points and their weights can be later returned to the corresponding face surfaces, thereby changing the face shape. Fig. M.6.3.2 shows an original cylindrical body **solid**, its side face is a cylinder, and Fig. M.6.3.3 shows the **result** body with a side face based on a NURBS surface. NURBS surface control points are also shown in Fig. M.6.3.3.

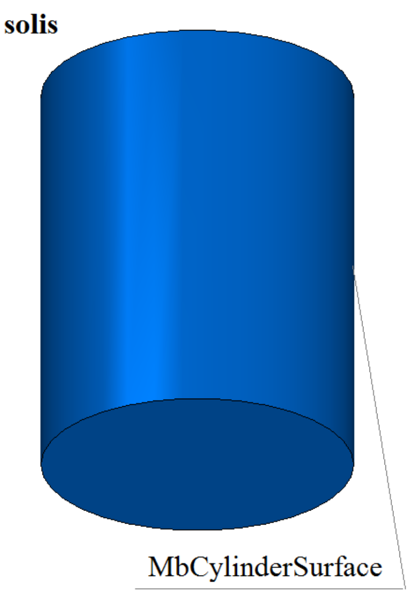


Fig. M.6.3.2.

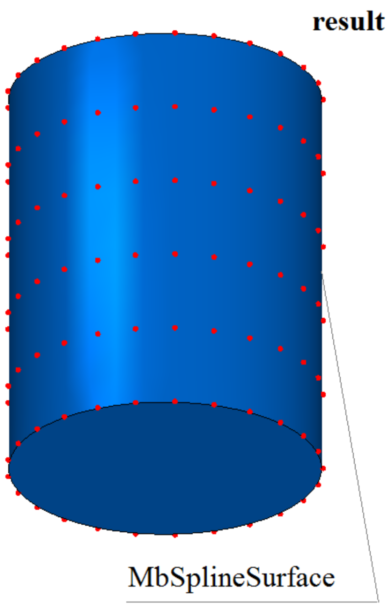


Fig. M.6.3.3.

The **ModifiedNurbsItem** methods add MbModifiedNurbsItem constructor in the newly constructed body log that contains all data required to execute the operation. The MbModifiedNurbsItem constructor is declared in the cr_modified_nurns_.h file.

The method

MbSurface *

GetControlSurface (const MbFace & **face**)

copies the NURBS surface of **face** for further editing.

The original **face** is an input parameter of the method.

When successful, the method returns a pointer to the **face** surface copy, otherwise the method returns null.

This method is declared in the action_direct.h file.

The method returns a non-zero pointer to the surface if **face** is based on a NURBS surface. This requirement will be met after successful execution of **ModifiedNurbsItem** methods described in the previous item. The method returns a pointer to the copy of the base **face** surface, instead of the base surface itself. Later you can modify the set of control points and their weights and then replace the body face surface with the modified surface.

The method

MbResultType

FaceControlPoints (const MbFace & **face**,
Array2<MbCartPoint3D> & **controlPoints**,
Array2<double> & *weights*)

returns the set of control points of the face surface points and the sets of their weights.

The method input parameters are:

- **face** is an original surface.

Output parameters of the method are as follows:

- **controlPoints** are face control points,
- *weights* are control point weights.

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

This method is declared in the action_direct.h file.

The method returns rt_Success if **face** is based on a NURBS surface. You can change the shape of the surface by changing the position of **controlPoints** and the *weights* of the control points. This method should be used after successful execution of **ModifiedNurbsItem** methods described above.

The **controlPoints** are passed as a rectangular matrix, its rows correspond to the first NURBS surface parameter, and its columns correspond to the second NURBS surface parameter. It means that one row of the **controlPoints** rectangular matrix contains the points lying along the coordinate line of the surface with the fixed second NURBS surface parameter, and one column of the **controlPoints** rectangular matrix contains the points lying along the coordinate line of the surface with the fixed first NURBS surface parameter. Fig. M.6.3.4 shows the control points and the coordinate lines of the surface on the background.

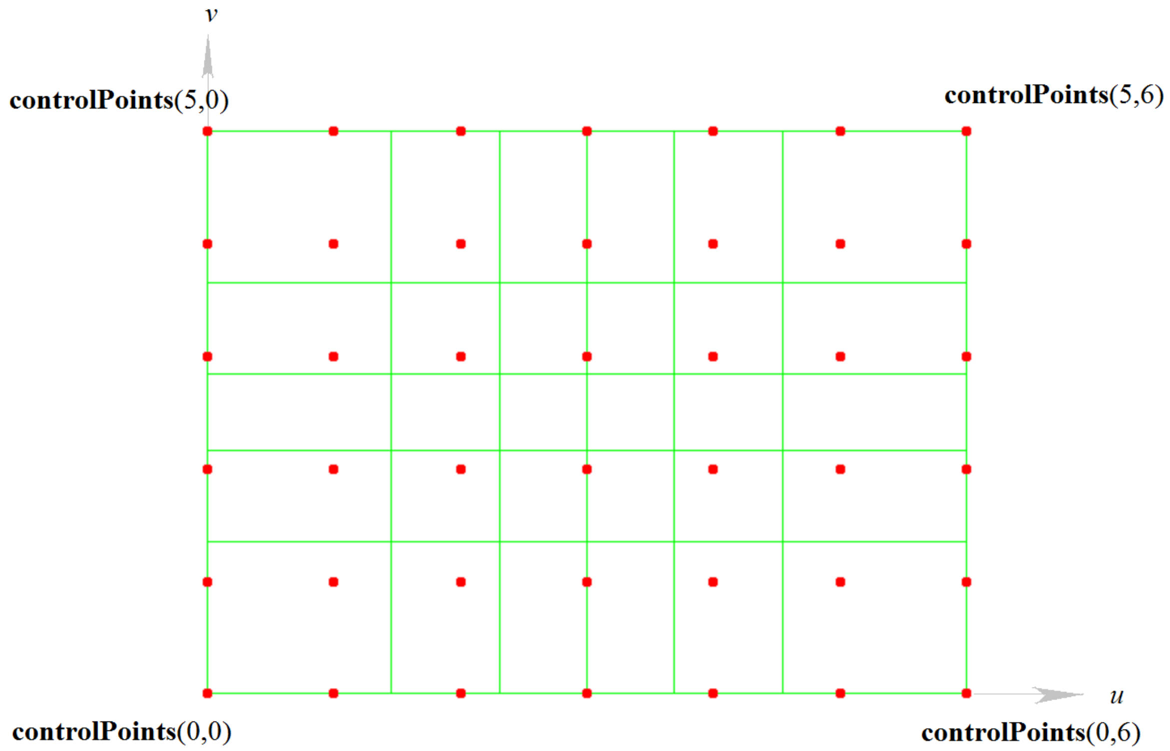


Fig. M.6.3.4.

The *weights* rectangular matrix corresponds to **controlPoints** rectangular matrix.

Fig. M.6.3.5 shows **face** and the **controlPoints** of its surface, and Fig. M.6.3.6 shows the edited **controlPoints** for further modification of **face**.

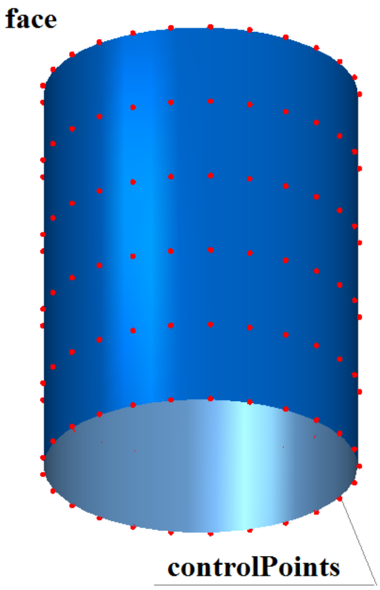


Fig. M.6.3.5.

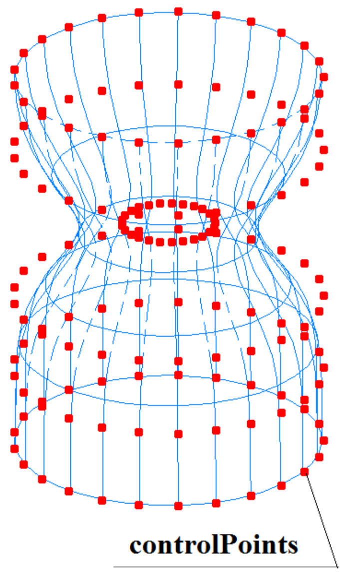


Fig. M.6.3.6.

The method

MbResultType

NurbsModification (MbSolid & **solid**,

MbeCopyMode *sameShell*,

MbFace * **face**,
MbSurface & **faceSurface**,
Array2<bool> & *fixedPoints*,
const MbSNameMaker & names,
MbSolid *& **result**)

deforms the given face of the original body copy by adding the input surface in the copy of the modified surface.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copying option,
- **face** is a body face to be modified,
- **faceSurface** is a surface of the face to be modified,
- *fixedPoints* is a mask with fixed control points of the surface,
- names is a face namer.

The output parameter of the method is the **result** constructed body.

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

This method is declared in the action_direct.h file.

This method copies the **solid** body and adds received **faceSurface** surface in the copy of **face** of the **solid** body.

The **solid** parameter contains the original body. The *sameShell* parameter controls the transfer of unchanged faces, edges, and vertices from the original body **solid** to the constructed body **result**. The *sameShell* parameter can take one of the following four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory* and *cm_Same*. The MbeCopyMode enumeration is declared in the mb_enum.h file and described in Item O.7.9. Copying a Set of Faces.

The **faceSurface** should be a NURBS surface. The *fixedPoints* parameter is a rectangular matrix, its rows and columns correspond to the rows and columns of the control point matrix for the **faceSurface** NURBS surface. The matrix elements with true value determine the fixed control points of the NURBS surface.

The names parameter is used to name new faces, edges, vertexes, and for operation versioning.

The method

MbResultType
NurbsModification (MbSolid & **solid**,
MbeCopyMode *sameShell*,
MbFace * **face**,
const Array2<MbCartPoint3D> & **controlPoints**,
const Array2<double> & *weights*,
Array2<bool> * *fixedPoints*,
const MbSNameMaker & names,
MbSolid *& **result**)

deforms the specified face of the original body copy based on given control points and their weights.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copying option,
- **face** is a body face to be modified,
- **controlPoints** are given control points of the modified face,
- *weights* are control point weights,
- *fixedPoints* is a mask with fixed control points of the face,
- names is a face namer.

The output parameter of the method is the **result** constructed body.

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

This method is declared in the action_direct.h file.

This method should be used after successful execution of **ModifiedNurbsItem**, **GetControlSurface** and **FaceControlPoints** methods described above. The **ModifiedNurbsItem** methods prepare the faces to modification. Fig. M.6.3.7 shows the original **solid** cylindrical body, its side face is based on a NURBS surface and contains the control points that should be modified. Fig. M.6.3.8 shows the **result** body with a modified side face. its **controlPoints** are shown in Fig. M.6.3.6.

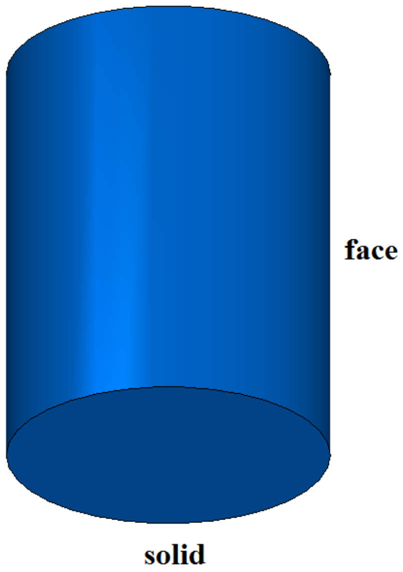


Fig. M.6.3.7.

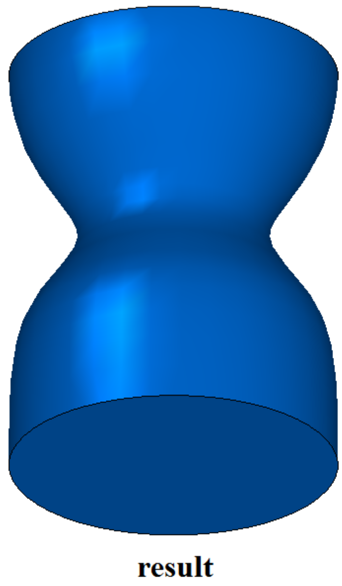


Fig. M.6.3.8.

The **NurbsModification** methods add the MbNurbsModification constructor in the newly constructed body log that contains all data required to execute the operation. The MbNurbsModification constructor is declared in the cr_modified_nurns_.h file.

M.6.4. Constructing a Deformable Prism

The method
MbResultType
NurbsBlockSolid (const MbPlacement3D& **place**,
double *x*, double *y*, double z,
bool *orientation*,
const MbSNameMaker & names,
SimpleName name,
NurbsBlockValues & *params*,
MbSolid *& **result**)
constructs a body shaped as a rectangular parallelepiped with deformable faces.

The method input parameters are:

- **place** is a local coordinate system,
- *x* is a dimension along the X axis,
- *y* is a dimension along the Y axis,
- *z* is a dimension along the Z axis,
- *orientation* is a parameter indicating whether the normals are directed outside of the body (true),
- names is a face namer,
- name is a main name,
- *params* are face construction parameters.

The output parameter of the method is the **result** body constructed from NURBS surfaces.

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

This method is declared in the action_direct.h file.

This method constructs a deformable rectangular prism-shaped body, its dimensions in the **place** local coordinate system are defined by the parameters *x*, *y*, z. All faces of the prism are constructed based on flat NURBS surfaces; surface degree and the number of control points are defined by the *params* parameter. The NurbsBlockValues parameter is declared in the op_shell_parameter_.h file.

Fig. M.6.4.1 shows a **result** prism constructed by the **NurbsBlockSolid** method, and its control points. Fig. M.6.4.2 shows the same prism after deformation by an offset of the face control points.

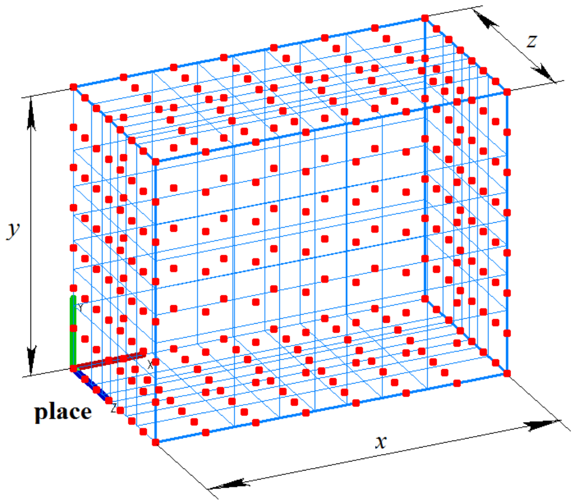


Fig. M.6.4.1.

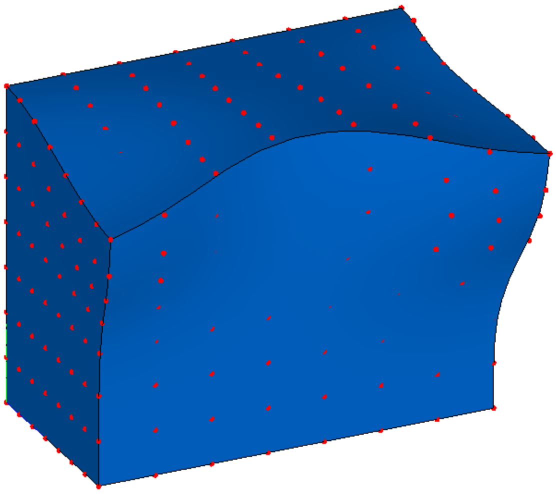


Fig. M.6.4.2.

The **NurbsBlockSolid** method adds the MbNurbsBlockSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbNurbsBlockSolid constructor is declared in the cr_nurbs_block_solid_.h file.

M.6.5. Constructing a Smoothed Surface

The method

MbResultType

SplineSurfaceSmoothing (const MbSplineSurface & **surface**,
size_t *udegree*,
size_t *vdegree*,
MbSplineSurface *& **result**)

smooths a copy of the original surface.

The method input parameters are:

- **surface** is an original surface,
- *udegree* is a parameter defining smoothing by the first surface parameter,
- *vdegree* is a parameter defining smoothing by the second surface parameter.

The output parameter of the method is constructed smoothed surface **result**.

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

This method is declared in the action_direct.h file.

This method smooths the original **surface** without changing its spline degrees and the number of control points. Smoothing is defined by the *udegree* and *vdegree* parameters. They should exceed the corresponding degrees of the original surface splines. The method returns **result**, a smoothed copy of the initial surface.

Fig. M.6.5.1 shows an original NURBS surface. Fig. M.6.5.2 shows a smoothed surface.

Fig. M.6.5.1.

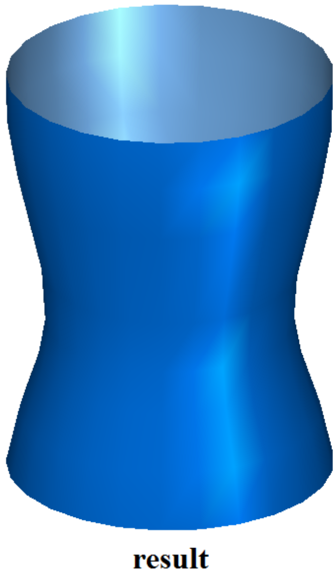


Fig. M.6.5.2.

M.7. SHEET METAL BODY CONSTRUCTION METHODS

The C3D Geometric Kernel supports methods that permit to construct models of sheet metal structures. There are the following sheet body construction methods: plate and shell, as well as the following operations with sheet bodies: bend by edges, bend along a line, incision, bend based on sketch, flat pattern, corner cap, cut, adding a plate, stamp, louver, and bead. There is also a number of service functions for work with sheet bodies. Let's call wide faces of the plate sheet plates, and call all other faces side faces. In this case the sheet bend face with smaller radius will be called the internal face of the bend, and the face with bigger radius will be called external bend face.

M.7.1. Constructing a Sheet Body

The method

MbResultType
CreateSheetSolid (const MbPlacement3D & **placement**,
RPArray<MbContour> & **contours**,
bool *unbended*,
const MbSheetMetalValues & params,
PArray<MbSNameMaker> * names,
PArray<MbSMBendNames> & bends,
MbSolid *& **result**)

constructs a sheet body by extruding flat contours.

The method input parameters are:

- **placement** is a local coordinate system,
- **contours** are a sheet body contours determined in the XY plane of the local coordinate system,
- *unbended* is a flag that indicates whether the body is formed in unbent state,
- params are construction parameters,
- names are namers of constructed faces.
- bends are generated bend parameters.

The output parameters of this method are the names of the sheet faces of the formed bends recorded in bends and the **result** body itself.

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

The method is declared in the action_sheet.h file.

If the contours are closed, then the sheet body is constructed using the given thickness and the given contour. One contour among the closed contours should be external, and it should contain all other internal contours. The cuts will be formed at the internal contours.

If the contours are non-closed, then the sheet body is constructed as follows. Non-closed contours are closed by filleting the angles between the segments using the radii set in params or bends, and closed contours formed as a result are extruded to the given distances. Then the formed sheet body contours are thickened in some direction. Thickening direction is specified in the params operation parameters. If the contour already contains an arc, then it should smoothly mate with adjacent segments. It is possible to construct a sheet body using bends for non-closed contours only. The *unbended* flag is valid only in this case. If the bends array of bend parameters is empty, then the bends are formed using unified parameters set in params. In this case, the names of sheet face parameters of the formed bends are not recorded at all. If bends parameter array is filled, then the number of elements in this array should be equal to the number of formed bends.

Bend parameters are determined as follows. Each contour in the **contours** array corresponds to a namer in the names array, and each namer contains an array of contour segment names. If the bend is formed in the place where two segments join, then the bend parameters are taken from the element of the bends array that contains the names of these segments in the segName1 and segName2 fields. If the bend is formed in the place of an arc, then its parameters are taken from the bends array element that has the arc name in the segName1, and segName2 is SYMPLENAME_MAX.

Fig. M.7.1.1 shows a sheet body formed based on several closed sketches..

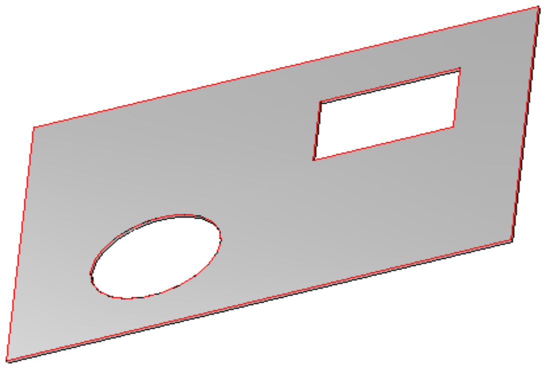


Fig. M.7.1.1

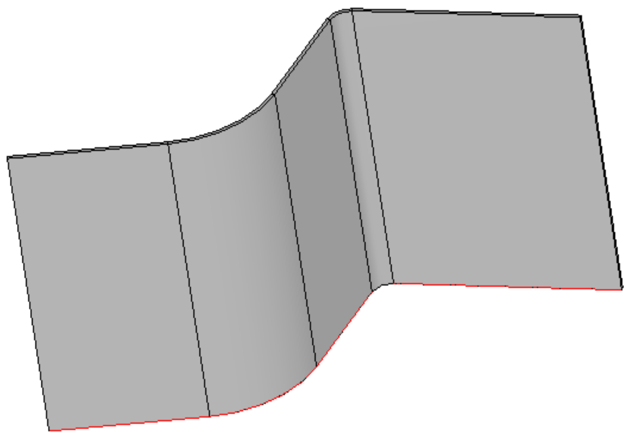


Fig. M.7.1.2

Fig. M.7.1.2 shows a sheet body formed based on a non-closed sketch containing an arc and a pair of segments joined at a certain angle.

The **CreateSheetSolid** method adds the MbSheetMetalSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbSheetMetalSolid constructor is declared in the cr_sheet_metal_solid.h file.

M.7.2. Constructing a Shell

The method

MbResultType

CreateRuledSolid (const MbRuledSolidValues & params,
const MbSNameMaker & names,
PArray<MbSMBendNames> & bends,
MbContour *& **resultContour**,
MbSolid *& **result**)

constructs a shell.

The method input parameters are:

- params are construction parameters,
- names is a namer of constructed faces,
- bends are generated bend parameters.

The output parameters of this method are the names of the sheet faces of the constructed bends recorded in bends, **resultContour** is the params.**contour**1 contour filleted according to the given parameters, and the **result** body itself. This operation also fills the contour splitting parameter arrays, if they were empty.

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

The method is declared in the action_sheet.h file.

A shell is constructed based on one or two flat contours.

If only one contour is used, then the shell is formed by extruding the contour filleted according to contour parameters with a possible slope, and then by thickening the formed surface. If the contour is closed, then a gap is created in the place specified by the operation parameters. If the shell is constructed with a slope, then it is possible to create conical bends or cylindric bends with constant radii in the places where the original contour was filleted. It is also possible to segment the contour arcs, in this case an arc is replaced by a set of segments that approximate it.

If the shell is formed based on two contours, then in general case the contour segments are joined by ruled surfaces, which are then thickened. In order to avoid surface twisting, the contours are automatically or manually split to smaller segments. Similar to construction based on one sketch, a gap is created in the place specified by the parameters to make the shell unbending feasible.

Fig. M.7.2.1 shows a shell constructed without keeping the bend radius.

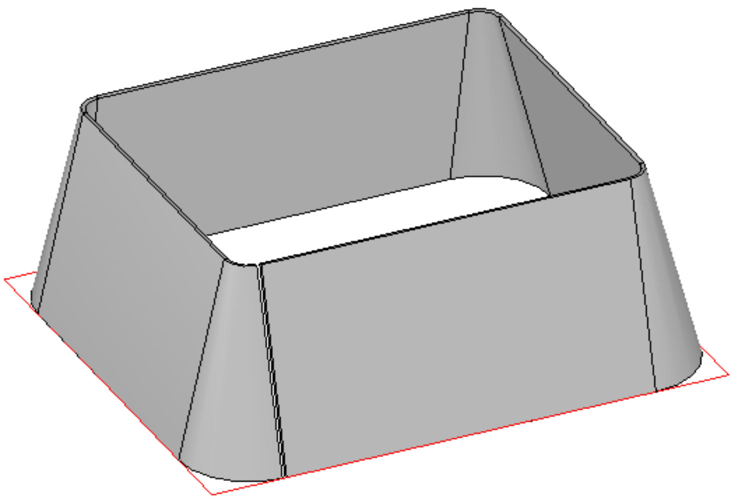


Fig. M.7.2.1

Fig. M.7.2.2 shows a shell constructed keeping the bend radius.

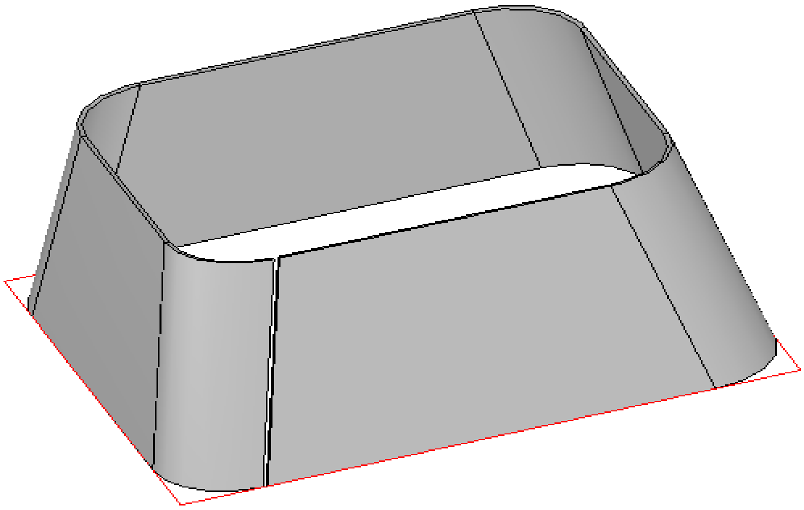


Fig. M.7.2.2

Fig. M.7.2.3 shows a shell constructed with segmentation of the second contour arcs.

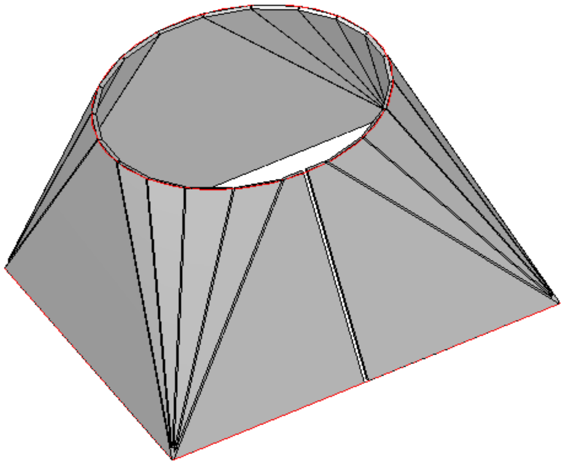


Fig. M.7.2.3

The **CreateRuledSolid** method adds the MbRuledSolid constructor in a newly constructed body log that contains all data required to execute the operation. The MbRuledSolid constructor is declared in the cr_sheet_ruled_solid.h file.

M.7.3. Forming a Sheet Body Bend Along a Line

The method
MbResultType
BendSheetSolidOverSegment (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const RPArray<MbFace> & **bendingFaces**,
MbCurve3D & **curve**,
bool *unbended*,
const MbBendOverSegValues & params,
MbSNameMaker & names,
MbSolid *& **result**)
bends a sheet body along a line lying in a sheet face of the body.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **bendingFaces** are faces that are bent,
- **curve** is a straight-line curve, the body is bent along it,
- *unbended* is a flag that indicates whether the element is formed in an unbent state,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

Straight-line **curve** can be either a segment lying in **bendingFaces** flat faces, or a line. All faces of the **bendingFaces** array should lie in one common plane. The **curve** segment can lie in several flat faces simultaneously, but the bends will be formed only at those faces that were added in the **bendingFaces** array.

Fig. M.7.3.1 shows a sheet body after bend along a line operation.

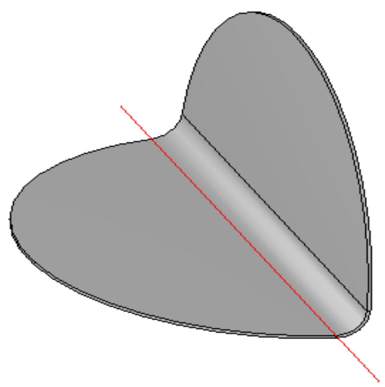


Fig. M.7.3.1

The **BendSheetSolidOverSegment** method adds the MbBendOverSegSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbBendOverSegSolid constructor is declared in the cr_sheet_bend_over_seg_solid.h file.

M.7.4. Constructing a Sheet Body Incision

The method
MbResultType
SheetSolidJog (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const RPArray<MbFace> & **bendingFaces**,
MbCurve3D & **curve**,
bool *unbended*,
const MbJogValues & params,
const MbBendValues & secondBendParams,
MbSNameMaker & names,
RPArray<MbFace> & **firstBendFaces**,
RPArray<MbFace> & **secondBendFaces**,
MbSolid *& **result**)
incises a sheet body along a straight line.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **bendingFaces** are faces that are bent,
- **curve** is a straight-line curve, along it the body will be incised,
- *unbended* is a flag that indicates whether the element is formed in unbent state,
- params are construction parameters,
- secondBendParams are parameters of the second bend,
- names is a namer of constructed faces.

The output parameters of the method are the **result** constructed body and:

- **firstBendFaces** are the sheet faces of the bends adjacent to the fixed part of the base faces,
- **secondBendFaces** are the sheet faces of bends raised over the base faces.

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

The method is declared in the action_sheet.h file.

The incision line can be either a segment lying in **bendingFaces** flat faces, or a straight line. All faces of the **bendingFaces** array should lie in one common plane. The **curve** segment can lie in several flat faces simultaneously, but the bends will be formed only at those faces that were added in the **bendingFaces** array. The incision is formed as two bends along the line displaced with respect to one another. Thus formed sheet faces of the bends are returned in **firstBendFaces** and **secondBendFaces** arrays. Fig. M.7.4.1 shows a sheet body with executed incision operation.

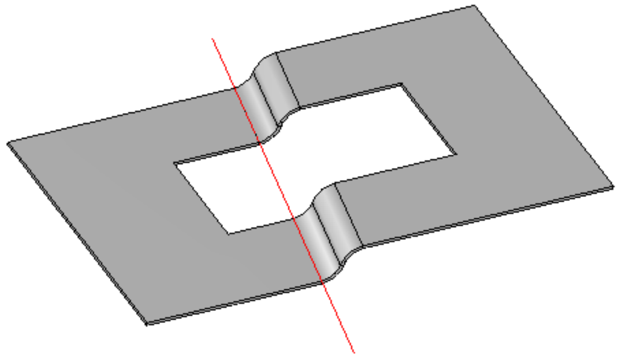


Fig. M.7.4.1

The **SheetSolidJog** method adds the MbJogSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbJogSolid constructor is declared in the cr_stamp_jog_solid.h file.

M.7.5. Bend Unbent Sheet Body

The method
MbResultType
BendSheetSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const RPArray<MbSheetMetalBend> & bends,
const MbFace & **face**,
const MbCartPoint & **point**,
MbSNameMaker & names,
MbSolid *& **result**)
transforms the given bends to a bent state.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- bends is a set of bends, it consists of face pair arrays: internal and external bend faces,
- **face** is a face that remains fixed,
- **point** is a point in the parametric region of the bend **face** surface,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

The method bends unbent bends of the sheet body relative to the fixed **face**. If **face** is a sheet surface belonging to one of the bends, then bend is performed so that only the surface tangential to the surface lying below **face** in **point** remains fixed. As a rule, each element of the bends array contains only one pair of sheet faces, namely the internal and external bend faces. However, if the ruled shell is formed based on two sketches, then it is possible that a chain of such adjacent bends should be bent simultaneously. In that case, the element of the bends array should contain the entire chain of such pairs that correspond to external and internal faces of the compound bend.

Fig. M.7.5.1 shows a sheet body in unbent state.

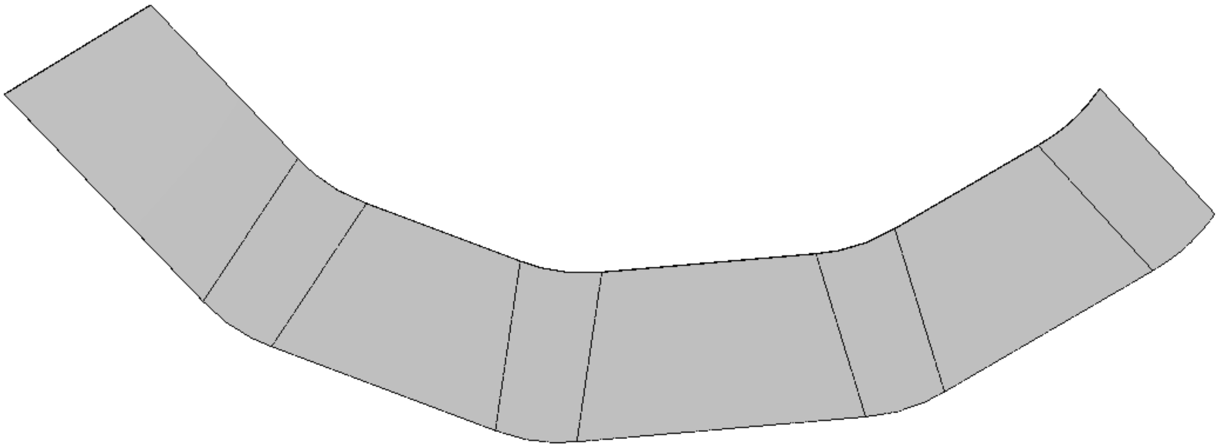


Fig. M.7.5.1

Fig. M.7.5.2 shows a sheet body from Fig. M.7.5.1 after bend operation.

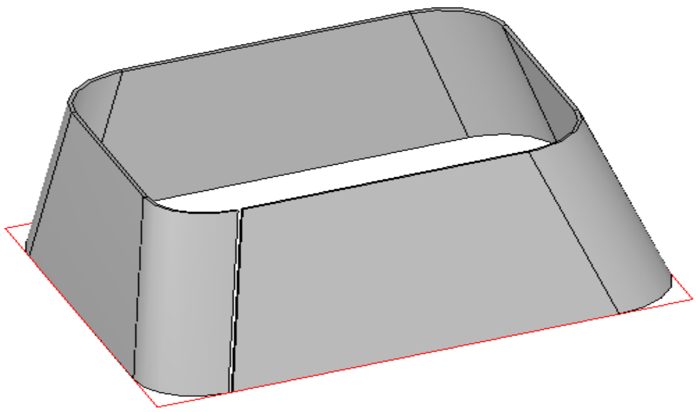


Fig. M.7.5.2

The **BendSheetSolid** method adds the MbBendUnbendSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbBendUnbendSolid constructor is declared in the cr_sheet_bend_unbend_solid.h file.

M.7.6. Unbend Sheet Body Bends

The method
MbResultType

UnbendSheetSolid (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const RPArray<MbSheetMetalBend> & bends,
const MbFace & **face**,
const MbCartPoint & **point**,
MbSNameMaker & names,
MbSolid *& **result**)
unbends the sheet body bends.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- bends is a set of bends containing face pairs: an internal bend face and an external bend face,
- **face** is a face that remains fixed,
- **point** is a point in the parametric region of the bend **face** surface,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

The method unbends the bends of the sheet body relative to the fixed **face**. If the **face** is a sheet surface belonging to one of the bends, then unbend operation is performed so that only the surface tangential to the surface lying under the **face** in **point** remains fixed. As a rule, each element of the bends array contains only one pair of sheet faces, namely the internal bend face and the external bend face. However, if a ruled shell is formed based on two sketches, then it is possible that a chain of such adjacent bends can only be unbent simultaneously. In that case, the element of the bends array should contain the whole chain of such pairs of corresponding external and internal faces of the compound bend.

Fig. M.7.6.1 shows a sheet body in bent state.

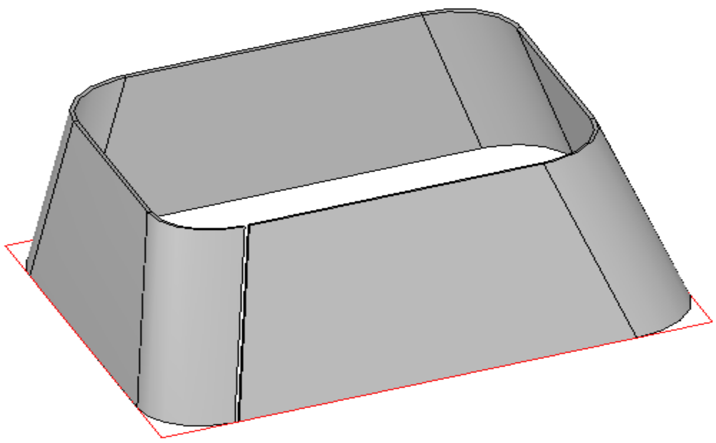


Fig. M.7.6.1

Fig. M.7.6.2 shows the sheet body from Fig. M.7.6.1 after unbend operation.

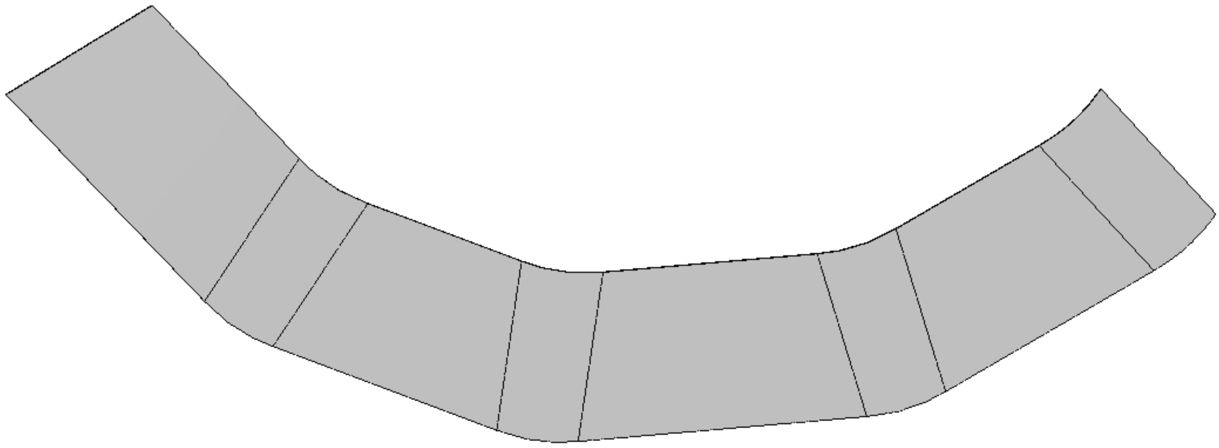


Fig. M.7.6.2

The **UnbendSheetSolid** method adds the MbBendUnbendSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbBendUnbendSolid constructor is declared in the cr_sheet_bend_unbend_solid.h file.

M.7.7. Add a Plate to a Sheet Body

The method
MbResultType
SheetSolidPlate (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **placement**,
RPArray<MbContour> & **contours**,
const MbSheetMetalValues & params,
PArray<MbSNameMaker> * names,
MbSolid *& **result**)
adds a plate to a sheet body.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **placement** is a local coordinate system,
- **contours** are plate contours given in the XY plane of the local coordinate system,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

The plate is constructed based on one or several closed non-intersecting contours; multiple external contours are possible.

Fig. M.7.7.1 shows a sheet body after plate operation.

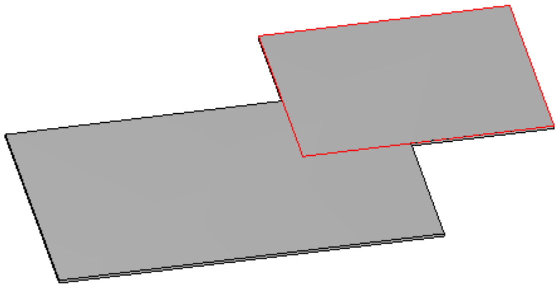


Fig. M.7.7.1

The **SheetSolidPlate** method adds the MbSheetMetalSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbSheetMetalSolid constructor is declared in the cr_sheet_sheet_metal_solid.h file.

M.7.8. Making a Cut in a Sheet Body

The method
MbResultType
SheetSolidHole (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbPlacement3D & **placement**,
RPArray<MbContour> & **contours**,
const MbSheetMetalValues & params,
bool *difference*,
PArray<MbSNameMaker> * names,
MbSolid *& **result**)
cuts holes in a sheet body based on closed contours.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **placement** is a local coordinate system,
- **contours** are sheet body cut/intersection contours given in the XY plane of the local coordinate system,
- params are construction parameters,
- *difference* is a flag that indicates the construction method: hole (true), intersection (false),
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

The method is notable in that besides a simple boolean cut it has a special cut by depth option. In this latter case the cut envelopes all the bends of the sheet body, i.e. the sheet body constructed based on contours using cut parameters with the original body thickness before cutting copies all the bends of the original body that it finds on its way.

Fig. M.7.8.1 shows a sheet body after cut sheet body operation with cut by depth option enabled.

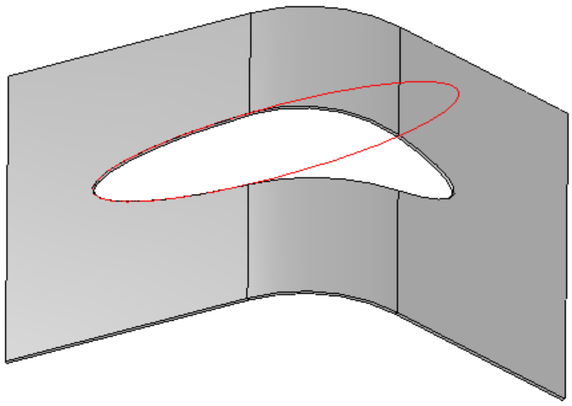


Fig. M.7.8.1

The **SheetSolidHole** method adds the MbSheetMetalSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbSheetMetalSolid constructor is declared in the cr_sheet_sheet_metal_solid.h file.

M.7.9. Construct a Sheet Body Bend at Edges

The method

MbResultType

BendSheetSolidByEdges (MbSolid & **solid**,
const MbeCopyMode *sameShell*,
const RPArray<MbCurveEdge> & **edges**,
const bool *unbended*,
const MbBendByEdgeValues & params,
MbSNameMaker & names,
MbSolid *& **result**)

makes a bend in a sheet body at specified rectangular edges.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **edges** is a set of edges used to form bends,
- *unbended* is a flag that indicates whether the bends are formed in unbent state,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

A bend is formed according to the parameters given for one or several bends belonging to a flat face of the sheet body by joining a bend either with a flat extension of the plate edge or without it. The faces or slopes at side faces can be extended depending on the operation parameters. You can cap the corners for chains of adjacent edges.

Fig. M.7.9.1 shows a sheet body after bend operation.

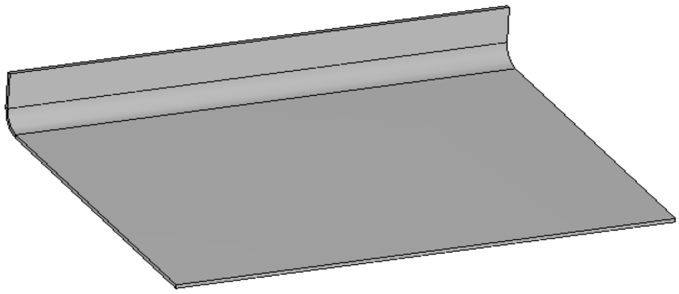


Fig. M.7.9.1

Fig. M.7.9.2 shows a sheet body after bend operation with slope option in the left part and extension option in the right part

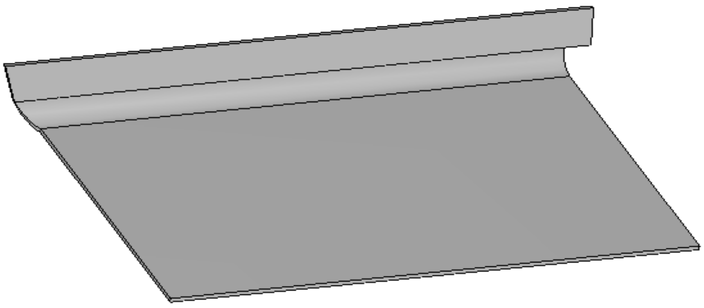


Fig. . M.7.9.2

Fig. M.7.9.3 shows a sheet body with executed bend operation and bend release option.

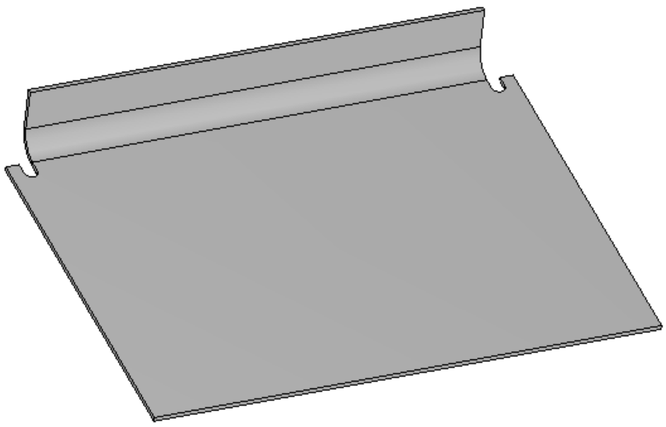


Fig. . M.7.9.3

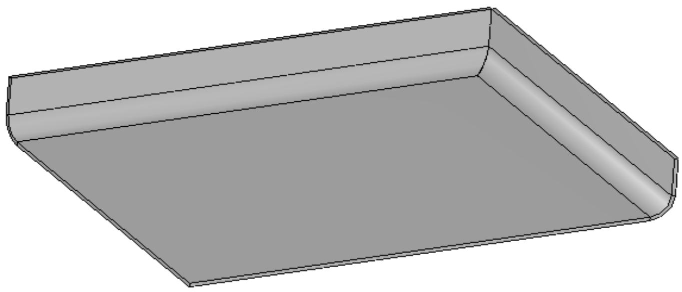


Fig. . M.7.9.4

Fig. M.7.9.4 shows a sheet body after bend operation with multiple edges selected and corner capping option.

The **BendSheetSolidByEdges** method adds the MbBendByEdgeSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbBendByEdgeSolid constructor is declared in the cr_sheet_bend_by_edge_solid.h file.

M.7.10. Forming a Bend Based on a Sketch

The method

MbResultType
SheetSolidJointBend (MbSolid & **solid**,
const MbeCopyMode *sameShell*,
const MbPlacement3D & **placement**,
const MbContour & **contour**,
const RPArray<MbCurveEdge> & **edges**,
const bool *unbended*,
const MbJointBendValues & params,
MbSNameMaker & names,
PArray< PArray<MbSMBendNames> > & **bends**,
MbSolid *& **result**)

forms a sheet body bend based on a sketch.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **placement** is a local coordinate system,
- **contour** is a contour of bends given in the XY plane of the local coordinate system,
- **edges** is a set of edges used to form bends,
- *unbended* is a flag that indicates whether the bends are formed in unbent state,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameters of the method are the **result** constructed body and the constructed **bends**.

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

The method is declared in the action_sheet.h file.

A combined bend of a sheet body (also called a bend based on sketch) can be constructed at one or several adjacent straight-line edges of the same sheet face or at several sheet faces located on the different sides of the same bend. The sketch containing segments and arcs should lie in the plane perpendicular to one of the construction edges. One end of the edge should lie on its projection onto this plane.

This sketch is applied to each edge involved in the construction process. Based on this sketch and its copies, sheet bodies are constructed for all edges, forming bends at arcs and non-smooth joints of straight-line contour segments, such bends are also formed between a contour and a sheet plate if they do not join smoothly. The constructed bodies are combined with the original body, and then the angles are capped according to the given parameters. After execution of this operation, the **bends** set contains all bends created by it.

Fig. M.7.10.1 shows a sheet body after bend based on sketch operation.

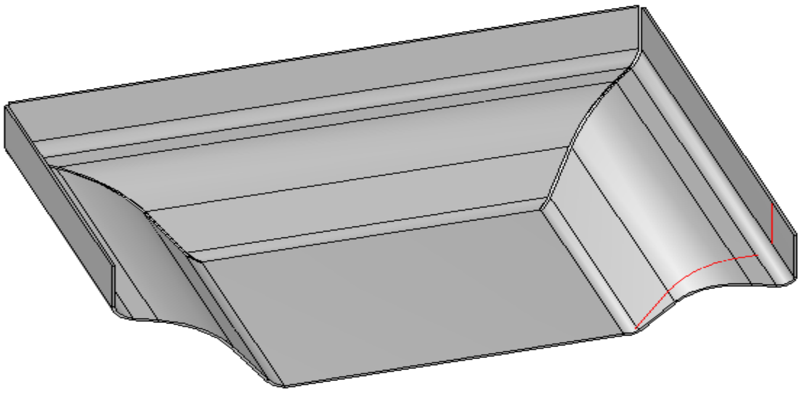


Fig. M.7.10.1

The **SheetSolidJointBend** method adds the MbJointBendSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbJointBendSolid constructor is declared in the cr_sheet_joint_bend_solid.h file.

M.7.11. Cap a Sheet Body Corner

The method
MbResultType
CloseCorner (MbSolid & **solid**,
MbeCopyMode *sameShell*,
MbCurveEdge * **edgePlus**,
MbCurveEdge * **edgeMinus**,
const MbClosedCornerValues & params,
MbSNameMaker & names,
MbSolid *& **result**)
caps a sheet body corner.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **edgePlus** is a bend edge that is conventionally considered positive,
- **edgeMinus** is a bend edge that is conventionally considered negative,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

If two bends are formed on adjacent edges of the sheet body or on the edges separated by a bend, then the angle that is formed between them can be filled with material by extending the corresponding sides of these bends, and that is executed in this operation. Whenever required, trimming can be executed instead of extension. Parameters permit to define the gap size and the capping types individually for the bends and individually for their flat extensions. It is also possible to select several corner processing methods if the corner is capped between the bends formed on the adjacent edges of the same face.

Fig. M.7.11.1 shows a sheet body with two bends after corner cap operation without bend processing.

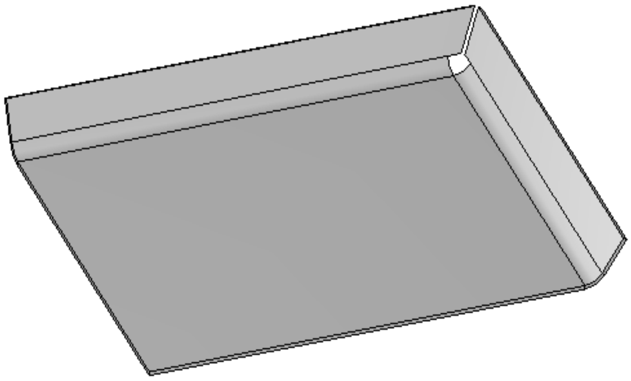


Fig. M.7.11.1

Fig. M.7.11.2 shows a sheet body with two bends after corner cap operation with close capping, joining at the edge and a gap.

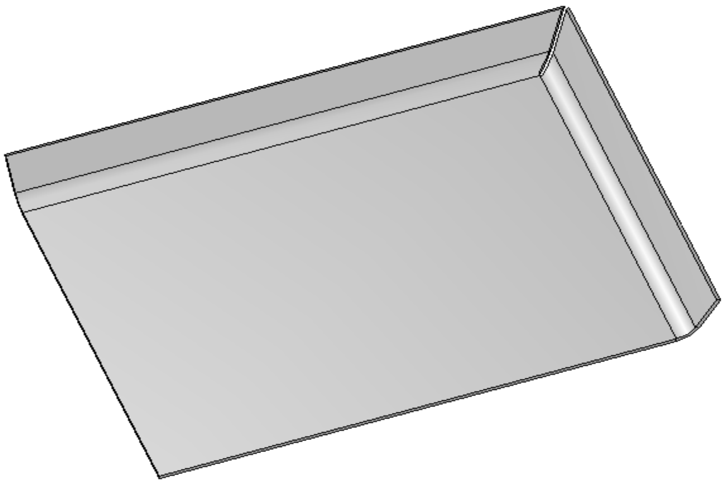


Fig. M.7.11.2

Fig. M.7.11.2 shows a sheet body with two bends after corner cap operation with close capping, a gap, and round corner processing.

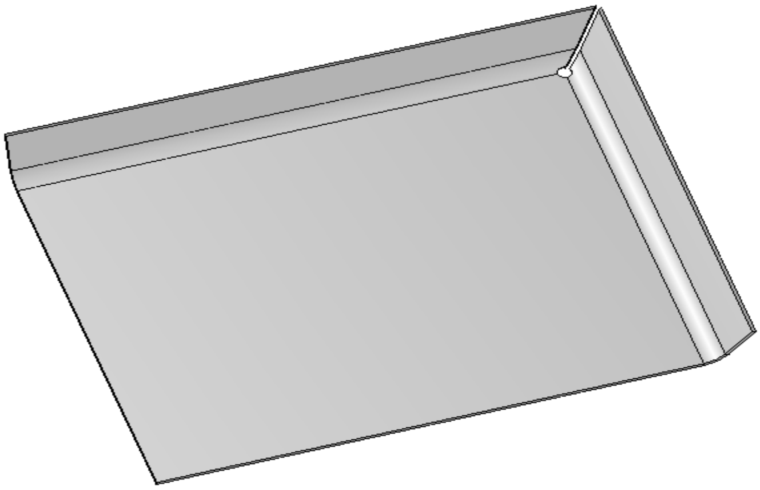


Fig. M.7.11.3

The **CloseCorner** method adds the MbClosedCornerSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbClosedCornerSolid constructor is declared in the cr_sheet_closed_corner_solid.h file.

M.7.12. Construct a Stamped Body

The method
MbResultType
Stamp (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbFace & **face**,
const MbPlacement3D & **placement**,
const MbContour & **contour**,
const MbStampingValues & params,
MbSNameMaker & names,
MbSolid *& **result**)
stamps a given form on the specified face.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **face** is a stamping face,
- **placement** is a local coordinate system,
- **contour** is a stamping contour given in the XY plane of the local coordinate system,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

The stamping is constructed based on one closed or non-closed contour lying on a flat sheet face. A closed sketch may completely or partially lie on a sheet face, and a non-closed sketch should start and end beyond face boundaries. Stamping is trimmed by the boundaries of the sheet face where the sketch is located. The sketch defines the shape of the stamping bottom. The stamping can be open, this depends on the parameters. It means that a sheet plate is punched through along the contour.

Fig. M.7.12.1 shows a sheet body after the closed stamp operation.

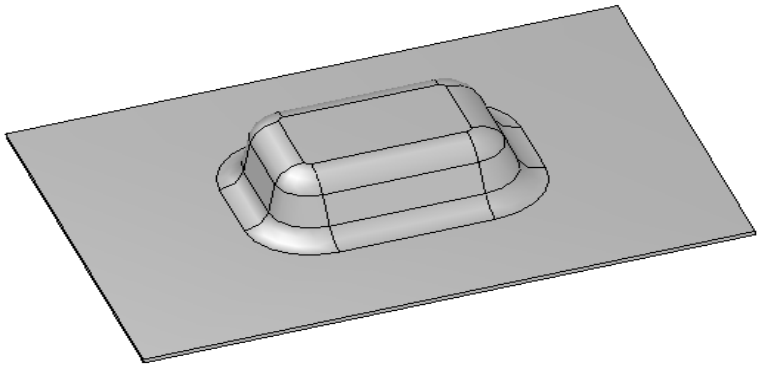


Fig. M.7.12.1

Fig. M.7.12.2 shows a sheet body after open stamp operation.

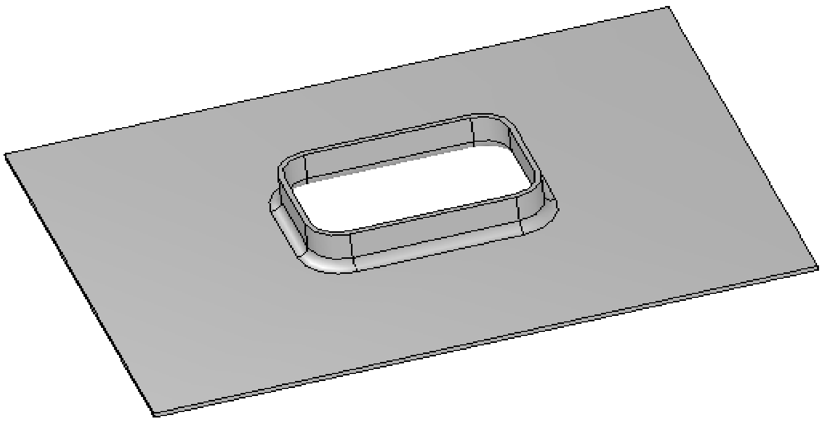


Fig. M.7.12.2

Fig. M.7.12.3 shows a sheet body after closed stamp operation based on a non-closed sketch.

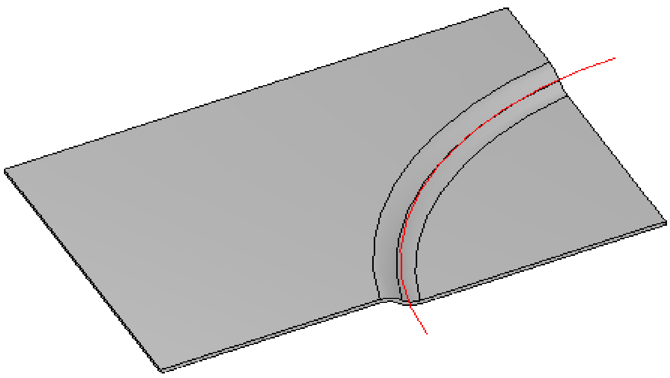


Fig. M.7.12.3

The **Stamp** method adds the MbStampSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbStampSolid constructor is declared in the cr_stamp_solid.h

file.

M.7.13. Construct a Sheet Body Bead

The method
MbResultType
CreateBead (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbFace & **face**,
const MbPlacement3D & **placement**,
const RPArray<MbContour> & **contours**,
const MbBeadValues & params,
MbSNameMaker & names,
MbSolid *& **result**)
constructs a sheet body bead.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **face** is a bead face,
- **placement** is a local coordinate system,
- **contour** are bead contours given in the XY plane of the local coordinate system,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

The bead is constructed along one or several closed or non-closed contours lying on a flat sheet face. If the contour extends beyond the boundaries of this face, then the bead is trimmed by its boundaries. The bead formed along a non-closed contour has ends, their shape is determined by operation parameters.

Fig. M.7.13.1 shows a sheet body after form bead with round section operation with a closed end.

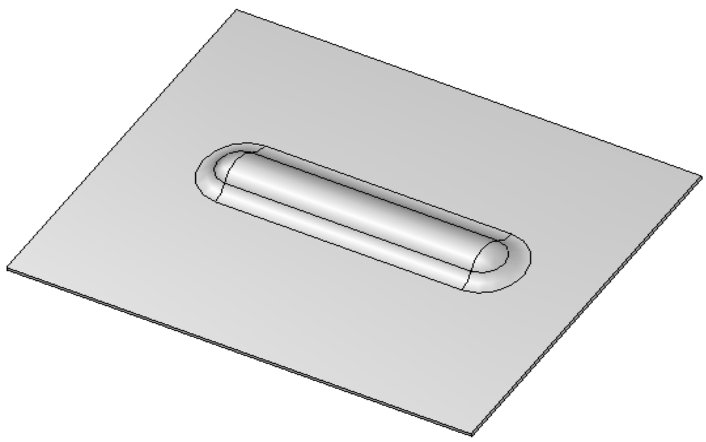


Fig. M.7.13.1

Fig. M.7.13.2 shows a sheet body after form bead operation with U-shaped bead section and an open end.

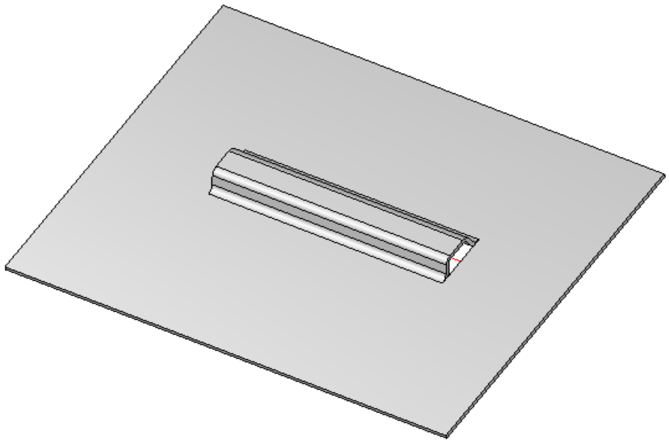


Fig. M.7.13.2

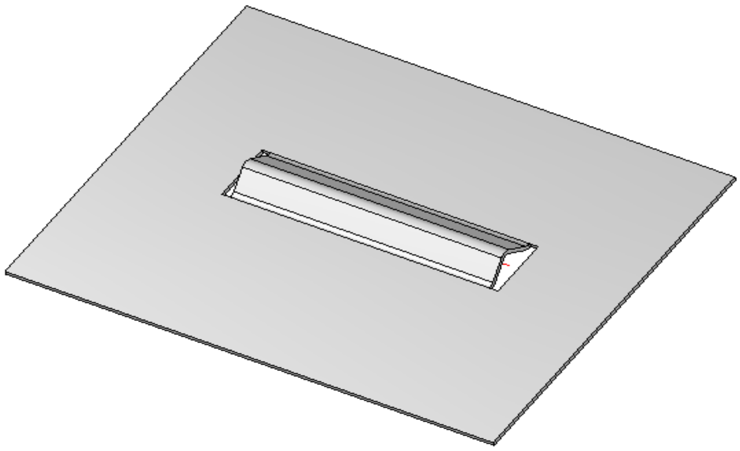


Fig. M.7.13.3

Fig. M.7.13.4 shows a sheet body after form bead operation, the bead has a V-shaped section and a chopped end.

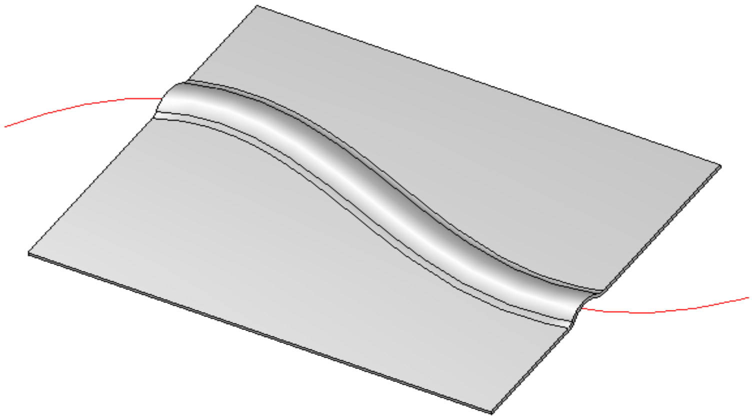


Fig. M.7.13.4

Fig. M.7.13.4 shows a sheet body after form bead operation, the bead has a round section according to the sketch that extends beyond the sheet boundaries.

The **CreateBead** method adds the MbBeadSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbBeadSolid constructor is declared in the cr_stamp_bead_solid.h file.

M.7.14. Construct a Sheet Body Louver

The method
MbResultType
CreateJalousie (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const MbFace & **face**,
const MbPlacement3D & **placement**,
const RPArray<MbLineSegment> & **segments**,
const MbJalousieValues & params,
MbSNameMaker & names,
MbSolid *& **result**)
constructs a sheet body louver.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **face** is a louver face,
- **placement** is a local coordinate system,
- **segments** are louver segments given in the XY plane of the local coordinate system,
- params are construction parameters,
- names is a namer of constructed faces.

The output parameter of the method is the **result** constructed body.

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

The method is declared in the action_sheet.h file.

Louvers are constructed on one or several segments lying on a flat sheet face. Louvers can't go beyond the face border or intersect each other. There are two types of louvers: drawn and trimmed louvers. Drawn louver is a half of an element cut along a straight-line bead, and a trimmed louver looks like a bent plate.

Fig. M.7.14.1 shows a sheet body after louver operation with drawn option.

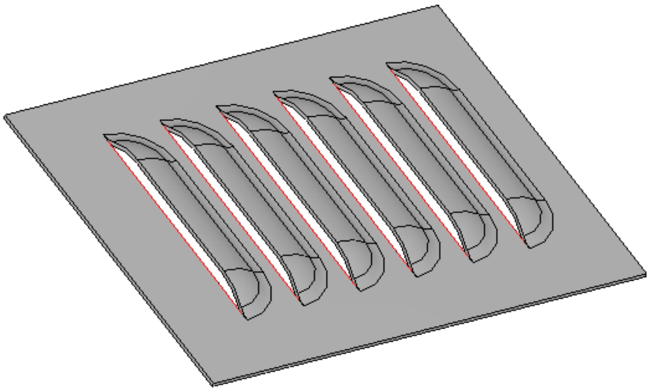


Fig. M.7.14.1

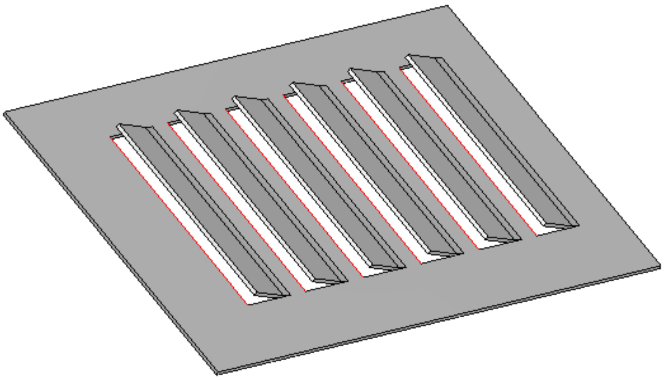


Fig. M.7.14.2

Fig. M.7.14.2 shows a sheet body after louver operation with trim option.

The **CreateJalousie** method adds the MbJalousieSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbJalousieSolid constructor is declared in the cr_stamp_jalousie_solid.h file.

M.7.15. Restore the Edges of Sheet Body Bends

The method

MbResultType
RestoreSideEdges (MbSolid & **solid**,
MbeCopyMode *sameShell*,
const RPArray<MbFace> & **faces**,
const bool *strict*,
PArray<MbSheetMetalBend> & **bends**,
MbSNameMaker & names,
MbSolid *& **result**)

restores side edges of sheet body bends.

The method input parameters are:

- **solid** is an original body,
- *sameShell* is an original body copy option,
- **faces** is a set of external bend faces with side edges that should be restored,
- *strict* is a restoration strictness flag (false means restoring wherever possible),

- names is a namer of constructed faces.

The output parameters of the method are the **result** constructed body and **bends** that have restored side edges.

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

The method is declared in the action_sheet.h file.

The method can be used to restore the side edges of bends after operations that could delete them, for example, cut or filleting. Now side edge restoration method is called automatically during a boolean operation with the *strict* flag set to false and during filleting operation.

Fig. M.7.15.1 shows side edges of a bend.

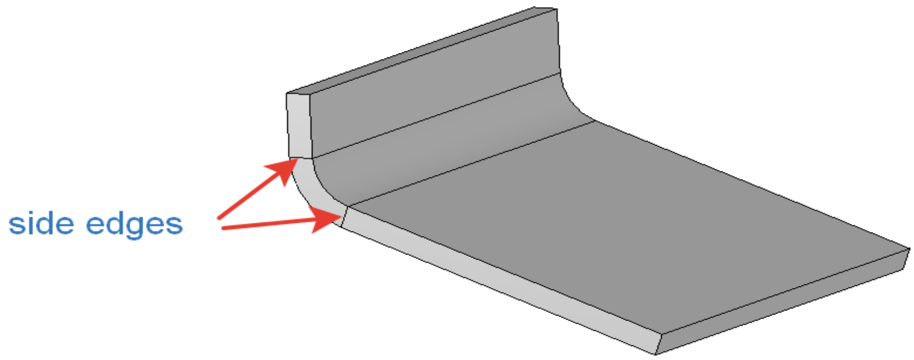


Fig. M.7.15.1

The **RestoreSideEdges** method adds the MbRestoredEdgesSolid constructor in the newly constructed body log containing all data required to execute the operation. The MbRestoredEdgesSolid constructor is declared in the cr_sheet_restored_edges_solid.h file.

M.7.16. Group Sheet Body Bends

The method

bool
SeparateBendsBySubshells (const MbSolid & **solid**,
const RPArray<MbSheetMetalBend> & **bends**,
const MbName & fixedFaceName,
PArray< RPArray<MbSheetMetalBend> > & bendsGroups,
RPArray<const MbFace> & **fixedFaces**)

groups bends by sheet body topological parts that they belong to.

The method input parameters are:

- **solid** is a sheet body,
- **bends** are body bends,
- fixedFaceName is a name of the fixed face.

The output parameters of the method are **bendsGroups**, it is a set of bends grouped by the topological part of the body they belong to, and fixed faces that correspond to these parts **fixedFaces**.

If successful, the method returns true, otherwise it returns false.

The method is declared in the action_sheet.h file.

The method is used after operations that cut the body into several separate parts in order to determine the parts that contain the bends and the corresponding portions of the fixed face. The method sets one-to-one correspondence between bend groups and fixed face parts that correspond to these groups.

Fig. M.7.16.1 shows two groups of bends with their fixed faces.

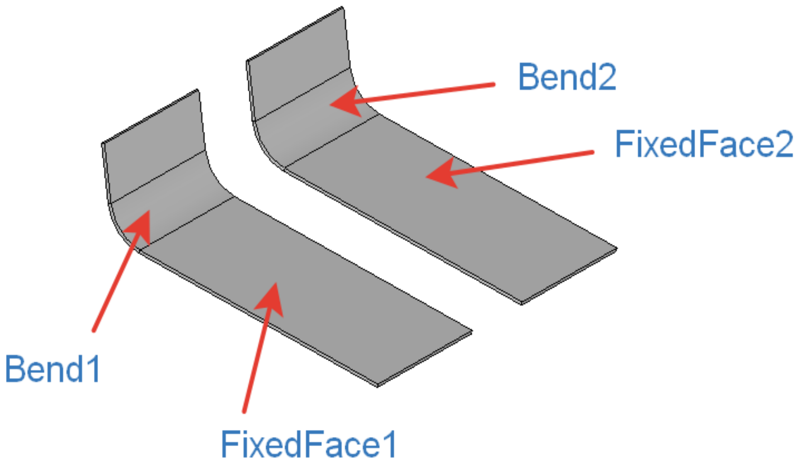


Fig. M.7.16.1

M.7.17. Couple Sheet Body Bends

The method
bool
CollectBends (const MbFaceShell & **shell**,
const RPArray<MbSolid> & **innerFaces**,
const RPArray<MbSolid> & **outerFaces**,
PArray<MbSheetMetalBend> & **bends**)
couples the faces of sheet body bends.

The method input parameters are:

- **shell** is a set of sheet body faces,
- **innerFaces** are internal bend faces,
- **outerFaces** are external bend faces,

The output parameter of the method is a set of found face pairs **bends** that form bends.

If successful, the method returns true, otherwise it returns false.

The method is declared in the action_sheet.h file.

The method looks for internal and external faces that form a bend in the unordered set of external and internal bend faces. A found pair is used to form a bend that is added in the **bends** set.

Fig. M.7.17.1 shows external and internal faces of a bend.

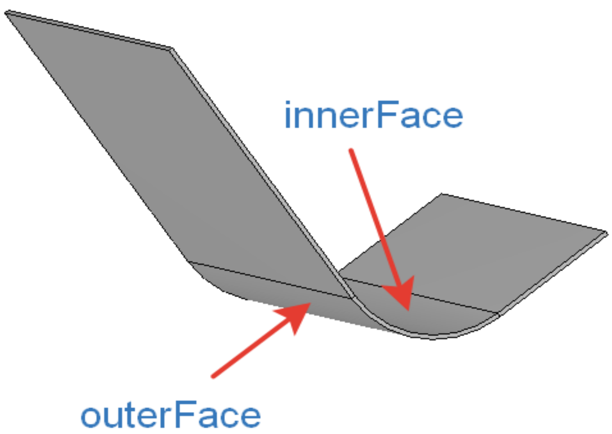


Fig. M.7.17.1

M.7.18. Check Whether the Face Can Be Fixed

The method
bool
IsSuitableForFixed (const MbFace & **face**)
checks whether it is possible to use the face as a fixed face when the sheet body is bent or when its bend is unbent.

The checked **face** is an input parameter of the method.

The method returns true if the face can be selected as a fixed face when a sheet body is bent or when its bend is unbent, otherwise the method returns false.

The method is declared in the action_sheet.h file.

M.7.19. Look for Faces for a Curve

The method
void
FindCurveFaces (const RPArray<MbFace> & **faces**,
const MbCurve3D & **curve**,
RPArray<MbFace> & **curveFaces**)
looks for the faces containing a given straight-line curve.

The method input parameters are:

- **faces** is a set of faces for the search,
- **curve** is a straight-line curve that lies in some faces of the set.

The output parameter of the method is **curveFaces**, the set of faces containing the **curve**.

This method does not return any values.

The method is declared in the action_sheet.h file.

Fig. M.7.19.1 shows two sheet faces located under a segment.

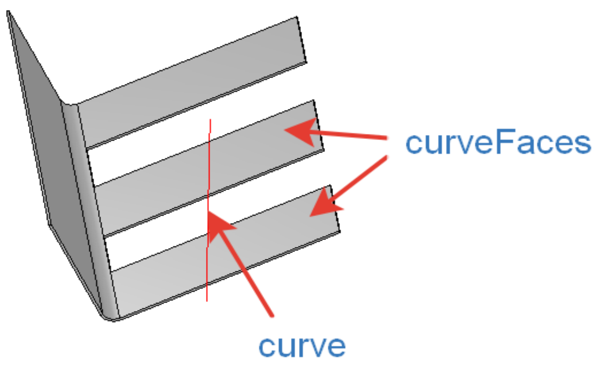


Fig. M.7.19.1

M.7.20. Look for a Sheet Body Face

The method
MbFace *
FindSheetFace (const MbCurveEdge & **edge**)
looks for top or bottom sheet body face that contains the given edge.

The output parameter of the method is **edge**, an edge of the sheet face.
When successful, the method returns a pointer to the found face, otherwise the method returns NULL.
The method is declared in the action_sheet.h file.
Top or bottom face of the sheet body is looked for among two faces joining in the **edge**.
Fig. M.7.20.1 shows a sheet face adjacent to an edge.

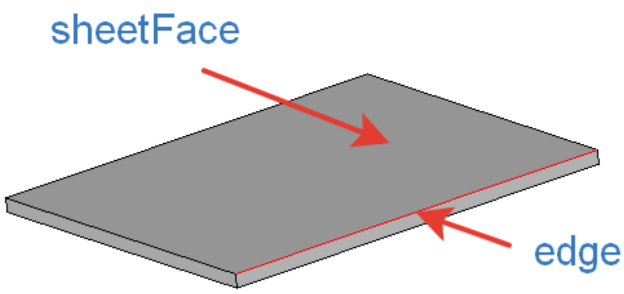


Fig. M.7.20.1

M.7.21. Look for a Pair Face for a Sheet Body Bend

The method
MbFace *
FindPairBendFace (const MbFace & **face**)
looks for a pair face of the sheet body bend.
The input parameter of the method is **face**, a face of the sheet body bend.
When successful, the method returns a pointer to the found face, otherwise the method returns NULL.
The method is declared in the action_sheet.h file.
The method looks for a sheet body bend face opposite to the given face.
Fig. M.7.21.1 shows the bend face and its pair face.

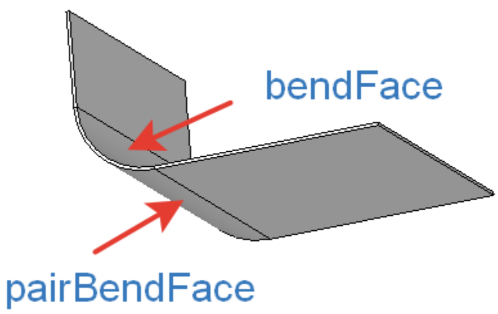


Fig. M.7.21.1

M.7.22. Look for a Flat Face in the Sheet Body

The method
MbFace *
GetPairPlanarFaceByEdge (const MbCurveEdge & **edge**,

const double *begDistance*,
const double *endDistance*)
looks for a flat face of the sheet body based on a given edge and distances from its ends.

The method input parameters are:

- **edge** is an edge used in the search,
- *begDistance* is a distance from the edge origin,
- *endDistance* is a distance from the edge endpoint.

When successful, the method returns a pointer to the found face, otherwise the method returns NULL.

The method is declared in the action_sheet.h file.

The method looks for the pair face of the sheet body for the sheet face containing **edge**. The method is used for edge bend operation in order to determine sheet thickness in the place where the bend is pasted. The method is used for sheet bodies with variable thickness if the selected sheet body face has several corresponding pair faces located at various distances from it. If the *begDistance* and *endDistance* distances are positive, then the indentation is measured in the direction outside along the edge, and if the *begDistance* and *endDistance* distances are negative, then the indentation is measured in the direction inside along the edge.

Fig. M.7.22.1 shows a flat sheet face found based on the edge and distances from its ends.

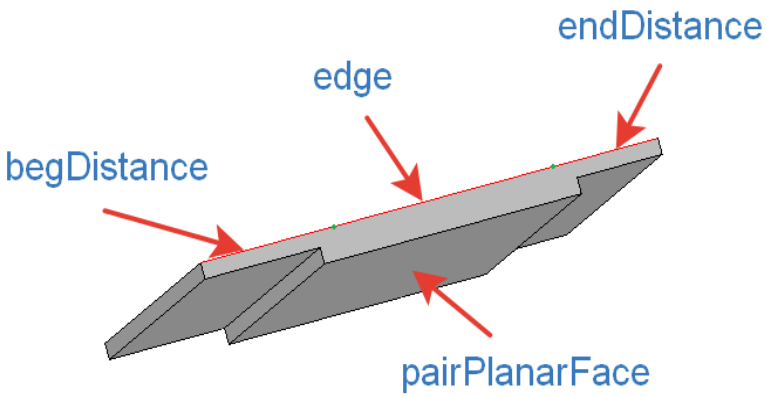


Fig. M.7.22.1

M.7.23. Look for a Pair Face in a Sheet Body

The method
MbFace *
GetPairPlanarFaceByCurve (const MbFace & **face**,
const MbCurve3D & **curve**)
looks for a flat pair face in a sheet body based on a straight-line **curve** lying in the **face**.

The method input parameters are:

- **face** is a flat face of the sheet body,
- **curve** is a straight-line curve lying in it.

When successful, the method returns a pointer to the found face, otherwise the method returns NULL.

The method is declared in the action_sheet.h file.

Look for a sheet body pair face is an auxiliary method for the method that forms a bend at the line. The method is used for sheet bodies with variable thicknesses, if the selected sheet face has several corresponding pair faces located at various distances from it.

Fig. M.7.23.1 shows a flat sheet face found based on a face opposite to it and a curve lying in it.

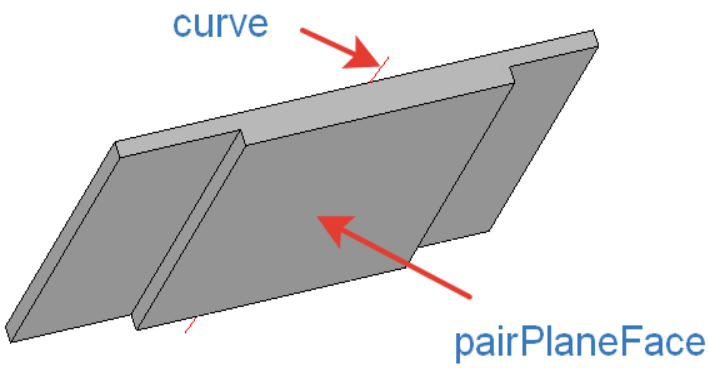


Fig. M.7.23.1

The method
MbFace *
GetPairPlanarFaceByContour (const MbFaceShell & **shell**,
const MbFace& **face**,
const MbPlacement3D & **place**,
const RPArray<const MbCurve > & **segments**)
looks for a flat pair face of a sheet body based on a set of contour elements.

The method input parameters are:

- **shell** is a set of sheet body faces,
- **face** is a sheet body face,
- **place** is a local coordinate system in the **face**,
- **segments** are curves lying in the XY plane of the **place** local coordinate system.

When successful, the method returns a pointer to the found face, otherwise the method returns NULL.

The method is declared in the action_sheet.h file.

Looking for a sheet body pair face is an auxiliary method for sheet body constructing functions that are based on contours. The method is used for sheet bodies with variable thicknesses, if the selected sheet face has several corresponding pair faces located at various distances from it.

Fig. M.7.23.2 shows a flat sheet face found based on a face opposite to it and a contour lying in it.

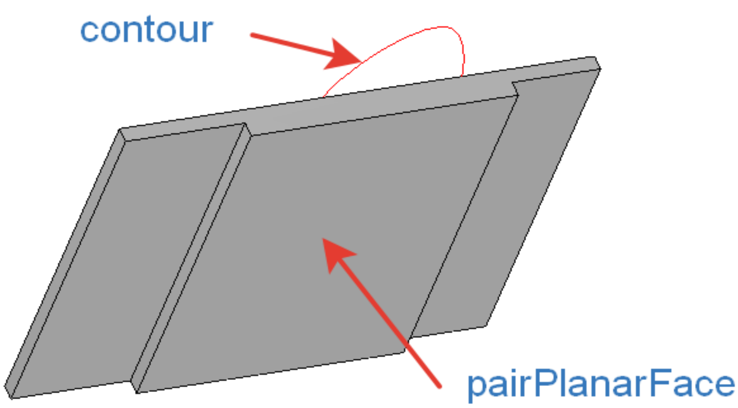


Fig. M.7.23.2

The method
MbFace *
GetPairPlanarFace (const MbFaceShell * **shell**,

const MbFace & **face**)
looks for a pair face based on the given face of the sheet body.

The method input parameters are:

- **shell** is a set of sheet body faces,
- **face** is a sheet body face,

When successful, the method returns a pointer to the found face, otherwise the method returns NULL.

The method is declared in the action_sheet.h file.

The search is executed first over the edges of the external **face** cycle. If the search fails, then the vertices of that cycle are looked for. If the face is not found, then the method looks through all connected faces or faces in the **shell** set. In the latter case, the faces that are located closer are looked for first.

Fig. M.7.23.3 shows a flat sheet face found by its opposite flat sheet face.

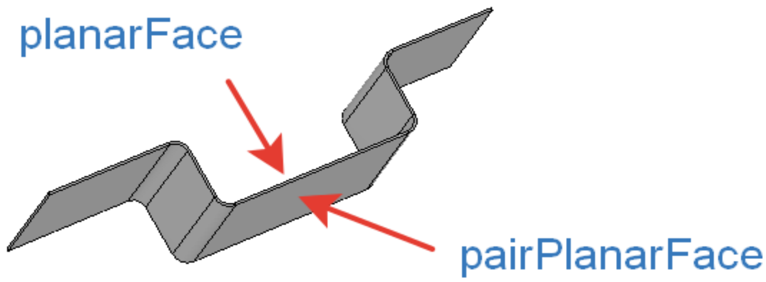


Fig. M.7.23.3

M.7.24. Determine the Distance Between Similar Faces

The method
double
GetDistanceIfSameAndOpposite (const MbFace & **face**1,
const MbFace * **face**2)
determines the distance between a pair of similar leaf faces.

The method input parameters are:

- **face**1 is the first face,
- **face**2 is the second face,

If successful, the method returns the distance between the faces, otherwise it returns zero.

The method is declared in the action_sheet.h file.

Pairs of flat, cylindrical and conical faces are considered similar. As for ruled bends, a pair of ruled offset faces is considered similar if their normals are collinear and have opposite directions. The distance is considered positive if the faces are located at the side opposite to the direction of the normal, otherwise it is considered negative.

Fig. M.7.24.1 shows a pair of similar faces and a distance between them.

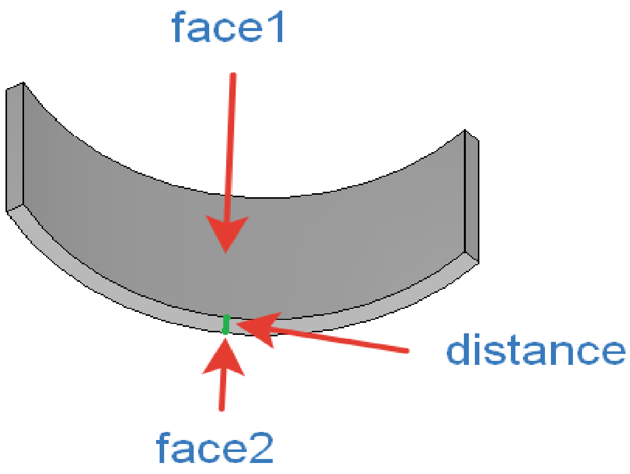


Fig. M.7.24.1

M.7.25. Look for Similar Bends

The method
void
GetSimilarCylindricBends (const **MbFaceShell** & **shell**,
PArray<MbSheetMetalBend> & bends)
looks for similar bends.

The input parameters of the method are a set of sheet body faces **shell** and the array of bends named bends, and similar bends should be found for them.

The output parameter of the method is a set of found similar bends named bends together with the original bends.

This method does not return any values.

The method is declared in the action_sheet.h file.

The method looks for bent cylindrical, conical, and ruled bends, that should be added to the bends from the bends set in order to unbend them later on, i.e. it looks for jointly unbent bends only.

Fig. M.7.25.1 shows a pair of similar bends that can only be bent or unbent together.

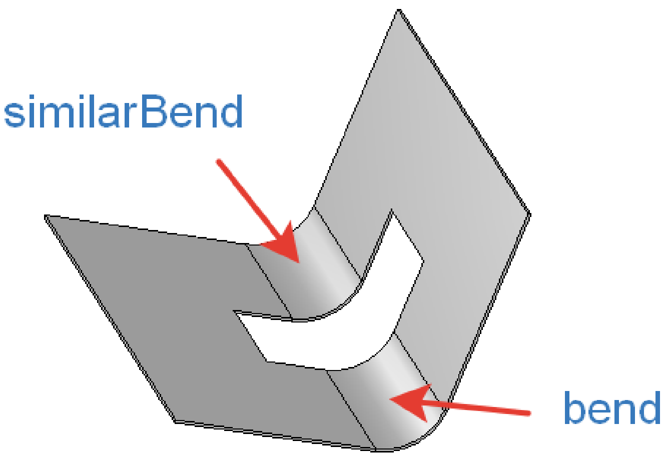


Fig. M.7.25.1

M.7.26. Look for a Tangency Point in a Sheet Body Bend

The method
bool
CalculateTangentPoint (const MbFace & **face**,
const MbPlane & **plane**,
MbCartPoint & **point**)
calculates a tangency point for the sheet body bending/unbending.

The method input parameters are:

- **face** is a sheet body face that contains the tangency point,
- **plane** is a tangential plane,
- **point** is a tangency point.

The output parameter of the method is a tangency **point**.

If successful, the method returns true, otherwise it returns false.

The method is declared in the action_sheet.h file.

point can lie in the face (0.0<=x<=1.0 and 0.0<=y<=1.0) or be located beyond its boundaries. In the first case, the tangency point is recalculated to the coordinates of the surface lying under the face, and in the second case the method finds one of tangency points of the surface lying under **face** and the **plane**.

M.7.27. Look for a Bend Centerline

The method
bool
CalculateConicAxisLine (const **MbFace** & **face**,
MbLineSegment & **axis**)
calculates the centerline of unbent conical bend.

The input parameter of the method is a **face** of unbent conical bend of the sheet body.

The output parameter of the method is an **axis** centerline.

If successful, the method returns true, otherwise it returns false.

The method returns the centerline in the coordinates of the flat **face** parametric field.

The method is declared in the action_sheet.h file.

M.8. AUXILIARY METHODS

In certain cases need to be calculated parameters required for the construction of bodies or other elements. This chapter lists the methods that perform auxiliary calculations and constructs.

M.8.1. Calculating Extrusion Body Depth or Rotation Body Angle

The method

bool
GetSweptValue (const MbSweptData & **sweptData**,
const MbAxis3D & **axis**,
const MbVector3D & **direction**,
const bool rotation,
const bool operationDirection,
const MbCartPoint3D & **point**,
double & *value*)

calculates the extrusion depth or the rotation angle for further construction of the body by extrusion or rotation of the generatrix.

The method input parameters are:

- **sweptData** is generatrix data,
- **axis** is a rotation axis (for rotation calculation),
- **direction** is an extrusion direction (it is used to calculate extrusion),
- rotation is a calculation type flag: rotation (true), or extrusion (false),
- operationDirection is a direction of the following operation execution: forward (true), or backward (false),
- **point** is a point, to which the generatrix should be extruded or rotated.

The output parameter of the method is *value*, this is the extrusion depth or rotation angle.

If the calculation is successful, the method returns true, otherwise it returns false.

This method is auxiliary for the **ExtrusionSolid**, **RevolutionSolid**, **ExtrusionResult** and **RevolutionResult** functions.

M.8.2. Determining the Curve Image for Extrusion or Rotation

The method

GetSweptImagePosition (const MbCurve3D & **curve**,
const MbSurface & **surface**,
const MbVector3D & **direction**,
const MbAxis3D & **axis**,
const bool rotation,
MbCartPoint& ***imagePosition***,
MbResultType & resType)

calculates the position of the generatrix curve point image on a surface to further construct the body by extrusion or rotation of the generatrix up to the given surface.

The method input parameters are:

- **curve** is a generatrix,
- **surface** is a surface, up to which the curve is extruded or rotated,
- **direction** is an extrusion direction (it is used to calculate extrusion),
- **axis** is a rotation axis (it is used to calculate rotation),
- rotation is a calculation type flag: rotation (true) or extrusion (false).

The method output parameters are as follows:

- ***imagePosition*** is a point on the surface where the image of the generatrix lies,

- resType is an operation result code: if successful, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

This method is auxiliary for the **ExtrusionSolid**, **RevolutionSolid**, **ExtrusionResult** and **RevolutionResult** functions.

M.8.3. Determining Extrusion or Rotation Parameters

The method

GetRangeToSurface (const MbSurface & **surface**,
const MbCurve3D & **curve**,
const MbVector3D & **direction**,
const MbAxis3D & **axis**,
const bool rotation,
const bool operationDirection,
const MbCartPoint & ***imagePosition***,
double range[2],
MbRect & rectOnSurface,
MbResultType & resType)

calculates the extrusion depths in forward and backward directions, or the rotation angles in forward or backward directions to further construct the body by extruding or rotating its generatrix up to the selected surface, and also calculates the curve image size.

The method input parameters are:

- **surface** is a surface, up to which the curve is extruded or rotated,
- **curve** is a generatrix,
- **direction** is an extrusion direction (it is used to calculate extrusion),
- **axis** is a rotation axis (it is used to calculate rotation),
- rotation is a calculation type flag: rotation (true), or extrusion (false),
- operationDirection is a direction of the following operation execution: forward (true), or backward (false),
- ***imagePosition*** is a point on the surface where the image of the generatrix lies.

The method output parameters are as follows:

- range is distances to the surface: range[0] is a distance in reverse direction, range[1] is a distance in forward direction,
- rectOnSurface is a size of the curve image on surface,
- resType is an operation result code: if successful, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

This method is auxiliary for the **ExtrusionSolid**, **RevolutionSolid**, **ExtrusionResult** and **RevolutionResult** functions.

M.8.4. Determine the Orientation of the Generatrix

The method

double
AreaSign (const MbCurve3D & **curve**,
const MbAxis3D & **axis**,
const MbVector3D & **direction**,
bool rotation)

calculates the area of the curve projection on the virtual coordinate plane to determine the orientation of the generatrix for further construction of the body by extrusion or rotation of the generatrix. A non-closed generatrix will be closed by a segment.

The method input parameters are:

- **curve** is a generatrix,

- **axis** is a rotation axis (it is used to calculate rotation),
- **direction** is an extrusion direction (it is used to calculate extrusion),
- rotation is a calculation type flag: rotation (true) or extrusion (false).

The method output parameter is the curve projection surface.

This method is auxiliary for the **ExtrusionSolid**, **RevolutionSolid**, **ExtrusionResult** and **RevolutionResult** functions.

M.8.5. Determine the Orientation of the Secant Surface

The method
AnalyzeSurfaceRelationToSweptOperation (const MbSurface & **surface**,
const MbCartPoint & ***imagePosition***,
const MbCurve3D & **curve**,
const MbVector3D & **direction**,
const MbAxis3D & **axis**,
const bool rotation,
bool operationDirection,
bool & relativeSense,
MbResultType & resType)
determines the orientation of the secant surface relative to the body, it is constructed by extruding or rotating the generatrix up to the given surface.

The method input parameters are:
- **surface** is a surface, up to which the curve is extruded or rotated,
- ***imagePosition*** is a point on the surface where the generatrix image lies,
- **curve** is a generatrix,
- **direction** is an extrusion direction (it is used to calculate extrusion),
- **axis** is a rotation axis (it is used to calculate rotation)
- rotation is a calculation type flag: rotation (true), or extrusion (false),
- operationDirection is a direction of the following operation execution: forward (true) or backward (false).

The method output parameters are as follows:
- relativeSense is a surface orientation for the extrusion or rotation operation,
- resType is an operation result code: if successful, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

This method is auxiliary for the **ExtrusionSolid**, **RevolutionSolid**, **ExtrusionResult** and **RevolutionResult** functions.

M.8.6. Sweep Body Curve Orientation

The method
MbResultType
EvolutionNormalize (const MbPlacement3D & **place**,
const MbContour & **contour**,
const MbCurve3D & **guide**,
EvolutionValues & parameters,
MbAxis3D & **axis**,
double & *angle*,
VERSION version)
orients the generating contour and the guiding curve to construct a sweep body.

The method input parameters are:
- **place** is a local coordinate system of the generating contour,
- **contour** is a generating contour,

- **guide** is a guiding curve,
- parameters are sweep operation parameters,
- version is an operation version.

The method output parameters are as follows:

- **axis** is an axis of additional generatrix rotation,
- *angle* is an angle of additional generatrix rotation,

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

This method is auxiliary for the **EvolutionSolid** and **EvolutionResult**.

M.8.7. Copy Guiding Curve of the Sweep Body

The method

MbCurve3D *
TrimClosedSpine (MbCurve3D & **curve**, double *t*)
copies a closed curve that begins in a point defined by parameter *t*.

The method input parameters are:

- **curve** is a guiding curve,
- *t* is a curve parameter.

If successful, the method returns a constructed copy of the curve that begins in the given point, otherwise the method returns null.

This method is auxiliary for the **EvolutionSolid** and **EvolutionResult**.

M.8.8. Constructing a Rib

The method

```
MbResultType
RibElement ( const MbSolid & solid,
             const MbPlacement3D & place,
             MbContour & contour,
             size_t        index,
             RibValues & params,
             const MbSNameMaker &  names,
             MbSolid *&  result )
```

constructs a rib for the original body.

The method input parameters are:

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

The output parameter of the method is the **result** constructed body.

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

The method constructs a separate rib, but it doesn't connect it to the original body.

M.8.9. Check a Curve for Ruled Body Construction

The method

CheckRuledCurve (const MbCurve3D & **curve**1,
const MbCurve3D & **curve**2,
bool & isInverted,
bool & isShifted,
VERSION version)

checks whether the second curve can be used together with the first curve to construct a non-closed ruled body, and modifies the second curve as required.

The method input parameters are:

- **curve**1 is the first curve,
- **curve**2 is the second curve,
- version is an operation version.

The method output parameters are as follows: the isInverted flag indicates whether the direction of the second curve was inverted, the isShifted flag indicates whether the second curve beginning point was shifted.

The method does not return any value.

This method is an auxiliary method used to construct the **RuledShell** non-closed ruled body based on two curves.

M.8.10. Check Curve Parameters for Ruled Body Construction

The method

bool

CheckRuledParams (const MbCurve3D & **curve**,
SArray<double> & params,
bool isAscending)

checks the curve parameters and normalizes the closed curve parameters.

The method input parameters are:

- **curve** is a curve,
- params is a set of curve parameters,
- isAscending is a flag that indicates whether set parameters are listed in ascending order.

The output parameter of the method is params, this is a set of curve parameters.

If successful, the method returns true, otherwise it returns false.

This method is an auxiliary method for constructing the **RuledShell** non-closed ruled body.

M.8.11. Check Curve for Constructing a Joint Body

The method

CheckJoinedCurve (const MbCurve3D & **curve**1,
const MbCurve3D & **curve**2,
bool & isInverted1,
bool & isShifted1,
VERSION version)

checks whether the second curve can be used together with the first curve to construct a non-closed joint body, and modifies the second curve if required.

The method input parameters are:

- **curve**1 is the first curve,
- **curve**2 is the second curve,
- version is an operation version.

The method output parameters are as follows: the isInverted flag indicates whether the direction of the second curve was inverted, the isShifted flag indicates whether the second curve beginning point was shifted.

The method does not return any value.

This method is auxiliary for the method that constructs a non-closed joint body based on two **JoinShell** curves.

M.8.12. Check Curve Parameters for the Joint Body Construction

The method
bool
CheckJoinedParams (const MbCurve3D & **curve**,
SArray<double> & params,
bool isAscending)
checks the curve parameters and normalizes the closed curve parameters.

The method input parameters are:

- **curve** is a curve,
- params is a set of curve parameters,
- isAscending is a flag that indicates whether set parameters are listed in ascending order.

The output parameter of the method is params, it is a set of curve parameters.

If successful, the method returns true, otherwise it returns false.

This method is an auxiliary method for constructing the **JoinShell** non-closed body.

M.8.13. Construct a Curve from a Set of Edges

The method
MbCurve3D *
CreateJoinedCurve (const RPArray<MbCurveEdge> & **edges**,
const SArray<bool> & orientations,
const MbMatrix3D & **matrix**,
MbResultType & result)
constructs a curve from a set of edges.

The method input parameters are:

- **edges** is a set of edges,
- orients is an edge orientation,
- **matrix** is an edge conversion matrix.

The output parameter of this method is the result value from the MbResultType enumeration.

When successful, the method returns a pointer to the constructed curve, otherwise the method returns null.

This method is an auxiliary method for constructing the **JoinShell** non-closed body.

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.

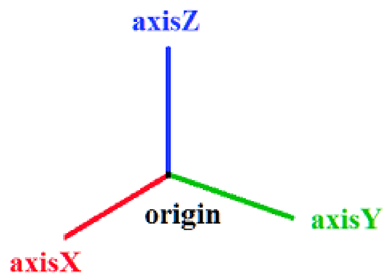


Fig. O.1.4.1.

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, body or several bodies) 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.

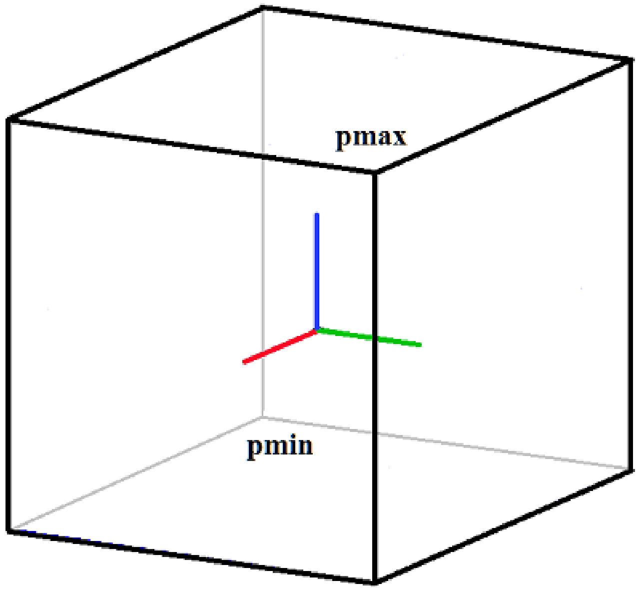


Fig. O.1.6.1.

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.

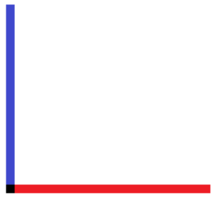


Fig. O.1.12.1.

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 ***a***=[a_1 a_2] and ***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.

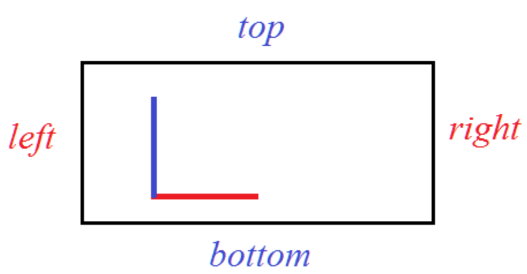


Fig. O.1.14.1.

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, bodies 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.

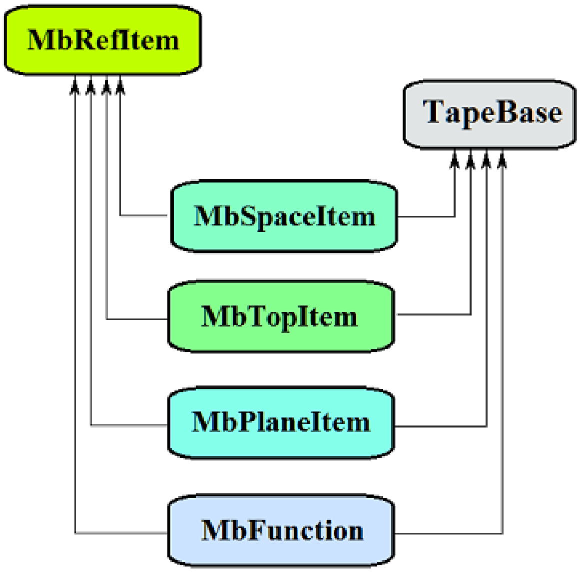


Figure O.2.1.1.

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.

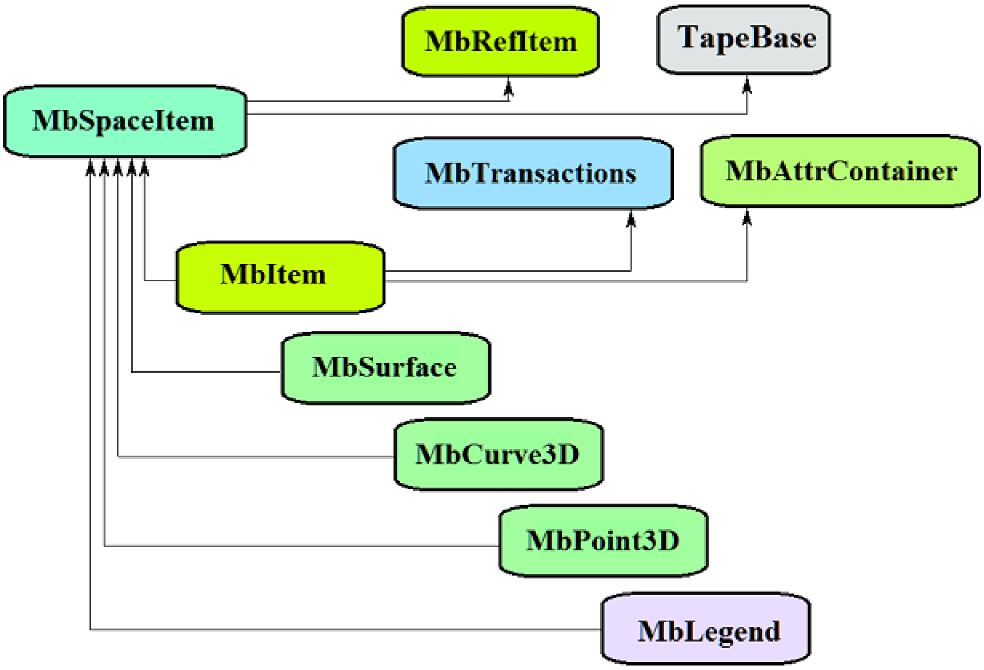


Figure O.2.2.1.

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.

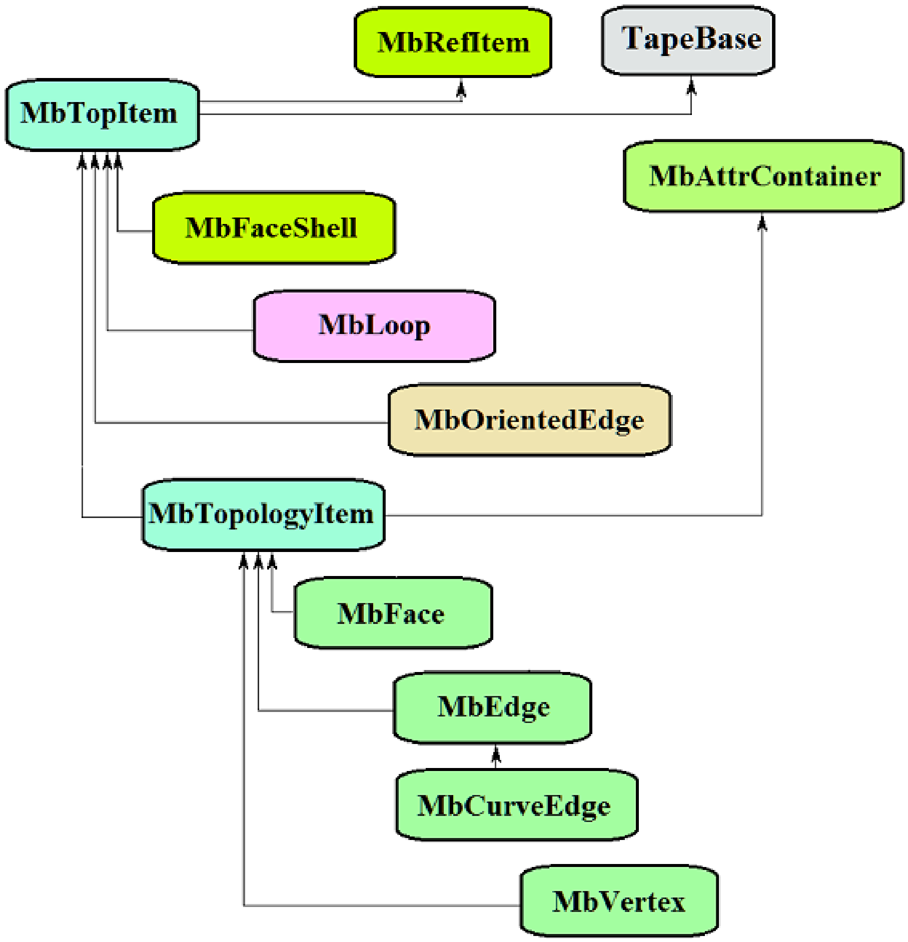


Figure O.2.3.1.

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.

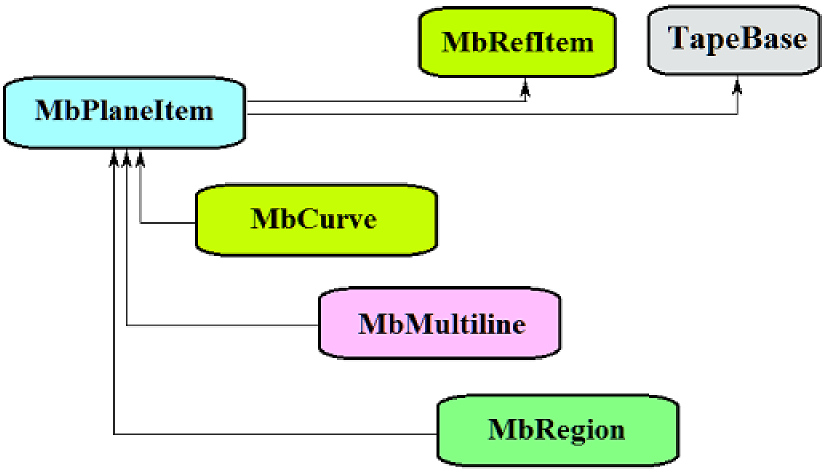


Figure O.2.4.1.

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.

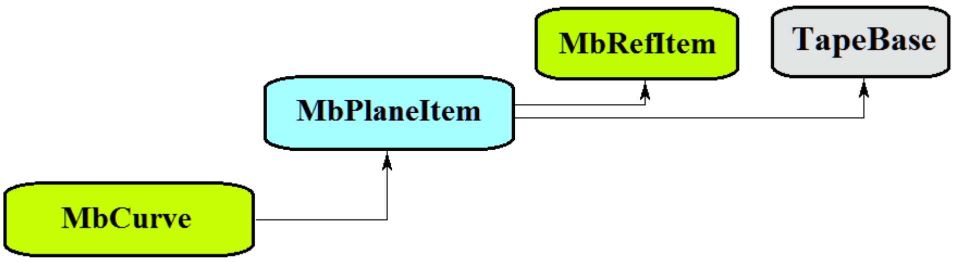


Fig. O.3.1.1.

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,
MbCharacterCurve – 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***$(t \pm kp)$=***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 & ***r***).

It returns ***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 & ***r***)

returns ***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.

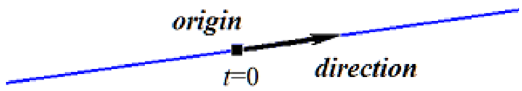


Fig. O.3.2.1.

In **PointOn**(double & *t*, MbCartPoint & ***r***) method, radius vector of ***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.

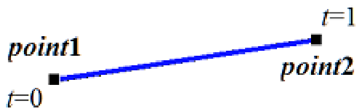


Fig. O.3.3.1.

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

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

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 & ***r***) method, the radius vector of ***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.

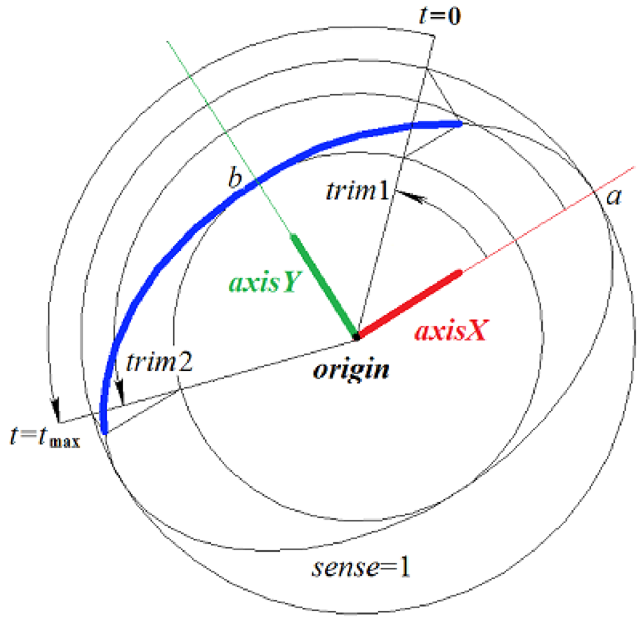


Fig. O.3.4.1.

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.

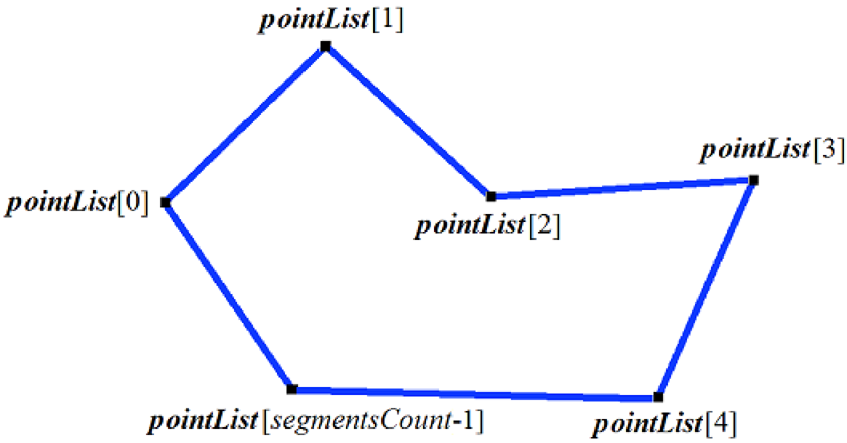


Fig. O.3.5.1.

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

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

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.

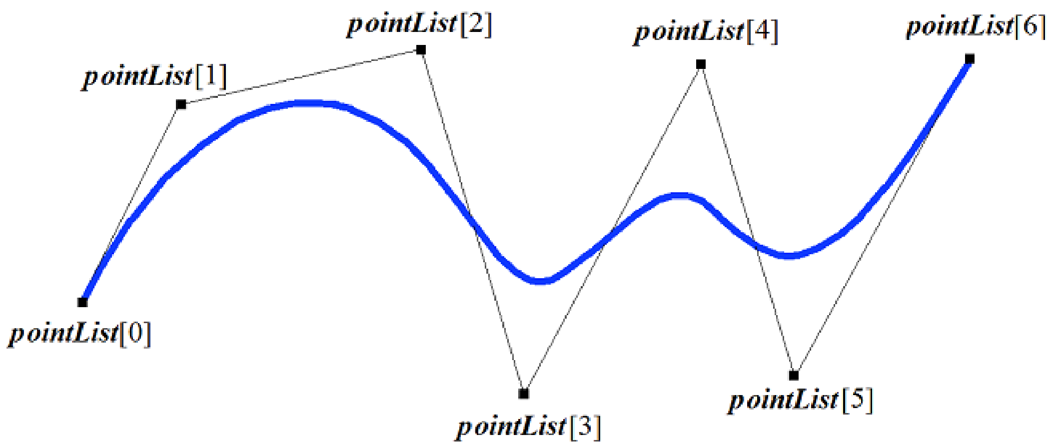


Fig. O.3.6.1.

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

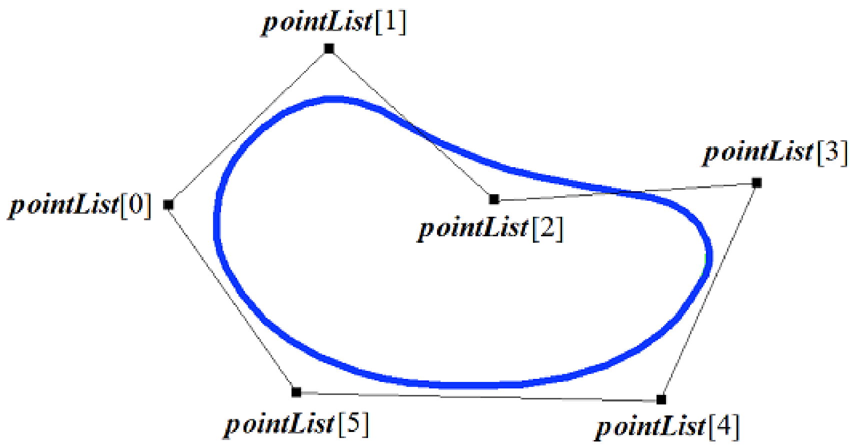


Fig. O.3.6.2.

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 $tList[i] \le t \le tList[i+1]$ 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*:

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

where $w = \dfrac{t - \text{tList}[i]}{\text{tList}[i+1] - \text{tList}[i]}$, and $tList[i] \le t \le tList[i+1]$.

A Hermite curve is shown in Figure O.3.7.1.

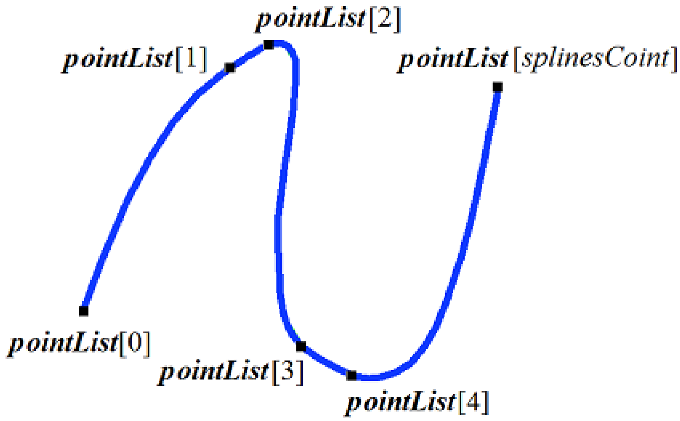


Fig. O.3.7.1.

t curve parameter takes values in $tmin \le t \le tmax$ 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 $w=t-n,\ n\le t\le n+1,\ 0\le w\le 1,\ B_j^3(w)=\frac{3!}{j!(3-j)!}w^j(1-w)^{3-j}$ are third-order Bernstein functions for *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.

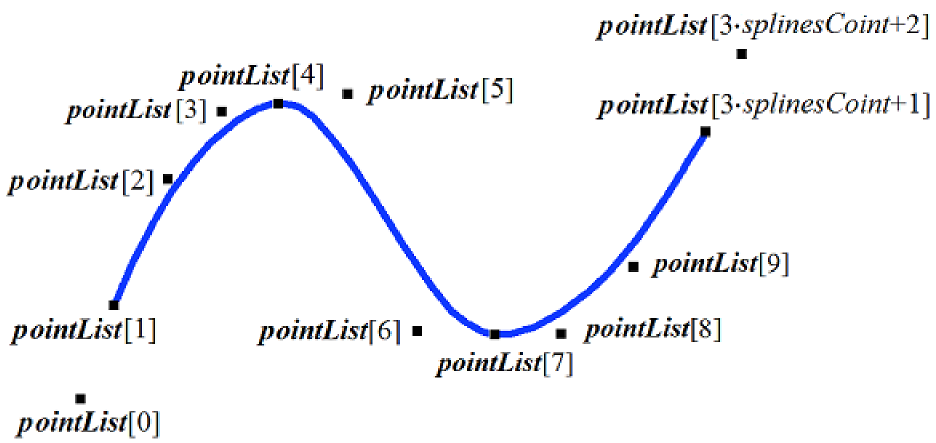


Fig. O.3.8.1.

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

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

vector function,

where $w = \dfrac{t - \mathrm{tList}[i]}{\mathrm{tList}[i+1] - \mathrm{tList}[i]}$, 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.

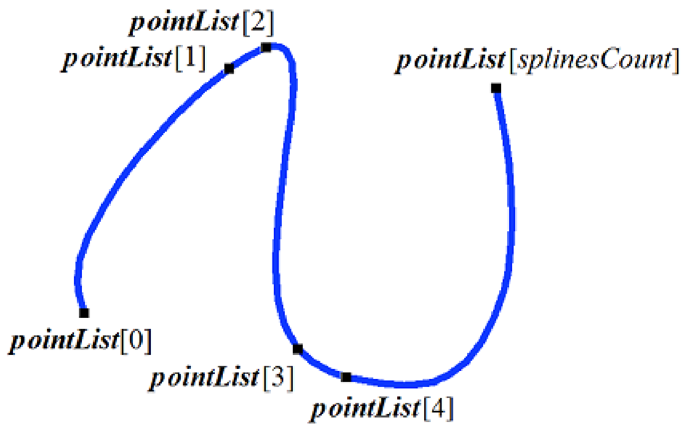


Fig. O.3.9.1.

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

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

A truncated curve is shown in Figure O.3.10.1.

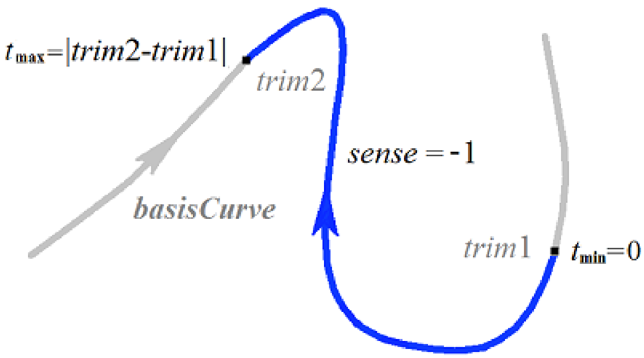


Fig. O.3.10.1.

t curve parameter takes values in $0 \le t \le sense(trim2 - trim1)$ 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 *trim2=trim1+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 $v(t) = b\text{min}\dfrac{\text{trim2} - t}{\text{trim2} - \text{trim1}} + b\text{max}\dfrac{t - \text{trim1}}{\text{trim2} - \text{trim1}}$, *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.

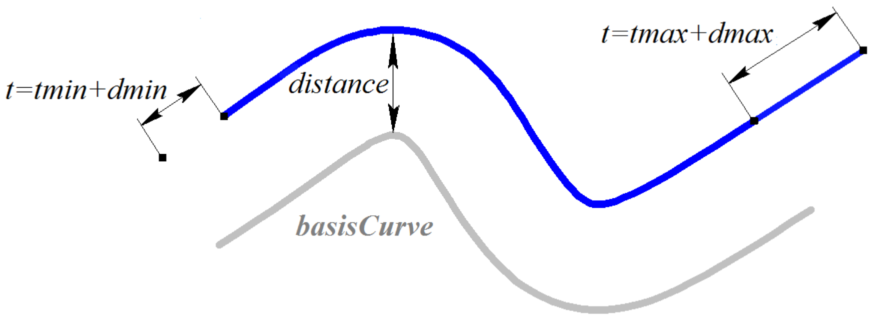


Fig. O.3.12.1.

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. MbCharacterCurve 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: $tmin \le t \le tmax$ range.

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

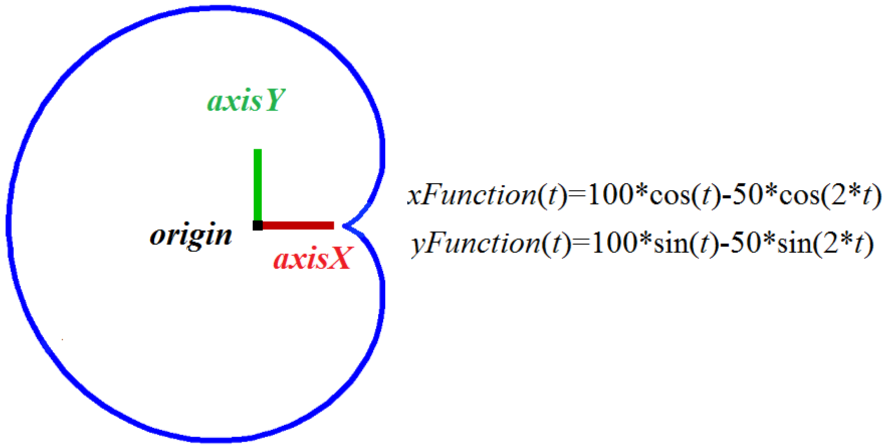


Fig. O.3.13.1.

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

The syntax expression parser recognizes the basic mathematical operations and taking in account priority and parentheses: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), and unary minus (-). Operations are performed on variables, constants and functions.

The syntax expression parser recognizes the names of the basic mathematical functions, which are given in Table O.3.13.1.

Table O.3.13.1. Mathematical functions used in symbolic expressions.

Function name	**Description**
sin	sine
cos	cosine
tan	tangent
sind	sine, input argument is in degrees
cosd	cosine, input argument is in degrees
tand	tangent, input argument is in degrees
asin	arcsine
acos	arccosine
atan	arctangent
asind	arcsine, input argument is in degrees
acosd	arccosine, input argument is in degrees
atand	arctangent, input argument is in degrees
sqrt	square root

exp	exponent
ln	natural logarithm
lg	decimal logarithm
deg	function to convert radians to degrees
rad	function to convert degrees to radians
abs	modulus

Unnamed constants (numbers) can be set as floating point numbers (a string of digits separated by dot) or in integer format.

Symbolic expressions can use the named constants shown in Table O.3.13.2.

Table O.3.13.2. Named Constants.

Constant name	**Description**
M_PI	π – the ratio of a length of circle to its diameter
M_PI_2	$\pi / 2$
M_PI_4	$\pi / 4$
M_SQRT2	$\sqrt{2}$
M_E	e – exponent
M_FI	φ – golden ratio
M_RADDEG	$180 / \pi$ – radians to degrees conversion factor
M_DEGRAD	$\pi / 180$ – degrees to radians conversion factor

The variable name of an analytic expression can be a sequence of string and numeric literals (the first character is a literal of string type), that does not match with function and constant names.

Limitations of symbolic curve expression:

- the maximum length of a variable name is 512 characters;
- the maximum length of an expression is 2048 characters;
- the operations of integration, differentiation, calculation of limits of functions are not allowed;
- there is no possibility of simplifying mathematical expressions (for example, reduction of fractions).

Due to the existing limitations, the correct calculation of the value of the analytical expression can be obtained for relatively simple mathematical expressions and functions. For example, the value of the ratio sin(x)/x at zero may be calculated incorrectly.

The example code below demonstrates the ability to create a symbolic 2D curve. The constructed symbolic curve is shown in Figure O.3.13.1.

```
MbCharacterCurve * CreateAnalyticalCurve2D() {
  MbCharacterCurve * curve( nullptr );
  // String representations of given coordinate functions.
  string_t xFunction = _T( "100*cos(t)-50*cos(2*t)" );
  string_t yFunction = _T( "100*sin(t)-50*sin(2*t)" );
  string_t argument  = _T( "t" );
  double t1 = 0.0, t2 = 2.0 * M_PI;
  // Creating Analytic Coordinate Functions using the Function Factory.
  MbFunctionFactory factory;
  SPtr<MbFunction> x( factory.CreateAnalyticalFunction(xFunction, argument, t1, t2, true) );
```

```
  SPtr<MbFunction> y( factory.CreateAnalyticalFunction(yFunction, argument, t1, t2, true) );
  if ( x != nullptr && y != nullptr ) {
    MbCartPoint point( 0.0, 0.0 ); // Origin of the local coordinate system.
    MbVector axisX( 1.0, 0.0 ), axisY( 0.0, 1.0 );
    MbPlacement place( point, axisX, axisY ); // Local coordinate system.
    MbeLocalSystemType cs = ls_CartesSystem; // Type of local coordinate system.
    // Create a symbolic 2D curve.
    curve = new MbCharacterCurve( *x, *y, cs, place, t1, t2 );
  }
  return curve;
}
```

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

$$\boldsymbol{r}(t) = \boldsymbol{position.origin} + (((tmin+t-phase) / frequency)\ \boldsymbol{position.axisX}) + (amplitude \cos(tmin+t)\ \boldsymbol{position.axisY})$$ vector function.

Cosine wave is shown in figure O.3.14.1.

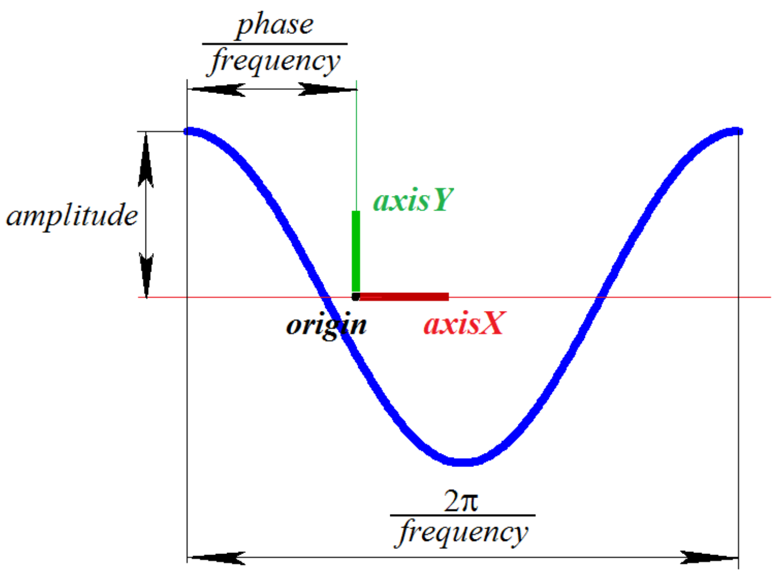


Fig. O.3.14.1.

t curve parameter takes values in $tmin \le t \le tmax$ 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 $tmin \le t \le tmax$ 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

$$\mathbf{deriveU} \cdot (\mathbf{spaceCurve}(t) - \mathbf{surface}(\, u, v\,)) = 0,$$
$$\mathbf{deriveV} \cdot (\mathbf{spaceCurve}(t) - \mathbf{surface}(\, u, v\,)) = 0,$$

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.

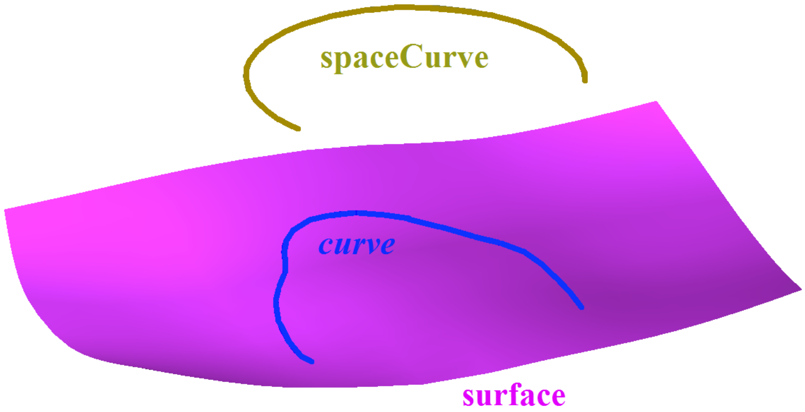


Fig. O.3.16.1.

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 ith 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.

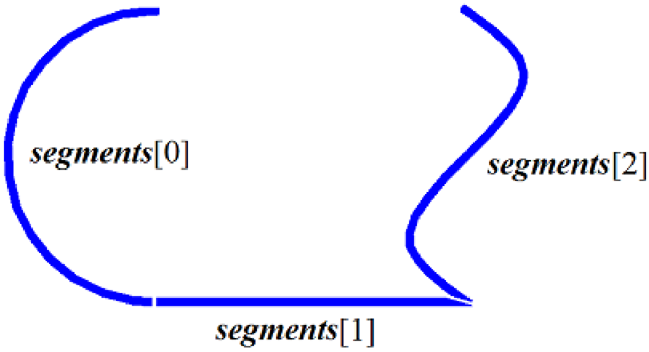


Fig. O.3.17.1.

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.

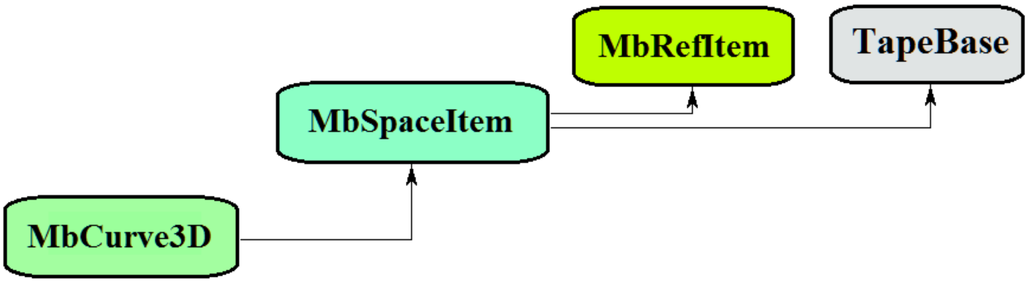


Fig. O.4.1.1.

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.

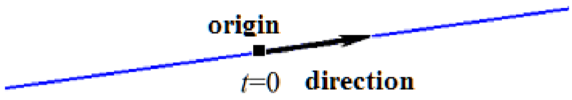


Fig. O.4.2.1.

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.

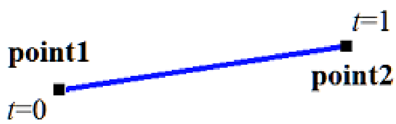


Fig. O.4.3.1.

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.

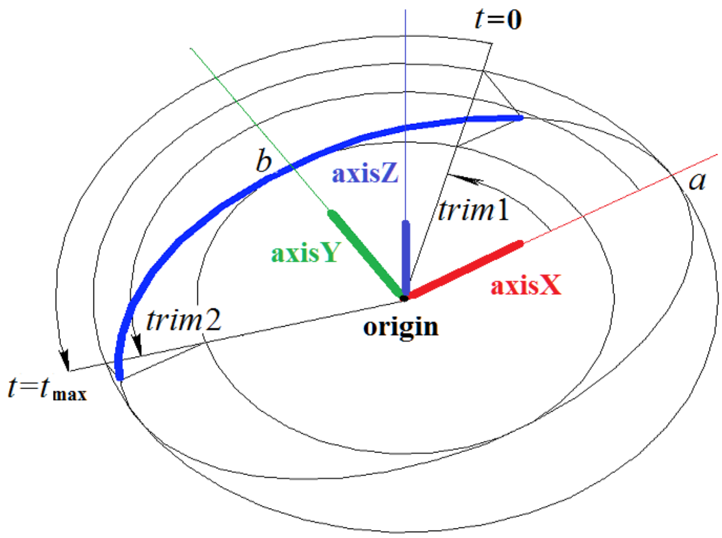


Fig. O.4.4.1.

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,...,*segmentsCount.* set of points at *t*=0,...,*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.

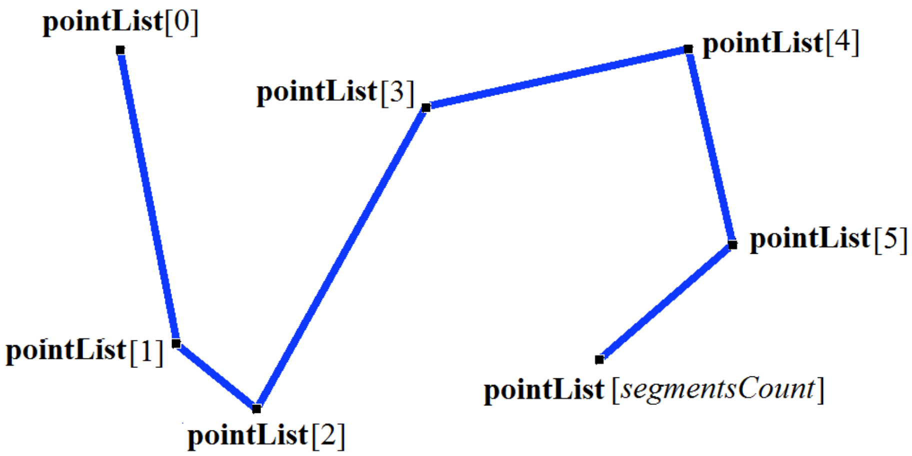


Fig. O.4.5.1.

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.

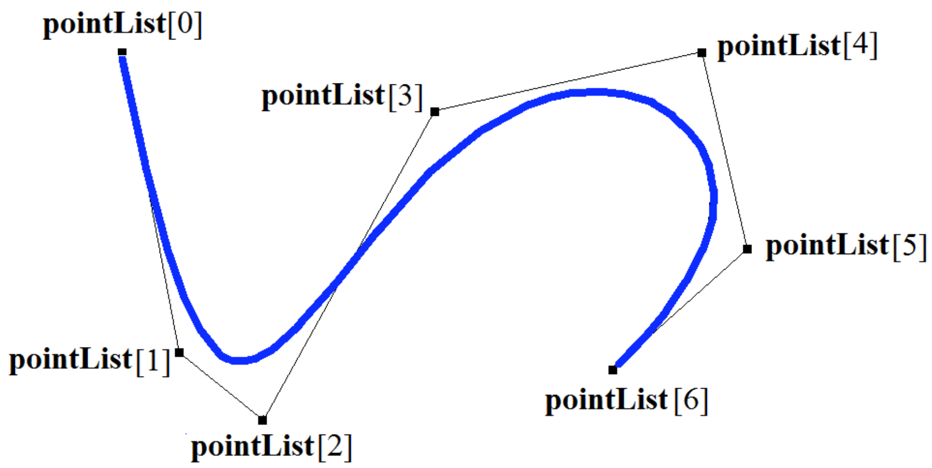


Fig. O.4.6.1.

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.

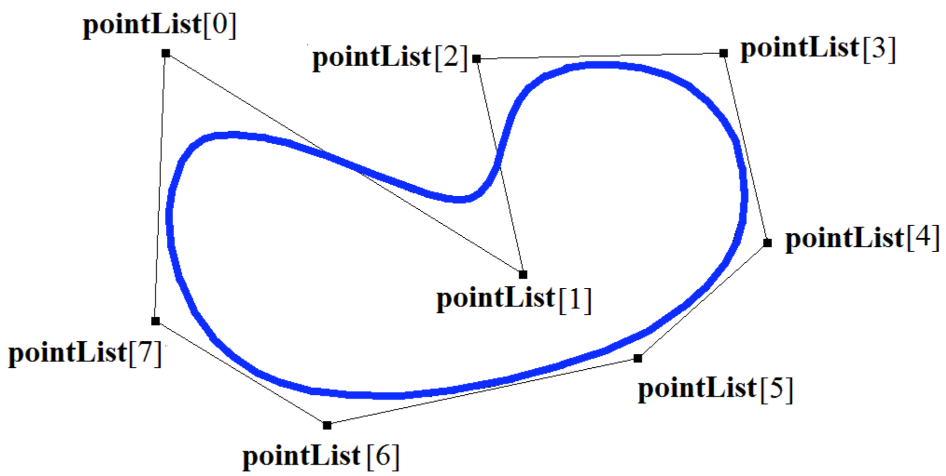


Fig. O.4.6.2.

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*

$$r(t)=(1-3w^2+2w^3)\mathbf{pointList}[i]+(3w^2-2w^3)\mathbf{pointList}[i+1]+$$
$$+\left((w-2w^2+w^3)\mathbf{vectorList}[i]+(-w^2+w^3)\mathbf{vectorList}[i+1]\right)(tList[i+1]-tList[i]),$$

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.4.7.1.

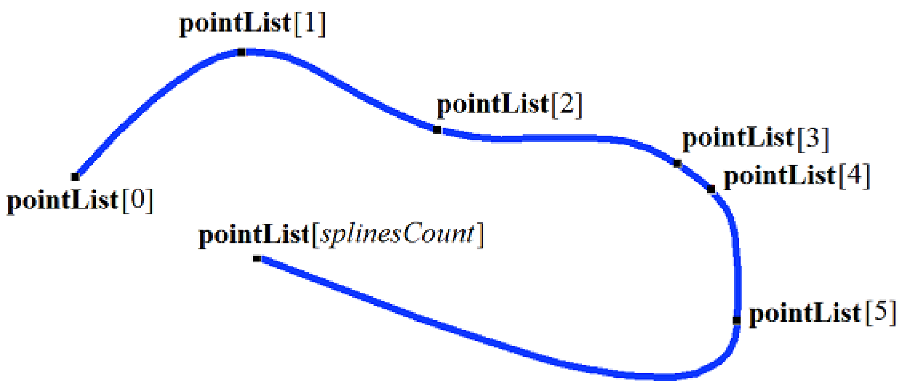


Fig. O.4.7.1.

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.

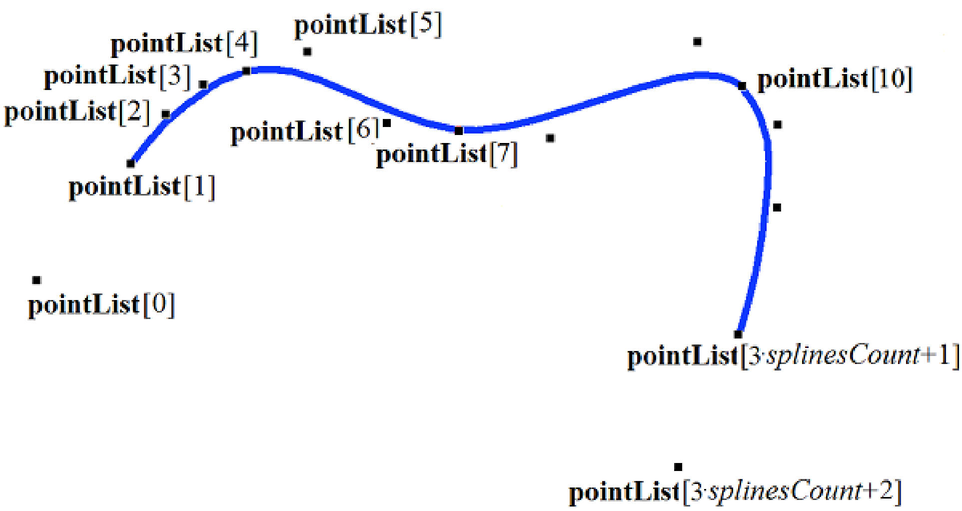


Fig. O.4.8.1.

t curve parameter takes values in $0 \le t \le splinesCount$ 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

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

vector function, where $w = \dfrac{t-\mathrm{tList}[i]}{\mathrm{tList}[i+1]-\mathrm{tList}[i]}$, 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.

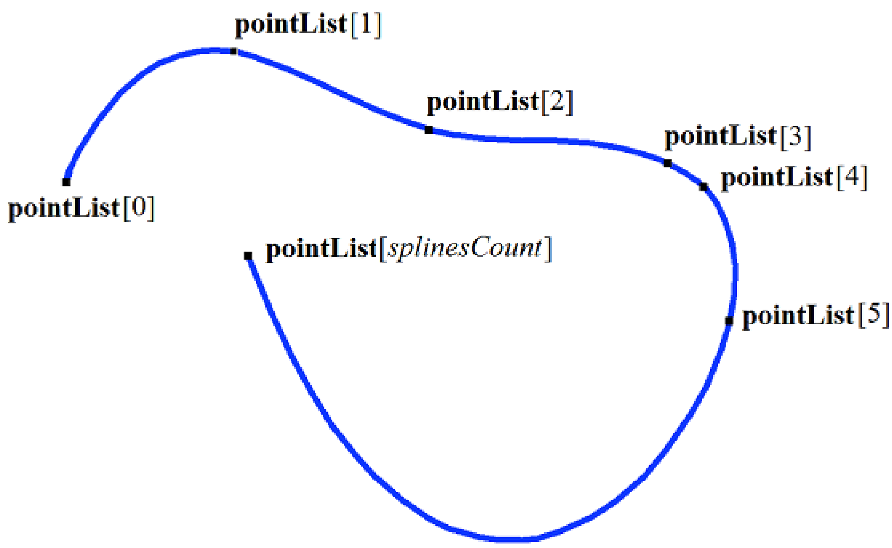


Fig. O.4.9.1.

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.

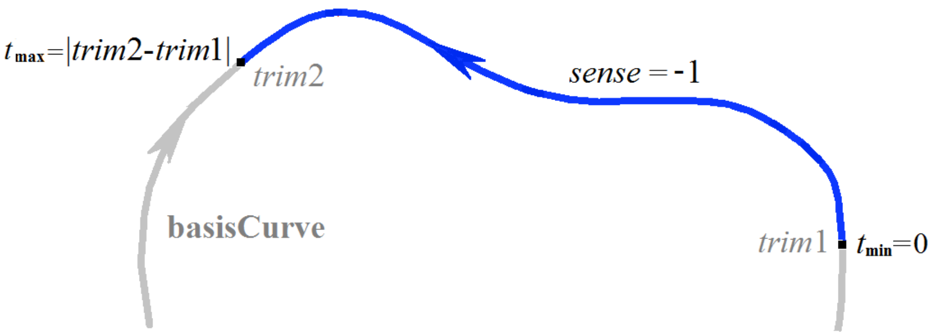


Fig. O.4.10.1.

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 $v(t) = b\text{min}\dfrac{\text{trim2} - t}{\text{trim2} - \text{trim1}} + b\text{max}\dfrac{t - \text{trim1}}{\text{trim2} - \text{trim1}}$, *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.

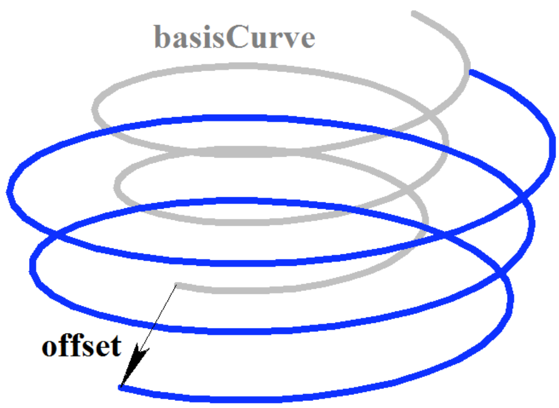


Fig. O.4.12.1.

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 $tmin \le t \le tmax$ *range.*

In **PointOn**(double & t, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

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

A character curve is shown in Figure O.4.13.1.

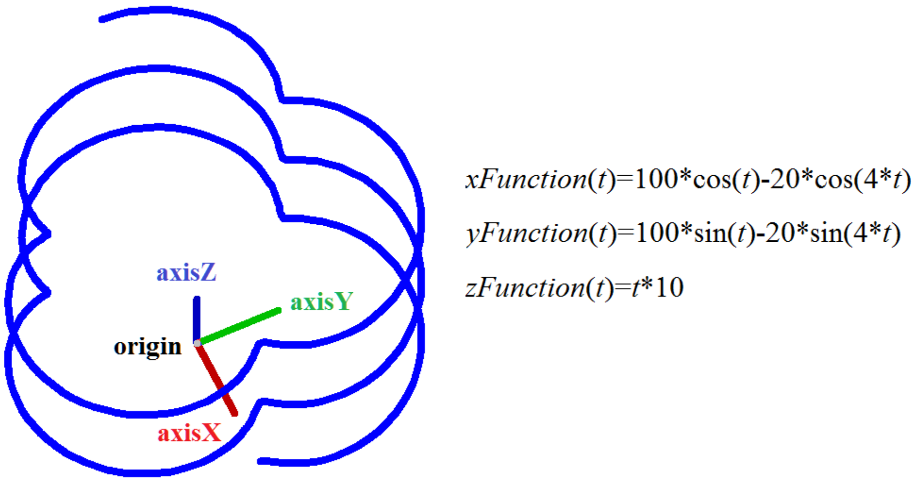


Fig. O.4.13.1.

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

The syntax expression parser recognizes the basic mathematical operations and taking in account priority and parentheses: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), and unary minus (-). Operations are performed on variables, constants and functions.

The syntax expression parser recognizes the names of the basic mathematical functions, which are given in Table O.4.13.1.

Table O.4.13.1. Mathematical functions used in symbolic expressions.

Function name	**Description**
sin	sine
cos	cosine
tan	tangent
sind	sine, input argument is in degrees
cosd	cosine, input argument is in degrees
tand	tangent, input argument is in degrees
asin	arcsine
acos	arccosine
atan	arctangent
asind	arcsine, input argument is in degrees

acosd	arccosine, input argument is in degrees
atand	arctangent, input argument is in degrees
sqrt	square root
exp	exponent
ln	natural logarithm
lg	decimal logarithm
deg	function to convert radians to degrees
rad	function to convert degrees to radians
abs	modulus

Unnamed constants (numbers) can be set as floating point numbers (a string of digits separated by dot) or in integer format.

Symbolic expressions can use the named constants shown in Table O.4.13.2.

Table O.4.13.2. Named Constants.

Constant name	**Description**
M_PI	π – the ratio of a length of circle to its diameter
M_PI_2	$\pi / 2$
M_PI_4	$\pi / 4$
M_SQRT2	$\sqrt{2}$
M_E	e – exponent
M_FI	φ – golden ratio
M_RADDEG	$180 / \pi$ – radians to degrees conversion factor
M_DEGRAD	$\pi / 180$ – degrees to radians conversion factor

The variable name of an analytic expression can be a sequence of string and numeric literals (the first character is a literal of string type), that does not match with function and constant names.

Limitations of symbolic curve expression:

- the maximum length of a variable name is 512 characters;
- the maximum length of an expression is 2048 characters;
- the operations of integration, differentiation, calculation of limits of functions are not allowed;
- there is no possibility of simplifying mathematical expressions (for example, reduction of fractions).

Due to the existing limitations, the correct calculation of the value of the analytical expression can be obtained for relatively simple mathematical expressions and functions. For example, the value of the ratio sin(x)/x at zero may be calculated incorrectly.

The example code below demonstrates the ability to create a symbolic 3D curve. The constructed symbolic curve is shown in Figure O.4.13.1.

```
MbCharacterCurve3D * CreateAnalyticalCurve3D() {
  MbCharacterCurve3D * curve( nullptr );
  // String representations of given coordinate functions.
  string_t xFunction = _T( "100*cos(t)-20*cos(4*t)" );
```

```
  string_t yFunction = _T( "100*sin(t)-20*sin(4*t)" );
  string_t zFunction = _T( "t*10" );
  string_t argument  = _T( "t" );
  double t1 = 0.0, t2 = 5.0 * M_PI;
  // Creating Analytic Coordinate Functions using the Function Factory.
  MbFunctionFactory factory;
  SPtr<MbFunction> x( factory.CreateAnalyticalFunction(xFunction, argument, t1, t2, true) );
  SPtr<MbFunction> y( factory.CreateAnalyticalFunction(yFunction, argument, t1, t2, true) );
  SPtr<MbFunction> z( factory.CreateAnalyticalFunction(zFunction, argument, t1, t2, true) );
  if ( x != nullptr && y != nullptr && z != nullptr ) {
    MbCartPoint3D point( 0.0, 0.0, 0.0 ); // Origin of the local coordinate system.
    MbVector3D axisX( 1.0, 0.0, 0.0 ), axisY( 0.0, 1.0, 0.0 );
    MbPlacement3D place( point, axisX, axisY ); // Local coordinate system.
    MbeLocalSystemType3D cs = ls_CartesianSystem; // Type of local coordinate system.
    // Create a symbolic 3D curve.
    curve = new MbCharacterCurve3D( *x, *y, *z, cs, place, t1, t2 );
  }
  return curve;
}
```

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 $tmin \le t \le tmax$ range.

In **PointOn**(double & *t*, MbCartPoint3D & **r**) method, **r** curve radius vector is described by

$$\mathbf{r}(t) = \mathbf{position.origin} + (radius + t\ tgAlpha\ step\_2pi)\ (\cos(t)\ \mathbf{position.axisX} + \sin(t)\ \mathbf{position.axisY}) + ((t\ step\_2pi)\ \mathbf{position.axisZ})\ \text{vector function.}$$

A conical spiral is shown in Figure O.4.14.1.

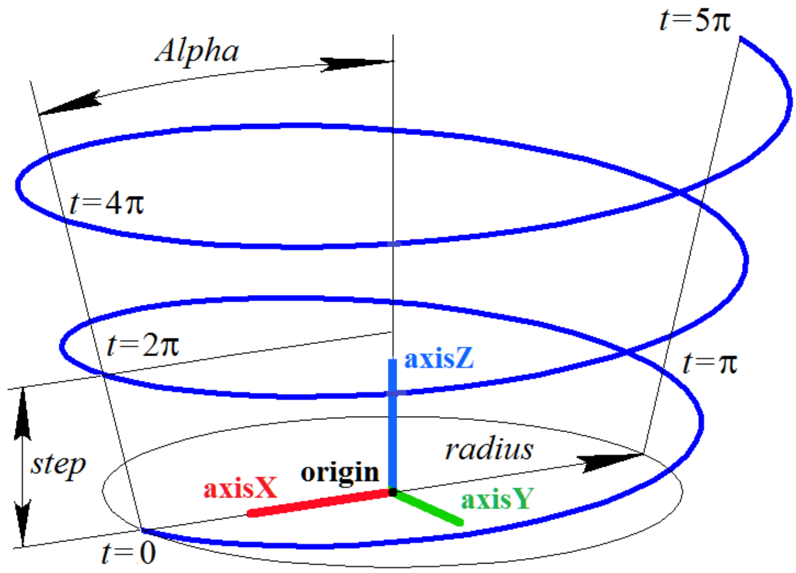


Fig. O.4.14.1.

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

$$\mathbf{r}(t) = \mathbf{position.origin} + \\ radius(t)\,(\cos(t)\,\mathbf{position.axisX} + \sin(t)\,\mathbf{position.axisY}) + \\ ((t \cdot step\,/\,2\pi)\,\mathbf{position.axisZ})\ \text{vector function,}$$

where *radius*(*t*) is local radius. *radius*(*t*) local radius is calculated as follows. We use defined *t* parameter to calculate $t{\cdot}step/2\pi$ 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 $t{\cdot}step/2\pi$ 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.

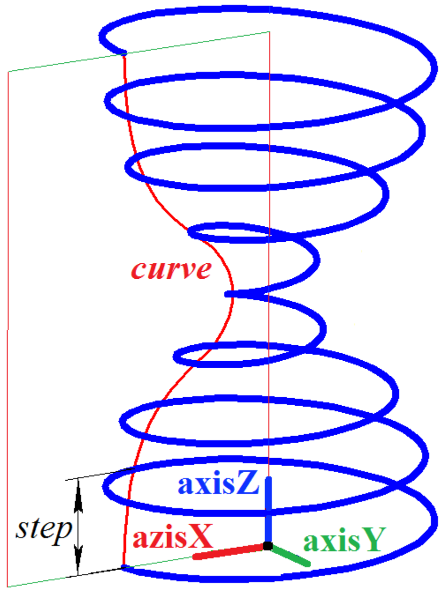


Fig. O.4.15.1.

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.

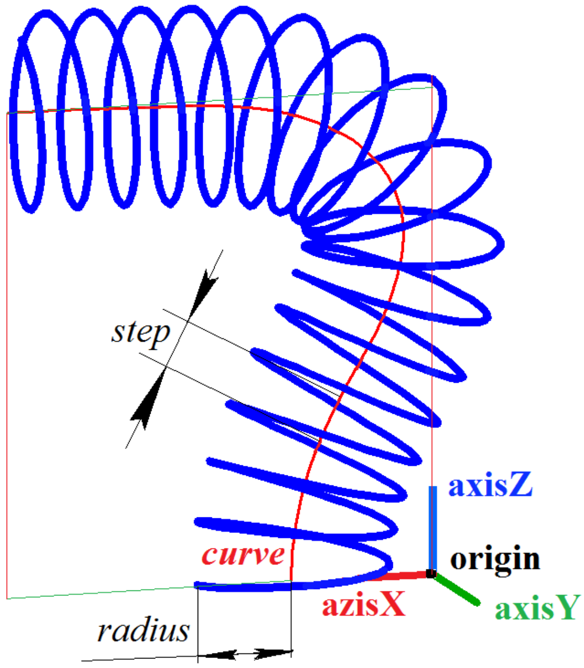


Fig. O.4.16.1.

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

$$r(t) = (1 - 3w^2 + 2w^3)\mathrm{point1} + (3w^2 + 2w^3)\mathrm{point2} + \\ + ((w - 2w^2 + w^3)\mathrm{derive1} + (-w^2 + w^3)\mathrm{derive2})(\mathrm{tmax} - \mathrm{tmin})$$

vector function, where $w = \dfrac{t - \mathrm{tmin}}{\mathrm{tmax} - \mathrm{tmin}}$ 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.

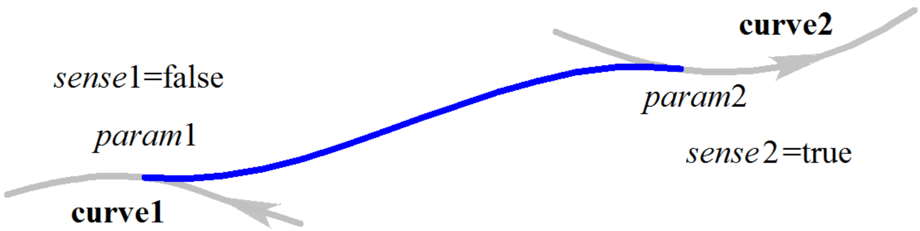


Fig. O.4.17.1.

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.

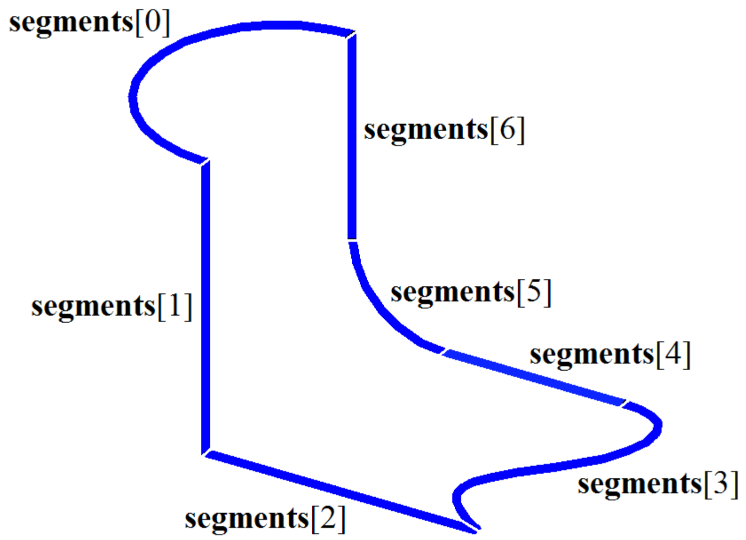


Fig. O.4.18.1.

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.

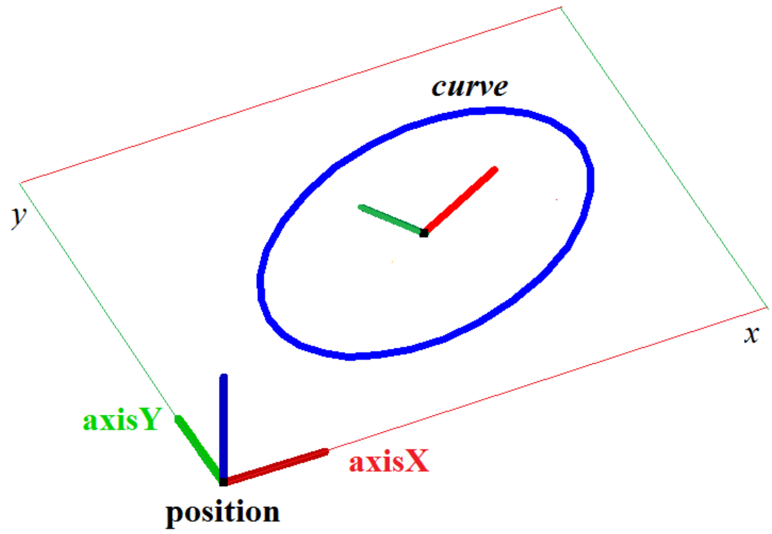


Fig. O.4.19.1.

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.

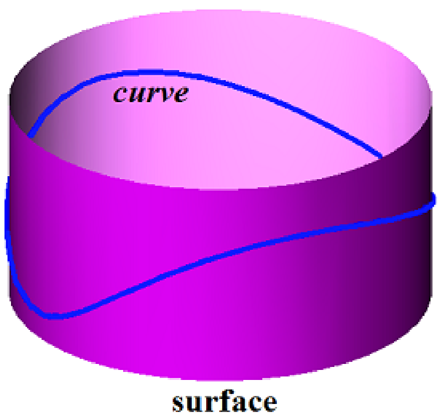


Fig. O.4.20.1.

Two-dimensional curve in surface parameter definition area is shown in Figure O.4.20.2.

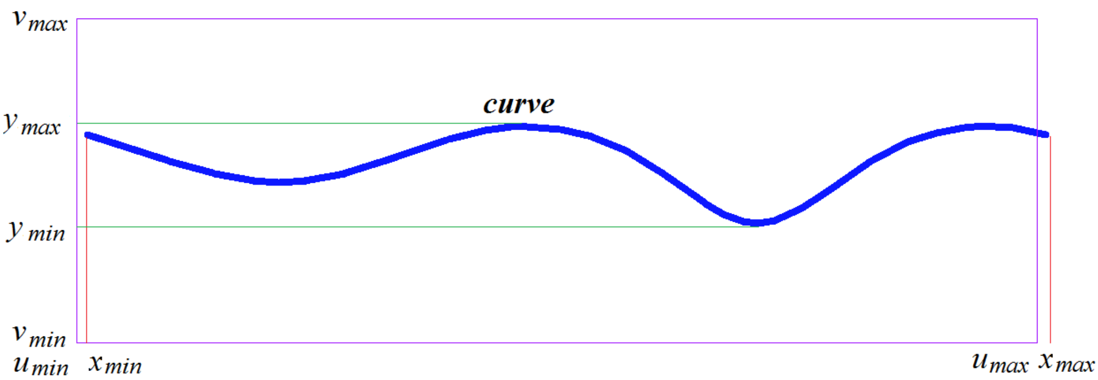


Fig. O.4.20.2.

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.

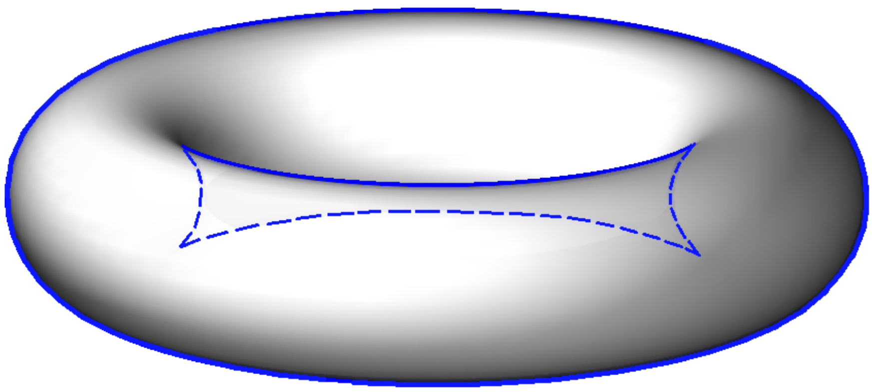


Fig. O.4.21.1.

The silhouette curve of torus surface from other observation point is shown in Figure O.4.21.2.

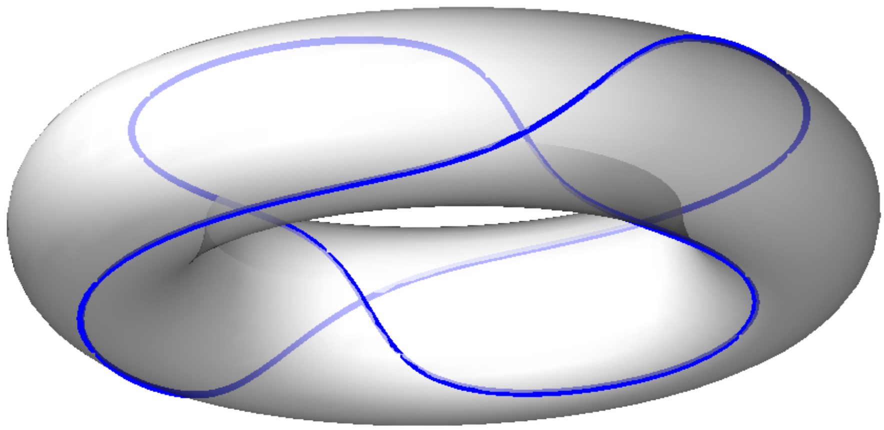


Fig. O.4.21.2.

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.

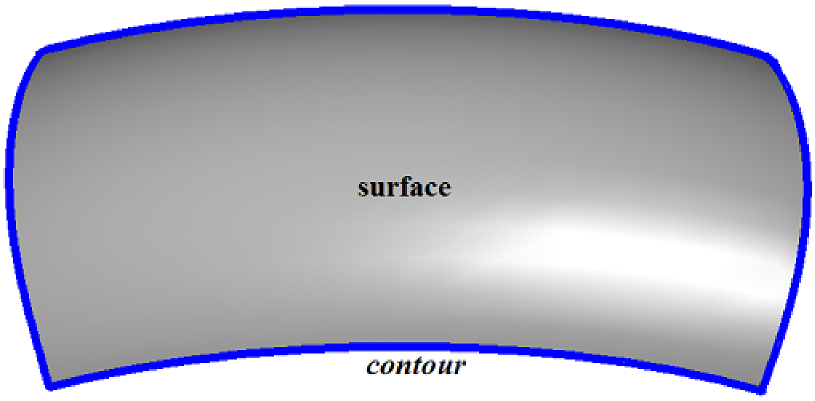


Fig. O.4.22.1.

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.

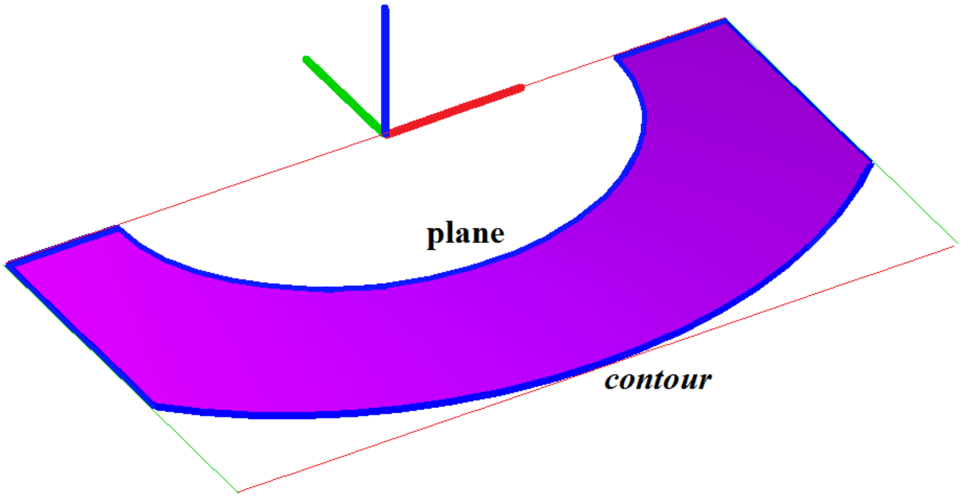


Fig. O.4.23.1.

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.

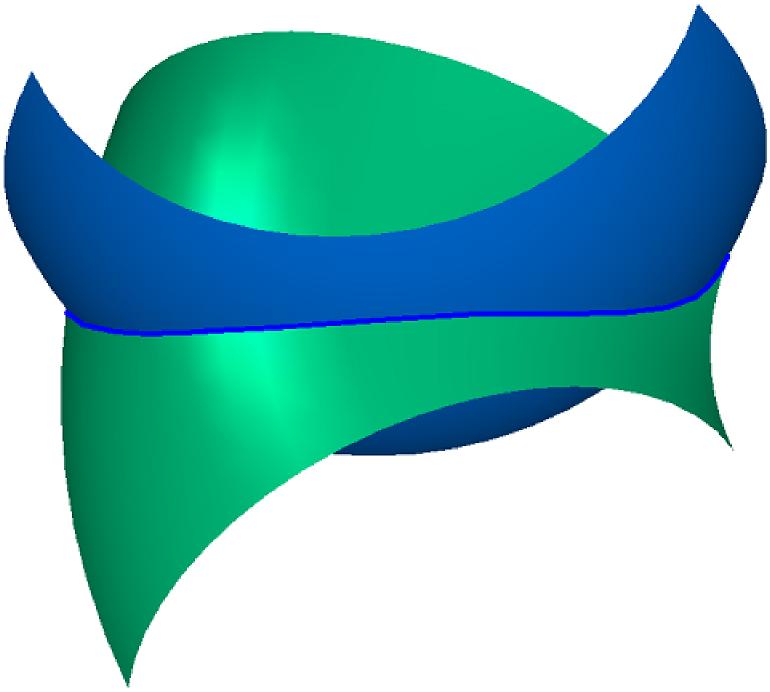


Fig. O.4.24.1.

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.

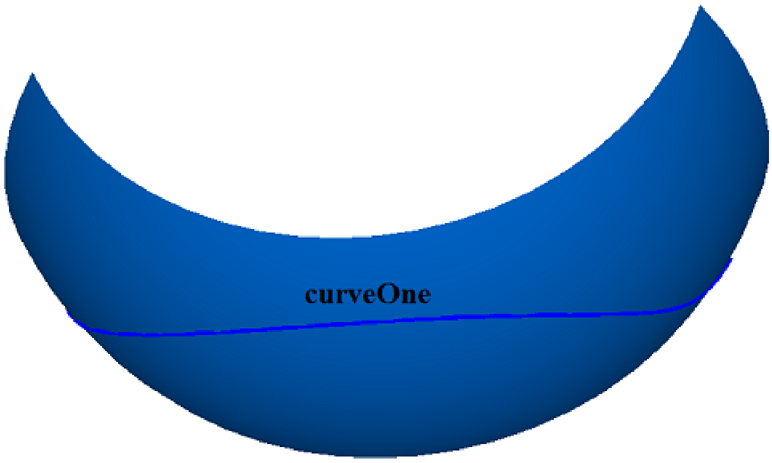


Fig. O.4.24.2.

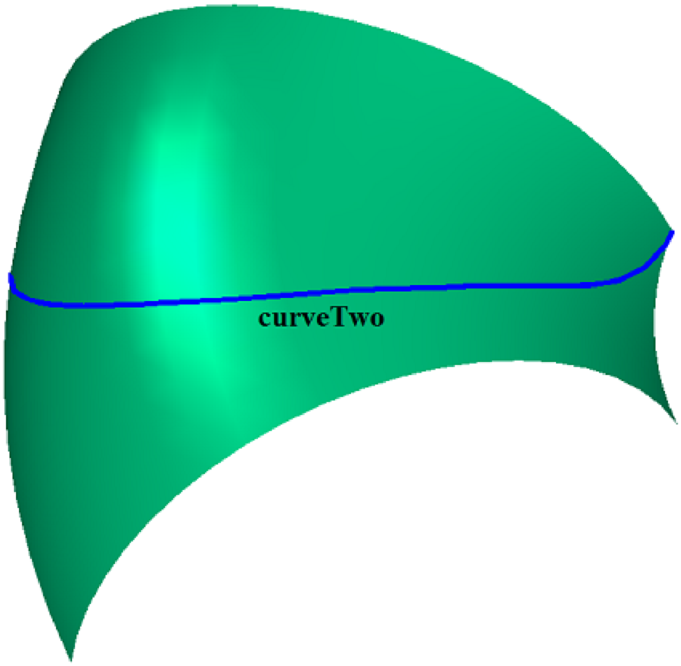


Fig. O.4.24.3.

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

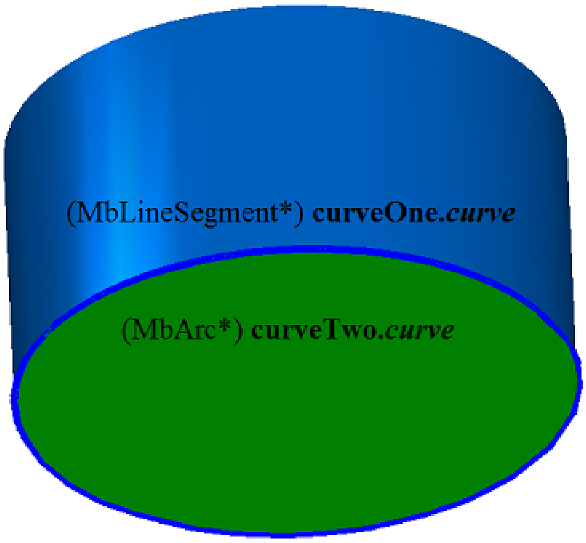


Fig. O.4.24.4.

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

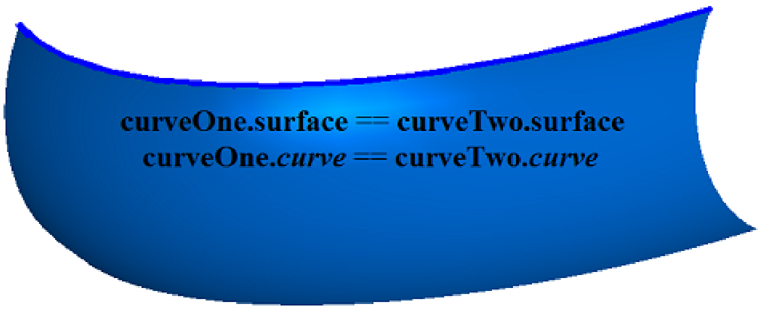


Fig. O.4.24.5.

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:

$$\textbf{curveOne.surface}(u_1,v_1) = \textbf{plane}(x,y),$$
$$\textbf{curveTwo.surface}(u_2,v_2) = \textbf{plane}(x,y),$$

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.

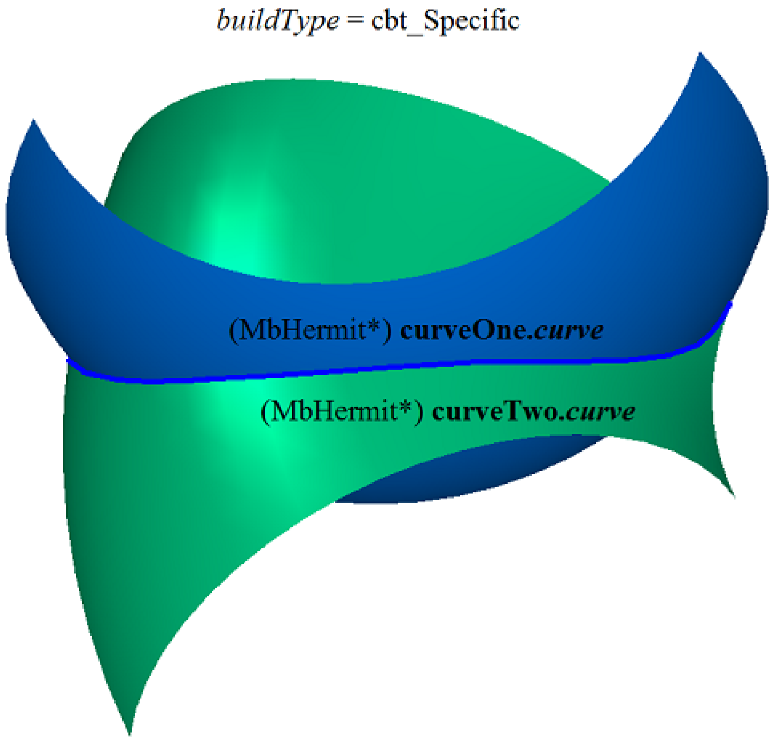


Fig. O.4.24.6.

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.

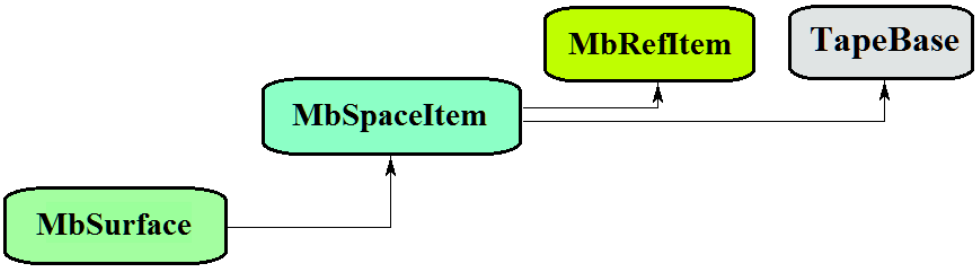


Figure O.5.1.1.

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 $u_{min} \le u \le u_{max}$, $v_{min} \le v \le v_{max}$ 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:

$$s_u=\frac{\partial \text{surface}(u,v)}{\partial u};\ s_v=\frac{\partial \text{surface}(u,v)}{\partial v};$$

$$s_{uu}=\frac{\partial^2 \text{surface}(u,v)}{\partial u^2};\ s_{vv}=\frac{\partial^2 \text{surface}(u,v)}{\partial v^2};\ s_{uv}=\frac{\partial^2 \text{surface}(u,v)}{\partial u \partial v}=\frac{\partial^2 \text{surface}(u,v)}{\partial v \partial u};$$

$$s_{uuu}=\frac{\partial^3 \text{surface}(u,v)}{\partial u^3};\ s_{uuv}=\frac{\partial^3 \text{surface}(u,v)}{\partial u \partial u \partial v};\ s_{uvv}=\frac{\partial^3 \text{surface}(u,v)}{\partial u \partial v \partial v};\ s_{vvv}=\frac{\partial^3 \text{surface}(u,v)}{\partial v^3}$$

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 $\mathbf{s}(u,v)$ 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.

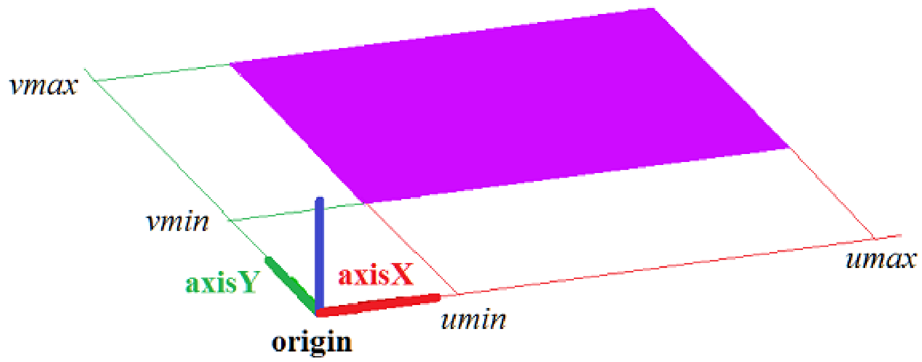


Figure O.5.2.1.

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.

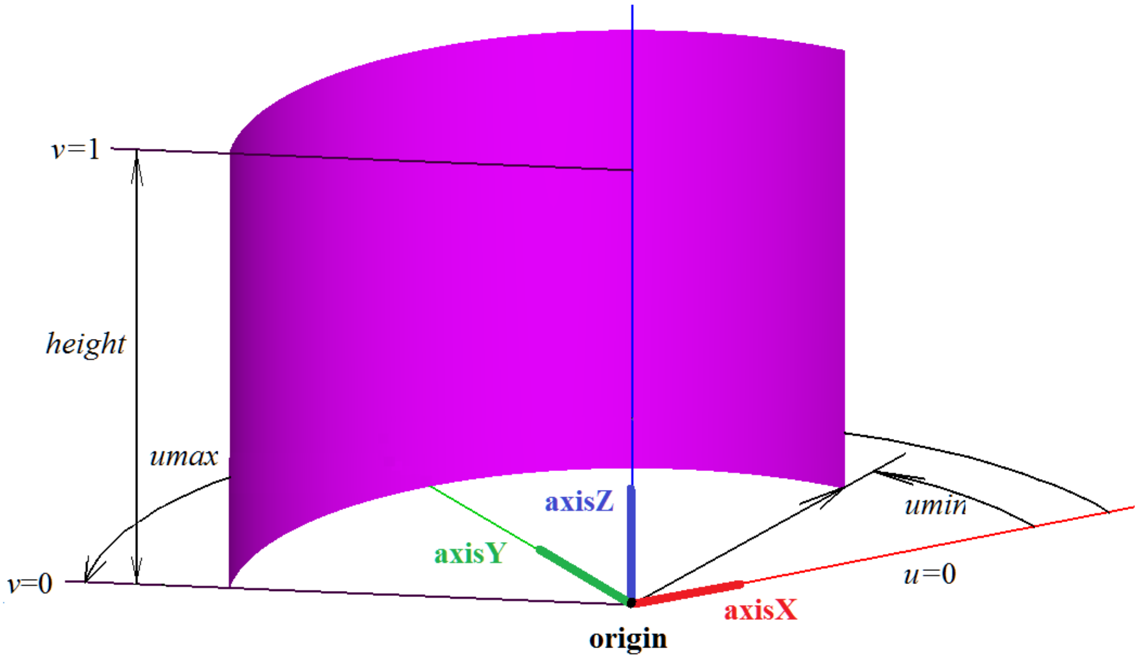


Figure O.5.3.1.

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. $umax-umin=2\pi$ holds for a periodic surface; $umax-umin<2\pi$ 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

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (radius + height\ v\ tg(angle))\ (\cos(u)\ \mathbf{position.axisX} + \sin(u)\ \mathbf{position.axisY}) + height\ v\ \mathbf{position.axisZ}$$ vector function.

A conical surface is shown in Figure O.5.4.1.

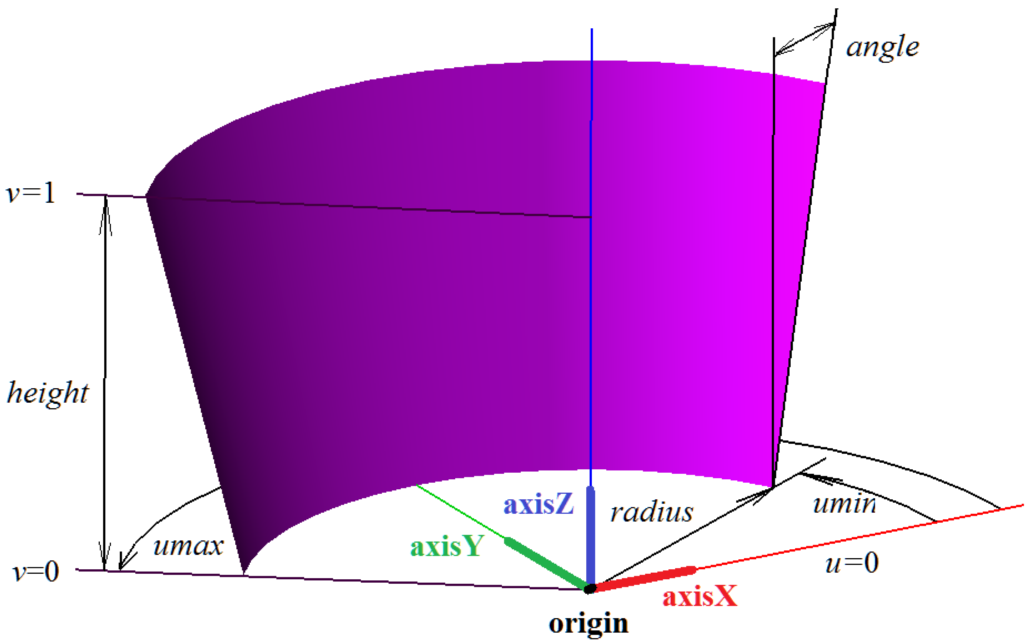


Figure O.5.4.1.

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

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + radius\,\sin(v)\,\mathbf{position.axisZ}$$ vector function.

A sphere is shown in Figure O.5.5.1.

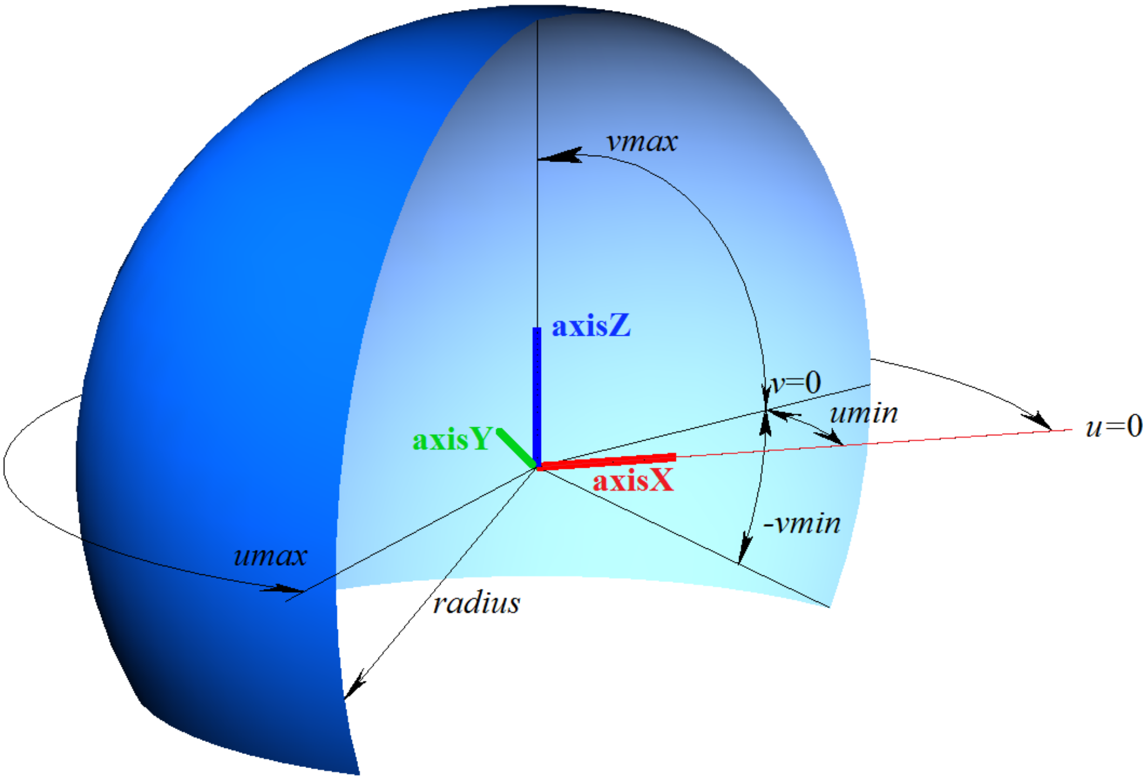


Figure O.5.5.1.

The radius of sphere should be greater than zero: *radius*>0. A sphere has poles for its parameter ($v\pi/2$ and $v=-\pi/2$). The following inequalities should hold for surface limiting parameters: *umin*<*umax*, *vmin*<*vmax*, *vmax*<=$\pi/2$, *vmin*>=$-\pi/2$.

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 $umin \le u \le umax$ 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 along the arc from XY plane of local coordinate system towards **position.axisZ** vector. Surface second parameter (v) takes values in $vmin \le v \le vmax$ 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

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (majorRadius + (minorRadius\ \cos(v))\ (\cos(u)\ \mathbf{position.axisX} + \sin(u)\ \mathbf{position.axisY}) + minorRadius\ \sin(v)\ \mathbf{position.axisZ}$$

vector function.

A toroidal surface is shown in Figure O.5.6.1.

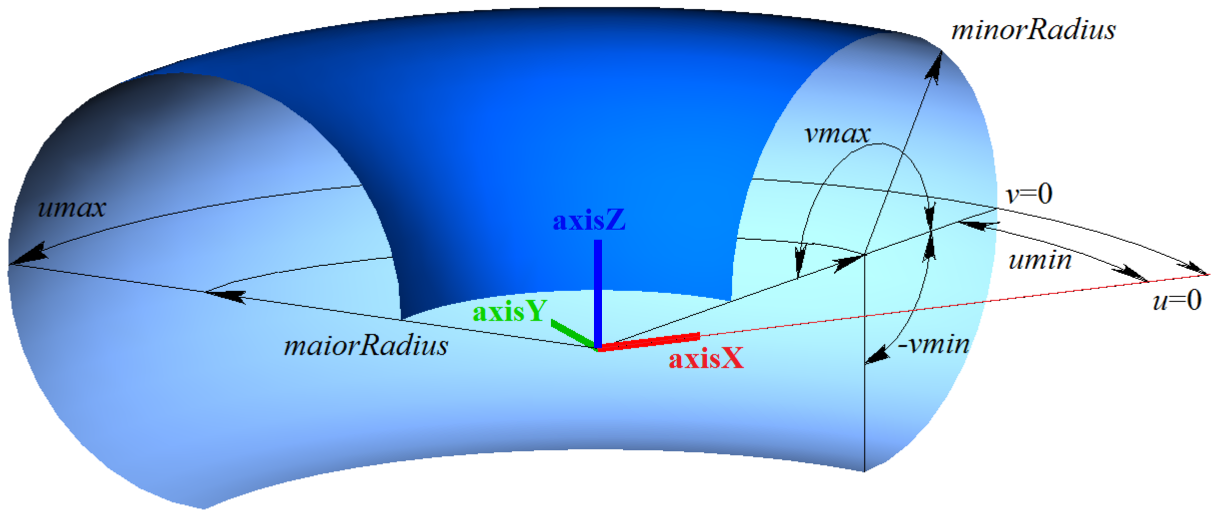


Figure O.5.6.1.

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

s(*u*,*v*) = **curve**(*u*) + (**direction** *distance v*) vector function.

Extrusion surface is shown in Figure O.5.7.1.

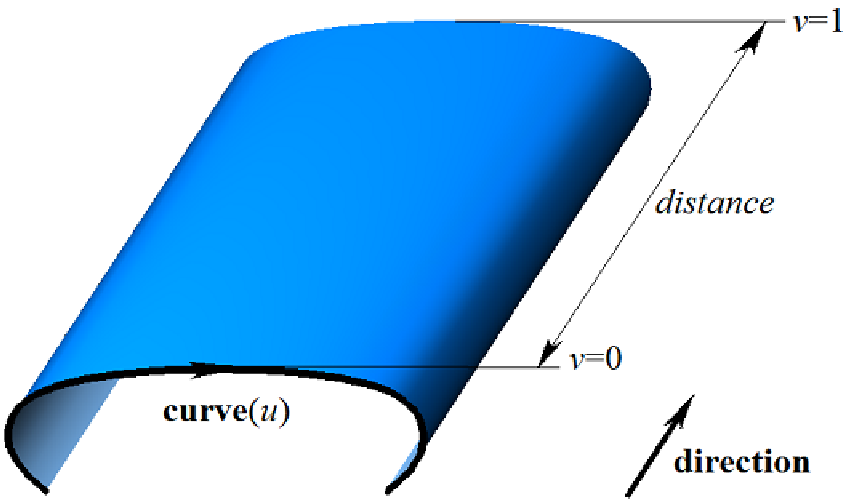


Figure O.5.7.1.

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=2π 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 **M**(v) is rotation matrix. Please note that multiplication of **(curve**(u)–**position.origin**) vector by **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.

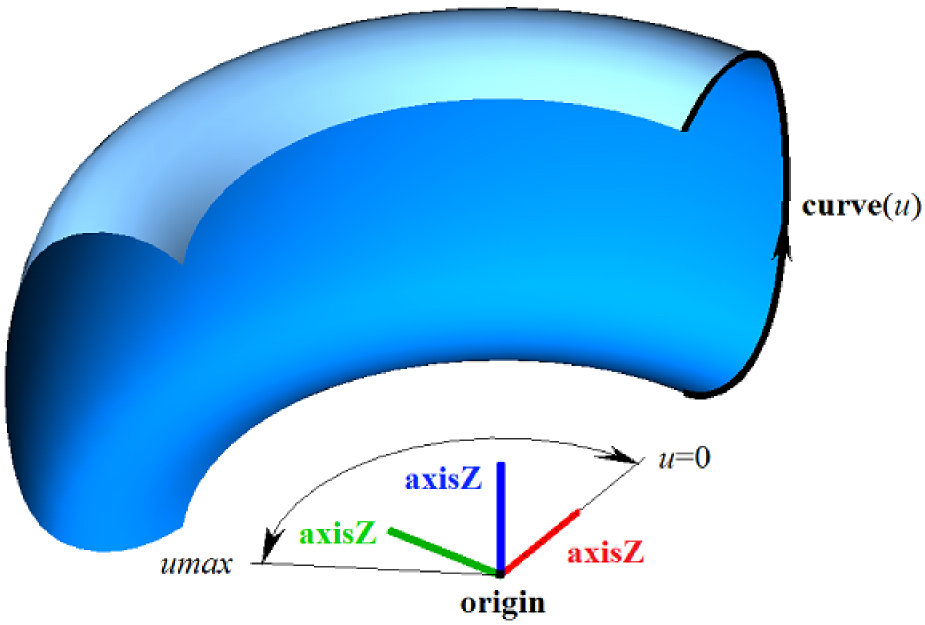


Figure O.5.8.1.

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

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + \mathbf{curve}(u) - \mathbf{origin} \text{ vector function.}$$

The general case for sliding surface is shown in Figure O.5.9.1.

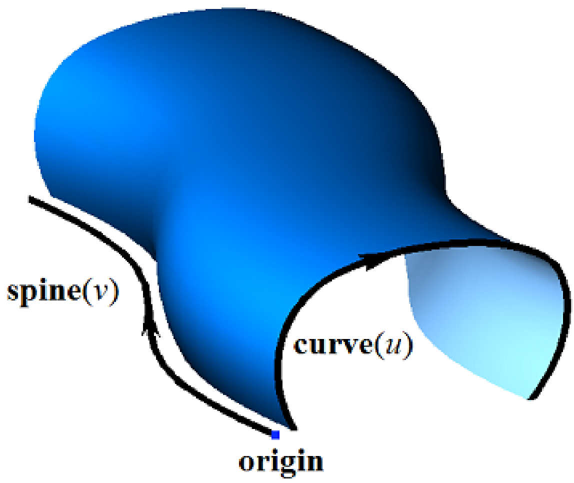


Figure O.5.9.1.

In a special case for **PointOn**(double u, double v, MbCartPoint3D & **s**) method, **s** plane radius vector is described by

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + (\mathbf{curve}(u) - \mathbf{origin})\,(1-w) + (\mathbf{brink}(t) - \mathbf{ending})\,w \text{ vector function,}$$

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.

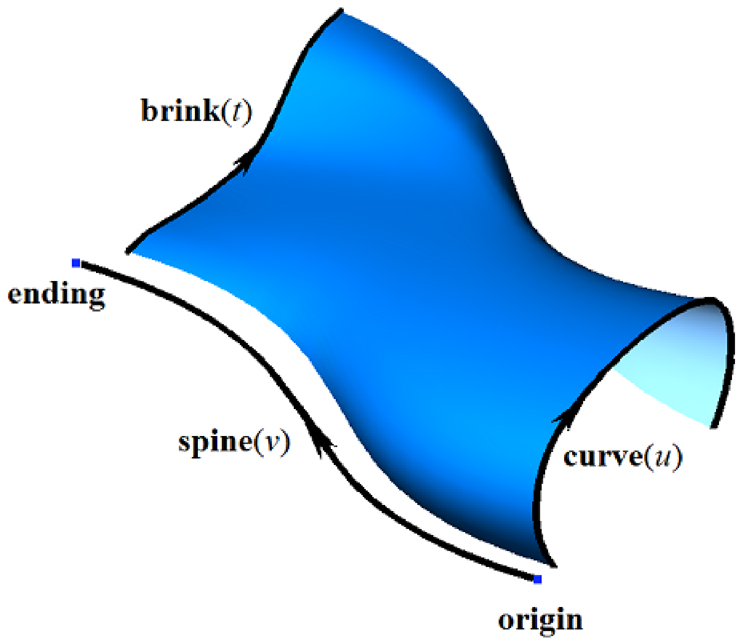


Figure O.5.9.2.

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.

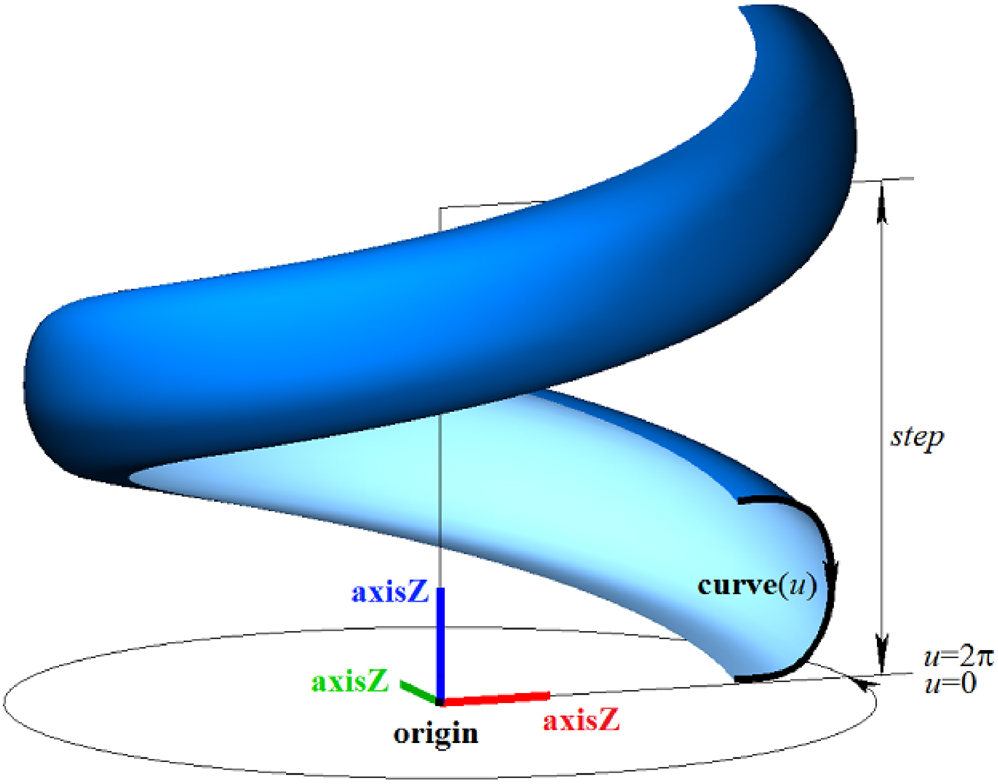


Figure O.5.10.1.

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 $umin \leq u \leq umax$ 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 $vmin \leq v \leq vmax$ 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 $\mathbf{A}(v)$ 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. $\mathbf{A}(v)$ matrix depends on second surface parameter. Rows of $\mathbf{A}(v)$ 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.

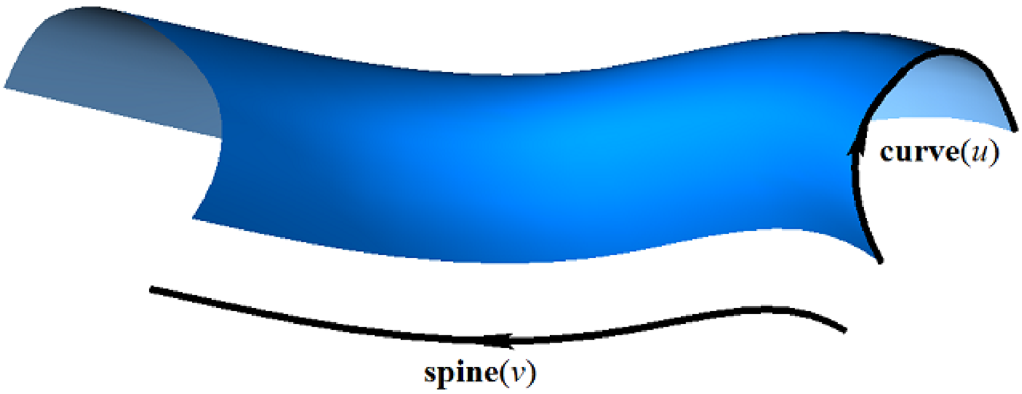


Figure O.5.11.1.

$\mathbf{i}_1(v)$ 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 bodies with the help of kinematic operations with kinked composite guiding curves, see Figure O.5.12.1.

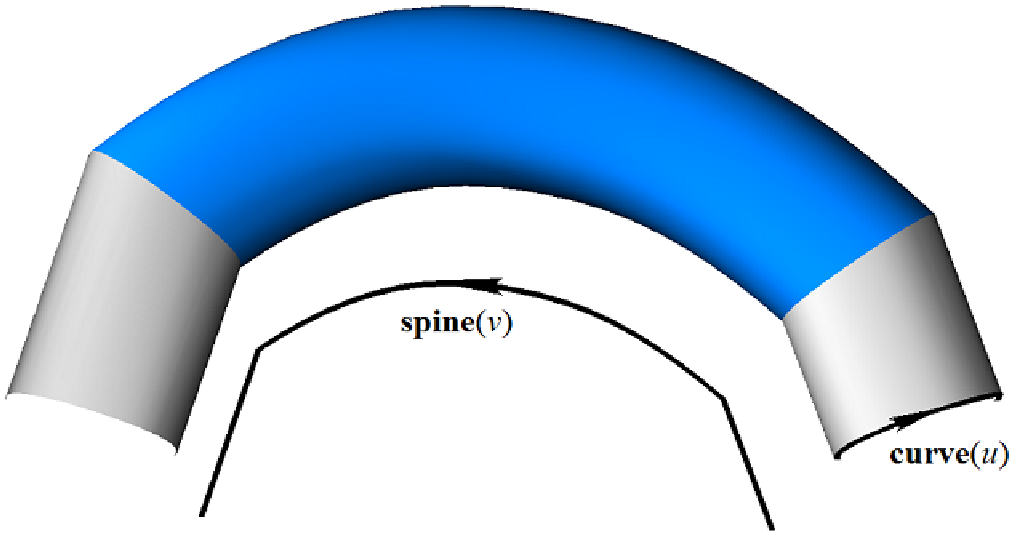


Figure O.5.12.1.

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 $umin \le u \le umax$ 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 $vmin \le v \le vmax$ 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 $w = \dfrac{v - \mathrm{vmin}}{\mathrm{vmax} - \mathrm{vmin}}$. A sectorial surface is shown in Figure O.5.13.1.

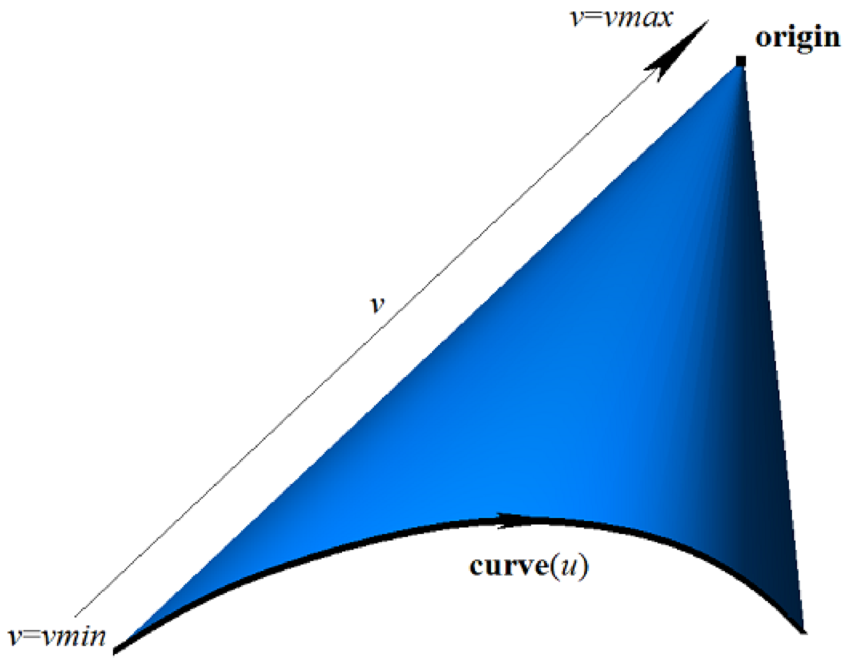


Figure O.5.13.1.

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

s(*u*,*v*) = **curve**(*u*) (1–*w*) + **sline**(*t*) *w* vector function,

where $w = \dfrac{v - \mathrm{vmin}}{\mathrm{vmax} - \mathrm{vmin}}$, *t*=*tmin*+(*u*–*umin*)*dt*. A ruled surface is shown in Figure O.5.14.1.

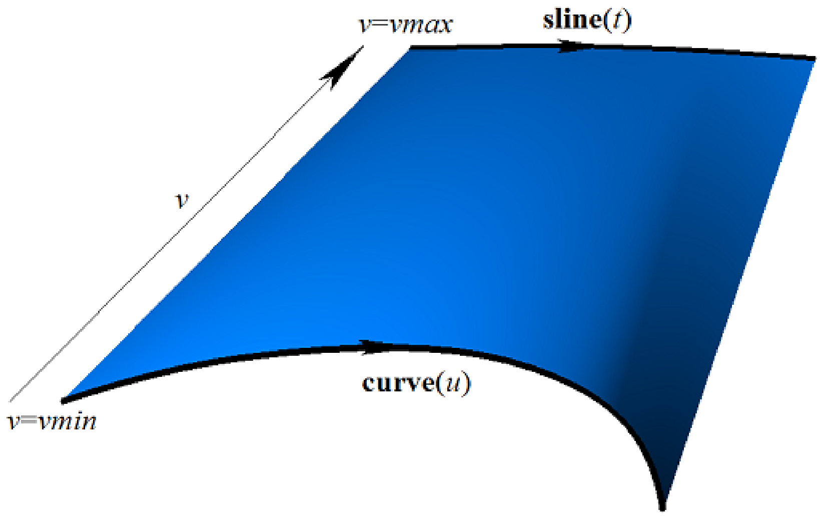


Figure O.5.14.1.

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 $umin \le u \le umax$ 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 $vmin \le v \le vmax$ 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 $vParams[i] \le v \le vParams[i+1]$. 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.

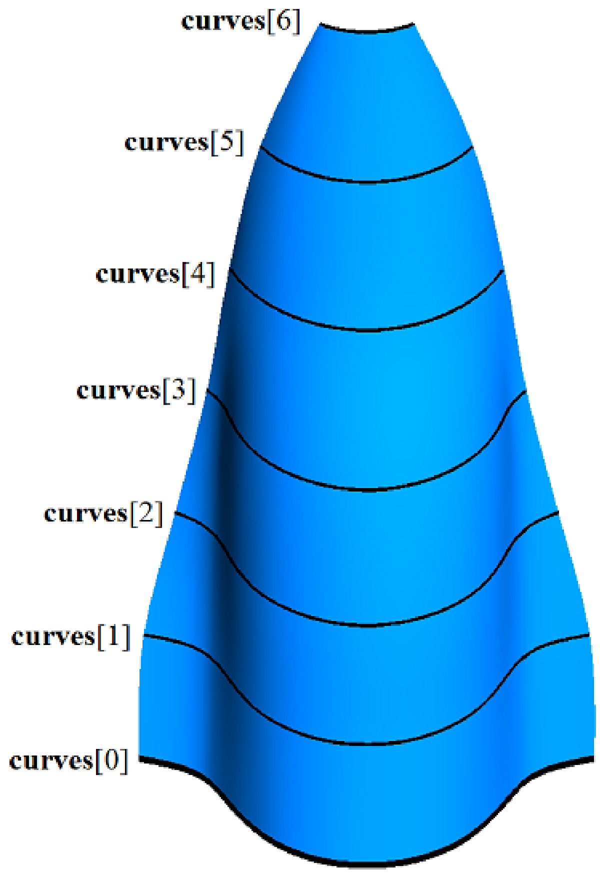


Figure O.5.15.1.

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 $vmin \le v \le vmax$ 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 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.

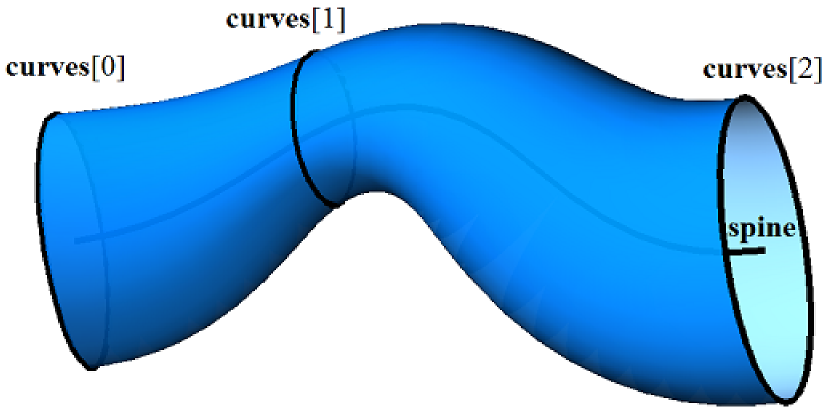


Figure O.5.16.1.

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: $t0min$, $t0max$, $t1min$, $t1max$, $t2min$, $t2max$. 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'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 $0 \le v \le 1$ range. The surface can't be periodic by the second parameter.

curve0, **curve**1, **curve**2 curves should have intersection points or crossing points. $t0min$, $t0max$, $t1min$, $t1max$, $t2min$, $t2max$ 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 $w0=1-u$, $w1=0.5(u-uv)$, $w2=0.5(u+uv)$ are barycentric coordinates of the surface, $t0=w2 \cdot t0min+(1-w2) \cdot t0max$, $s0=(1-w1) \cdot t0min+w1 \cdot t0max$ are parameters of **curve**0 curve, $t1=w0 \cdot t1min+(1-w0) \cdot t1max$, $s1=(1-w2) \cdot t1min+w2 \cdot t1max$ are parameters of **curve**1 curve, $t2=w1 \cdot t2min+(1-w1) \cdot t2max$, $s2=(1-w0) \cdot t2min+w0 \cdot t2max$ are parameters of **curve**2 curve. A surface based on three crossing curves is shown in Figure O.5.17.1.

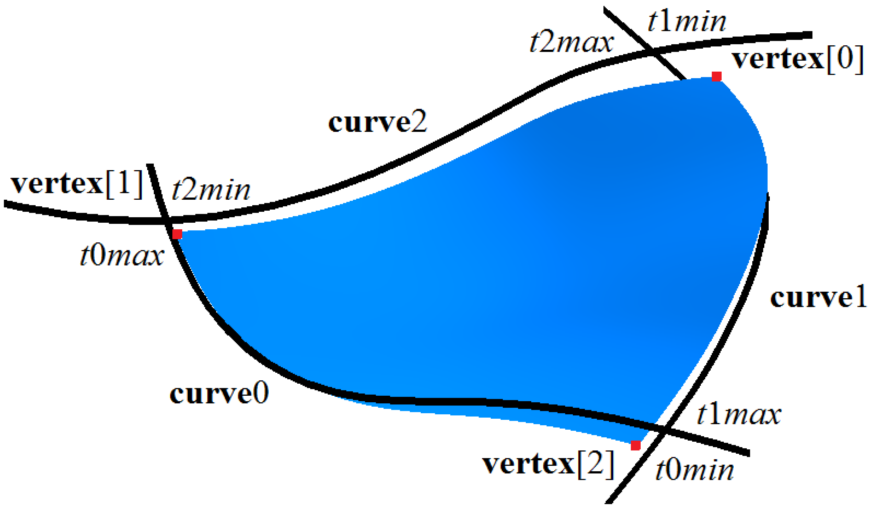


Figure O.5.17.1.

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.

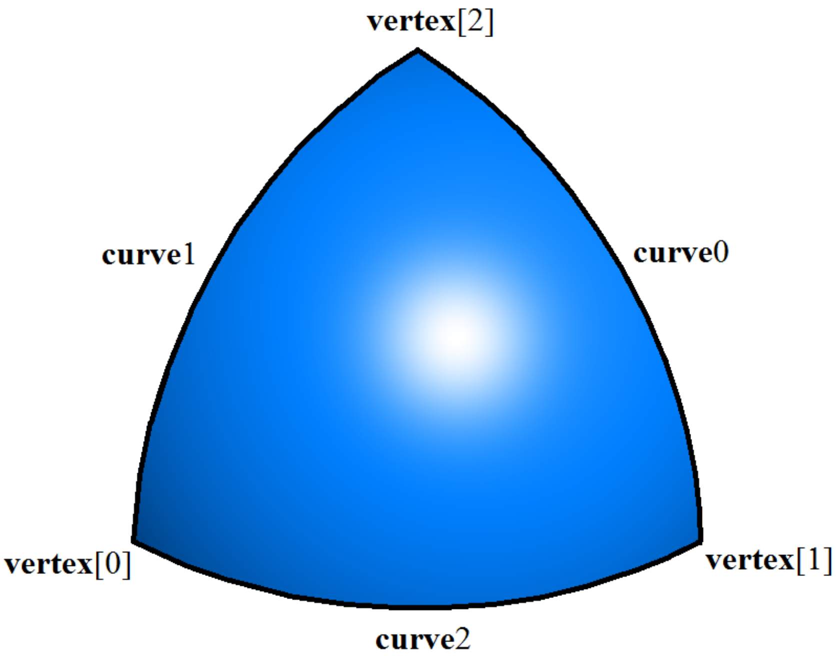


Figure O.5.17.2.

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 $w0$=0 then the surface contains a segment of **curve**0 curve; if $w1$=0 then the surface contains a segment of **curve**1 curve; if $w2$=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, t*2*min, t*2*max*, *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

$$\begin{aligned}\mathbf{s}(u,v) = \; & (1-v)\,(\mathbf{curve}0(t0) - (1-u)\,\mathbf{vertex}[0]) + \\ & + u\,(\mathbf{curve}1(t1) - (1-v)\,\mathbf{vertex}[1]) + \\ & + v\,(\mathbf{curve}2(t2) - u\,\mathbf{vertex}[2]) + \\ & + (1-u)\,(\mathbf{curve}3(t3) - v\,\mathbf{vertex}[3]) \text{ vector function,}\end{aligned}$$

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, $t1=(1-v)\cdot t1min+v\cdot t1max$ is the parameter of **curve**1 curve, $t2=(1-u)\cdot t2min+u\cdot t2max$ is the parameter of **curve**2 curve, $t3=(1-v)\cdot t3min+v\cdot t3max$ is the parameter of **curve**3 curve. Coons surface based on four crossing curves is shown in Figure O.5.18.1.

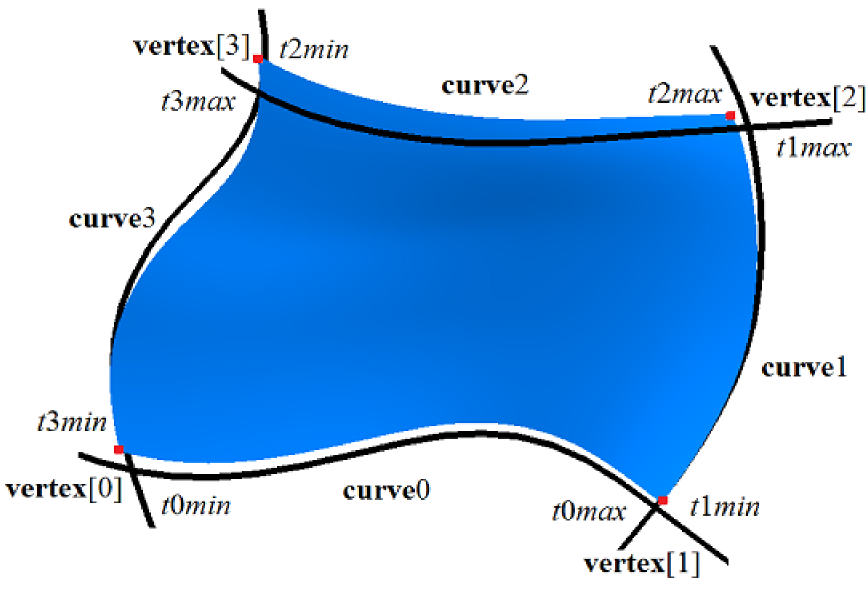


Figure O.5.18.1.

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*, *t*3*min*, *t*3*max*), *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 $0 \leq u \leq 1$ 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 $0 \leq v \leq 1$ 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.

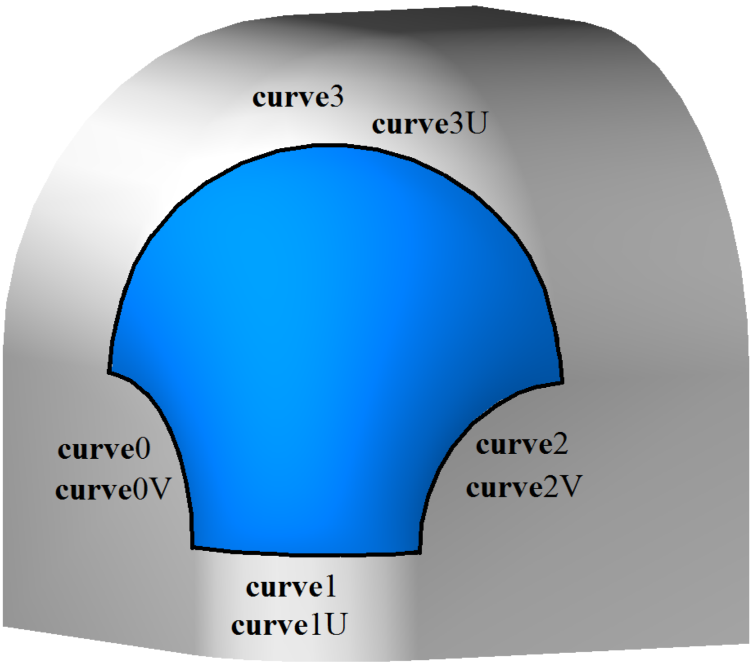


Figure O.5.19.1.

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 $uParams[0] \le u \le uParams[uParams.\mathrm{MaxIndex}()]$ 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 $vParams[0] \le v \le vParams[vParams.\mathrm{MaxIndex}()]$ 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.

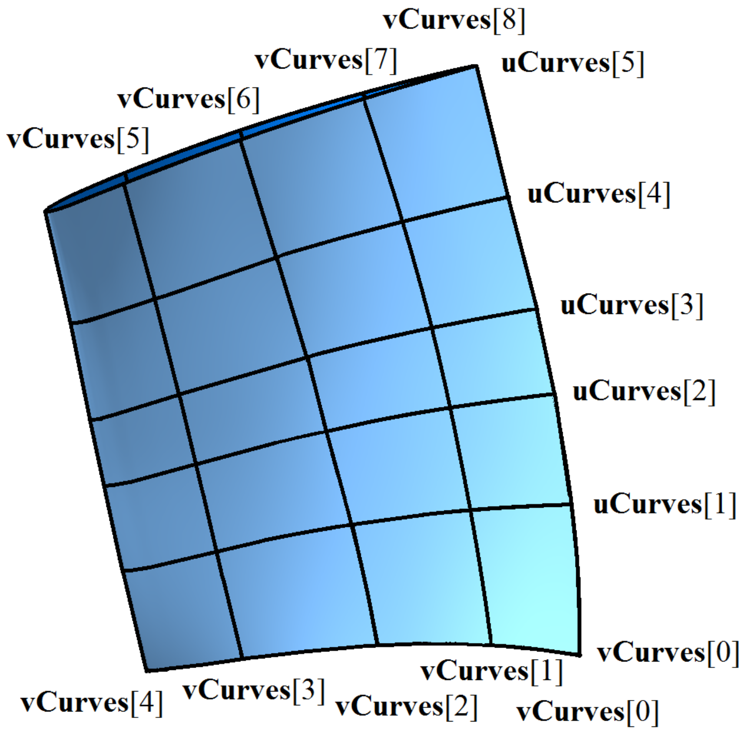


Figure O.5.20.1.

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.

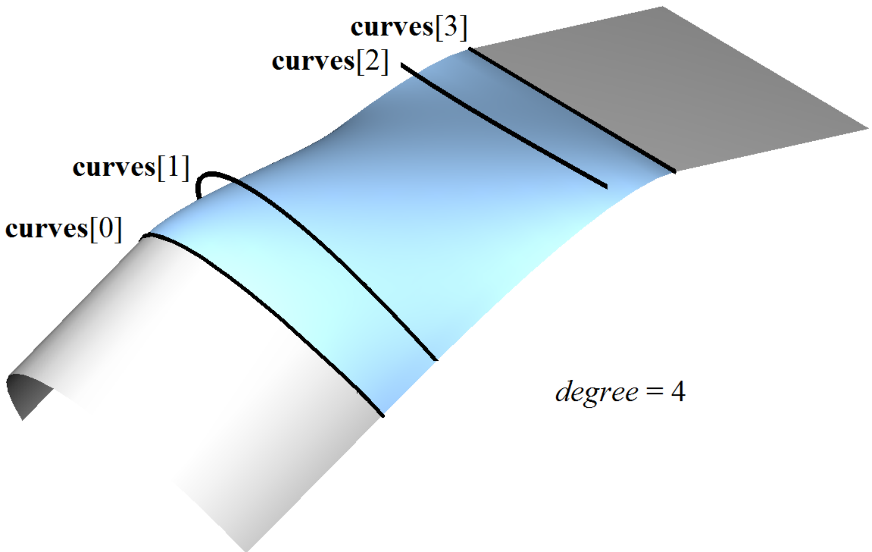


Figure O.5.21.1.

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.

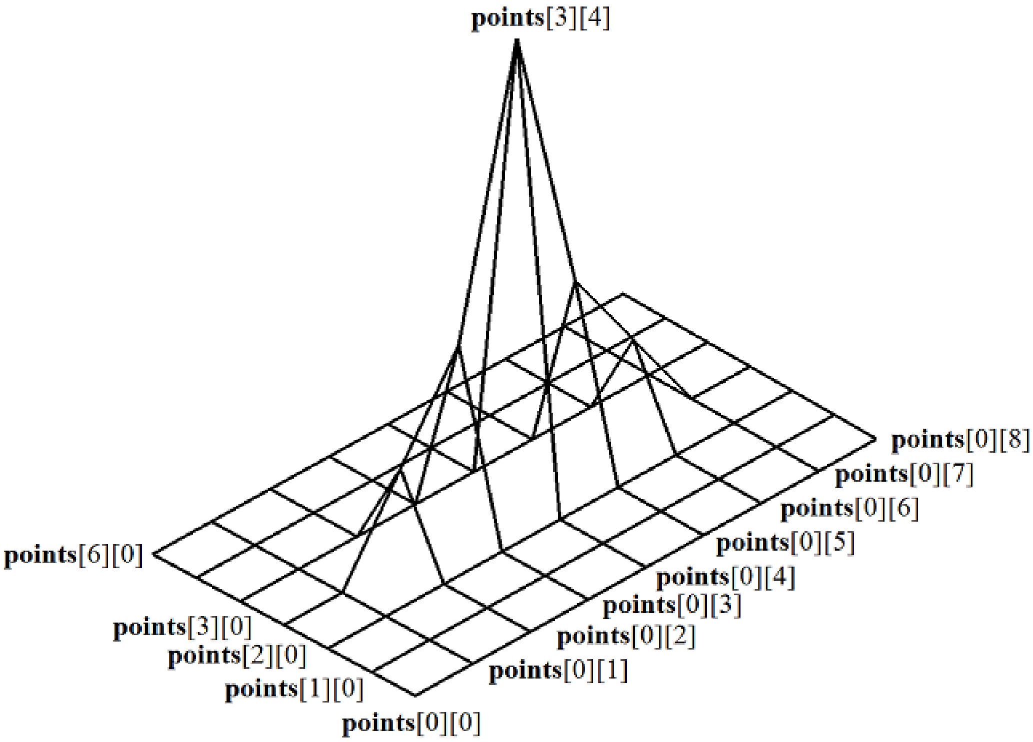


Figure O.5.22.1.

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.

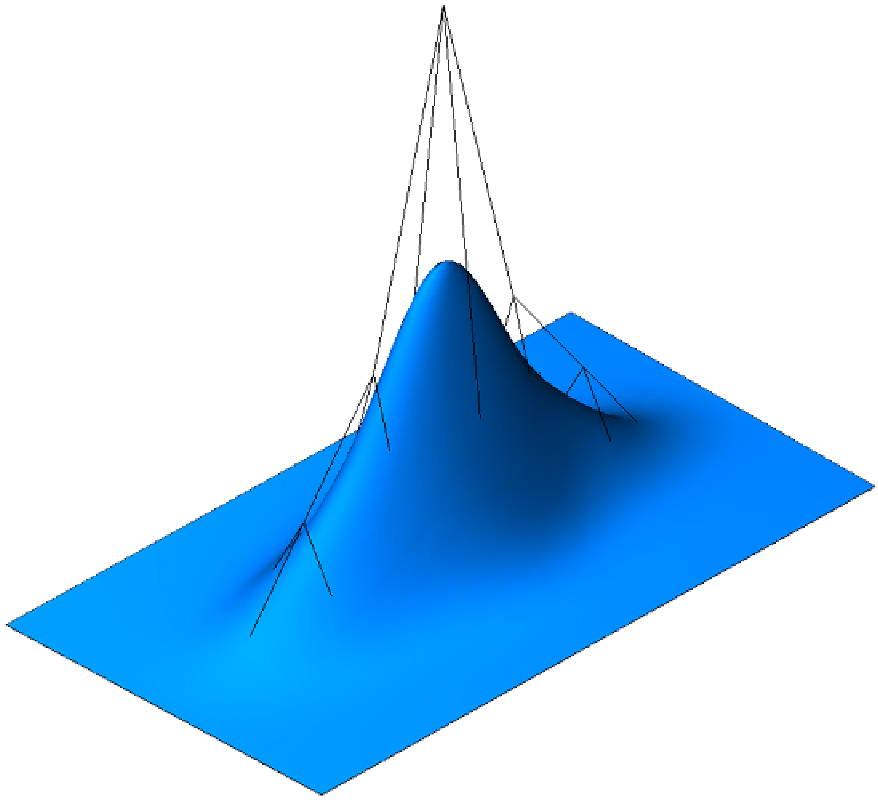


Figure O.5.22.2.

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.

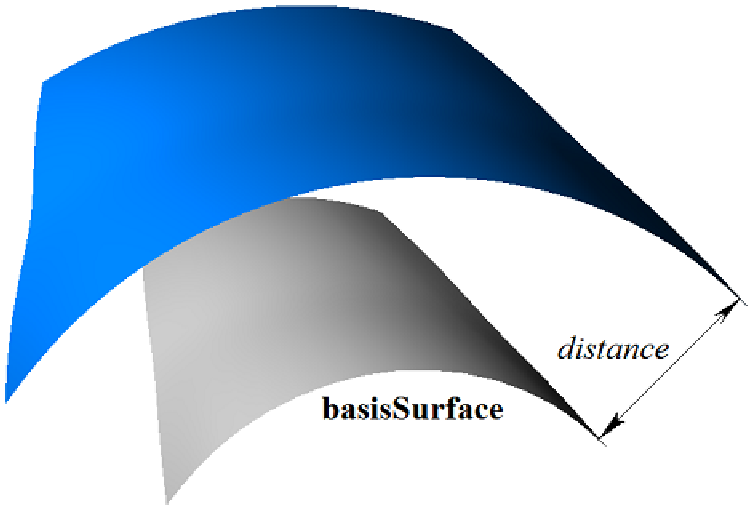


Figure O.5.23.1.

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.

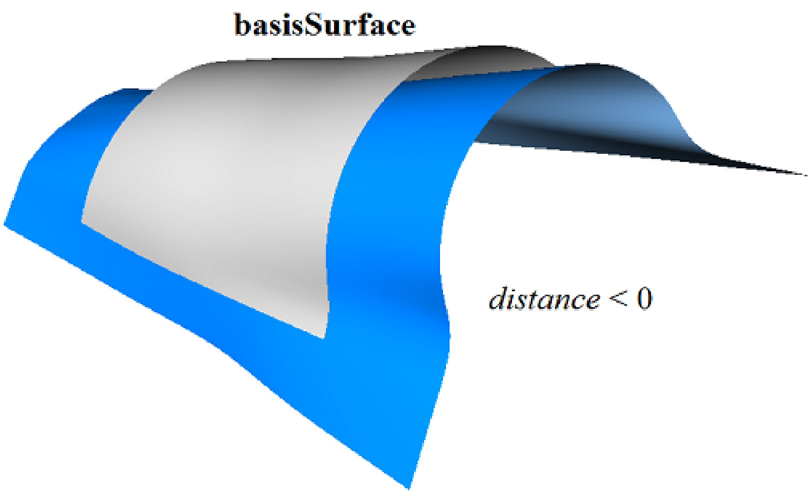


Figure O.5.23.2.

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 $w = \dfrac{v - \text{vmin}}{\text{vmax} - \text{vmin}}$. A chamfer surface is shown in Figure O.5.24.1.

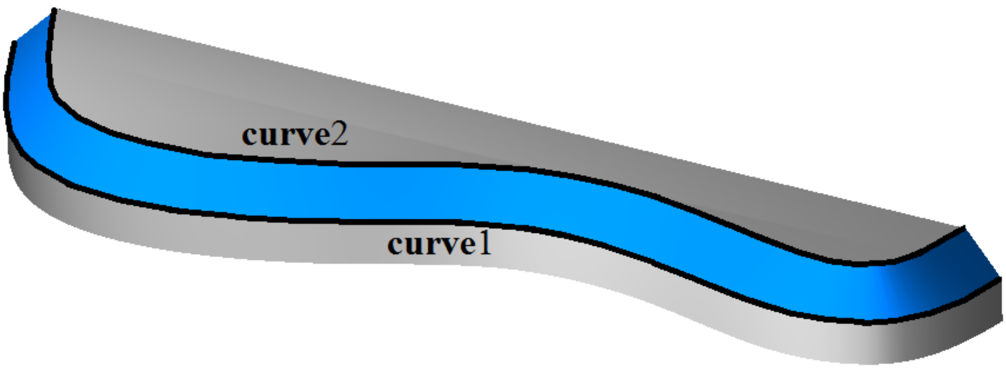


Figure O.5.24.1.

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 $umin \le u \le umax$ 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 $vmin \le v \le vmax$ 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

$$s(u,v) = \frac{(1-v)^2 \mathrm{curve}\,1(u) + 2t(1-v)\mathrm{weight}\,0(u)\mathrm{curve}\,0(u) + v^2 \mathrm{curve}\,2(u)}{(1-v)^2 + 2v(1-v)\mathrm{weight}\,0(u) + v^2}.$$

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.

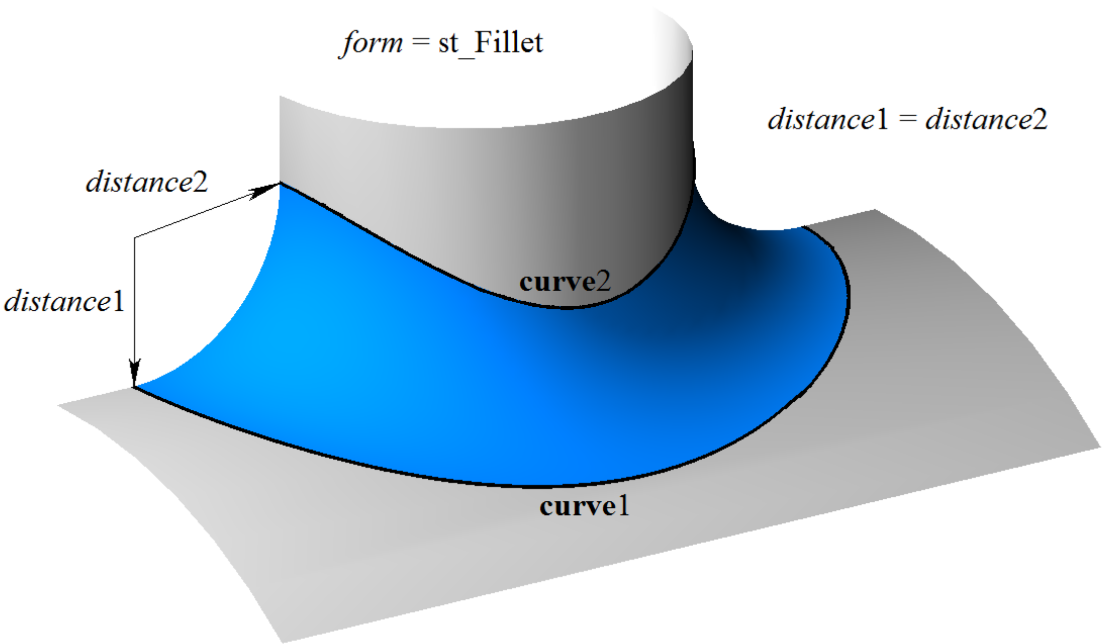


Figure O.5.25.1.

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.

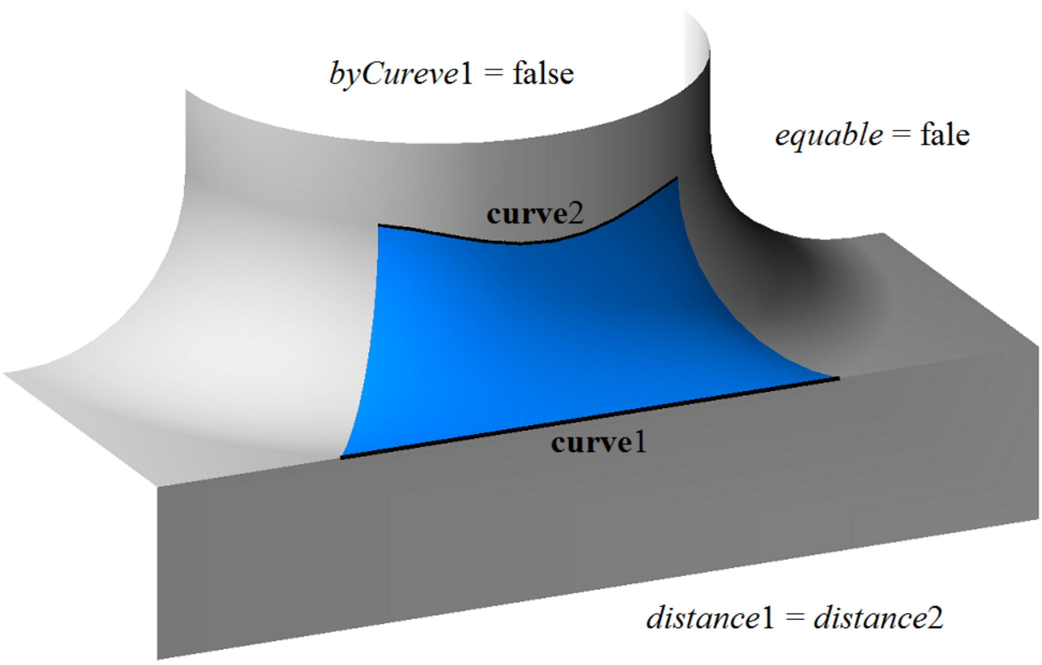


Figure O.5.25.2.

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.

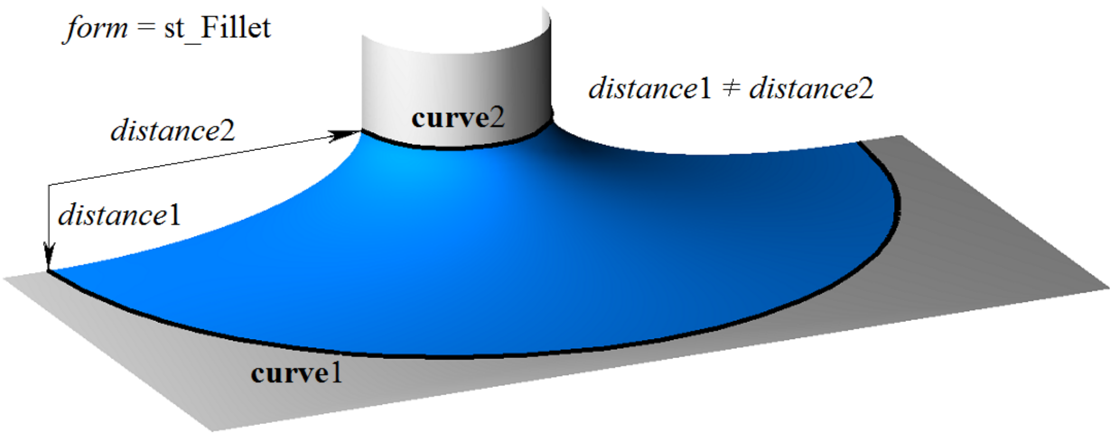


Figure O.5.25.3.

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.

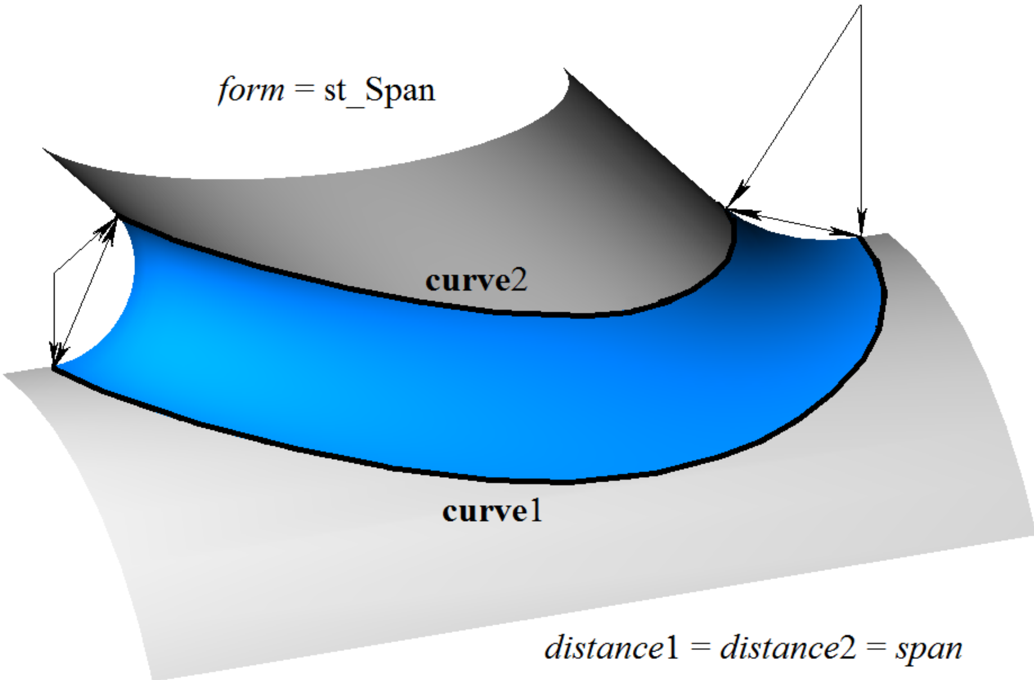


Figure O.5.25.4.

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.

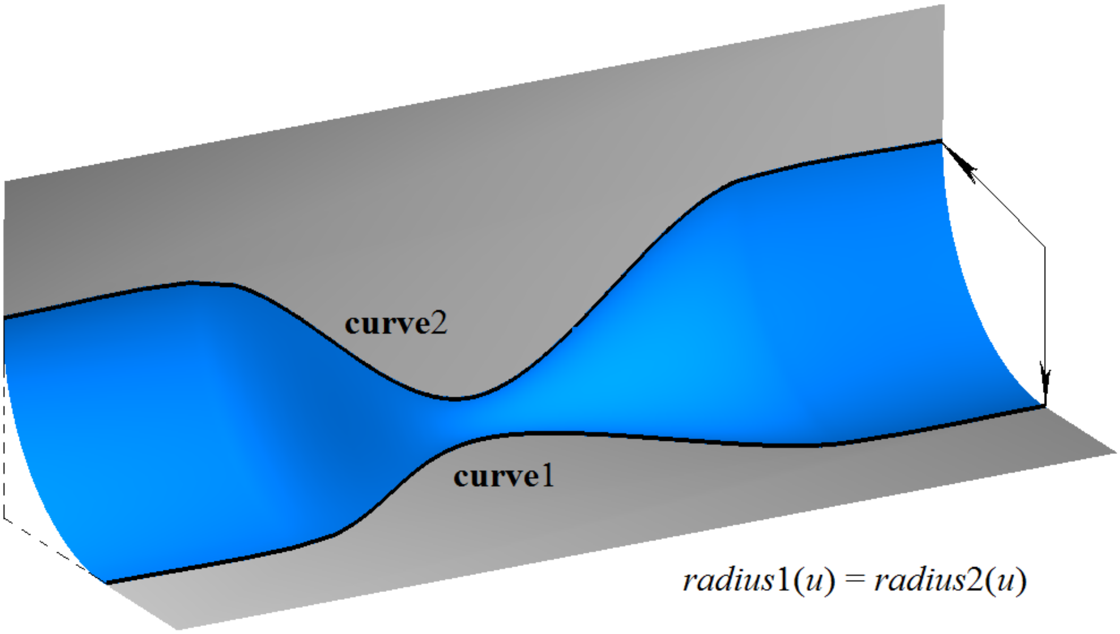


Figure O.5.26.1.

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.

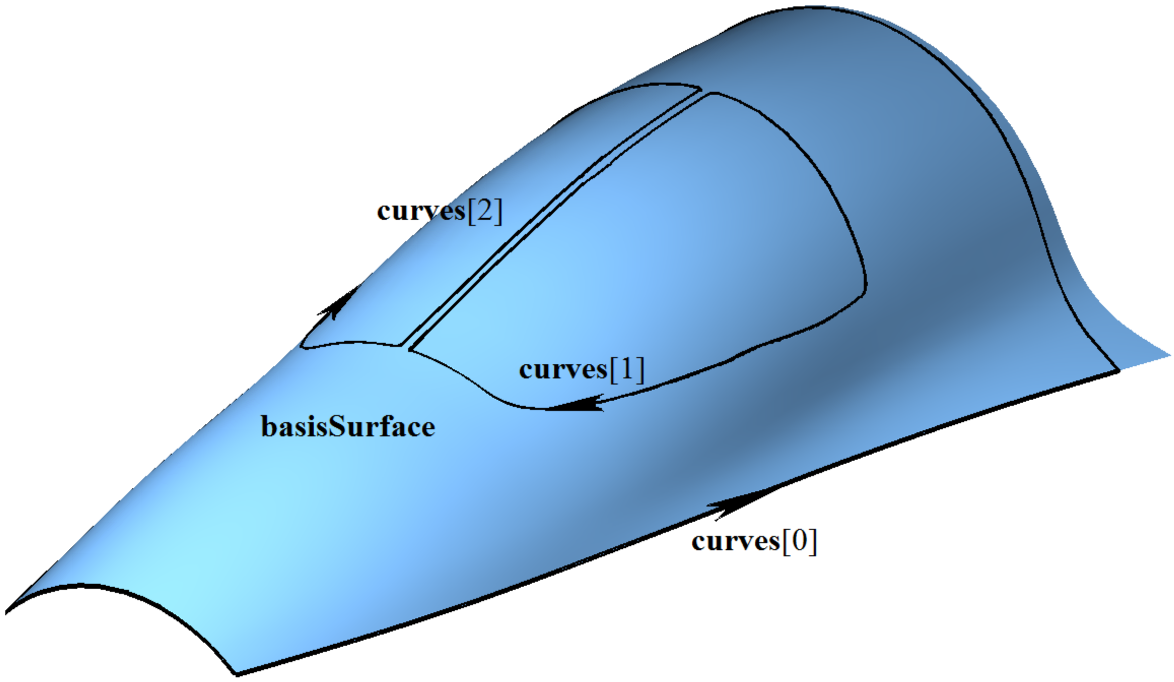


Figure O.5.27.1

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.

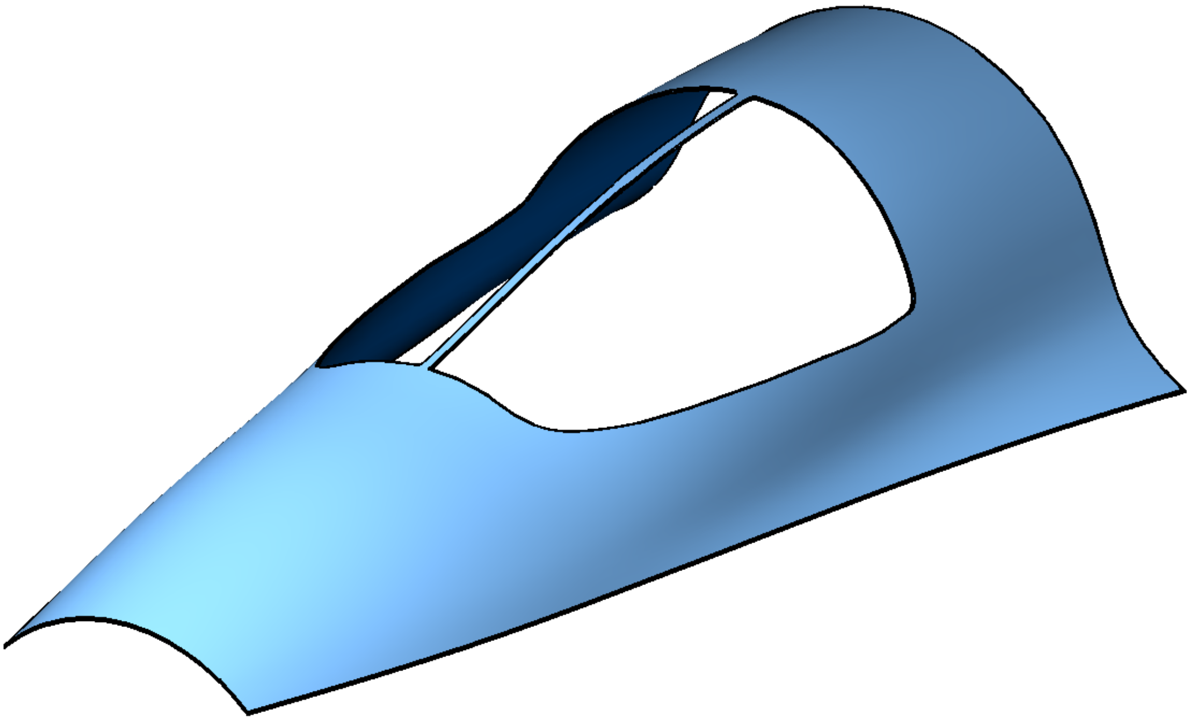


Figure O.5.27.2.

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.

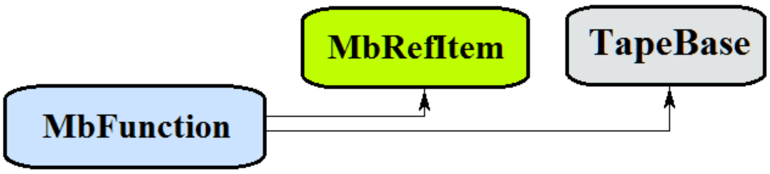


Fig. O.6.1.1.

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

$$function(t) = f(t) \ scalar \ function$$

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

$$f(t) = (1 - 3w^2 + 2w^3)\mathrm{valueList}[i] + (3w^2 + 2w^3)\mathrm{valueList}[i+1] +$$
$$+ ((w - 2w^2 + w^3)\mathrm{valueList}[i] + (-w^2 + w^3)\mathrm{valueList}[i+1])(\mathrm{tList}[i+1] - \mathrm{tList}[i]) \text{ function,}$$

where $w = \dfrac{t - \mathrm{tList}[i]}{\mathrm{tList}[i+1] - \mathrm{tList}[i]}$ is the local parameter of *tList*[*i*]≤*t*≤*tList*[*i*+1] working segment. Cubic Hermite function is shown in Figure O.6.4.1.

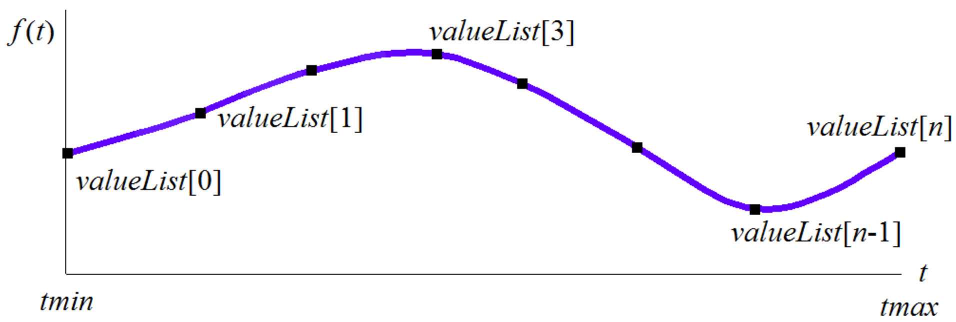


Fig. O.6.4.1.

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

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

function,where $w = \dfrac{t - \mathrm{tList}[i]}{\mathrm{tList}[i+1] - \mathrm{tList}[i]}$ is the local parameter of $tList[i] \le t \le tList[i+1]$ working segment. Cubic spline function is shown in Figure O.6.5.1.

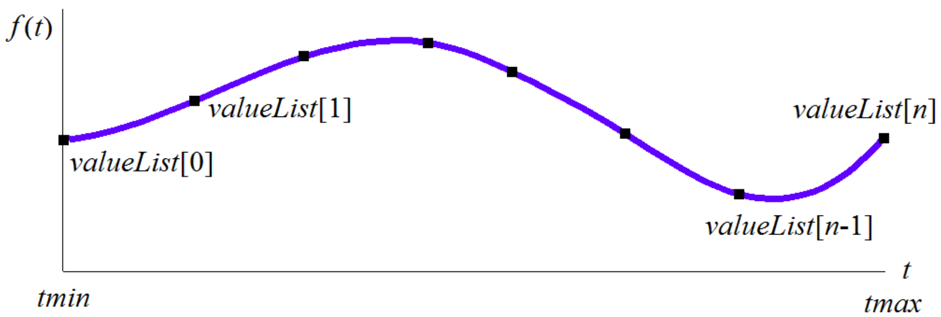


Fig. O.6.5.1.

Function parameter range is within $tmin \le t \le tmax$ 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 $tmin \le t \le tmax$ 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.

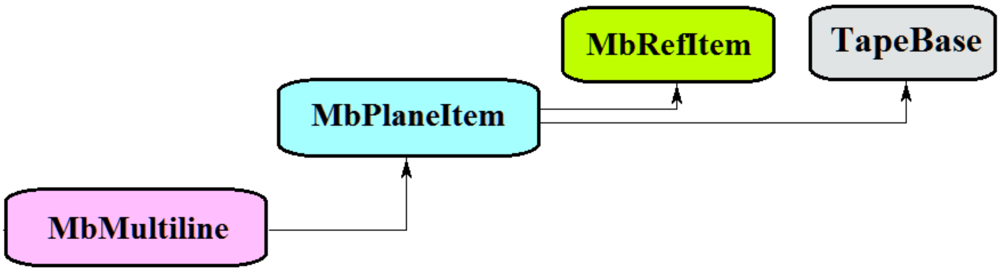


Fig. O.6.7.1.

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.

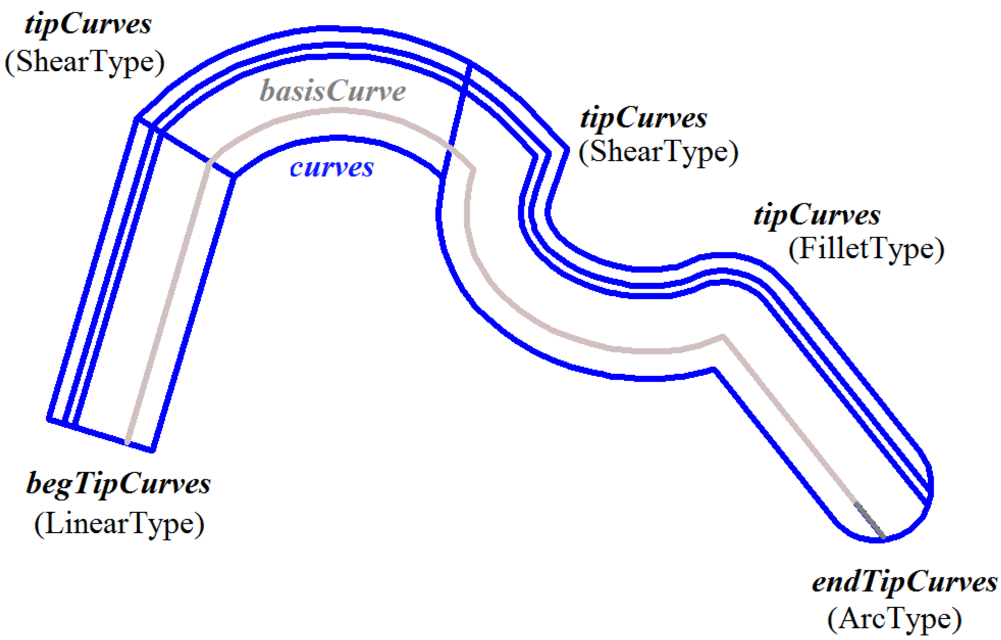


Fig. O.6.7.2.

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.

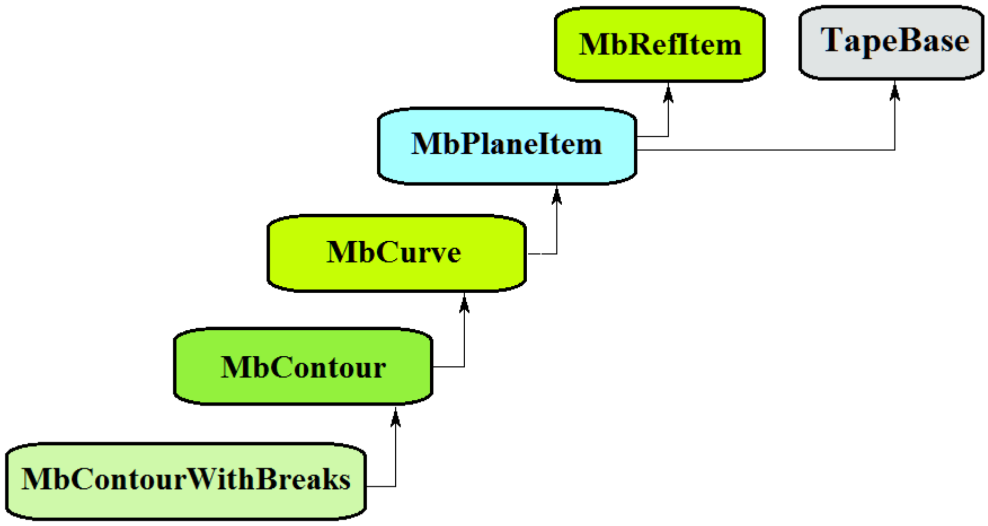


Fig. O.6.8.1.

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.

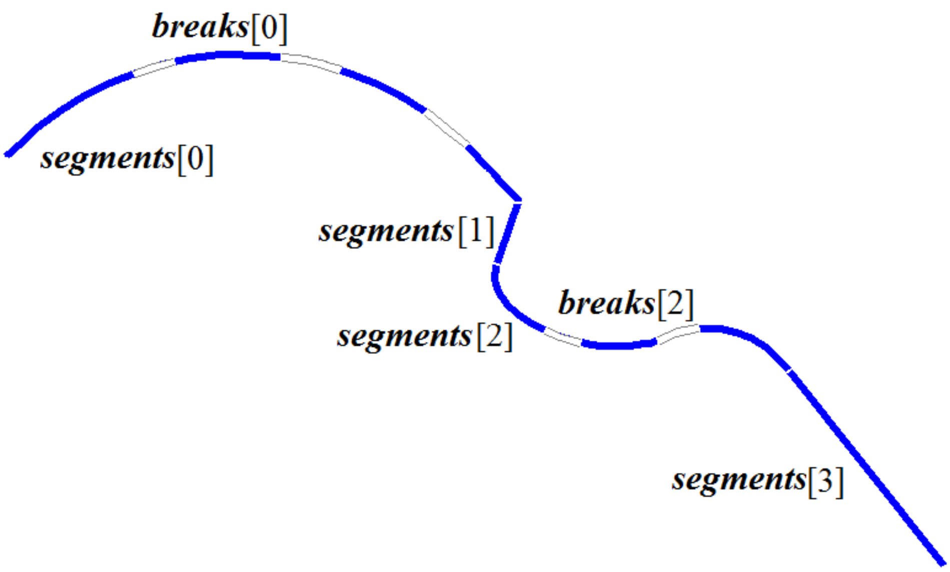


Fig. O.6.8.2.

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.

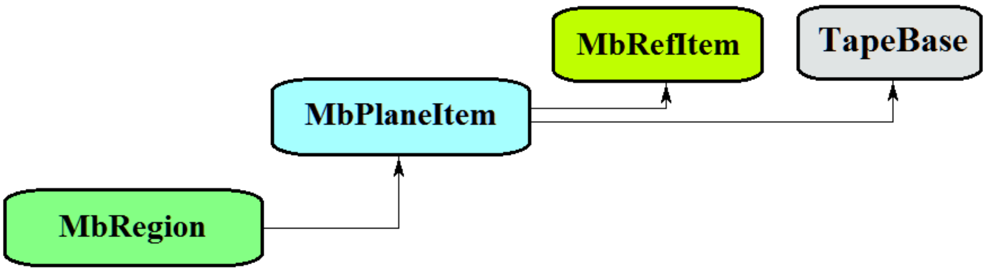


Fig. O.6.9.1.

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.

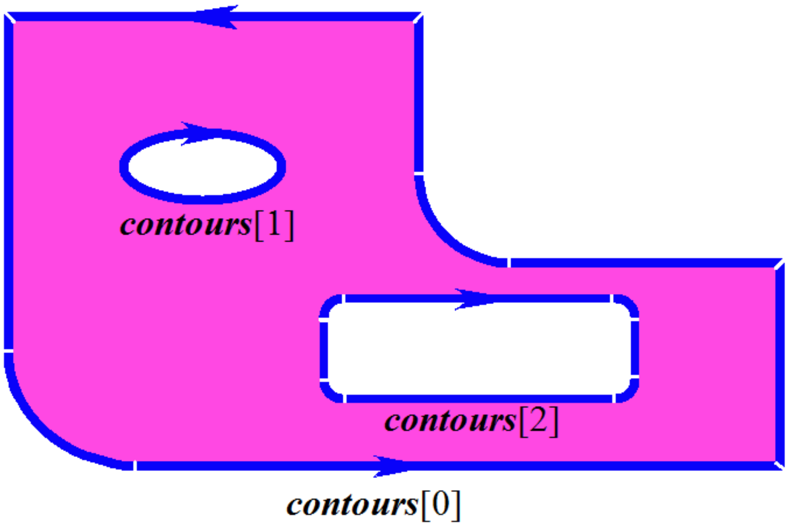


Fig. O.6.9.2.

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.

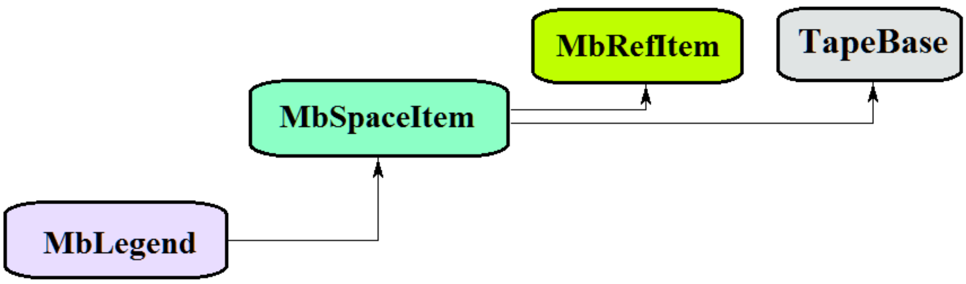


Fig. O.6.10.1.

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 body, *length* thread length, *angle* thread cone angle, **name** thread name, and **bodys** set of bodies. 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 body faces from **solids** set. Thread graphic symbol is shown in Figure O.6.12.1.

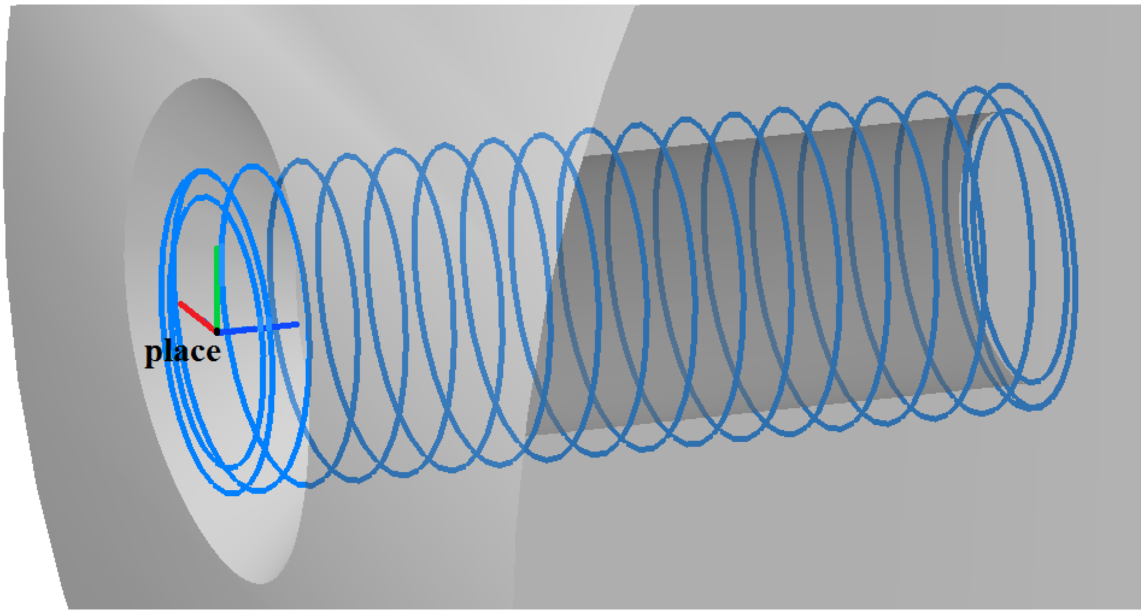


Fig. O.6.12.1.

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.

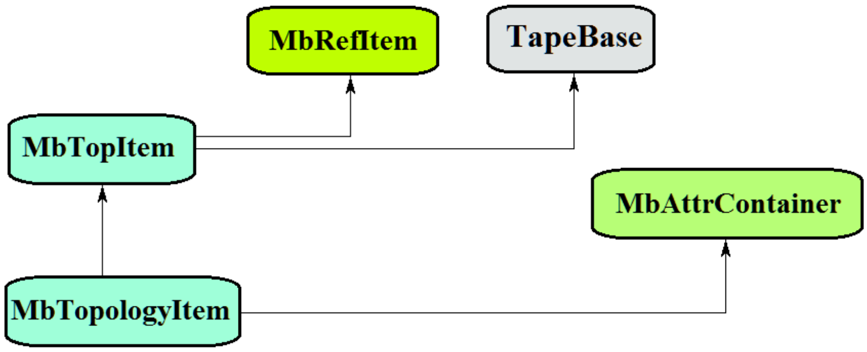


Fig. O.7.1.1

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.

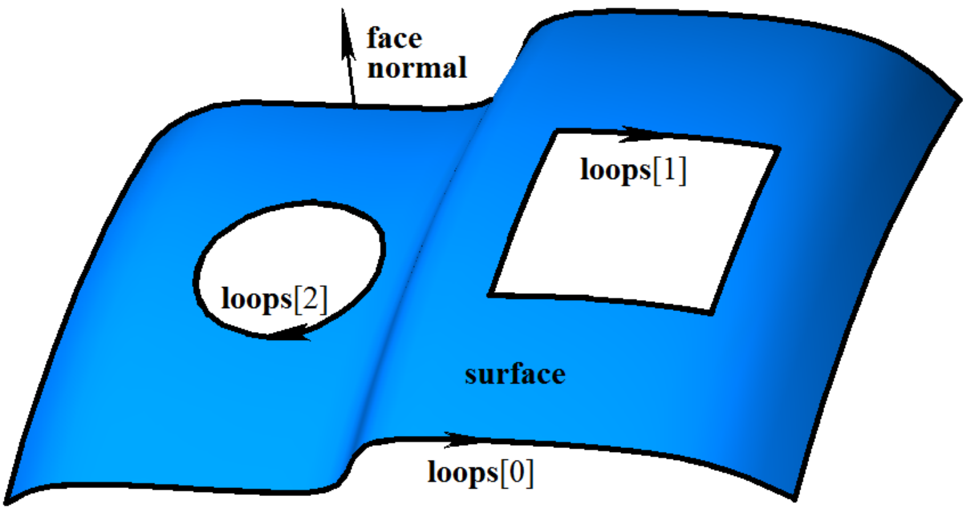


Fig. O.7.2.1.

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 body-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.

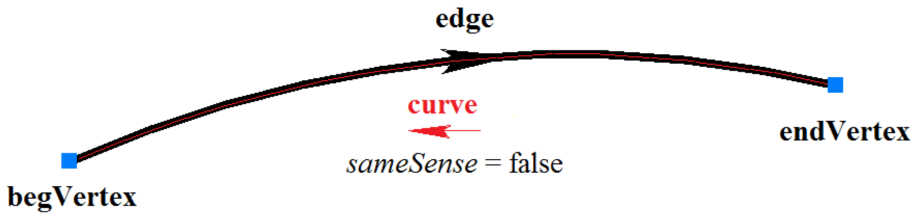


Fig. O.7.3.1.

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.

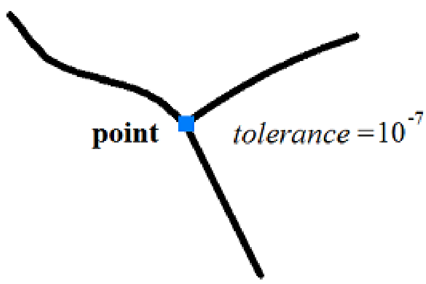


Fig. O.7.4.1.

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.

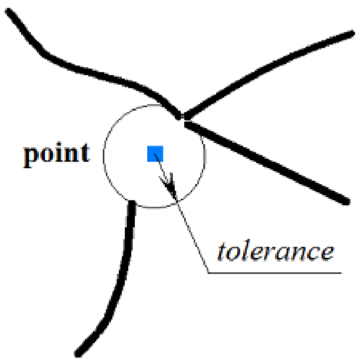


Fig. O.7.4.2.

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.

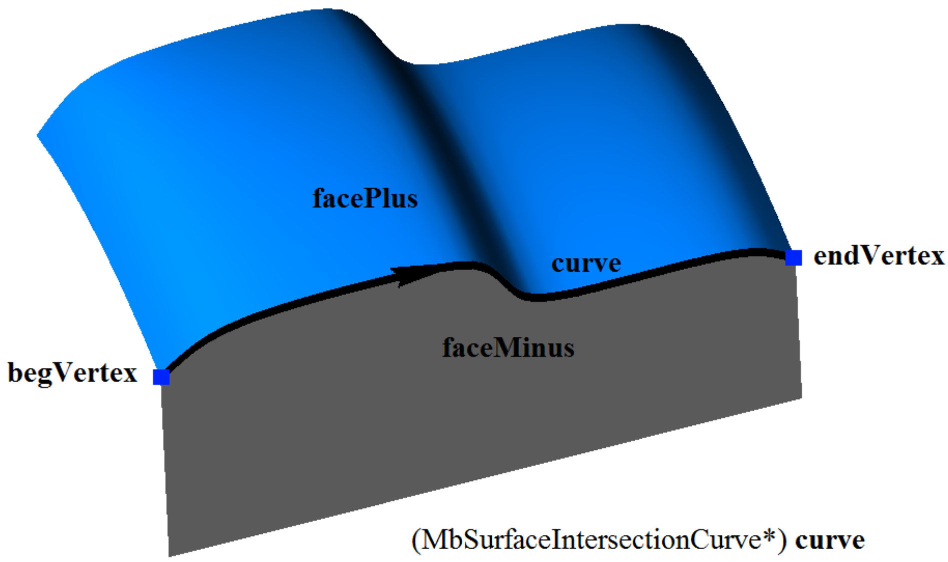


Fig. O.7.5.1.

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.

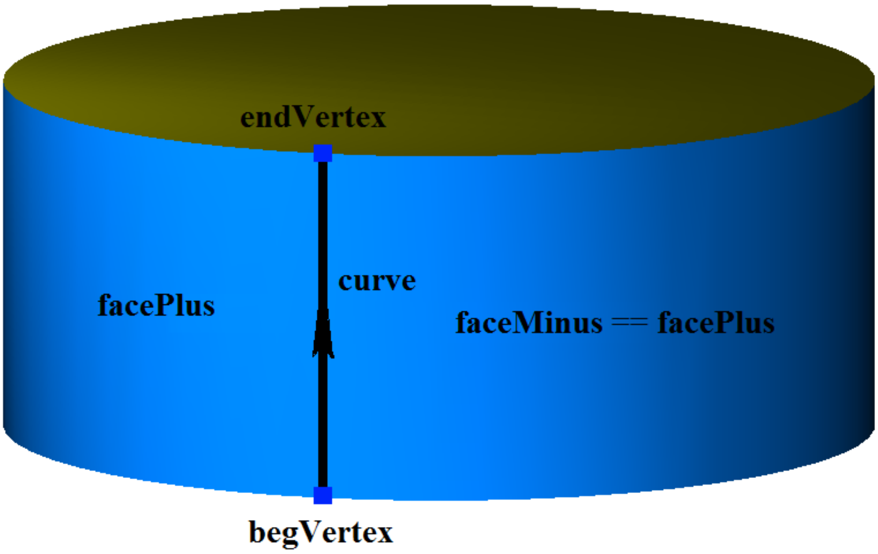


Fig. O.7.5.2.

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.

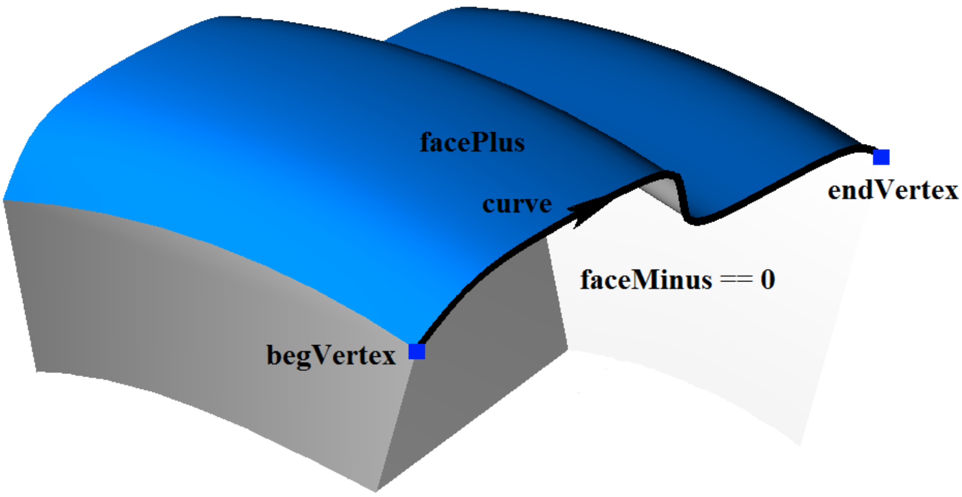


Fig. O.7.5.3.

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.

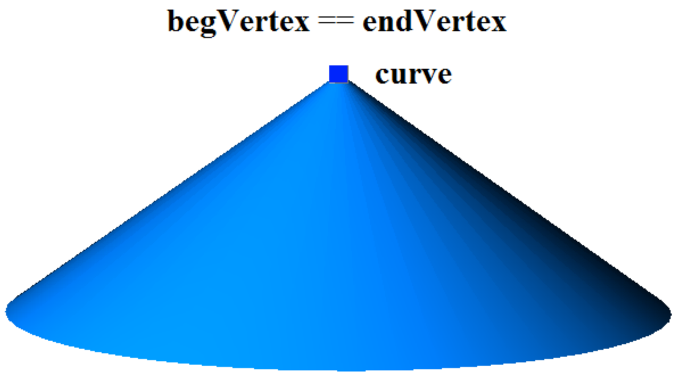


Fig. O.7.5.4.

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 body-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.

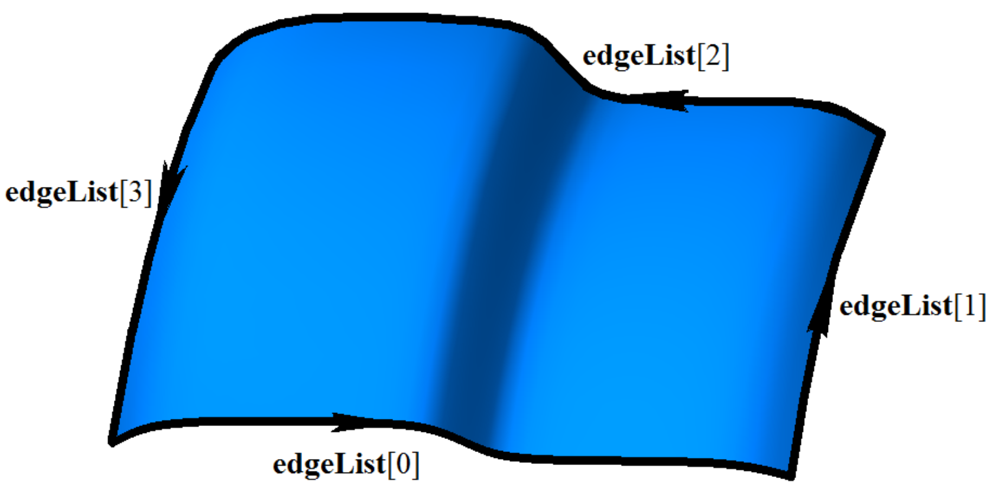


Fig. O.7.6.1.

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.

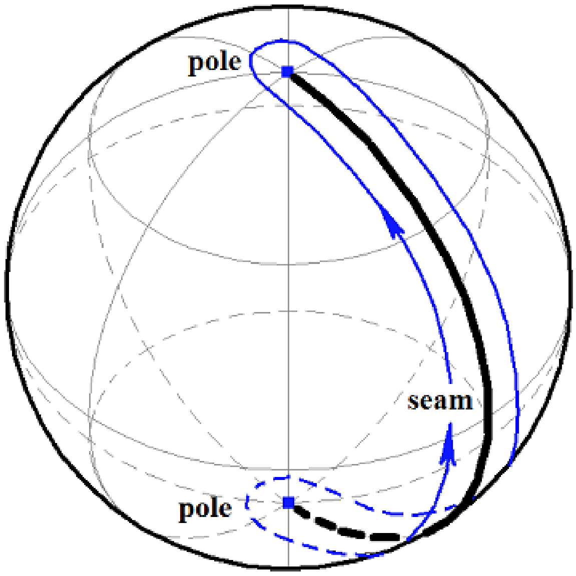


Fig. O.7.6.2.

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.

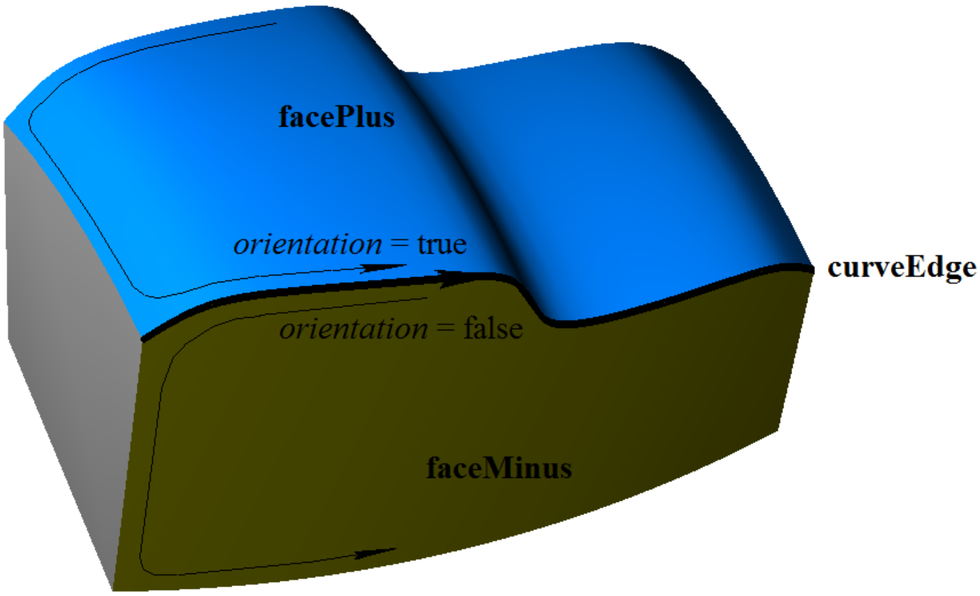


Fig. O.7.7.1.

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**.

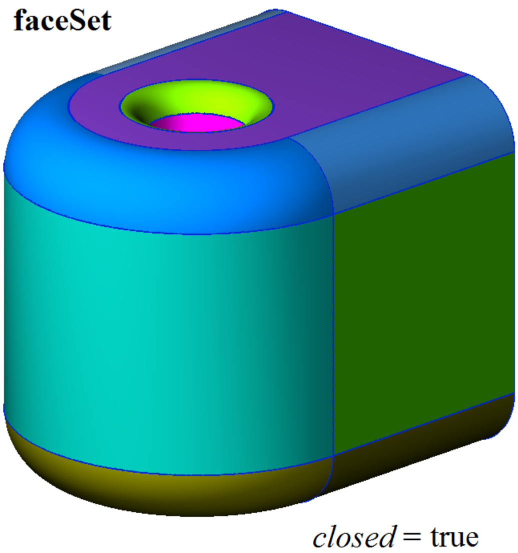


Fig. O.7.8.1.

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.

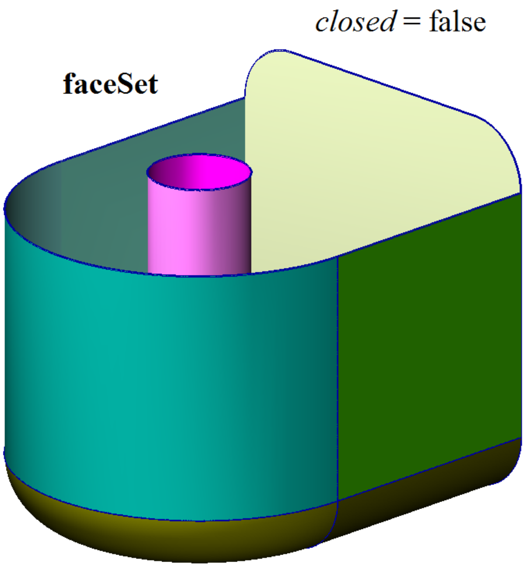


Fig. O.7.8.2.

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.

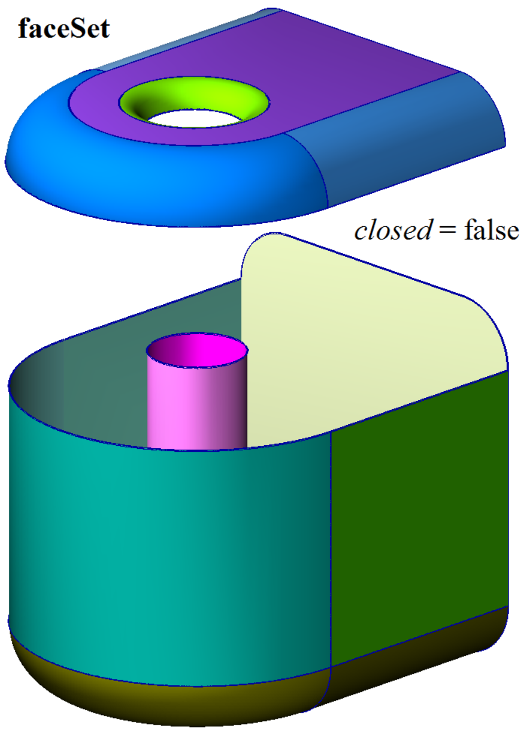


Fig. O.7.8.3.

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.

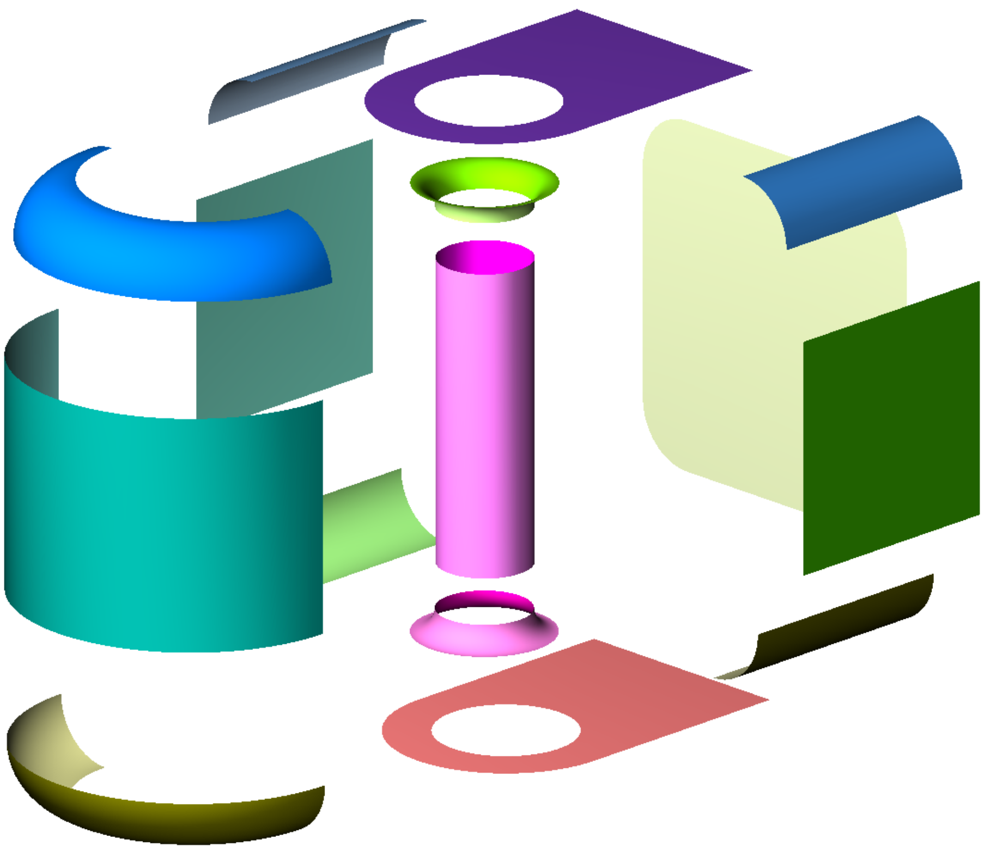


Fig. O.7.8.4.

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 bodies. 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 body construct operations with input parameters containing other bodies

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. The container of simple names can have either a set of unique numbers, or one integer, or it can be empty.

If the simple name container contains one integer, the remaining required simple names MbSNameMaker will create itself by adding the numbers of the natural sequence to the original integer. If the simple name container is empty, then the starting integer will be zero.

Each integer from the container of simple names will correspond to one of the geometric input parameters (the segment of the shaping contour of the sketch, the processed edge of the operation, the modifiable face of the operation, and so on) and will be used to name the new face born by the operation.

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. As a face, edge, or vertex identifier, you can use a number that will return the MbName::Hash() method of the corresponding topological object. The faces, edges, and vertices can be searched in the body by the known name MbName using the **FindFaceByName**(...), **FindEdgeByName**(...), **FindVertexByName**(...) methods.

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 body, surface and direct simulation. Bodies are also constructed when objects made of sheet metal are simulated. Besides a body, 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.

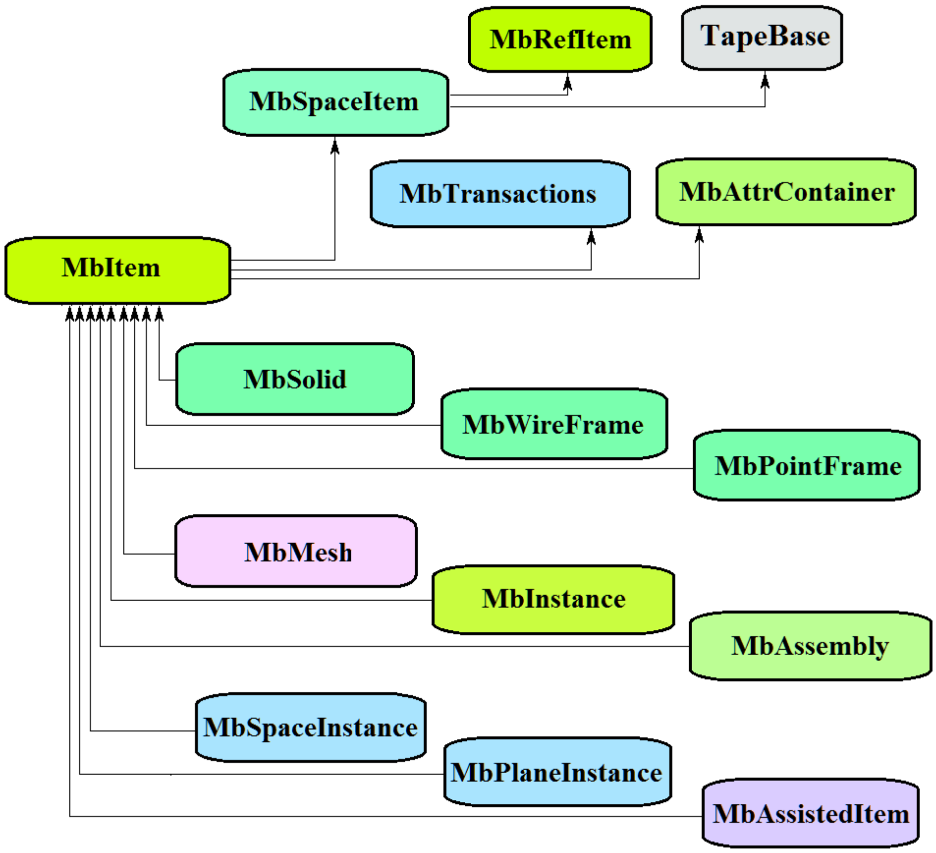


Figure O.8.1.1.

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 body) is an inheritor of MbItem class, and it is described by MbFaceShell* **outer** set of faces and *multiState* connectivity type.

A body is a set of edges that are joined together by edges and describe the surface of the simulated object. A body can describe one or more interconnected sets of points. *multiState* connectivity type indicates that a body describes one or several interconnected sets of points (in the latter case, a body can be divided into several bodies).

outer set of body 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 body describes a set of points located on surfaces of faces and on inside surfaces of these faces. Such body is called *closed* and is described by a closed shell. A closed body is shown in Figure O.8.2.1.

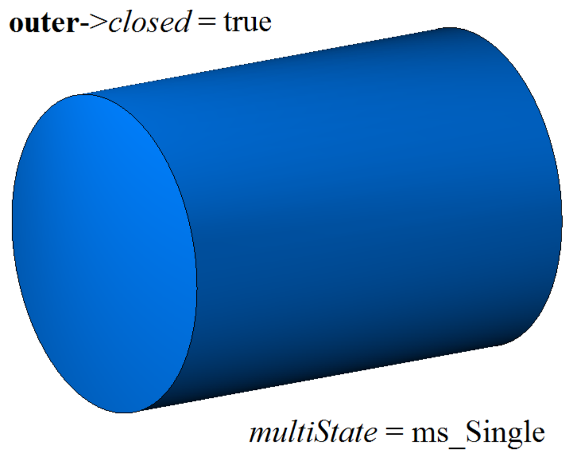


Fig. O.8.2.1.

If the set of faces has one or several edges, then the body describes a set of points located on face surfaces of only. Such a body is called *open* and is described by an open shell. An open body is shown in Figure O.8.2.1.

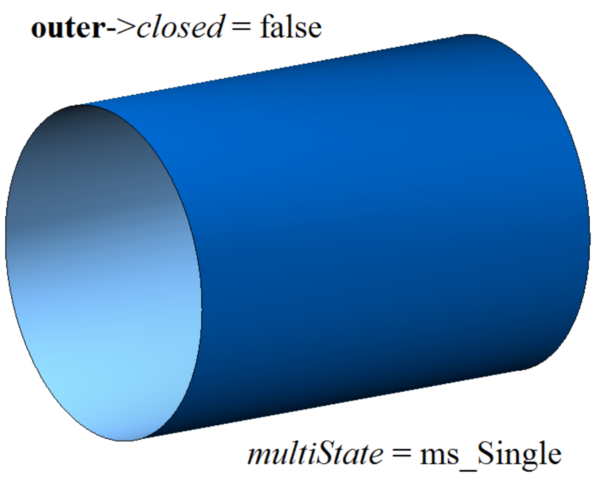


Fig. O.8.2.2.

In most cases, a closed body is described by one interconnected set of faces, that is, by one shell. If a body has cavities, then they are described by several interconnected sets of faces. In Figure O.8.2.3, you can see a closed body described by two closed shells, an external shell and an internal one (the latter shell is located inside the first one).

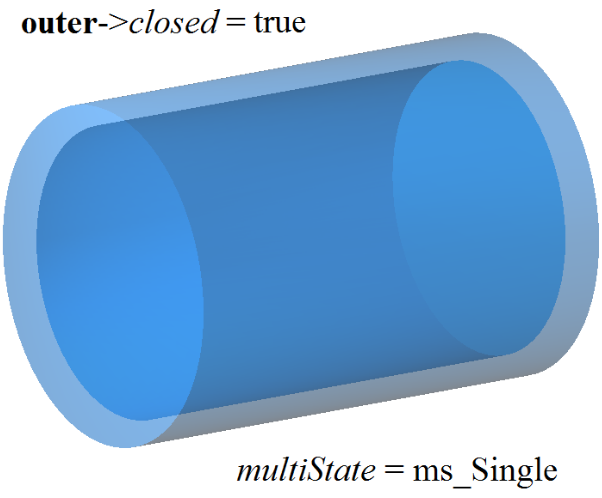


Fig. O.8.2.3.

The body is semitransparent, so you can see a cavity inside it.

Bodies support various operations (such as Boolean operations) on them, these are sets of actions that result in construction of bodies having different shape. The Result of subtraction of two closed bodies is shown in Fig. O.8.2.4.

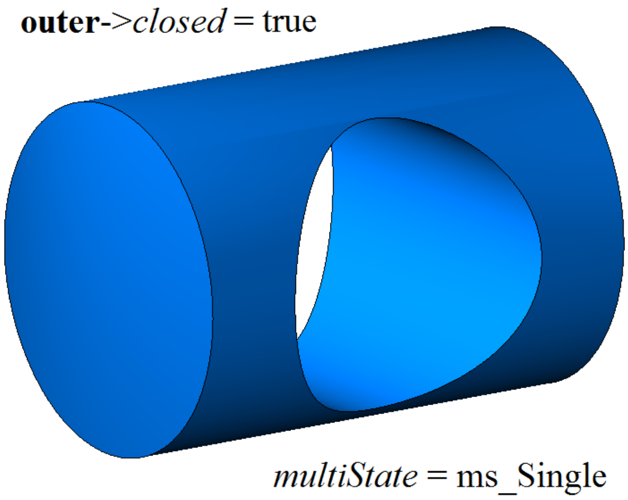


Fig. O.8.2.4.

Results of operation on a closed body and a non-closed body are completely different as the operations are executed on different sets of points. A result of subtraction of two closed bodies is shown in Figure O.8.2.5.

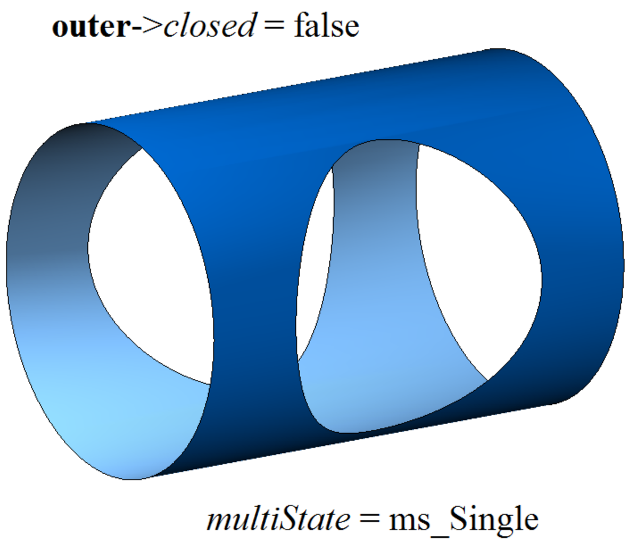


Fig. O.8.2.5.

A body 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 body described by two closed shells is shown in Figure O.8.2.6.

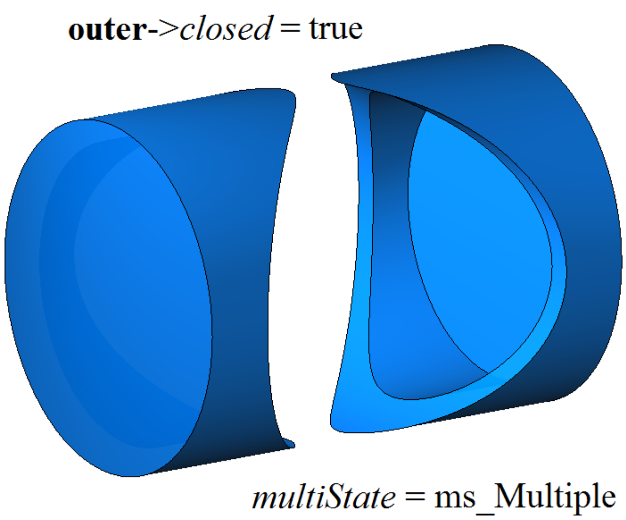


Fig. O.8.2.6.

Such a body can be divided into two simply connected closed bodies using **::DetachParts** (...) method. A body in Figure O.8.2.6 is semitransparent. A doubly-connected body described by two non-closed shells is shown in Figure O.8.2.7.

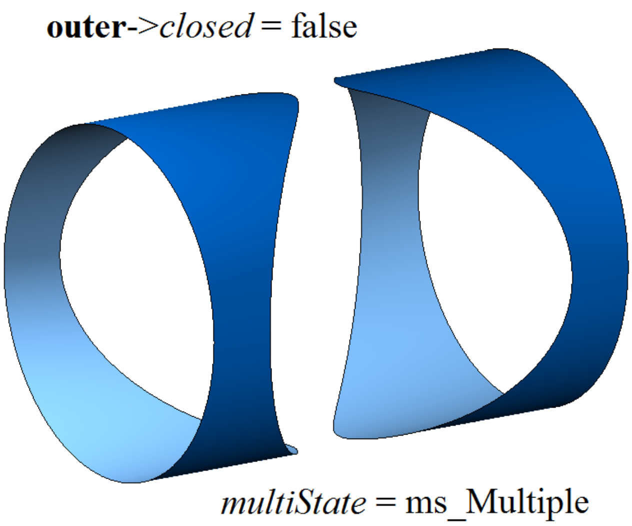


Fig. O.8.2.7.

Such a body can be divided into two simply connected non-closed bodies.

Bodies are constructed using the methods provided by C3D geometric kernel. Bodies with simple shapes are constructed by points, curves and surfaces. Operations permit you to construct more complex bodies based on simple ones. You can edit initial bodies and construct modified ones by changing the parameters in MbTransactions construction log or by directly modifying the elements of previously constructed bodies. Open bodies are used for surface simulation. Open body 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.

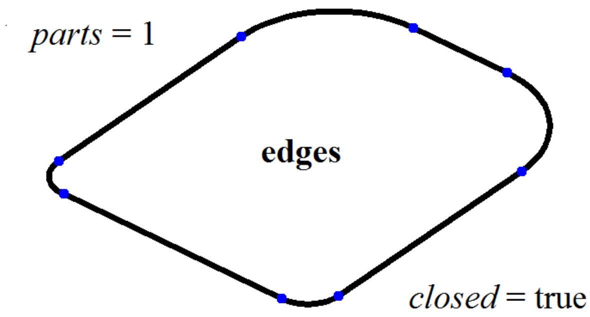


Fig. O.8.3.1.

Open wireframe is shown in Figure O.8.3.2.

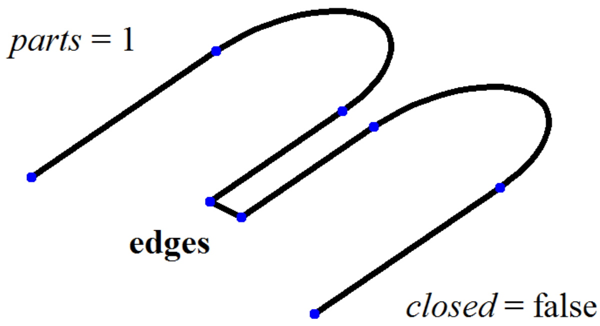


Fig. O.8.3.2.

An open wireframe consisting of two interconnected sets of edges is shown in Figure O.8.3.3.

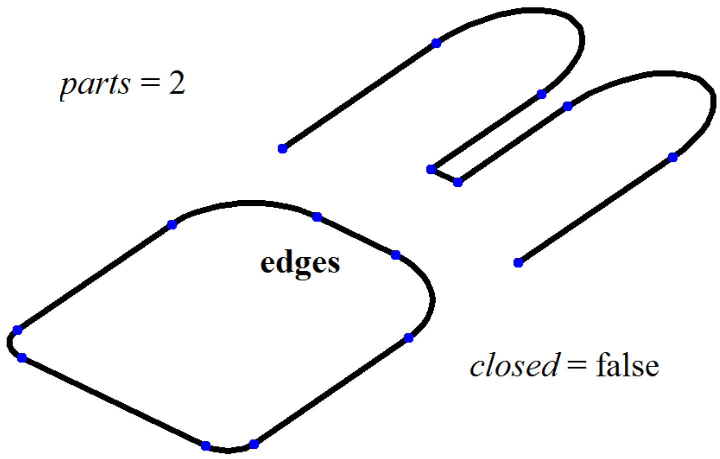


Fig. O.8.3.3.

Closed wireframe consisting of two interconnected sets of edges is shown in Figure O.8.3.4.

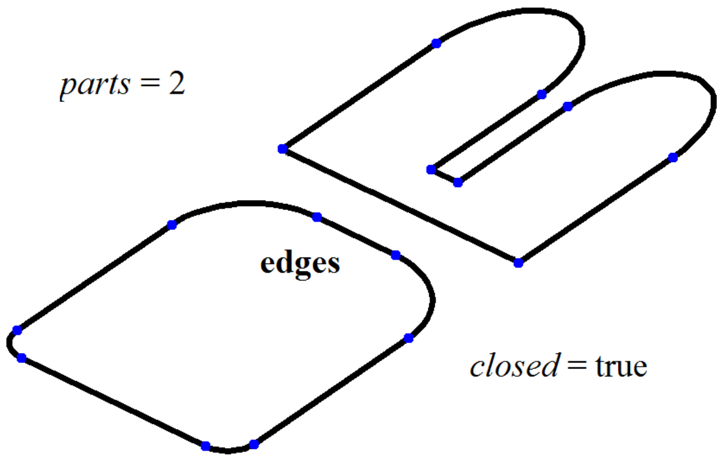


Fig. O.8.3.4.

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.

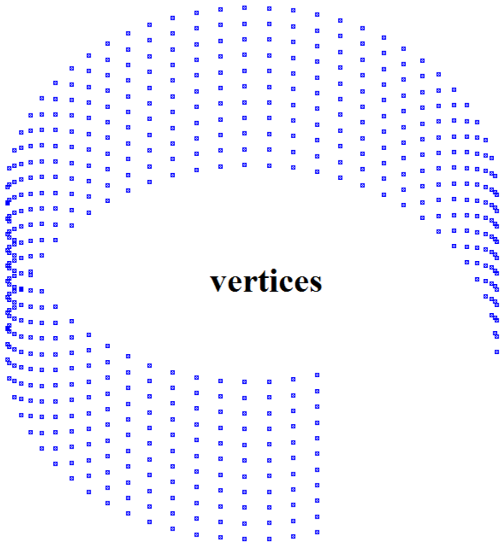


Fig. O.8.4.1.

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 body. Every *i*th body face is approximated by **grids**[*i*] triangulation, every *j*th body edge is approximated by **wires**[*j*] polygon, every *k*th vertex is associated with **peaks**[*k*] apex, *closed* closure sign corresponds to body 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.

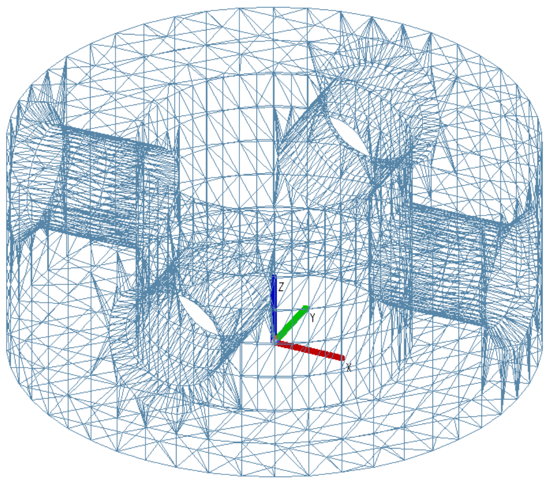


Fig. O.8.5.1.

A toned image of a polygonal object is shown in Figure O.8.5.2.

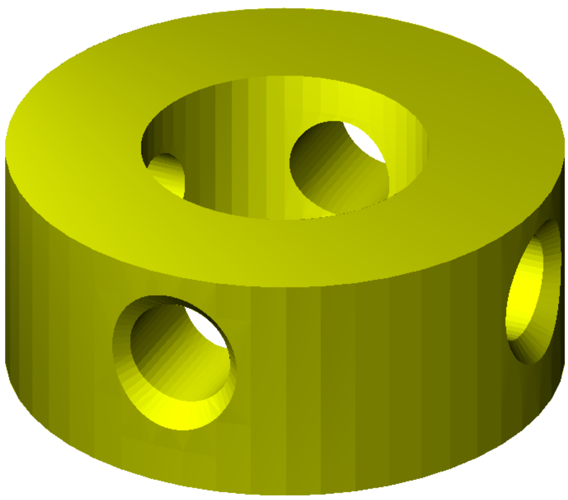


Fig. O.8.5.2.

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.

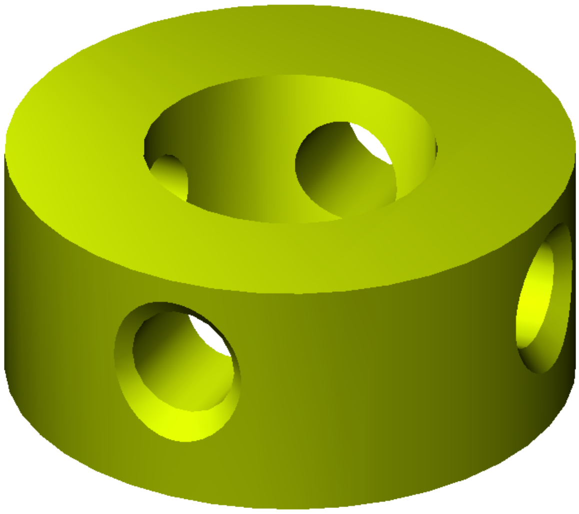


Fig. O.8.5.3.

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 body, 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.

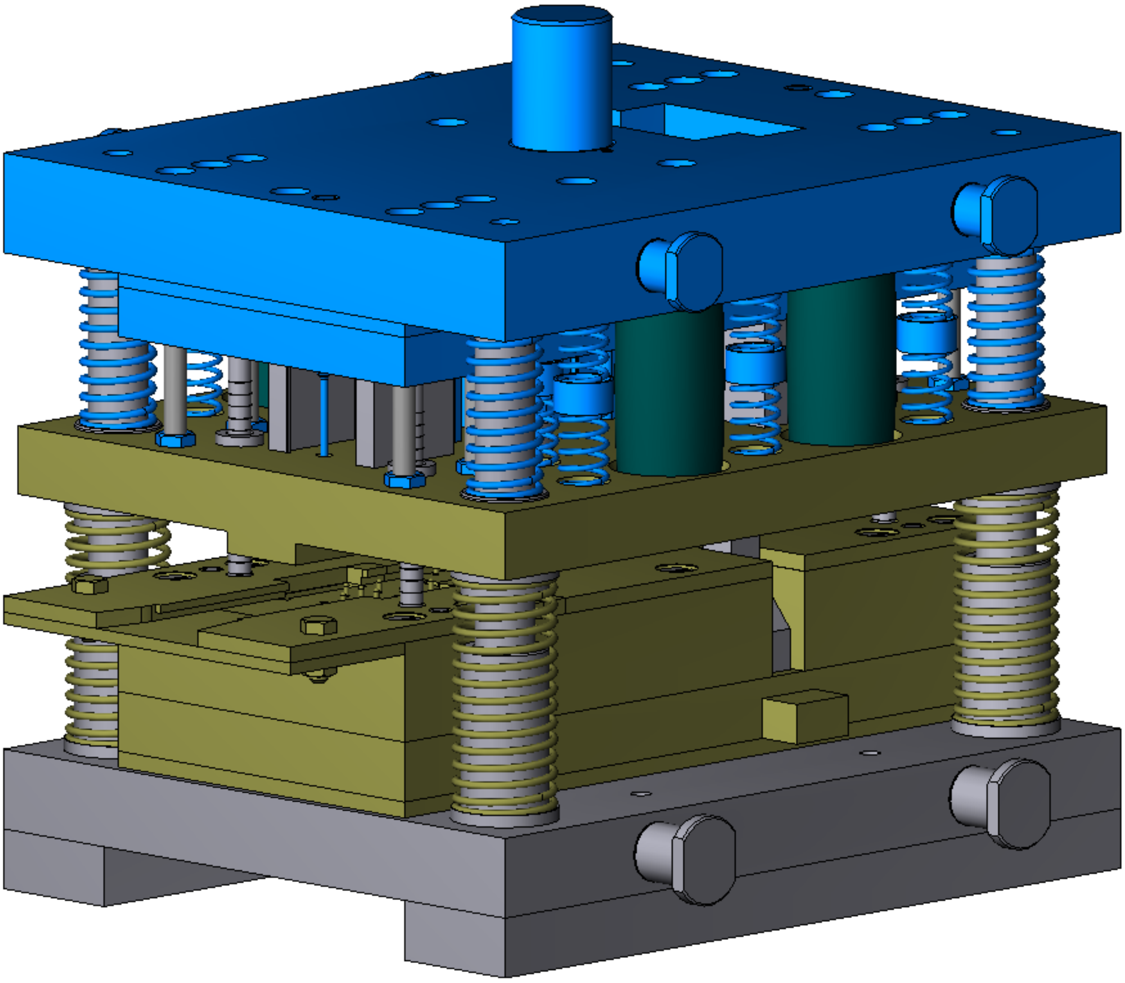


Fig. O.8.7.1.

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.

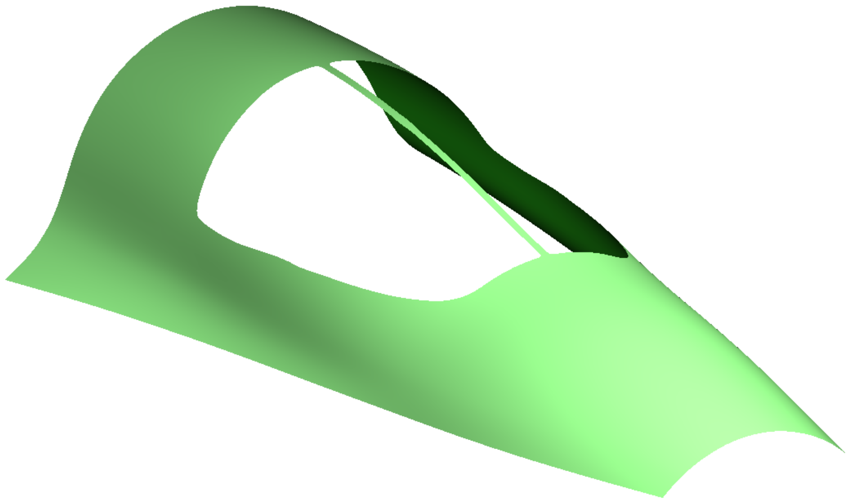


Fig. O.8.8.1.

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 body, 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.

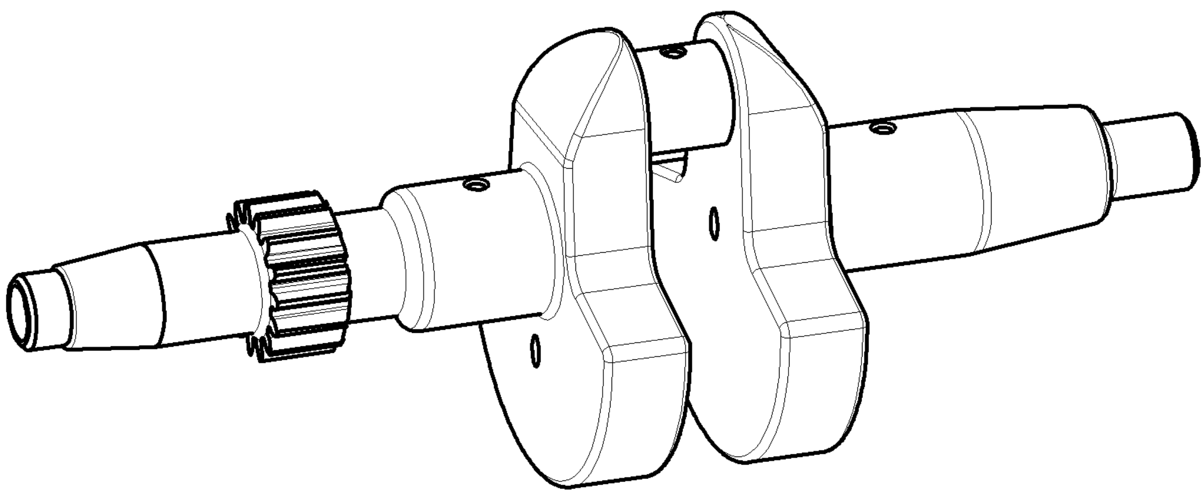


Fig. O.8.9.1.

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.

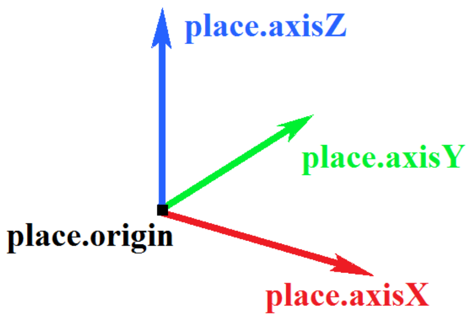


Fig. O.8.10.1.

O.9. MULTITHREADING

Multithreading support in the mathematical kernel implies:

- Use of parallel calculations in the kernel.
- Support of user multithreading.

The mathematical kernel implements the mechanisms that enable thread-safe use of the kernel interfaces in parallel calculations.

To establish internal multithreading the kernel uses the **OpenMP** technology.
The main parallel operations in the kernel includes (but not limited to):

- Construction of planar projections
- Calculation of polygonal meshes
- Calculation of mass-inertia properties
- Converters operations

If an interface implementation employs parallel calculations, then the corresponding information is included in the comments to the interface.

To ensure thread-safe use of kernel interfaces in multithreaded applications synchronization objects based on system mechanisms are used.

O.9.1. Thread-safety of kernel objects

Thread-safety of the kernel objects is implemented with the special mechanism – multithreaded caching which provides a thread-safe access to an object data and enables effective parallel calculations in case if the object is processed in several threads simultaneously.

Each thread works with its own copy of cached data that prevents data contention between threads. The cache manager controls the multithreaded caches and is responsible for creating, storing and providing cached data for the current thread.

It is important to note that multithreaded caching works effectively for both parallel and sequential calculations. The definite plus is that migration to using multithreaded caches requires minimal code refactoring. Although side effects of using multithreaded caches should be taken into account:

- Some increase in memory usage.
- Some small time overhead for retrieving cached data. If there is no parallelism, it may make some value.

Multithreaded caching could be switched on or off by changing the kernel multithreading mode.

O.9.2. Multithreaded caches implementation

The base class for cached data and the cache manager class are defined in the tool_multithreading.h file.

O.9.2.1. Cache manager CacheManager

The cache manager is defined as a template class working with cached data and includes the members:

- **longTerm** - data of the main thread; is used in serial execution and to initialize data of multithreaded caches.

- **tcache** - a list of caches with data which are used in parallel calculations. Each thread uses only its own copy of the data which is available by the thread identifier **threadKey**. For multithreaded processing of dependent (with shared data) objects, the **mtm_SafeItems** multithreading mode or higher should be used.
- **lock** — cache manager lock which is used when selecting caches and changing the main thread cache.

The template class **CacheManager<class T>** implements the methods:

- **T * operator()** - Returns a pointer to the cached data of the current thread. Always returns non-null value.
- **void Reset (bool resetLongTerm)** - Invalidates caches. If resetLongTerm = true, deletes the main thread cache.
- **void CleanAll(bool doPostproc, bool force)** - Deletes all caches and unsubscribes from the garbage collection. Should be called in sequential code.
- **T * LongTerm()** - Returns a pointer to cached data of the main thread. Always returns non-null value. All operations with the main thread cache should be protected by the cache manager lock.
- **CommonMutex * GetLock()** - Returns a pointer to the lock for operations with the main thread cache, considering whether the code runs in parallel. Can return null value (good for use with **ScopedLock**).
- **CommonMutex * GetLockHard()** - Returns a pointer to the lock for operations with the main thread cache. Always returns non-null value.
- **bool ResetCacheData()** - Cleaning function, used by the garbage collector.

The cache manager algorithm depends on the kernel multithreading mode **MbeMultithreadedMode.**

If the kernel multithreading mode is defined as **mtm_SafeItems** or higher, the cache manager initiates use of dedicated cache for each thread. In this case, the cache manager selects a copy of cached data for a thread by its thread ID.

After switching to serial execution the cache manager calls a **Postprocess()** function for caches post-processing. The specified function iterates through the caches used in parallel computing and calls the function **longTerm.MergeWith()** with the data of the each cache as a parameter. After the function **Postprocess()** finished the caches are destroyed. Then, when a cache is requested, the Cache Manager returns the main thread cache (**longTerm**).

The **CacheManager** class is derived from the **CacheCleaner** class, that allows cleaning caches on demand.

O.9.2.2. Base cached data class AuxiliaryData

Any cached data class should be derived from the base class **AuxiliaryData** and contain default constructor and copy constructor.

Also, if caches post-processing is needed after exiting parallel calculations, a cached data class should override the method **void MergeWith(AuxiliaryData *)** which is called by the **CacheManager** for the main cache with the data of the each cache as a parameter. Default implementation of **MergeWith()** does not perform any post-processing.

An object class that needs data caching should define its cached data as a class derived from **AuxiliaryData** class, and include an instance of **CacheManager** as a class member.

For parallel processing of dependent (with shared data) objects the kernel multithreading mode **mtm_SafeItems** or higher should be used.

O.9.3. Garbage collection in Cache Manager

The cache manager **CacheManager** deletes unused caches at the first call from a sequential code.

In addition, garbage collection runs automatically on exiting a parallel region.

The mathematical kernel provides an interface for clearing cached data – the class **MbGarbageCollection**.

The class **MbGarbageCollection** is the garbage collector which clears caches in registered objects of the **CacheCleaner** type. By a request it deletes caches in the registered objects.

In order to enable a garbage collection in an object of type **CacheCleaner**, the object needs to implement the method **ResetCacheData()** for caches cleaning and to subscribe to the garbage collection in the class **MbGarbageCollection**.

The CacheManager class is derived from the **CacheCleaner** class, that makes it possible to run caches clearing in the CacheManager manually. For that, when creating a cache for a thread the CacheManager registers itself in the **MbGarbageCollection** class.

O.9.3.1. Base class for objects that require garbage collection

In order to subscribe to the garbage collection, a class should be derived from the class CacheCleaner and should implement the method ResetCacheData for deleting cached data.

CacheCleaner is the base class for objects that require garbage collection. It implements the methods:

- **void SubcribeOnCleaning()** - Subscribe the object for garbage collection.
- **void UnsubcribeOnCleaning()** - Unsubscribe the object from garbage collection.

The class CacheCleaner also defines the method:

- **bool ResetCacheData()** - Reset cached data. Should return true if the object was unsubscribed from garbage collection.

O.9.3.2. Class MbGarbageCollection

The class **MbGarbageCollection** is the garbage collector for objects of type **CacheCleaner**, which keep cached data. By a request the MbGarbageCollection clears caches in registered objects by calling the **ResetCacheData** for each object.

The class MbGarbageCollection implements the methods:

- **void Subscribe(CacheCleaner * obj)** - Subscribe the object to the garbage collection.
- **void Unsubscribe(CacheCleaner * obj)** - Unsubscribe the object from the garbage collection.
- **static bool Run(bool force = false)** - Perform the garbage collection. Should be called in sequential code. When called in a parallel region, does nothing. If force = false, then run garbage collection in caches created for threads which are finished, if true, then force garbage collection in all caches. Returns TRUE if the garbage collection is done.
- **static void Enable(bool allow = true)** - Enable/disable collecting data for garbage collection. By default it is enabled.

O.9.4. Kernel multithreading modes

The kernel multithreading mode controls the thread-safety of the kernel objects and defines which kernel operations will be paralleled.

The kernel multithreading mode is defined as **MbeMultithreadedMode** enumeration in the file tool_multithreading.h.

The kernel can work in the following modes:

- **mtm_Off** - The kernel multithreading is off. In this mode all operations are executed sequentially. The mechanism providing the kernel objects thread-safety is switched off.
- **mtm_Standard** — In the standard multithreading mode the limited operations are paralleled, namely, only operations that process independent data. The mechanism providing the kernel objects thread-safety is switched off.
- **mtm_SafeItems** – The objects thread-safety mode. In this mode the mechanism of multithreading caches is switched on, but still only limited operations are paralleled. This mode is used to support multithreaded operations in user applications.
- **mtm_Items** – The objects multithreading mode. In this mode the full multithreading of the kernel operations is on. The kernel operations that process both independent and dependent data are paralleled. This mode also can be used to support multithreaded operations in user applications.
- **mtm_Max** – The maximal kernel multithreading is on. Currently this mode is equal to mtm_Items mode.

By default the kernel works in **mtm_Max** mode.
The kernel provides interfaces for dynamic switching the multithreading mode.

The interfaces for switching the kernel multithreading mode are declared in the file mb_variables.h.
The following static methods of the Math class can used for switching the kernel multithreading mode:

- **bool Multithreaded()** - Checks if the **mtm_Standard** mode is used (whether multithreading is on).
- **void SetMultithreaded(bool b)** - If b = false, disables multithreading (sets the mode to **mtm_Off**). If b = true, enables limited multithreading (sets the mode to **mtm_Standard**).
- **MbeMultithreadedMode MultithreadedMode()** - Returns the current multithreading mode.
- **bool CheckMultithreadedMode (MbeMultithreadedMode mode)** - Checks if the current mode implies the specified mode (is not lower than the specified mode).
- **void SetMultithreadedMode (MbeMultithreadedMode mode)** - Sets the specified mode**.**

O.9.5. Synchronization objects

The synchronization objects are defined in the file tool_mutex.h.

O.9.5.1. Locks

By default the kernel uses lock classes implemented on base of system synchronization API that enables safe use of alternative parallel frameworks (not only OpenMP) in user applications.

But the kernel can be switched to using locks based on OpenMP locks.

On Windows system locks by default are implemented on base of critical sections. On Linux system locks are implemented on base of pthread_mutex_t.

The lock classes implementation is controlled by **C3D_NATIVE_LOCK** variable. Redefining the variable C3D_NATIVE_LOCK in user applications is prohibited.

The classes **CommonMutex** and **CommonRecursiveMutex** define lock and recursive lock. They provides the methods lock() and unlock().

The classes **ScopedLock** and **ScopedRecursiveLock** define scoped lock and scoped recursive lock. They can accept a null pointer to a mutex. Locking occurs if the pointer to the mutex is nonzero and the code runs in parallel.

O.9.5.2. Base synchronization objects

The class **MbSyncItem** is the base synchronization object with lazy initialization. Creates a lock (mutex) when needed. Implements methods Lock() and Unlock().

The class **MbNestSyncItem** is the base synchronization object with lazy initialization which supports nested locks. Creates a lock (mutex) when needed. Implements methods Lock() and Unlock().

The class **MbPersistentSyncItem** is the base synchronization object implementing methods Lock() and Unlock(). Always creates a lock (mutex).

The class **MbPersistentNestSyncItem** is the base synchronization object which supports nested locks. It implements methods Lock() and Unlock(). Always creates a lock (mutex).

The kernel objects that need synchronization are inherited from one of these classes.

O.9.6. Support of multithreading in user application

For using the kernel interfaces in several threads, the multithreading mode **mtm_SafeItems** or higher should be defined for the kernel. By default the kernel works in maximal multithreading mode (**mtm_Max**).

If the user application uses the kernel interfaces in parallel computation, it must notify the kernel about entering and exiting a parallel region. For that, the class **ParallelRegionGuard** (a scoped guard of parallel region) or pair of the functions **EnterParallelRegion** and **ExitParallelRegion** could be used.

O.9.6.1. Protection of parallel code in user application

The class ParallelRegionGuard notifies the kernel that code in the scope runs in parallel. It should be used to protect a parallel region.

Instead of the class ParallelRegionGuard, the next pair of functions also could be used to notify the kernel about a parallel region boundaries:

- The function EnterParallelRegion notifies the kernel about entering a parallel region.
- The function ExitParallelRegion notifies the kernel about exiting a parallel region.

Interfaces ParallelRegionGuard or EnterParallelRegion and ExitParallelRegion must be used in cases if the kernel is used in several threads.

Note, that before start using parallel computing it is necessary to ensure that the multithreading mode **mtm_SafeItems** or higher is set in the kernel (by default the maximal multithreading mode **mtm_Max** is set).

O.9.6.2. The examples of notifying the kernel about its use in parallel computing

ParallelRegionGuard should be called one time before the start working in several threads. For example:

```
<...Sequential code...>
if( Math::CheckMultithreadedMode( mtm_Items ) ) {
    ParallelRegionGuard guard; // works in the scope
    std::thread t1( func1 );
    std::thread t2( func2 );
    t1.join();
    t2.join();
}
<...Sequential code...>
```

The same can be done using a couple of functions **EnterParallelRegion** and **ExitParallelRegion** instead of **ParallelRegionGuard**:

```
<...Sequential code...>
if( Math::CheckMultithreadedMode( mtm_Items ) ) {
    EnterParallelRegion();
    std::thread t1( func1 );
    std::thread t2( func2 );
    t1.join();
    t2.join();
    ExitParallelRegion();
}
<...Sequential code...>
```

The purpose of the interfaces **ParallelRegionGuard**, **EnterParallelRegion** and **ExitParallelRegion** is notifying the kernel about parallel calculations in a user application. These notifications are necessary to guarantee correct work of thread-safety mechanisms in the kernel.

O.9.6.3. Quick reference on organizing parallel computing using C3D interfaces

1. Before running parallel calculations the application must check that the multithreading mode of the kernel allows parallel computing. To use kernel interfaces in several threads, the multithreading mode of the kernel must be set to at least **mtm_SafeItems** or higher.

   Example of checking the kernel multithreading mode:

   ```
   bool useParallel = Math::CheckMultithreadedMode( mtm_SafeItems );
   ```

2. When using parallel calculations the user application must notify the kernel about entering and exiting a parallel region.
   - If a parallel cycle is used, the application must call **EnterParallelRegion** before the cycle start and **ExitParallelRegion** after the cycle finish.

     Example:

     ```
     if( Math::CheckMultithreadedMode( mtm_Items ) ) {
        EnterParallelRegion();
        #pragma omp parallel for
        for ( ptrdiff_t i = 0; i < count; ++i ) {
            /* Cycle body */
        }
        ExitParallelRegion();
     }
     ```
   - If the kernel interfaces are simply called from several threads, then in each thread **EnterParallelRegion** must be placed before the kernel calls and **ExitParallelRegion** must be placed after the kernel calls.

Example:

```
Thread 1:
EnterParallelRegion();
// The kernel calls...
ExitParallelRegion();

  Thread 2:
  EnterParallelRegion();
  // The kernel calls...
  ExitParallelRegion();
```

After the first call of **EnterParallelRegion** the kernel start using multithreaded caches and after the last call of **ExitParallelRegion** it returns to the base mode of sequential execution.

3. It is not recommended calling **ParallelRegionGuard,** **EnterParallelRegion** and **ExitParallelRegion** without need (in multithreading absence), because use of multithreaded caches in sequential execution entails some time overhead of obtaining cached data.

O.10. FORMAT C3D

A geometric model could be saved in a compact form to a memory buffer or a disk file. There are two types of C3D format for storing geometric model – compact and extended.

This article describes C3D formats for storing geometric model and the interfaces the kernel provides for writing and reading geometric model data in C3D format.

O.10.1. Format for storing geometric model

O.10.1.1. Notions and terms

Recall that **serialization** is a process of translating an object (class) to a sequence of bytes for storing it in a memory buffer (a file). **Deserialization** is an opposite process of restoring an object from a sequence of bytes.

Stream operations is operations of writing and reading a model and its objects in a serialized form (work with write and read streams).

Objects that can be serialized (represented as a sequence of bytes) are called **streaming objects**.

A continuous memory block (a sequence of bytes) is called a **cluster**, and a set of clusters is called a **file space**. A file space has a single point of writing (reading) – a position in a current file space.

O.10.1.2. Geometric model serialization

Geometric model C3D could be serialized (represented as a sequence of bytes) to a set of file spaces.

All main objects of geometric model are streaming, that is, they could write and read their data in a file space. Each model object writes itself starting from the last unfinished cluster of the current file space.

Writing a geometric model starts with MbModel object.

Each model object first writes data of its parent object (if it exists), then its own data and objects it contains or references.

After a geometric model is fully serialized to a set of file spaces, using a special function, a single sequential memory buffer is created to which a special header is written and data of all clusters (from all file spaces) sequentially are copied.

Created in this way memory buffer, contains geometric model data as a sequence of bytes and could be transferred to another location (another application) via memory for subsequent restoring the model or could be saved to a disk file.

O.10.1.3. Compact format C3D

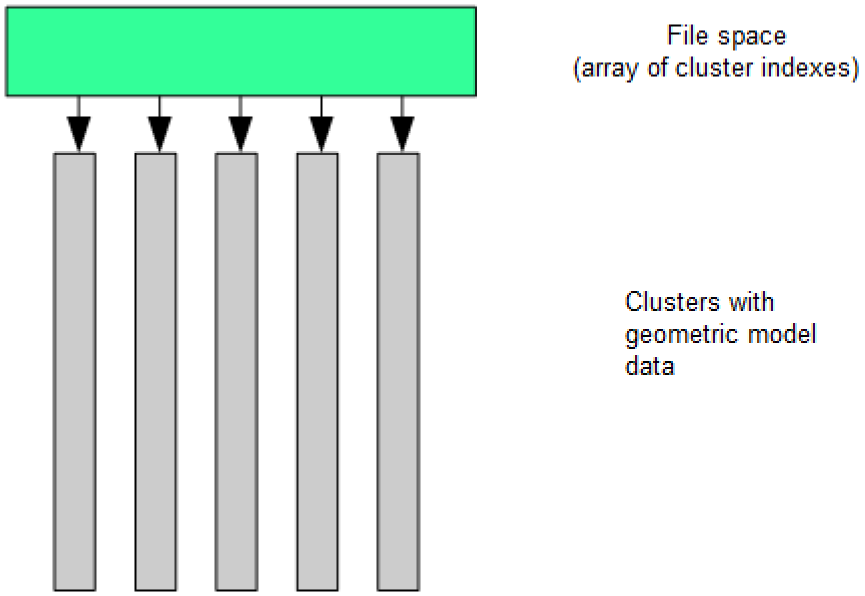
The **compact format** provides a compact and fast storing data of geometric model to a memory buffer (file) and fast reading the whole model from the memory buffer (file).

Main features of the compact format:

- Minimal overhead while writing and reading a model.
- Compact data storage.
- Allowed reading the whole model.
  - Information about model structure and about model objects is available only after reading the whole model.
  - Impossible to read model objects in an arbitrary order or read a few selected objects.

The compact format puts all model data in a single file space, that is, assumes only one point of data writing or reading – a current cluster of given file space.

The Scheme 1 shows the structure of serialized data of geometric model before saving it in a sequential memory buffer when using the compact format.

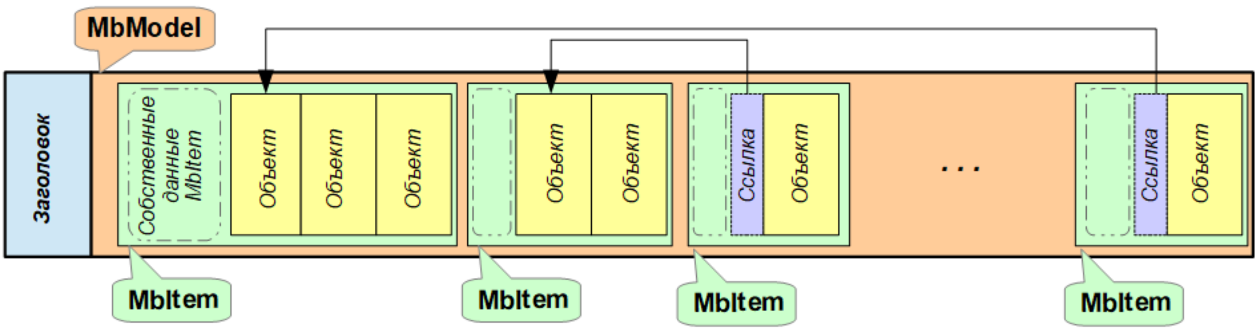


Scheme 1. Model data structure in a file space (compact format).

While writing a model, if a geometric object contains another object or references to another object, then the latter object is written inside the referring object.

An object referenced by multiple objects is declared as **registered.** While writing first time, such object is registered in a special Table Of Registered Objects. All subsequent objects, that refer to this registered object, contain only a reference to the record in the table.

The Scheme 2 shows a structure of a geometric model data in a sequential memory buffer for the compact format.



Scheme 2. Structure of a geometric model data in a sequential memory buffer (compact format).

O.10.1.4. Extended format C3D

The **extended format** provides independent storing objects of geometric model. In addition to the model objects data, the extended format keeps objects storage positions in the buffer and a brief table of contents of the model, that allows obtaining information about the model structure without full reading it, and also provides the ability of selective reading of the model objects.

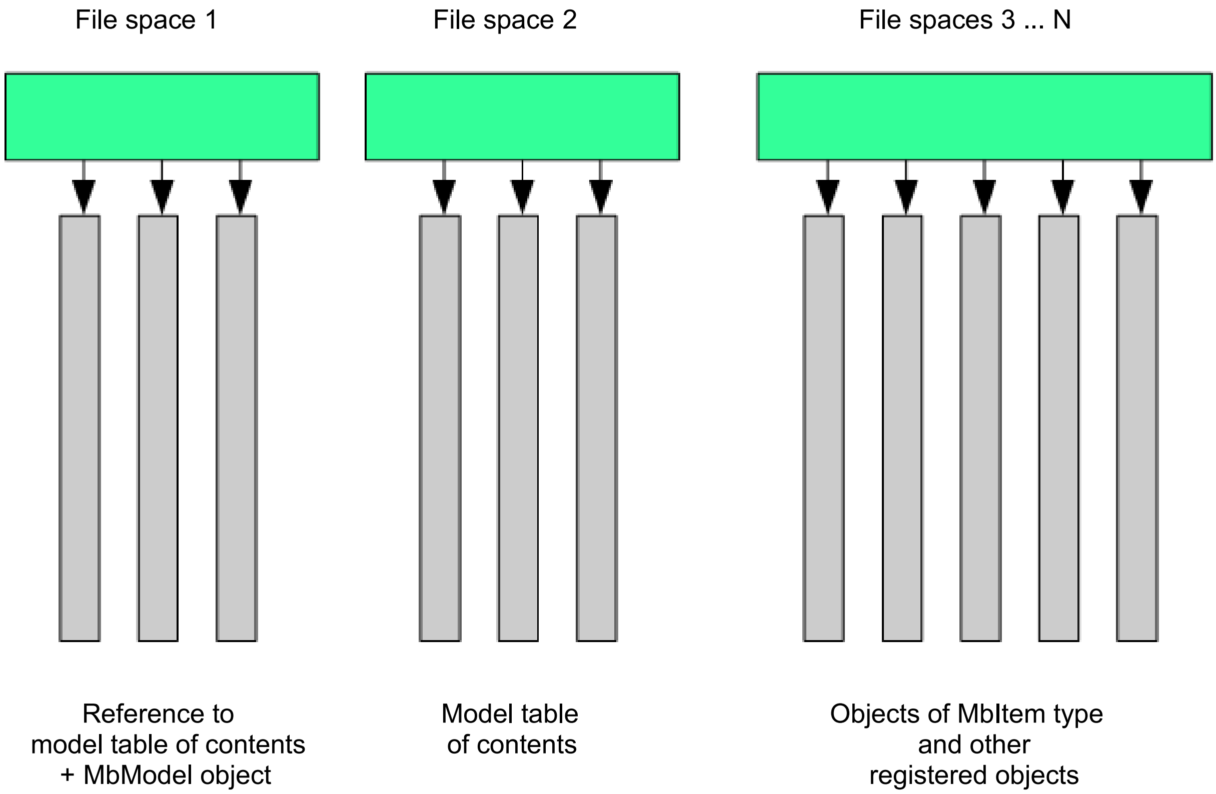
Main features of the extended format:

– Independent storage of objects, which provides the ability of selective reading of the model objects.
– Support of a table of contents for a model which keeps information about the model objects

and links to their storage positions.

– Some increase in the size of the stored data.

The extended format writes model objects into several file spaces, that is, supports multiple writing (reading) points – the current cluster in each file space.

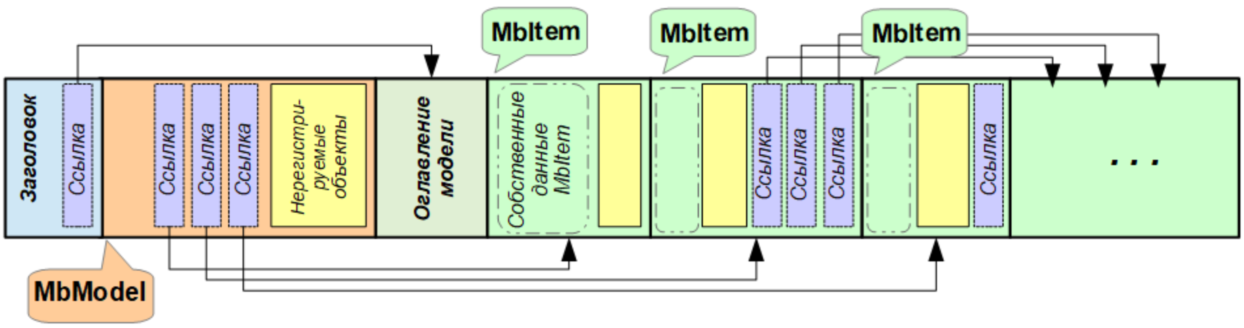


Scheme 3. Model data structure in file spaces (extended format).

The Scheme 3 shows the structure of serialized data of a geometric model before saving it to a sequential memory buffer when using the extended format.

Objects of each level of the model are written to a separate space file (a number of file spaces used is determined by the nesting depth of the model objects).

The Scheme 4 shows a structure of a geometric model data in a sequential memory buffer for the extended format.



Scheme 4. Structure of a geometric model data in a sequential memory buffer (extended format).

Model table of contents

The extended format implements generation of a geometric model tree and writing/reading it in a memory buffer (file).

The tree is created for all objects of type MbItem of the geometric model and is written as a table of contents of the model in a separate file space, a link to which is added to the file header.

The table of contents of the model, written in the file, contains:

1. Data of model tree nodes.
   Each tree node stores information for one object of the geometric model. A node keeps the following data:
   - A unique ID of the node in the model tree.
   - Object type (MbeSpaceType).
   - Object name (SimpleName).
   - Object gabarit.
   - Information about the object attributes.
   - A list of the immediate descendants of the node.
   - Writing/reading position in a file.
2. Information about the model tree roots.

The model table of contents could be read separately, that allows obtaining information about the model structure and its main objects without the whole model loading. Using data of the model table of contents it is possible to choose and separately read any model objects listed in the table of contents.

Working with embodiments

The extended C3D format provides an opportunity to work with embodiments (variants of model implementation):

- A model with embodiments should be saved (serialized) to a file using the class writer_ex (in the extended format) and should be read using the class reader_ex.
- Upon reading a model catalog from the file with embodiments, it is possible to look through the embodiments hierarchy, select an embodiment and read its contents as a separate model.
- When reading a file with embodiments without use of a model catalog (for example, by direct call to ReadModelItems function), the first in order (default) embodiment will be read.

A model with embodiments is created as an object MbModel, which contains a set of assemblies (MbAssembly objects). Each assembly represents one embodiment (contains objects of one embodiment) and has an attribute of at_Embodiment type. The first in order embodiment in the model is the default embodiment.

An assembly is considered an embodiment and is treated as an embodiment only if it is located directly in the MbModel object (is one of the roots of the model tree) and has an attribute of at_Embodiment type. MbAssembly objects which are located inside embodiments (deeper than the root in the model tree) are treated as regular assemblies.

An attribute of at_Embodiment type serves as an indicator of an embodiment. It must contain the name (SimpleName) of the current embodiment and the name (SimpleName) of the parent embodiment (or UNDEFINED_SNAME if there is no parent).

To provide the ability to view the hierarchy of embodiments and selective reading, a model with embodiments should be serialized using the class writer_ex. After that the model with embodiments should be read using the class reader_ex.

When reading C3D file with embodiments, only one embodiment can be read.

If reading a file with embodiments in a usual way, that is, without use of model tree (for example, using direct call to the function ReadModelItems), then the first (default) embodiment will be read.

In order to read any other embodiment, a model tree should be read first. Then, using the model tree, it is possible to look through the hierarchy of embodiments and load the selected one as a separate model.

O.10.2. Read and write streaming objects

Classes for reading and writing of streaming objects of the model (read and write streams) are defined in the file tape.h. They inherit from the base class tape.

O.10.2.1. Base class for read and write streams

The base class for read and write streams **tape** contains references to a data buffer, a manager of write/read streams and a structure for registration of written (read) objects.

It implements methods for managing the data buffer, registering objects and working with a progress indicator:

- Access to a data buffer:
  const iobuf_Seq & GetIOBuffer() - Access to a data buffer.
  iobuf_Seq & GetIOBuffer() - Access to a data buffer.
  bool IsOwnBuffer() - Whether owning the buffer?
  void SetOwnBuffer(bool) - Set a flag of owning the buffer.
- Managing an operation mode of data buffer:
  uint8 mode() - Get an operation mode of the buffer.
  void setMode(uint8) - Set an operation mode of the buffer.
  void clearState(state) - Clear the buffer state.
  void setState (state) - Add the buffer state.
- Requesting the data buffer state:
  int fresh() - Is the buffer fresh?
  bool good() - Whether the buffer state is correct?
  uint8 eof() - Is the end of the file reached?
  uint32 state() - Get the flag of the buffer state.
  io::pos tell() - Get the current position in the stream.
- Managing versions:
  void SetVersionsByStorage() - Set the current version to be equal to the storage version.
  VERSION MathVersion() - Return the main version (version of the mathematical kernel).
  VERSION AppVersion(size_t ind) - Return the additional version (version of the target application).
  const VersionContainer & GetVersionsContainer() - Get access to the version container.
  void SetVersionsContainer(const VersionContainer &) - Set the version of open file.
  VERSION SetStorageVersion(VERSION) - Set the storage version.
- Managing registration of objects:
  void registrate(const TapeBase *) - Register the pointer.
  void unregistrate(const TapeBase *) - unregister the pointer.
  bool exist (const TapeBase *) - Does a registered object exist?
  void flushRegister() - Flush the registration array.
  size_t RegisteredCount() - Get the number of registered objects.
  size_t GetMaxRegisteredCount() - Get the maximal possible number of objects for registration.
  void ReserveRegistered(size_t n) - Reserve memory for n objects.
  uint8 GetIndexType(size_t index) - Get index type.
- Working with a progress indicator:
  void InitProgress(IProgressIndicator * pr) - Initialize the progress indicator.
  void InitProgress(ProgressBarWrapper & pr) - Initialize the progress indicator.
  void ResetProgress() - Release the current progress indicator. Set the parent progress indicator, if it exists.
  ProgressBarWrapper * GetProgress() - Get the progress indicator.

O.10.2.2. Write streams

Class writer

The class **writer** provides writing the model to a data buffer (to a file space) in the compact format. It inherits from the base class tape.

The class **writer** implements:

- Static functions for creating the class instance:
  - writer_ptr CreateWriter(std_unique_ptr<iobuf_Seq>, uint16) - Creates the class instance for a given sequential memory buffer of type iobuf_Seq.
  - writer_ptr CreateMemWriter(membuf &, uint8) - Creates the class instance for a given memory buffer of type membuf.
- The methods for writing objects of different types to the buffer:
  - void writeObject(const TapeBase *) - Write an object. void writeObjectPointer(const TapeBase *) - Write a pointer to an object.
  - void writeByte(uint8) - Write a byte to the buffer.
  - void writeBytes (const void *, size_t) - Write a bytes sequence to the buffer. void writeUInt64(const uint64 &) - Write unsigned 64-bit integer. Returns a number of written bytes.
  - void writeInt64 (const int64 &) - Write a 64-bit integer. Returns a number of written bytes.
  - writer & __writeChar (const char *) - Write CHAR string to the stream. (ANSI coding, Russian locale).
  - writer & __writeWchar(const TCHAR *) - Write WCHAR string to the stream (stored in the stream as UTF-16).
  - writer & __writeWcharT(const wchar_t *) - Write WCHAR string to the stream (stored in the stream as UTF-16).
  - size_t __lenWchar(const TCHAR *) - Length of WCHAR string in the stream (stored in the stream as UTF-16).
- The methods for writing a model tree and access to it (these methods are not supported by this class - have an empty implementation):
  - void WriteModelCatalog() - Write a model tree.
  - const c3d::IModelTree * GetModelTree() const - Get a pointer to the model tree.

Class writer_ex

The class **writer_ex** provides writing the model to a data buffer (to a set of file spaces) in the extended format. It inherits from the class writer.

The class **writer_ex** implements:

- Static functions for creating the class instance:
  - std_unique_ptr<writer_ex> CreateWriterEx(std_unique_ptr<iobuf_Seq>, uint16) - Creates the class instance for a given sequential memory buffer of type iobuf_Seq.
  - std_unique_ptr<writer_ex> CreateMemWriterEx(membuf &, uint8) - Creates the class instance for a given memory buffer of type membuf.
- The methods for writing a model tree and access to it:
  - void WriteModelCatalog() - Write a model tree.
  - const c3d::IModelTree * GetModelTree() const - Get a pointer to the model tree.

O.10.2.3. Read streams

Class reader

The class **reader** provides reading from a data buffer (a file space), written in the compact format. It inherits from the base class tape.

The class **reader** implements:

- Static functions for creating the class instance:
  - reader_ptr CreateReader (std_unique_ptr<iobuf_Seq>, uint16) - Creates the class instance for a sequential memory buffer.
  - reader_ptr CreateMemReader (membuf &, uint8) - Creates the class instance for a memory buffer.
- The methods for reading objects of different types from the buffer:
  - TapeBase * readObject (TapeBase *) - Read an object.
  - TapeBase * readObjectPointer() - Read a pointer to an object.
  - bool readUInt64(uint64 &) - Read unsigned 64-bit integer from the buffer.
  - bool readInt64(int64 &) - Read 64-bit integer from the buffer.
  - int readByte() - Read a byte from the buffer.
  - bool readBytes(void *, size_t) - Read bytes sequence from the buffer.
- The methods for managing reading of the model tree and partial reading of the model objects (these methods are not supported by this class - have an empty implementation):
  - void ReadObjectCatalog() - Read the model table of contents.
  - membuf * ReadObjectByPosition (const ClusterReference &) - Read an object by its position in the buffer.
  - bool SetReadPosition (ClusterReference &) - Set reading position.
  - const c3d::IModelTree * GetModelTree() const - Get a pointer to the model tree.
  - bool IsFullRead() - Get an indicator of full reading of the current object (whether the current object is read in the whole).
  - void SetFullRead(bool) - Set an indicator of full reading of the current object (read the current object in the whole).
- The method to get reading errors:
  - uint32 GetLastError() - Get reading errors.
- The methods for working with progress indicator.:
  - void InitProgress(IProgressIndicator *) - Initialize the progress indicator.
  - void InitProgress(ProgressBarWrapper &) - Initialize the progress indicator.

Class reader_ex

The class **reader_ex** provides reading from a data buffer (a set of file spaces), written in the compact or extended format. It can read objects from several file spaces by given positions. It inherits from the class reader.

The class reader_ex **i**mplements:

- Static functions for creating the class instance:
  - std_unique_ptr<reader_ex> CreateReaderEx (std_unique_ptr<iobuf_Seq>, uint16) - Creates the class instance for a sequential memory buffer.
  - std_unique_ptr<reader_ex> CreateMemReaderEx(membuf&, uint8) - Creates the class instance for a memory buffer.
- The methods for managing reading of the model tree and partial reading of the model objects:
  - void ReadObjectCatalog() - Read the model table of contents.
  - TapeBase * ReadObjectByPosition (const ClusterReference &) - Read an object by its position in the buffer.
  - bool SetReadPosition (ClusterReference &) - Set reading position.
  - const c3d::IModelTree * GetModelTree() const - Get a pointer to the model tree.
  - bool IsFullRead() - Get an indicator of full reading of the current object (whether the current object is read in the whole).
  - void SetFullRead(bool) - Set an indicator of full reading of the current object (read the

current object in the whole).
- The method to get reading errors:
  - uint32 GetLastError() - Get reading errors.

Scoped progress indicator for reading classes

The class **ScopedReadProgress** implements a progress indicator in the scope for classes of model reading. It creates a child progress indicator in the scope for the current instance of reader or reader_ex.When exiting the scope, the current progress indicator is released and the parent progress indicator is set.

The class defines an operator for accessing the current progress indicator:
ProgressBarWrapper * operator()().

O.10.2.4. Read-write stream

The class **rw** provides reading and writing the model in the data buffer (in a file space) in the compact format. It inherits from the classes reader and writer.

The class implements a static function for creating the class instance:
rw_ptr CreateMemWriter(membuf &, uint8).

O.10.2.5. Model tree

The classes for the model tree are defined in the file io_tree.h.

A tree node data is declared as a structure **MbItemData**.

The class **IModelTreeNode** defines an interface for a tree node, which can have several descendants, and is able to write itself to the stream and read itself from the stream.

The class IModelTreeNode contains ordered arrays of pointers to the immediate descendants and the immediate ancestors of the node.

The class IModelTreeNode declares the methods:
- std::set<const IModelTreeNode*>& GetParents() - Access to the immediate node ancestors.
- const std::set<const IModelTreeNode*>& GetParents() const - Access to the immediate node ancestors.
- std::set<const IModelTreeNode*>& GetChildren() - Access to the immediate node descendants.
- const std::set<const IModelTreeNode>& GetChildren() const - Access to the immediate node descendants.
- void AddParent(IModelTreeNode* parent) - Add an ancestor.
- void AddChild(IModelTreeNode* child) - Add an descendant.
- MbItemData& GetData() - Access to the node data.
- const MbItemData& GetData() const - Access to the node data.
- ClusterReference& GetPosition() - Access to the position of the node reading/writing.
- bool PartialRead() - Check whether to read the node partially. While separate reading an object there can be a need to read some data from its parent. In this case the parent object is read partially and has a corresponding flag.
- void SetPartialRead (bool partial) - Set an indication of full or partial node reading.
- writer& operator >> (writer &) - Write the node to the stream.
- reader & operator << (reader&) - Read the node from the stream.

The class **IModelTree** defines an interface for a generic model tree, which is used for creating, writing and reading of the model table of contents in the memory buffer.

The class defines types of callback functions:
- FilterNodesFunc - The type of a function for selecting tree nodes by filters.
- NodeToAddFunc - The type of a function for determining, whether to add the object to the

model tree, and for filling the node data.

The class IModelTree defines a enumeration TreeType for tree types:

- mtt_Model - Tree contains a standard model (MbModel object).
- mtt_Embodiment - Tree contains embodiments.

The class declares the methods:

- TreeType GetType() const - Get the tree type.
- void SetType(TreeType type) - Set the tree type.
- void AddNode (const TapeBase*, const ClusterReference&) - Add a node to the tree.
- void CloseNode(const IModelTreeNode*) - Notification about the end of current node writing/reading (the function must be called when the read/write of the given node is complete).
- std_unique_ptr<const IModelTree> GetFilteredTree (const std::vector<MbItemData>& filters) - Build a tree with nodes, selected by filters. For a tree of mtt_Embodiment type, the function works with the first embodiment.
- std_unique_ptr<const IModelTree> GetFilteredTree (std::vector<const IModelTreeNode*>& nodes) - Build a tree for given nodes. Not applicable to a tree of mtt_Embodiment type (in this case, returns NULL).
- const IEmbodimentTree* GetEmbodimentsTree() const - Get pointer to the embodiments tree. Return NULL if not applicable (in case of tree of mtt_Model type).
- void SetNodeToAddFunction(NodeToAddFunc callback) - Define a function for selecting a geometric object for adding to the model tree, and filling the node data.
- void SetFilterFunction(FilterNodesFunc callback) - Define a function for selecting nodes from the model tree.
- writer & operator >> (writer &) - Write the tree to the stream.
- reader & operator << (reader &) - Read the tree from the stream.
- Access to the tree roots. A tree node could be nested recursively (for example, an Instance can contain an Assembly which contains another Instance which includes this Assembly).
  - const std::vector<const IModelTreeNode*>& GetRoots() - Get the tree roots.
  - std::vector<const IModelTreeNode*>& GetRoots() - Get the tree roots.
- VERSION GetVersion() - Get the tree version.
- void SetVersion(VERSION) - Set the tree version.
- static IModelTree* CreateModelTree() - Create a tree instance.

The class **IEmbodimentNode** defines an interface for an embodiment tree node. It contains ordered arrays of pointers to the immediate descendants of the node in an embodiment tree.

The class declares the methods:

- std_unique_ptr<const IModelTree> GetEmbodiment() const - Build a tree of a model which is contained in a given embodiment.
- const MbItemData& GetEmbodimentData() const - Access to the embodiment info.
- std::set<const IEmbodimentNode*>& GetChildren() - Access to the immediate node children.
- const std::set<const IEmbodimentNode*>& GetChildren() - Access to the immediate node children.
- void AddChild(IEmbodimentNode* child) - Add a child.

The class **IEmbodimentTree** defines an interface for an embodiment tree which presents a hierarchy of variants of model implementation (embodiments). Each node of the tree presents an embodiment.

The class declares the methods:

- const std::vector<const IEmbodimentNode*>& GetRoots() const - Access to the tree roots.
- std::vector<const IEmbodimentNode*>& GetRoots() - Access to the tree roots.

O.10.2.6. Streaming objects

Model objects which can be serialized (represented as a sequence of bytes) are called streaming objects.

Classes of streaming objects inherit from the base class TapeBase, which is defined in the file tape.h.

Streaming objects can be registered and not registered. This determines how they should be written to the stream. A registered object is written separately, and model objects, which refer to it, contain a link to it. Not registered object is written inside the object, which refers to it.

Types of streaming object registration

Types of streaming objects registration defined in the enumeration **RegistrableRec:**

- noRegistrable - Not registered object.
- registrable - Registered object.

Base class of streaming objects TapeBase

The class **ClassDescriptor** defines a packed name of streaming class. It keeps a class name hash and an application identifier.

Classes of streaming objects inherit from the base class **TapeBase**, which defines the following methods:

- RegistrableRec GetRegistrable() const - Get registration type of the streaming class.
- void SetRegistrable(RegistrableRec regs) - Set the registration state of the streaming class.
- ClassDescriptor GetClassDescriptor(const VersionContainer &) - Get the class descriptor.
- const char * GetPureName(const VersionContainer &) - Get the class name.
- bool IsFamilyRegistrable() const - Whether the object belongs to a registrable family.

O.10.2.7. Modes of streaming operations

Modes of streaming operations described in the enumeration **io::mode_flags** in the file io_buffer.h:

- in - Open stream for reading.
- out - Open stream for writing.
- trunc - Open existing file and clear its content.
- speedOnClose - Sort while closing.
- delIfEmpty - Delete file if it is empty.
- delOnClose - Delete file while closing.
- recovery - Recovery mode.
- appSpecial - An auxiliary flag of application.
- createNew - Create a new file. An error is generated if the file already exists.
- createAlways - Create a new file. If the file already exists, then it is to be rewritten.
- openExisting - Open an existent file. An error is generated if the file does not exist.
- openAlways - Open an existent file. If the file does not exist, a new file is created.
- truncExisting - Open the file with clearing of its content.

O.10.2.8. Stream states

Flags of stream state are defined in the enumeration **state** in the file io_buffer.h:

- good - Everything is all right (no bits are set).
- eof - The end of the file.
- outOfRead - Out of limits of the file.
- outOfMemory - Failed to allocate requested memory.
- fail - Input-output error.
- badData - Incorrect file structure.
- notFound - File not found.
- accessViolation - Access is denied.

- cantOpenStore - Can't open the storage.
- cantCreateStore - Can't create a storage.
- badSig - There is no signature or the signature is wrong.
- cantReadCatalog - Can't read the storage catalog.
- cantWriteCatalog - Can't write the catalog of the storage.
- cantFind - Can't find a file in the catalog.
- cantRead - Can't read the file.
- cantWrite - Can't write the file.
- badClassId - The class identifier was not find in the database.
- doubledClassId - Attempt for repeated registration of the class identifier.
- verViolation - The file version is older than the task version.
- hardFail - File operation error.
- closed - The buffer is closed.
- writeProtect - The "Read only" attribute is set to the file.
- cantWriteObject - Can't write the object. (the stream version is newer than the new objects class occurrence version).
- underflow64to32 - Can't read file with 64-bit data in 32-bit task (loss of upper word uint32 while reading uint64 in 32-bit task).
- encrypted - The file is protected or encrypted.
- skippedUnknown - Partial read of the file in the extended format (unknown objects skipped).
- readAborted - The file reading aborted by user.
- allMask - All errors.

O.10.3. Working with streaming buffer

O.10.3.1. Cluster

Cluster is a continuous memory region (a sequence of bytes), which is described by the class Cluster in the file io_buffer.h.

The class contains the cluster start (shift - for disk clusters, address - for memory) and the cluster length in bytes.

The class implements the methods for accessing the cluster data:

- uint16 _len() - Get the cluster length.
- size_t _off() - Get the cluster start field.
- const uint8 * _ptr() - Get the cluster start as an address.
- uint8 * _getMemPointer() - Get the cluster start as an address.
- void clear() - Clear the cluster.
- void AllocFile(size_t beg, uint16 len) - Memorize the shift in file and the bytes count.
- void SetClusterOffset(size_t off) - Set the cluster start.
- void SetClusterLength(uint16 len) - Set the cluster length.
- static size_t SizeOf(VERSION version) - Size of the cluster data in the stream for a given version.

O.10.3.2. File space

File space is a set of clusters. File space is described by the class FileSpace in the file io_buffer.h.

The class contains the array of cluster indexes and the number of used bytes in the last cluster.

The class implements the methods for accessing the array of cluster indexes:

- size_t Count() - Get the number of elements in the array of cluster indexes.
- void Flush() - Set the number of elements in the array to zero.

- void RemoveInd(size_t) - Delete an element from the array.
- size_t * Add() - Add an element to the end of the array.
- size_t * Add(const size_t &) - Add a given element to the end of the array.
- void SetSize(size_t, bool) - Set the new array size.
- void Reserve(size_t) - Reserve space for a given number of elements.
- bool IsExist(size_t &) - Return true, if the element found.
- size_t FindIt(size_t & el) - Find an index of the element (if not found, return -1).
- size_t * InsertInd(size_t index, const size_t & el) - Insert an empty element before the given one.
- const size_t * GetAddr() - Get the address of the beginning of the array.
- size_t * AddItems(size_t n) - Add *n* elements to the end of the array.
- size_t & operator [](size_t loc) - Access operator by index.
- uint16 & rest() - Get the number of used bytes in the last cluster.

O.10.3.3. Read/write position

The class ClusterReference describes the position in the cluster for reading/writing. It is defined in the file io_buffer.h.

The class contains the cluster index in the array of cluster indexes if the buffer iobuf_Seq and the offset in this cluster.

O.10.3.4. Streaming sequential buffer

The class iobuf_Seq describes streaming buffer, which provides only sequential writing/reading (the base class). The class iobuf_Seq and its inheritors serve for reading/writing operations in the interest of stream (the class tape). The class is defined in the file io_buffer.h.

The following terms are used:

- Storage – a disk file or a memory region.
- File - a file space inside a storage.

The main class data:

- FileSpace sys - "system" file space. Opens in the constructor of the class tape.
- PArray<FileSpace> files - An array of file spaces, which are contained in iobuf_Seq. The first element of the array is always an address of sys.
- uint32 stateFlag - The buffer state.
- VERSION storageVers - The storage version.
- VERSION curFileVers - The version of the current open file space (stream). In general case the storage version and a version of any file space in it may be different.
- uint8 bufferMode - Mode in which the buffer can operate.
- uint8 curFileMode The opening mode of the current file. In general case the storage mode and the mode of open file in this storage can be different. Constraint - if the buffer mode is io::in, then an attempt to open the file for writing (io::out) doesn't result in opening.
- uint8 * base — A pointer to the beginning of the buffer in memory.
- uint8 * ptr - A pointer to the next symbol in memory.
- uint8 * end - A pointer to the end of buffer in memory. When working with the disk, pointers are set to a fixed memory block the sections of file are loaded to while reading. When working with the memory, membuf sets them to the memory allocated for the cluster.

The main methods of the class:

- Access to the array of the clusters:
    - void Reserve(size_t n, bool addAdditionalSpace) - Reserve space for a given number of elements.
    - void Flush() - Set the number of elements to null.
    - void HardFlush() - Free the whole memory.
    - void Adjust() - Free the unnecessary memory.

- Cluster * Add() - Add an element to the end of the array.
- Cluster * Add(const Cluster &) - Add a given element to the end of the array.
- size_t Count() - Get the number of elements in array.
- Cluster & operator [](size_t) - Access by index operator.

- Access to file spaces:
  - void ReserveFiles(size_t n, bool addAdditionalSpace) - Reserve space for a given number of FileSpace.
  - ClusterReference getCurrentClusterPos() - Get current position in the buffer.
  - FileSpace & sysFile () - Get access to the system file.
  - FileSpace * openedFile() - Get access to the open file.
- Access to versions:
  - void SetVersionsByStorage() - Set the current version to be equal to the storage version.
  - VERSION MathVersion() - Return the main version (version of the mathematical kernel).
  - 448 AppVersion(size_t) - Return the additional version (version of the target application).
  - const VersionContainer & GetVersionsContainer() - Get the buffer versions.
  - VERSION GetStorageVersion() - Get the storage version.
  - VERSION GetFormatVersion() - Get the format version.
  - void SetFormatVersion (VERSION) - Set the format version.

O.10.3.5. Streaming buffer with arbitrary access

The class **iobuf** describes a streaming buffer with arbitrary access (the base class). It inherits from the class iobuf_Seq and extends its functionality with the ability to delete and overwrite file spaces. It is defined in the file io_buffer.h.

The class contains the file space with the list of freed clusters. When a file space is deleted, all its clusters are moved here. While writing, when it is necessary to allocate the new cluster, this array is firstly checked for availability of free clusters. Clusters in 'freed' are always ordered by shift from the beginning of the file space.

The class methods:

- bool open (FileSpace &, uint8, const VersionContainer &, bool fullCheck = true) - Open the file space if it is one's own file. The flag fullCheck = false switches off excessive checks (for the sake of performance).
- bool del(FileSpace &) - Free space allocated for the file space.
- bool truncate(FileSpace &, size_t) - Truncate the file space.
- bool detach(FileSpace &) - Detach the file space from the buffer.
- bool speedOnClose() - Whether should be ordered while closing.
- void speedOnClose(bool) - Set the flag of ordering while closing.
- void free(size_t) - Free the cluster.

O.10.3.6. Memory streaming buffer

The class membuf implements memory stream buffer. It inherits from the class iobuf. The class is defined in the file io_memory_buffer.h.

Memory stream buffer is intended for using in read and write streams. Besides the instruments needed for the buffer, it has functions for packing to a contiguous memory block, that enables transferring data to another application via the memory.

The class contains the array of file spaces, which stores all used file spaces except 'sys' defined in iobuf_Seq. When using the extended format, objects of each level are saved to a separate file space. Index of a file space in the array corresponds to the level of the object in the model.

The main class methods:

- bool isEmpty() - Whether the buffer is empty.
- size_t toMemory(const char *& m, size_t addSize = 0) - Write to contiguous memory. If

m != 0 (memory is already allocated), then addSize is equal to memory size, if m = 0, then addSize defines a number of bytes to be added at the beginning and zeroed.

- bool fromMemory(const char *) - Read from the contiguous memory.
- size_t getMemLen() - Compute necessary length of the contiguous memory block for a buffer.
- void clean() - Clear the buffer.
- void closeBuff() - Close the buffer.

O.10.3.7. Reading and writing memory buffer

The functions for writing a contiguous memory buffer to a disk file and reading to contiguous memory buffer from a disk file are defined in the file io_memory_buffer.h.

Reading to a contiguous memory buffer from a disk file:

- iobuf & createiobuf(const TCHAR * fileName).

Writing a contiguous memory buffer to a disk file:

- bool writeiobuftodisk(const TCHAR * fileName, membuf & buf).

O.10.4. Version container

The type VERSION defines a version. It is defined in the file system_types.h.

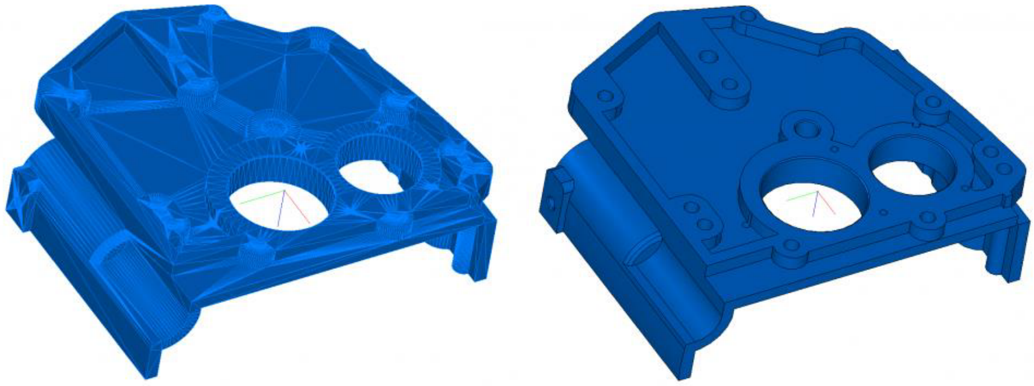
The class **VersionContainer** defines a container for versions of geometric model objects. It implements the following main methods:

VERSION GetMathVersion() - Get the main version (the mathematical kernel version).
VERSION GetAppVersion (size_t ind) - Get the additional version (the target application version).
void Flush() - Flush the container.
void SetVersion(size_t index, VERSION ver) - Set the version by index.
size_t ToMemory(const char *&) - Write the container to the memory block.
size_t FromMemory(const char *) - Read the container from the memory block.

The class VersionContainer is defined in the file io_version_container.h.

P.1. CONVERTING OF POLYGONAL MODELS

C3D B-Shaper lets you work with polygonal models in MCAD, AEC, BIM, and other CAD applications by converting the models to boundary representation (b-rep) bodies. Polygonal models are the typical result from 3D scanners and non-CAD 3D modeling software, such as those used to develop movies and games. B-rep is the primary method of representing 3D models in geometric software such as CAD.



The key applications for programs using C3D B-Shaper include the following:

- **Product Catalog Conversions** – download polygonal models from online libraries, and then turn them into CAD models
- **CAE Post-processing** – process the results of topological optimization from CAE systems
- **Enhancing Polygonal Models** – smooth grids, decimate surfaces, apply compression algorithms, and so on.

B-Shaper’s unique algorithm first segments meshes by dividing sets of polygons into subsets (segments), which become prototypes for probable faces. In the next step, selected areas are recognized as elementary surfaces (planes, cylinders, cones, spheres, or tori), revolution surfaces or NURBS surfaces.
Intersection curves are calculated between adjacent segments, and then these curves become the basis for constructing edges of the body’s faces.

Through its API, B-Shaper operates in two modes: automatic and interactive. It should be noted that working in automatic recognition mode is generally not suitable for models obtained as a result of 3D scanning or topological optimization. This mode is intended for use in polygonal models, which are based on some models created in CAD applications.

P.1.1. Automatic shell recognition mode by polygon mesh

The C3D B-Shaper automatic conversion interface is represented by two functions: **ConvertMeshToShell**() и **ConvertCollectionToShell**().

The method

```
MbResultType ConvertMeshToShell(       MbMesh & mesh,
                                  MbFaceShell *& shell,
                                  const MbMeshProcessorValues & params = MbMeshProcessorValues() )
```

performs shell creation in the boundary representation (B-Rep), corresponding to the model specified by the polygonal mesh. The module algorithm automatically recognizes and reconstructs the faces corresponding to elementary surfaces (plane, cylinder, cone, sphere, torus), revolution surfaces or NURBS surfaces.

Method input parameters are:

- mesh – polygonal mesh of the model;
- params – conversion settings.

The output parameter of the method is shell – the constructed shell.

If the algorithm works successfully, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

The method is declared in the action_b_shaper.h file.

The conversion settings include the recognition tolerance value, i.e. the maximum allowable distance from the vertices of the polygonal mesh within the boundaries of this segment to the recognized surface. This tolerance can be absolute or relative: when using relative tolerance, the deviation of the faces of the body from the grid is checked with respect to the size of the model. Also, the user has the option of switching recognition modes, which allows you to control the types of surfaces during reconstruction. These parameters are in the fields of the MbMeshProcessorValues class, which contains:

- useRelativeTolerance – relative tolerance flag;
- tolerance – recognition tolerance;
- surfReconstructMode – surface recognition mode, from the listing:
    1. srm_All – build all surfaces;
    2. srm_NoGrids – do not build surfaces based on triangulation;
    3. srm_CanonicOnly – build only elementary surfaces;
    4. srm_Default – default mode.

The method

```
MbResultType ConvertCollectionToShell ( MbCollection & collection,
                                        MbFaceShell *& shell,
                                  const MbMeshProcessorValues & params = MbMeshProcessorValues() )
```

performs shell creation in the boundary representation (B-Rep) corresponding to the collection of elements containing the polygonal mesh. The module algorithm automatically recognizes and reconstructs the faces corresponding to elementary surfaces (plane, cylinder, cone, sphere, torus).

Method input parameters are:

- collection – collection of elements containing the polygonal mesh;
- params – conversion settings.

The output parameter of the method is shell – the constructed shell.

If the algorithm works successfully, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

The method is declared in the action_b_shaper.h file.

The test application builds the shell in the boundary representation using the menu command "Create->Solid->Based on a mesh-> Convert to the solid with surface recognition".

P.1.2. MbMeshProcessor class – shell creation based on a polygon mesh with user settings

Advanced management of segmentation and surface recognition processes is provided by the **MbMeshProcessor** class interface.

```
class MbMeshProcessor {
public:
// Creating a processor instance using a collection of items.
MbMeshProcessor * Create( const MbCollection & collection );
// Destructor
~MbMeshProcessor();
// Recognition tolerance.
void SetRelativeTolerance( double tolerance );
void SetTolerance( double tolerance );
double GetTolerance() const;
// Mesh healing.
void SetUseMeshSmoothing( bool useSmoothing );
// Mesh segmentation.
const MbCollection & GetSegmentedMesh();
MbResultType SegmentMesh( bool createSurfaces = true );
void ResetSegmentation();
void UniteSegments( size_t firstSegmentIdx, size_t secondSegmentIdx );
MbResultType SegmentMeshBySeparators( const std::vector<std::vector<uint>> & sep );
// Surface recognition.
void SetReconstructionMode( MbeSurfReconstructMode mode );
void FitSurfaceToSegment( size_t idxSegment );
void FitSurfaceToSegment( size_t idxSegment, MbeSpaceType surfaceType );
const MbSurface * GetSegmentSurface( size_t idxSegment ) const;
// Building a B-Rep model.
MbResultType CreateBRepShell( MbFaceShell *& pShell );
..
}
```

The class description is in the action_b_shaper.h file.
Futher, more detailed descriptions of the methods of this class will be considered.

P.1.3. Recognition tolerance

One of the key settings for converting a polygonal mesh into a model with a boundary representation is the value of recognition tolerance. The tolerance of the method is determined by the maximum allowable deviation of the recognized surfaces from the vertices of the polygonal mesh. The user can set the tolerance required in the calculations, or the default value will be used. In the case when the parameter of the deviation of the faces from the grid is not known in advance, you can use the relative tolerance, which will be calculated based on the size of the original body. Thus, in the case of an unsatisfactory result of the algorithm, the user can influence it by changing the recognition tolerance parameter.
Several functions of the **MbMeshProcessor** class are used to control the tolerance value.

The method
void **SetRelativeTolerance** (double tolerance)
sets the value of relative tolerance. When using relative tolerance, the deviation of the faces of the body from the grid is checked relative to the dimensions of the current grid. The input parameter to the method is:

- tolerance – relative tolerance value.

To obtain an absolute recognition error, the tolerance value will be multiplied by the diagonal of the dimensional cube of the polygonal mesh or a collection of model elements. Thus, for a given relative tolerance equal to 1.0, the value of the absolute tolerance will be calculated based on the size of the model.

The method
void **SetTolerance** (double tolerance)
sets the absolute tolerance of surface recognition and expansion of mesh segments. The input parameter to the method is:

- tolerance – absolute tolerance value.

This method must be called before calling the **SegmentMesh**() method. The default absolute accuracy is 0.1.

The method
double **GetTolerance** () const
transmits the value of the current absolute tolerance used in recognizing surfaces and expanding grid segments.
The method returns the absolute tolerance by a double type value.

P.1.4. Polygon mesh segmentation editing

Often, polygon mesh models, being the result of 3D scanning, have a complex internal structure with the presence of "noise" - random outliers of points outside the dimensions of the face. The transformation of such grids implies a denser interaction with the user, since automatic recognition is difficult. Several tools are provided for correcting segmentation results.

The method
MbResultType **SegmentMesh** (bool createSurfaces = true)
performs segmentation of the given polygon mesh.
The input parameter of the method is:

- createSurfaces – flag using for creating surfaces on segments.

If the algorithm works successfully, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

The method
void **UniteSegments** (size_t firstSegmentIdx,
size_t secondSegmentIdx)
merges two segments in the current polygon mesh segmentation.
Method input parameters are:

- firstSegmentIdx – index of the first segment to merge;
- secondSegmentIdx – index of the second segment to merge.

The result of the union is available through the collection returned by the method **GetSegmentedMesh**().

The method
MbResultType **SegmentMeshBySeparators** (const std::*vector*<std::*vector*<uint>> & separators)
performs segmentation of the polygonal mesh by segment delimiters.
The input parameter of the method is:

- separators – an array of separators, each of which contains a path along the vertices of the grid whose edges separate the segments.

If the algorithm works successfully, the method returns rt_Success, otherwise the method returns an error code from the MbResultType enumeration.

P.1.5. Surface reconstruction on a segment

When using the C3D B-Shaper, developers can reconstruct a surface of a certain type on a segment. The following methods are used to control surface recognition.

The method
void **FitSurfaceToSegment** (*size_t* idxSegment)
recognizes the surface by a grid segment with a given index and fits it into the segmentation. A recognized surface can be obtained by calling the **GetSegmentSurface**() method. The input parameter idxSegment passes the index of the polygon mesh segment.

The method
void **FitSurfaceToSegment** (*size_t* idxSegment, MbeSpaceType surfaceType)
performs the construction of a surface of a given type approximating a mesh segment with a given index. A recognized surface can be obtained by calling the GetSegmentSurface() method. The input parameter idxSegment passes the index of the polygon mesh segment.

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 Fig. R.1.1.1 as their input.

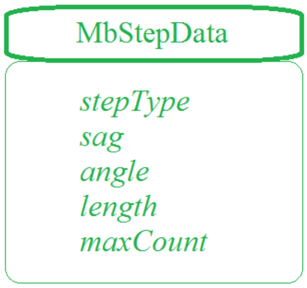


Fig. R.1.1.1.

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 Fig. R.1.1.2 as their input.

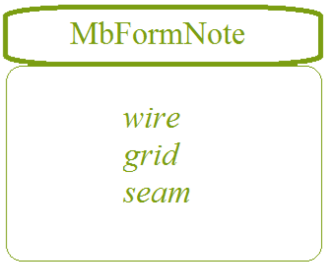


Fig. R.1.1.2.

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 body, 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 body 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 body object that was constructed with the following parameters: *note.wire*==true, *note.grid*==false, the object contains only polygons.

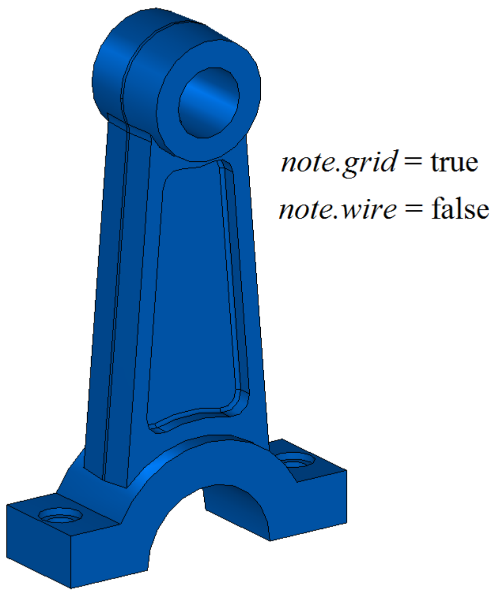


Fig. R.1.2.1.

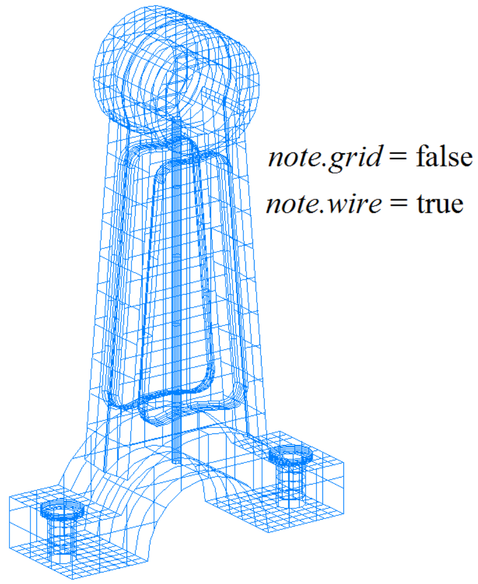


Fig. R.1.2.2.

In Fig. 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.

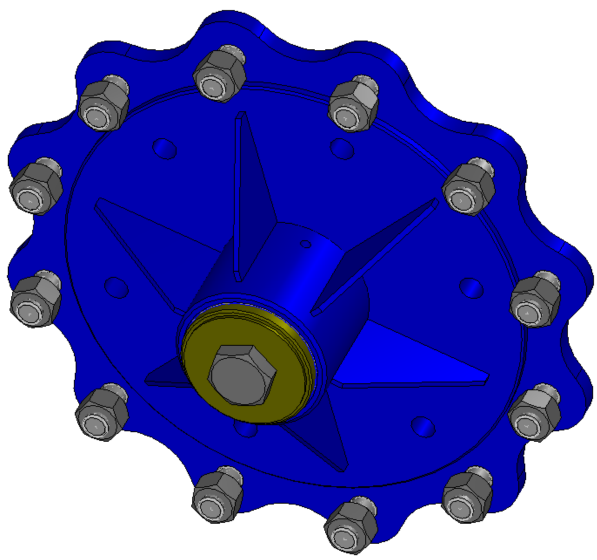


Fig. R.1.2.3.

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 Fig. 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.

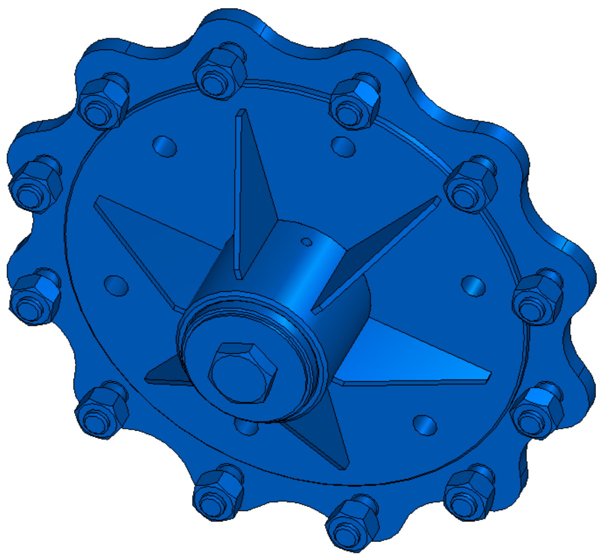


Fig. R.1.3.1.

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 Fig. R.1.4.1, you can see a curve and its polygonal object consisting of one polygon.

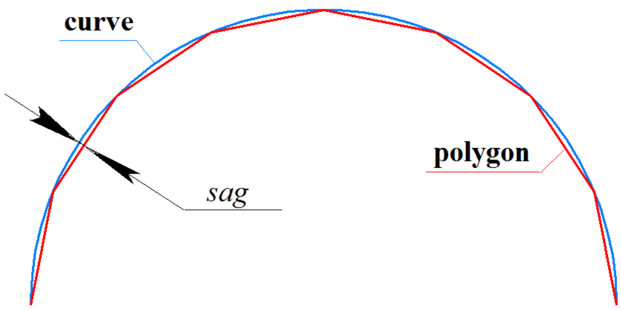


Figure R.1.4.1.

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 Fig. R.1.4.2, you can see a polygonal object of a surface that consists of several broken lines.

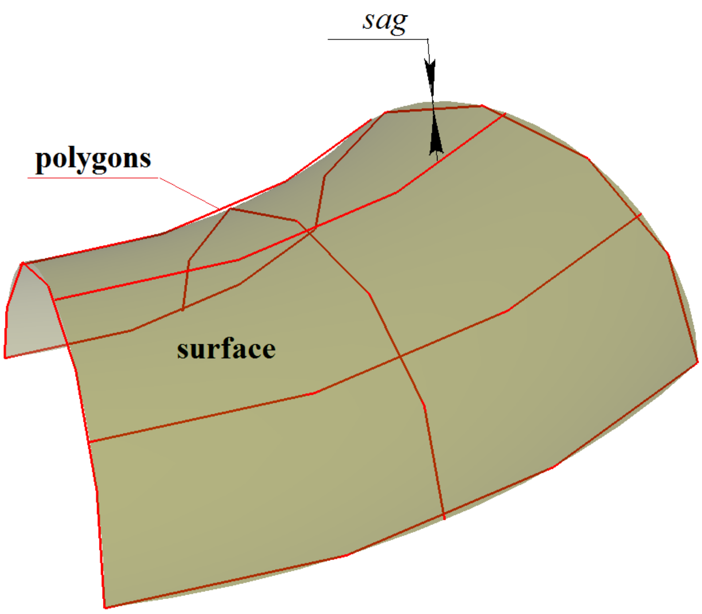


Fig. R.1.4.2.

The method uses a set of broken lines to approximate a body. 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 body edges are based. In Fig. R.1.4.3 (right), you can see a polygon object for the body.

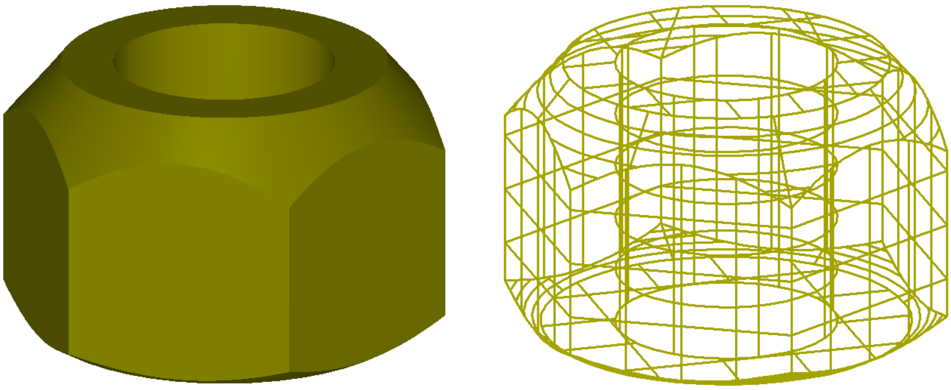


Figure R.1.4.3.

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 Fig. R.1.5.1, you can see triangulation of a flat face with *ist_SpaceStep* mask.

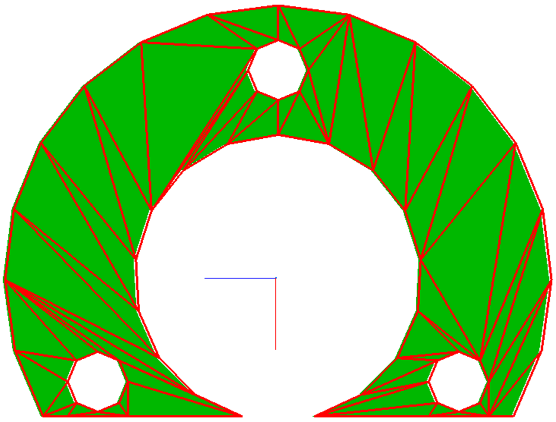


Fig. R.1.5.1.

In Fig. R.1.5.2, you can see triangulation of a curved face with *ist_SpaceStep* mask.

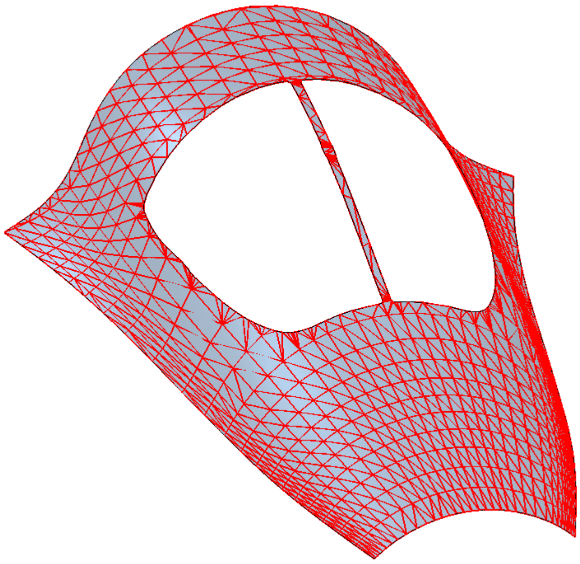


Fig. R.1.5.2.

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 Body

Method

void

CalculateGrid (const MbSolid & **solid**,
const MbStepData & *stepData*,
RPArray<MbGrid> & **grids**)

constructs a set of triangulations for the faces of a body.

Input parameters of the method are:

- **solid** is the body,
- *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** body 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** body 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 bodies. *stepData* parameter is described in Item R.1.1. Triangulation Calculation Control.

R.1.7. Constructing Polygonal Objects for a Set of Bodies

Method

void

CalculateGrid (const RPArray<MbSolid> & **solids**,
const MbStepData & *stepData*,
RPArray<MbMesh> & **meshs**)

constructs a set of polygonal objects for a set of bodies.

Input parameters of the method are:

- **solids** is the set of bodies,
- *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** bodies using **meshs** polygonal objects. When the method is called, **meshs** set should be empty. Each **solids**[*i*] body 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 bodies. *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 Fig. R.2.1.1 is used as method input to construct flat projections to present the bodies.

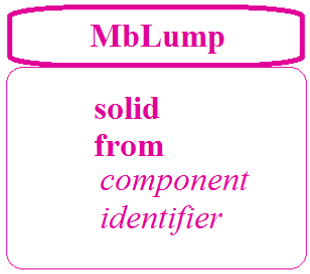


Fig. R.2.1.1.

MbLump structure is declared in lump.h file. **MbLump** contains a pointer to **solid** body, a matrix that transforms the body from **from** local coordinate system, *component* and *identifier* body identification parameters.

MbProjectionsObjects class shown in Fig. R.2.1.2 is used by the method to construct flat projections in order to display supplementary objects.

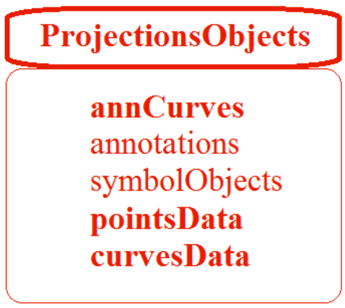


Fig.R.2.1.2.

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 Fig. R.2.1.3 is used to pass flat projection construction results.

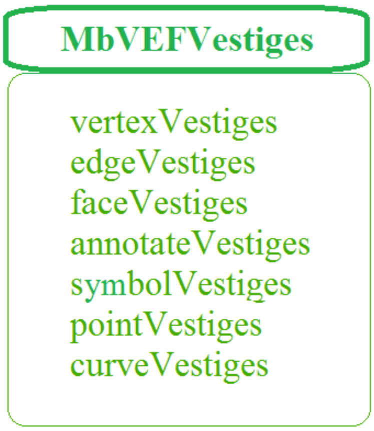


Fig. R.2.1.3.

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 bodies and other objects.

Input parameters of the method are:

- **place** is the projection plane,
- *znear* is observation point parameter,
- **lumps** are the projected bodies 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.

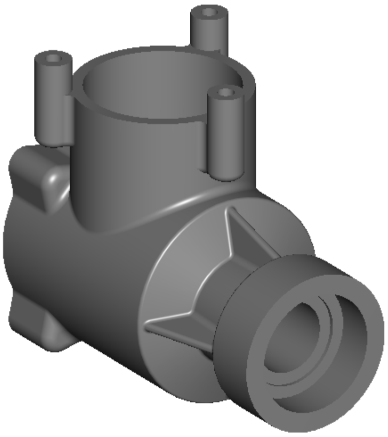
The method is declared in map_create.h file.

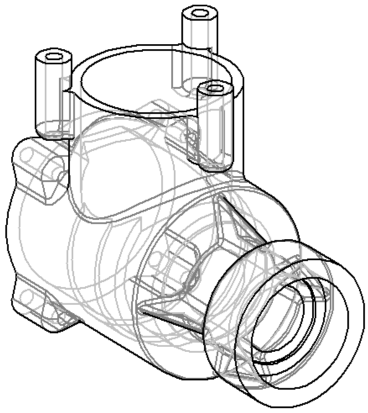
XY plane of **place** local coordinate system is the projection plane.

znear parameter defines image type. If *znear*=0, then a parallel projection of the objects is constructed.

lumps parameter (Fig. R.2.1.1) contains the bodies and matrices of their transformation from the local coordinate system. **objects** parameter (Fig. 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 Fig. R.2.2.1, you can see a body, its projection lines to a plane parallel to the screen are shown in Fig. R.2.2.2. In Fig. R.2.2.3, you can see only visible lines of the projection of the body shown in Fig. R.2.2.1.





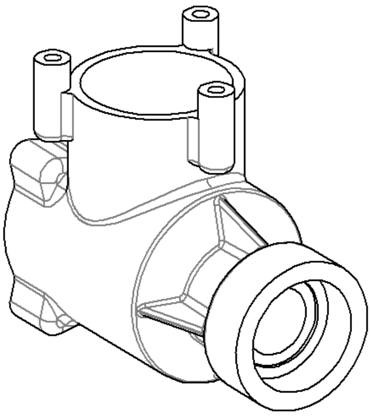


Fig. R.2.2.1. *Fig. R.2.2.2.* *Fig. R.2.2.3.*

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 bodies. It is similar to **GetVestiges** method, the difference is the absence of **objects** auxilliary objects.

R.2.3. Constructing Polygonal Projections of Bodies

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 bodies to the specified plane.

Input parameters of the method are:

- **lumps** are the projected bodies 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 (Fig. R.2.1.1) contains the bodies 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 bodies 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 body 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 Fig. R.3.1.1 is used to set inertial characteristics of a model.

Fig. 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 Fig. R.3.1.2 and Fig. 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.

Fig. R.3.1.2.

Fig. R.3.1.3.

SolidMIAttire class contains the following data:

- constMbSolid & **solid** is the body,
- double *density* is the density or specific gravity of unit area,
- MbMatrix3D **matrix** is the matrix used to transform the body to the nearest assembly system,
- **InertiaProperties** * **properties** are specified inertial characteristics of the body, 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 bodies of the assembly unit,
- MbMatrix3D **matrix** is the matrix used to transform the body into the nearest assembly system,
- **InertiaProperties** * **properties** are specified inertial characteristics of the assembly body, 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 **bodyMIAttire** 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 **bodyMIAttire** 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 Body Characteristics

Function

void

MassInertiaProperties (const MbSolid * **solid**,
double *density*,
double *deviateAngle*,
InertiaProperties & **properties**,
IfProgressIndicator * progress = 0)

calculates body surface area, volume, mass, center of gravity and moments of inertia.

Method input parameters are:

- **solid** is the body,
- *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 body, *density* parameter determines body density. For a non-closed body, **solid** *density* parameter determines specific gravity of body 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 Bodies

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 bodies.

Method input parameters are:

- **solids** is a set of bodies,
- *densities* is a set of densities or specific gravities per unit area,
- **matricies** is the set of matrices used to transform bodies into a common coordinate system,
- **mpSolids** is a set of available characteristics of bodies (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 bodies in **solids** set. The second parameter determines body density for a closed body or specific gravity of unit area for a non-closed body. **matricies** parameter contains a set of matrices used to convert bodies into a coordinate system where the calculation should to be used. **mpSolids** parameter contains the characteristics of the corresponding bodies 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 bodies 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 body volume.

Method input parameters are:

- **solid** is the body,
- *deviateAngle* is a parameter used to control calculation accuracy.

This method returns the body 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).

R.4. COLLISIONS DETECTION OF BODIES

The C3D geometric kernel provides two methods to evaluate solid body collisions. The first method is based on Boolean operation algorithms and designed for accurate evaluation of the intersection of two fixed bodies and calculation of the intersection face edges. The second method is based on polygonal models and designed for dynamic scenes defined by a given set of bodies. It is recommended to use the second method if it is required to execute many evaluations for collisions of moving solid bodies in real time.

R.4.1. Detecting the Intersection of Two Bodies

The method

MbResultType

InterferenceSolids(MbSolid & **solid**1, MbSolid & **solid**2,
std::vector<MbCurveEdge*> * **edges**,
std::vector<ptrdiff_t> (*faceNumbers)[4])

evaluates the intersection of two bodies.

The method input parameters are:

- **solid**1 is the first body,
- **solid**2 is the second body,
- **edges** are the body intersection edges (no edges is also a possible situation),
- (*faceNumbers)[4] is the face number container (it can be empty).

The output parameters of the method are the intersection **edges** of **solid**1 and **solid**2 body faces and the numbers of intersecting faces (*faceNumbers)[4].

The method returns either rt_Intersect if the bodies intersect or rt_NoIntersect if the bodies do not intersect.

The method is declared in the cdet_bool.h file.

The method modifies the **solid**1 and **solid**2 body edges, so if you need to keep the bodies unchanged, then you should pass to the method the copies received using MbSolid::Duplicate(). The bodies are modified by cutting the edges of the bodies with intersection edges.

The returned value, rt_Intersect, means that the pair of bodies has some intersection volume or some contact area. The function returns rt_NoIntersect if the bodies have no intersection volume. The tangency at one point is not considered to be intersection as in this case intersection volume is zero.

If the pointer to the **edges** container is not NULL, then the **solid**1 and **solid**2 intersection edges will be added to the **edges** container.

If the pointer to the faceNumbers face number container is not NULL, then the numbers of the intersecting faces of the first and second body correspondingly will be added to faceNumbers[0] and faceNumbers[1]. Furthermore, the numbers of the contacting faces of the first and second body correspondingly will be added to faceNumbers[2] and faceNumbers[3].

R.4.2. Determining Collisions in the Set of Bodies

This collision evaluation method is used when it is required to perform a series of collision tests for the same set of bodies that change their position in time. Fig. R.4.2.1 shows a mechanism constituting an assembly unit with seven parts that have kinematic links. It is required to estimate the boundaries within which the parts do not interfere with each other when the mechanism works. Fig. R.4.2.2 shows a position of mechanism parts when they collide. The faces of bodies that contacted each other when they moved beyond the free motion boundaries are colored red in Fig. R.4.2.2

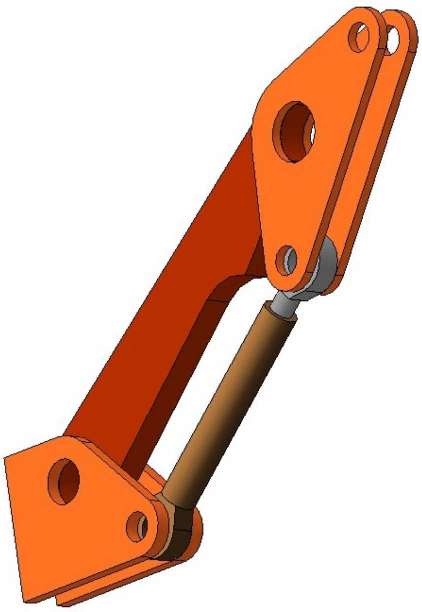


Fig. R.4.2.1.

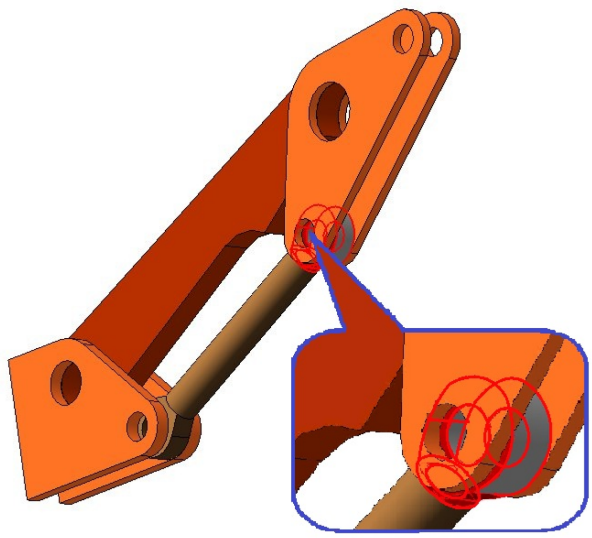


Fig. R.4.2.2.

In order to control collisions for a specific set of bodies, it is required to create an instance of the MbCollisionDetectionUtility class, and add the bodies to the collision control set using the **AddItem** method. At the moment when one or several bodies change their position, the application calls the **Reposition** method to provide new position to the bodies included in the set, and determines whether there is a collision in the geometric model by calling the **CheckCollisions** method.

The MbCollisionDetectionUtility class controls collisions, it is declared in cdet_utility.h file. The additional data types are declared in cdet_data.h.

The method
cdet_item
MbCollisionDetectionUtility::**AddItem** (const MbSolid & **solid**,
const MbPlacement3D & **place,**
cdet_app_item *appItem* = CDET_APP_NULL)
adds **solid** body to the collision control set.

The method input parameters are:

- **solid** is a solid body model,
- **place** is a local coordinate system, which determines the position of the **solid** body in space,
- *appItem* is a client application geometric object containing the body.

The method returns a collision control object descriptor.

The *appItem* parameter is used to identify the client application model objects when they collide. If the *appItem* parameter is not defined, then identification is based only on cdet_item type descriptor that was used to register the body in the set. It is assumed that more than one MbSolid body can belong to one *appItem* object, i.e. the **AddItem** method can be called more than once for the same object with *appItem* parameter. In this case, the bodies from the group added for the same *appItem* parameter are not checked for collisions between them.

The method
cdet_item
MbCollisionDetectionUtility::**RemoveItem** (cdet_item *cdItem*)
deletes a solid body from the collision control set.

The input parameter of the method is:

- *cdItem* is a descriptor of the collision control object that was added earlier using **AddItem** method**.**

The method returns the object descriptor.

The method
cdet_result
MbCollisionDetectionUtility::**CheckCollisions** (cdet_query **cdQuery** = defaultQuery)
checks for collisions between the objects in the control body set.

The input parameter of the method is:

- **cdQuery** is a collision data query structure.

The method returns the collision search result code.

The **CheckCollisions** method detects collisions between the bodies added to the control set by calling the **AddItem** method. The method returns CDET_RESULT_Intersected, if a collision is detected in this body set. If no collisions were detected, then the method returns CDET_RESULT_NoIntersection**.** The cdet_result code is the main result of the **CheckCollisions** algorithm. However, the **cdQuery** structure permits to get more detailed data on collision detection results, and to limit or even interrupt the search when required. If the **cdQuery** parameter is not defined, then the detection algorithm does not lose its time on studying all collision details, rather it stops when the first pair of intersecting faces is found.

If two or more bodies have been added in the same cdet_app_item object, then the collision check is not executed in such group.

R.4.3. Collision Detection Queries

Sometimes, it is required not only to detect a collision of bodies, but also to get a list of colliding faces, as shown in Fig. R.4.2.2, or reveal groups of bodies, for which collision detection is not required. This is achieved using a set of type classes (cdet_query inheritors) configured to request collision details:

- The cdet_query_result structure is the simplest and the quickest variant for detecting collisions in the body control group. This structure is used as a default parameter in the **CheckCollisions** method.
- The cdet_first_collided structure gives the first found pair of colliding faces.
- The cdet_collided_faces structure gives a set of colliding faces as an ordered set of pairs std::pair<cdet_app_item,const MbRefItem*>, where the first element in the pair is a solid body model in the client application, and the second element in the pair is a face belonging to the model. The cdet_collided_faces::**Group** method permits to group the objects of the control body set to detect only intersections between the groups.

The cdet_collided_faces::**Group** method groups a pair of bodies; in order to join ***n*** bodies in a group, you should call the cdet_collided_faces::**Group** method ***n***-1 times for the first body and all other bodies in the group.

Item R.4.4. Configuring a Collision Detection Query describes how the geometric kernel user can configure his/her own queries by using the cdet_query inheritance. However, in most cases you can simply use the cdet_collided_faces class for a query. To do this, please create cdet_collided_faces instance and use it as a parameter in **CheckCollisions** method. Before this, you can configure the detection filter using the cdet_collided_faces::**Group** and cdet_collided_faces::**ExcludeGroup** methods. If required, you can reset the search filter by calling the cdet_collided_faces::Reset method and reconfiguring the filter.

The method
cdet_collided_faces::**Group**(cdet_app_item *fst*,

cdet_app_item *snd*)
declares a group of bodies, the elements of which are not checked for collision with each other.

The method input parameters are:

fst and *snd* are a pair of bodies already included in the groups or grouped for the first time.

This method groups two bodies or adds one body to the group of the other body. So, if you want to group ***n*** bodies, you need to call the cdet_collided_faces::Group method ***n***-1 times for the first body and for all other bodies in the group.

The method

cdet_collided_faces::**ExcludeGroup**(cdet_app_item *member*)

excludes all group elements from the collision detection.

The input parameter of the method is:

member is any body in the group, the elements of which are excluded.

This method disables all collision checks in this group. It is recommended to use this method to temporarily disable the body check collisions without complete deletion from the set using the MbCollisionDetectionUtility::**RemoveItem method,** because adding them later using **AddItem** method requires more computation time than cancelling the exclusion using the cdet_collided_faces::**Reset** method. There are two possible ways to exclude a body set from collision detection: either call the **ExcludeGroup** method many times for all set elements or first group all the bodies that should be excluded and then call the **ExcludeGroup** method once for any body included in the group.

The method

cdet_collided_faces::**Reset**()

resets all the **Group** and **ExcludeGroup** call results. The method is used if it is required to reconfigure the groups and the exceptions for the previously defined cdet_collided_faces query structure.

R.4.4. Configuring a Collision Detection Query

If the cdet_collided_faces structure configuration options are insufficient, then C3D geometry core user can create a custom query. To do this, you need to define the cdet_query class inheritor and implement for it a special call-back function of the following type:
typedef cback_res (***cback_func**)(cdet_query *, message, cback_data &).

Below you can find C++ example code for cdet_query_result structure that inherits the cdet_query query structure. The example works until the first intersection is found and it determines whether there is at least one intersection in the control body set. You can find the example in cdet_data.h file.

```
//------------------------------------------------------------------------------
struct cdet_query_result: public cdet_query
{
 cdet_result result;

 cdet_query_result()
   : cdet_query( QueryFunc )
   , result( CDET_RESULT_NoIntersection )
  {}

private:
 static cback_res QueryFunc( cdet_query * query, message code, cback_data & )
  {
   cdet_query_result * q = static_cast<cdet_query_result*>( query );
   switch( code )
   {
   case CDET_QUERY_STARTED: // The collision query is executed for all solid bodies of the set
     {
      q->result = CDET_RESULT_NoIntersection;
      return CBACK_VOID_RESULT;
     }
   case CDET_INTERSECTED: // The first intersection is found.
     {
      q->result = CDET_RESULT_Intersected;
      return CBACK_SUFFICIENT;
     }
   case CDET_FINISHED: // A pair of solids is finished.
     return (q->result == CDET_RESULT_Intersected) ? CBACK_BREAK:CBACK_VOID_RESULT;

   default:
     return CBACK_VOID;
   }
  }
};
```

The query is based on the call-back function
static cback_res **QueryFunc**(cdet_query * **query**, message *code*, cback_data & **cData**),
the user can implement this function in any way he/she likes. This function is passed to the **CheckCollisions** algorithm via the cdet_query structure and it can be called by one of the following events (enum cdet_query::message):

CDET_QUERY_STARTED (The collision detection started in the control body group),
CDET_STARTED (The collision detection started for object pair),
CDET_FINISHED (The collision detection for object pair was ended),
CDET_INTERSECTED (The intersection of faces for a pair of objects was found),
CDET_TOUCHED (The contacting faces for a pair of objects was detected).

The **QueryFunc** function has the following input parameters:

- **query** is a pointer to the query structure,
- *code* is an integer encoding the detection event,
- **cData** is a data pass structure.

The function returns one of the cback_res enumeration values:

CBACK_SUFFICIENT (Client application response to the CDET_INTERSECTED or CDET_TOUCHED events;
the search is interrupted for this pair of bodies, but is continued for other pairs in the control body set),

CBACK_SKIP (Skipping the detection for this body based on CDET_STARTED event: the detection for other pairs in the set will be continued),
CBACK_BREAK (Interrupts further intersection detection for all events),
CBACK_VOID (Does not impact the collision detection query and continues the detection),
this enumeration value is passed to the C3D core by the client application to configure detection algorithm.

code is an integer that defines the detection event, it takes one of the following values:
CDET_QUERY_STARTED,
CDET_STARTED,
CDET_FINISHED,
CDET_INTERSECTED,
CDET_TOUCHED.

The **cData** data structure is used to pass the cdet_query::geom_element, it should have the following form:

```
struct cback_data
{
 geom_element first, second;
};
```

where an element of cdet_query::geom_element pair is a detection feature structure, it contains the following components:

- geom_element::**appItem** is a geometric object in the client application,
- geom_element::**refItem** is a MbRefItem type geometric object, usually it is MbFace face,
- geom_element::**wMatrix** is a matrix that defines the conversion from the local CS of the body to the general global CS.

Various implementations of the **QueryFunc** call-back method permit to use the following detection process configuration methods:

- Collect all face pairs that take part in collisions of the bodies;
- Remove certain bodies or pairs of bodies from collision analysis at any moment;
- Distinguish between face tangent case and face intersection case;
- Interrupt the detection in case of a certain result;
- Interrupt the detection if tangency or intersection is found;
- Group bodies to avoid detection of intersections within the group.

R.4.5. Grouping Bodies Included in the Control Set

Collision control often requires to detect collisions between body groups instead of collisions for all pairs in the control body set. For example, you can improve performance by skipping the intersection tests if it is known that the bodies in the group do not intersect, or if a detection in the group is not required due to some reason. Two methods are used to form such groups.

The first method is to use the same *appItem* value in the **AddItem** method when the bodies are added in the group**.** This method is useful when the bodies belonging to one "monolithic" object on the client application side are naturally grouped.

The second method is to configure cdet_query to skip the intersection test for pairs of bodies belonging to the same group. An example of this method is included in the cdet_collided_faces class source code. To do this, you need to return CBACK_SKIP for the CDET_STARTED event in the call-back method for a pair of bodies belonging to the same group.

S.1. TWO-DIMENSIONAL GEOMETRIC SOLVER

From the geometric point of view, any drawing can be presented as a set of plane objects, i.e. points, lines, segments, arcs, ellipses, and spline curves. However, not every arbitrary set of flat geometric objects is a drawing. In any drawing, geometric objects always depend on interrelations that determine the positions of some objects in respect to other ones. Such interrelations are always assumed to exist, no matter whether we draw a sketch by a pencil on paper or simulate it on the computer. As for computer-aided drawing, such interrelations are set by *geometric constraints,* that include *logical constraints,* such as coincidence, parallelism/perpendicularity, tangency, horizontality, verticality, symmetry, and *dimensional constraints* that set various *linear and angular dimensions*, as well as curve length or radius. A sketch/drawing that has at least one dimensional constraint is called a *parametric* one.

The 2D geometric solver permits to calculate the positions of geometric objects meeting all the given constraints and sketch dimensions. If the positions of objects during drawing conform to given constraints and dimensions, then this process is called *parametric drawing*.

S.1.1. Assignment of GCE Geometric Solver

The 2D solver included in C3D geometry core is called GCE. This is an internal name given for technical purposes, it is an *abbreviation of Geometric Constraint Engine*. GCE is a component of C3D Solver module that has another component, namely three-dimensional mating solver (GCM). The main purpose of the GCE component is to satisfy the system of constraints for geometric objects of flat drawing (sketch). The following flat geometric objects can be a subject of GCE calculations: a point, line, circle, ellipse, spline or parametric curve. Users can add links to them in the form of constraints selected from a given set of types, including the following logic constraints: parallelism, perpendicularity, tangency, point on curve, symmetry, and also various types of dimensional constraints that define the distances, lengths and angles. The solver can also work with curves that have ends (bounded curves), for example, line segments and circular arcs.

All functions of the GCE geometric solver are available via the gce_types.h and gce_api.h header files, they contain a set of simple data structures and functions. The solver type system reflects the main concepts required to define geometric constrain problem for a drawing and manipulate the states of geometric objects. The problem domain of the geometric solver includes such concepts, as geometric solver, geometric constraint, dimension, constraint system, numeric variable, etc. The application interacts with the constraint solver based on simple C++ data structures, all these structures are declared in gce_types.h file.

S.1.2. Embedding in an Application

The GCE geometric solver is designed to be a general-purpose parametric drawing component. It means that it can be embedded into any application that deals with planar geometry, drawings or sketches. We assume that such application already has its data structures that represent the sketch geometry. In fact, the developer who embeds GCE component in his application should provide to the solver control over his geometric objects and a due system of constraints. Solver API has its own abstract data type system that provides the minimal functionality required to define constraint satisfaction problem. Therefore, it is required to implement a special wrapper to provide sketch control to the geometric solver. The wrapper would be a bridge between the C3D Solver and the application, it would pass geometry and constraints data into the solver and also apply solver computation results to the sketch geometry. Fig. S.1.2.1 shows a sample flow chart that describes the interaction of the constraint solver with a sketch. **GCE wrapper** serves sketch parametric model and performs the following functions:

- Loads sketch geometric model expressed as C3D Solver data types into the solver;
- Processes editor requests, e.g., adds or deletes constraint system data, calculates the sketch and drags geometric objects;
- Controls the state of sketch geometry with applied constraints;
- Vice versa, updates solver data based on the actual state of sketch geometry;
- Hides (adapts) solver API using its mathematic data types and provides more convenient API with

application data types.

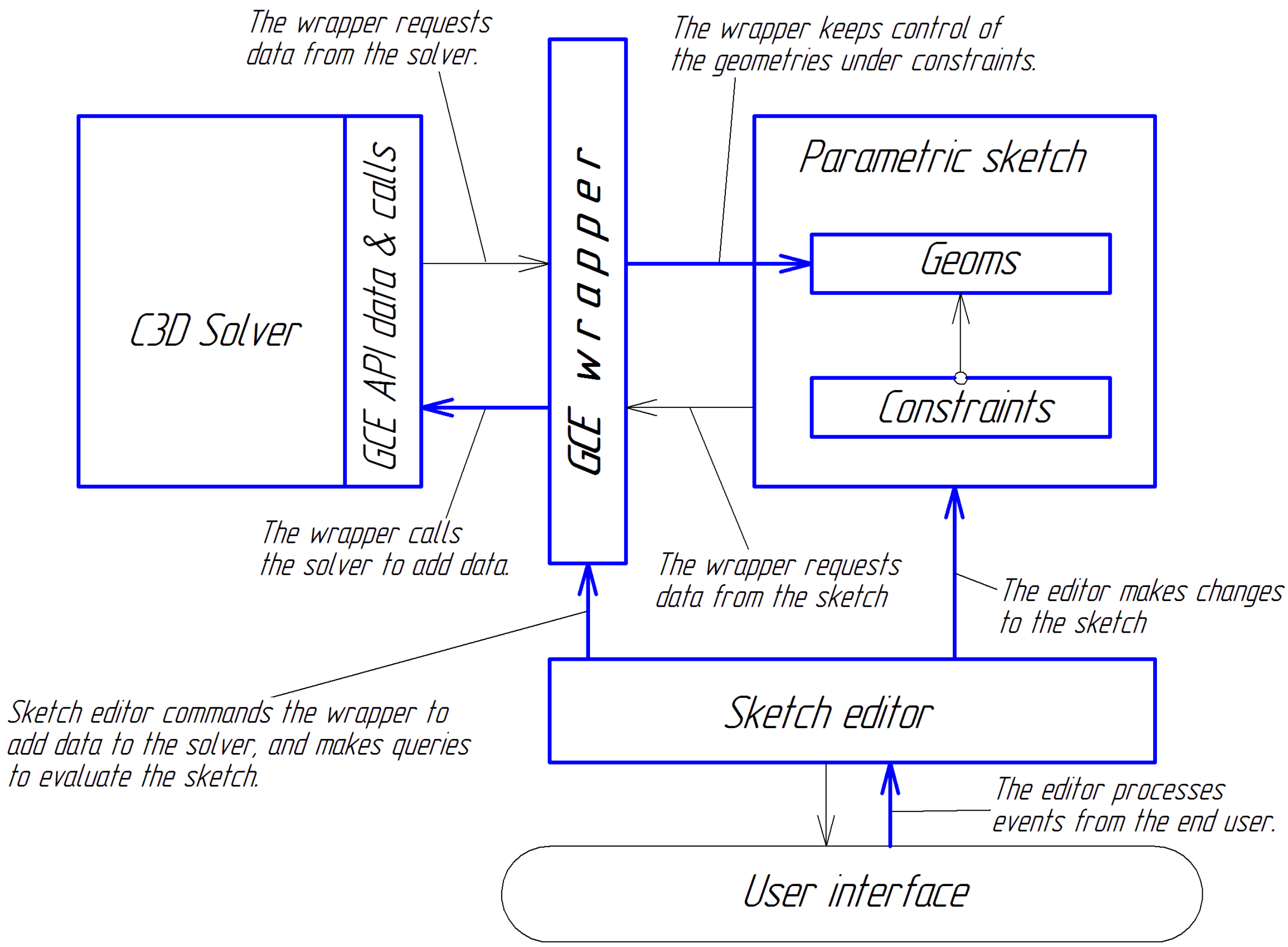


Fig. S.1.2.1.

The flow chart on Fig. S.1.2.1 does not show the only solver embedding option, it just shows an example of a possible way to integrate it in an application with adaptation of GCE mathematic data structures to application data types.

The GCE component does not permit to save user data in a document file, so application developer should implement read/write sketch constraint system in the application document. Furthermore, one should take into account that the set of geometric solver types is not able to cover all diversity of application geometric types. For example, the solver does not have a rectangle type, but it may be represented as four points and lines with some constraints. A possible software implementation is based on creating its method for any drawing object that describes it in the solver.

S.1.3. Supported Geometry Types

The GCE geometric solver supports a basic set of geometry types sufficient to build various sketch drawing objects on its basis. The basic type set includes the following geometry types:

- A point is defined by two Cartesian coordinates x and y;
- A line is defined by a point and a normal vector;
- A circle is defined by a center point and a radius;
- An ellipse is defined by a center point, a guiding vector, and two semi-axis values;
- A spline is defined by NURBS data structure;

- A parametric curve is defined by a MbCurve general curve type object.

The solver also permits to create separate parts or arcs based on infinite curves. Bounded curve type and its special case line segment were designed for this purpose.

- A bounded curve is defined by one base curve and two end points;
- A line segment is defined by a pair of points.

S.1.4. Types of Geometric Constraints and Dimensions

Any geometric constraint links geometric objects that are called constraint *arguments*. It is possible to use as a constraint argument any object listed as a geometry type in Item S.1.3. Supported Geometry Types . For instance, a circle can be **tangency** constraint argument. Constraints are classified as unary, binary and ternary based on the number of geometry arguments. They link one, two, or three objects correspondingly. **Symmetry** is an example of a ternary constraint. It includes three objects: two points and a symmetry axis. In general case, a constraint may have N arguments. *Logical constraints* assume dependencies between geometric objects only. *Dimensional constraints* establish a link between geometric objects and numeric parameter that defines a specific linear or angular dimension. Therefore dimensional constraints always have a last numeric argument, that is also called a *scalar*.

The GCE solver supports the constraint types specified in Table S.1.4.1.

Table S.1.4.1. Constraint types

Logical constraints:	**Dimensional constraints**:
fixing	segment length
horizontality	radius
verticality	distance
segment length	distance with a direction
coincidence	angle;
point on curve	
alignment of points	
collinearity	**Special constraints**:
equal length	driving parameter
equality of radii	fix spline derivative
parallel	point on curve with a position based on curve parameter
perpendicularity	
tangency	
middle point	
mirror symmetry	Linear equation of the following form:
bisector	$a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + \ldots + a_n \cdot v_n + c = 0$

Each of these constraints will be further described below. Logical constraints are described in Chapter S.3. TWO-DIMENSIONAL LOGICAL CONSTRAINTS. Dimensional constraints and the issues of adding constraints to the sketch are described in Chapter S.2. TWO-DIMENSIONAL DIMENSIONS**.**

S.1.5. Basic Data Types of GCE Solver API

The application interacts with the constraint solver based on simple C++ data structures, all these structures are declared in gce_types.h file. The key role is played by descriptor data types that identify any objects controlled by the solver (see Table S.1.5.1.).

Table S.1.5.1. Descriptor data types

Solver data type	Implementation	Interpretation
GCE_system	void *	constraint system descriptor
geom_item	size_t	geometric object descriptor
constraint_item	size_t	constraint descriptor
var_item	size_t	numeric variable descriptor

Data type Data types with finite enumerations of possible values are listed in Table S.1.5.2.

Table S.1.5.2. Enumeration data types

Solver data type	Interpretation
geom_type	geometric object type
constraint_type	constraint type
point_type	type of requested control point
coord_name	geometric object coordinate name
GCE_result	function diagnostic code
GCE_bisec_variant	selection between bisector solutions

Data structures listed below (Table S.1.5.3.) are used to pass object parameter set. For example, GCE_ellipse structure passes ellipse data in the solver: center coordinates, semi-axis dimensions, and a guiding vector.

Table S.1.5.3. Data structures

Solver data type	Interpretation
GCE_vec2d	2D vector coordinates
GCE_point	Coordinates of a point on a surface
GCE_point_dof	point freedom degree record
GCE_line	line coordinates
GCE_circle	circle coordinates
GCE_ellipse	ellipse coordinates
GCE_spline	spline coordinates and characteristics
GCE_dim_pars	dimensional constraint parameters
GCE_adim_pars	angular dimensional constraint parameters
GCE_ldim_pars	linear dimensional constraint parameters

GCE_dragging_point	dragging control point

S.1.6. Geometric Constraint System

Geometric constraint system is a set of geometric objects and constraints that link these objects. The types of supported geometric objects and constraints are described in items S.1.3. Supported Geometry Types ,S.1.4. Types of Geometric Constraints and Dimensions correspondingly. It is assumed that parametric sketch created by the application with all its objects and constraints has its own constraint system. From software engineering viewpoint it means that each parametric sketch is associated with the constraint system using GCE_system descriptor, and each its geometric object and constraint are represented by their unique type descriptors geom_item and constraint_item.

Before proceeding to work with sketch geometric constraints it is required to declare a constraint system for it by calling

GCE_system **GCE_CreateSystem**()

The function returns the constraint system as GCE_system descriptor, it is a pointer to internal data structure instance in the geometric solver. All further manipulations with the constraint system will be performed using this descriptor. For example, if you want to declare a point in the sketch, then it is required to call the following function for its constrain system:

geom_item **GCE_AddPoint**(GCE_system **gSys**, GCE_point **pVal**)

The function returns a descriptor of the geometric point that belongs to the **gSys** geometric constraint system**.** The **pVal** parameter defines the start coordinates of this point <X,Y>.

When you finish to work with the sketch, you should always call the function that deletes the constraint system:

void **GCE_RemoveSystem**(GCE_system **gSys**)

This function completely releases the memory occupied by the constraint system with all object and constraint data. After calling the **GCE_RemoveSystem** function, the constraint system descriptor becomes invalid, i.e. if you try to use the descriptor then the application can crash.

The function

void **GCE_ClearSystem**(GCE_system **gSys**)

clears the constraint system, it deletes objects and constraints only from the memory, but it keeps the constraint system valid for further work.

S.1.7. Representation of Geometric Objects

Geometric constraint solver works with a certain geometric object representation form shown in Fig.S.1.7.1. All objects are expressed using point, vector and number coordinates (scalars).

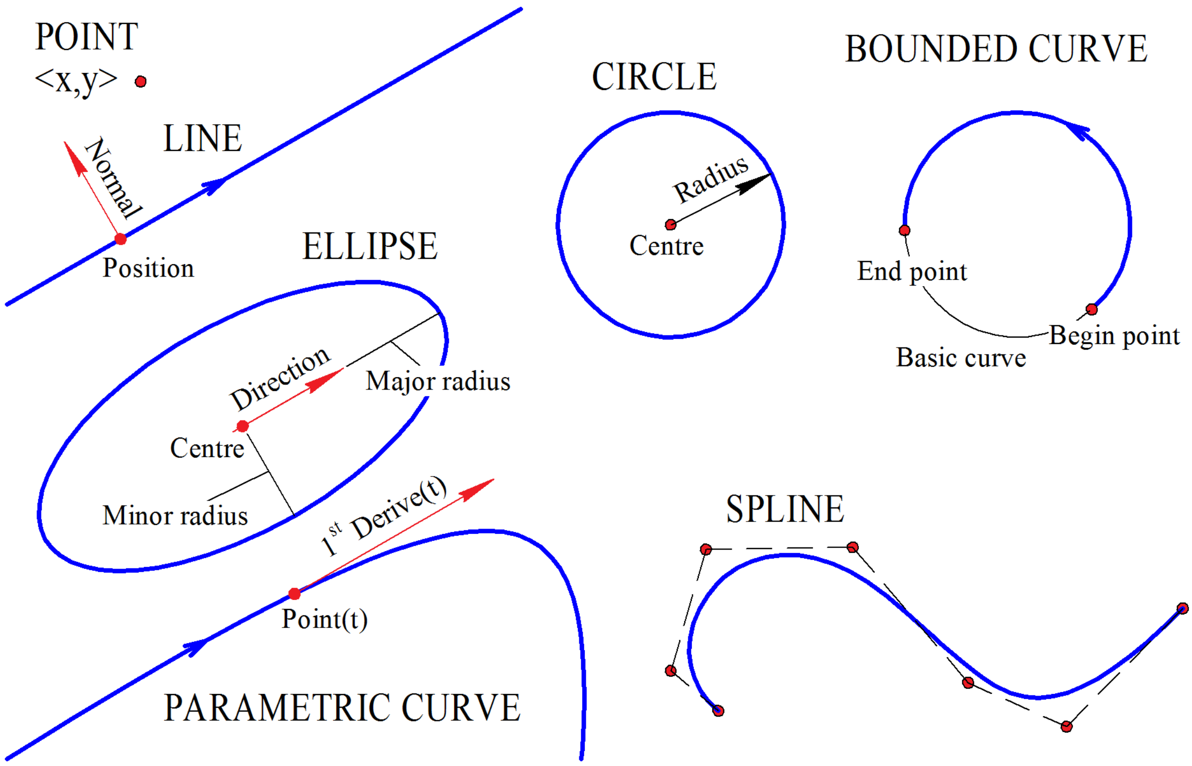


Fig. S.1.7.1.

The application can have its own representation of geometric objects that differs from solver representation. However, passing object status data in the solver and passing the calculation results back are based on the fact that each geometry type has its representation:

- A **point** is represented by a pair of Cartesian coordinates <X,Y>.
- A **line** is defined by its position point and a normal vector. It is assumed that a curve has a guiding vector equal to the normal vector rotated 90 degrees clockwise. In other words, the normal with <Y,–X> coordinates will correspond to the guiding vector with <X,Y> coordinates.
- A **circle** is defined by its center point and a radius. At this moment, radius can be a positive non-zero number only.
- An **ellipse** is defined by its central point, radii along the major and minor semi-axes and the guiding vector of the main semi-axis. Ellipse parameters are also defined by a periodic parameter ranging from 0 to 2π running the ellipse counterclockwise along its starting point at the main semi-axis.
- A **parametric curve** is passed to the solver as MbCurve class. Such curve is considered to be fixed, and the calculations associated with this curve are based on the following virtual functions: MbCurve::PointOn, MbCurve::FirstDer and MbCurve::SecondDer, that return, correspondingly, the first point on the curve based on the parameter, the first or the second derivatives in the point. Parametric curves are described in more detail in Item S.5.2. General Parametric Curves

Note. In the current version, you should create MbCurve instance in order to create a parametric curve in the constraint system. However, an alternate variant based on simple user-implemented functions will be implemented in future releases. C3D core user can also implement custom inheritors for the MbCurve class.

- A **spline** uses NURBS representation based on a list of control points. Work with splines is described in more detail in Item S.5.2. General Parametric Curves.
- A **bounded curve** is a curve portion limited by end points on both sides. It is defined by three elements: a base curve, curve portion start point and curve portion end point.

S.1.8. Degree of Freedom

Every geometry type has a degree of freedom equal to the minimum number of coordinates required to determine the state of the geometric object. For example, the degree of freedom is 2 for a 2D point. For a circle the degree of freedom is 3, as it is completely defined by three parameters <X,Y,R>, namely, center coordinates and radius. According to Item S.1.7. Representation of Geometric Objects., a line is represented by a position point and a normal vector. This presentation is convenient, but it is redundant; minimum sufficient line presentation can be a pair of values, such as offset value and slope angle, so for a line the degree of freedom is 2. For an ellipse, the degree of freedom is 5. For a spline, the degree of freedom is the sum of degrees of freedom of its control points. A parametric curve is completely determined on the application side, i.e. its degree of freedom is zero. Table S.1.8.1. lists the degrees of freedom for all types supported by the solver.

Table S.1.8.1. Degrees of freedom for geometric objects

Geometry type	Degree of freedom
Point	2
Line	2
Circle	3
Ellipse	5
Spline	2 · Number of control points
Parametric curve	0
Bounded curve	2 + Degree of freedom of the base curve

Every geometric object included in the system adds a number of its degrees of freedom to the overall degree of freedom for the sketch. From the other side, every added constraint takes away one or more degrees of freedom. To define the state of all sketch geometric objects, it is required to add some number of constraints that take away all degrees of freedom for the object.

Most constraints take away one degree of freedom. These are constraints like parallelism, perpendicularity, horizontality/verticality, equality of radii, equality of lengths, a point on curve, tangency, and most dimensional constraints. Other constraints take away two degrees of freedom: middle point, collinearity, symmetry, and bisector.

It can be said that the task of parametric drawing is to completely determine geometric objects in the sketch. The number of geometric constraints required to completely determine the sketch is the sum of degrees of freedom of all the sketch objects.

S.1.9. Add and Delete Geometric Objects

The geometric solver mainly works with geometric objects, so to start to create constraint system, the application declares the geometric objects that will become constraint arguments in the constraint system.

Every geometric object declared in the constraint system will have its unique identifier, namely, geom_item descriptor. Its geometric type (geom_type) is not changed during the entire object life. API C3D Solver calls that add geometric objects to the system are described below.

A **point** is added using

geom_item **GCE_AddPoint**(GCE_system **gSys**, GCE_point **pVal**) method.

The function returns a descriptor of the geometric point that belongs to the **gSys** geometric constraint system**.** The **pVal** parameter sets the start values of <X,Y> point coordinates.

A **line** is added using

geom_item **GCE_AddLine**(GCE_system **gSys**, GCE_line **lVal**) method.

The function returns a descriptor of the line on a plane that belongs to the **gSys** geometric constraint system**.** The **lVal** sets the starting values of position **lVal.p** and a normal to the line **lVal.norm**.

A **circle** is added using

geom_item **GCE_AddCircle**(GCE_system **gSys**, GCE_circle **cVal**) method.

The function returns a descriptor of the circle that belongs to the **gSys** geometric constraint system. The **cVal** parameter defines the starting values of the circle center **cVal.centre** and its radius **cVal.radius**.

An **ellipse** is added using

geom_item **GCE_AddEllipse**(GCE_system **gSys**, GCE_ellipse **eVal**) method.

The function returns a descriptor of the ellipse that belongs to the **gSys** geometric constraint system. The **eVal** parameter sets the ellipse start parameters:

<**eVal**.centre, **eVal**.direct, **eVal**.majorR, **eVal**.minorR>, that define correspondingly the center, guiding vector, and radii along the main and the minor semi-axes.

A **spline** is added using

geom_item **GCE_AddSpline**(GCE_system **gSys**, GCE_spline **spl**) method.

The function returns a descriptor of the spline that belongs to the **gSys** constraint system**.** The **spl** data structure determines the original state of the spline. Parts of spline curves and methods used to define them are reviewed in Item S.5.1. Spline Curves

A parametric curve of general form is added using

geom_item **GCE_AddParametricCurve**(GCE_system **gSys**, const MbCurve & **crv**) method.

The function returns a descriptor of the parametric curve added to the **gSys** constraint system. The **crv** object completely determines abstract mathematical description of the curve in parametric form. 2D curve class is described in more detail in Item O.3.1. MbCurve Two-Dimensional Curve**.** It should be noted that the MbCurve data type is included in the C3D geometric core class hierarchy, so its lifetime is determined by the reference counter. In other words, the constraint system guarantees the validity of MbCurve instance until the parametric curve is deleted from the system. Work with parametric curve is described in more detail in Item S.5.2. General Parametric Curves.

A **bounded curve** is added using

geom_item **GCE_AddBoundedCurve**(GCE_system **gSys, geom_item curve, geom_item p**[2]) method.

The function returns a descriptor of the bounded curve that belongs to the **gSys** constraint system. The created curve is based on the **curve** with **p**[0] and **p**[1] end points as boundaries. In fact, this method links three objects (a base curve and two points) into a single object, and automatically creates two "point on curve" constraints for the end points. A code fragment given below shows an example how to create a **circular arc**.

```
GCE_system gSys = GCE_CreateSystem();
GCE_circle circPars; // Circle values
GCE_point endP1, endP2; // Coordinates of the end points.
// …
// The code that assigns start values of the circle and its end points should be inserted here
// …
geom_item p[2] = { GCE_AddPoint(gSys, endP1), GCE_AddPoint(gSys, endP2) };
geom_item circItem = GCE_AddCircle( gSys, circPars ); // Creating a circle. It's a base curve of the arc.
geom_item arc = GCE_AddBoundedCurve( gSys, circItem, p ); // Creating the circular arc.
// …
GCE_RemoveSystem( gSys );
```

Likewise, you can call **GCE_AddBoundedCurve** method to create elliptical arcs, parametric curve portions, line segments, etc. A bounded curve created based on a straight line is called a **segment**, and all the constraints applicable to a straight line are also applicable to a segment.

A geometric object added to the constraint system by any of the above methods and free of any constraints, can be removed by calling

bool **GCE_RemoveGeom**(GCE_system **gSys**, **geom_item** **g**) method.

This method deletes the geometric object from the system and makes its descriptor **g** invalid. The function will return **true** if successful. The function will return **false** and the deletion will not be performed if the geometric object at the moment of the call is an argument of one of the constraints. To find out if a geometric object is available for deletion by **GCE_RemoveGeom**, you can use the **GCE_IsConstrainedGeom** function. Also, it is allowed to delete the geometric object on which other objects are based before others. For example, one of endpoints or base curve may be removed before the **bounded curve** based on them, but it will be inevitably deleted after the deletion of bounded curve or the last of the geometric objects associated with it.

S.1.10. Fixing and Freezing Geometric Objects

Any geometric object created in the constraint system is initially free, i.e. it has all the degrees of freedom inherent to particular object type. During calculations, the C3D Solver may change the state of geometric objects when it is needed to satisfy the constraints. Sometimes it is necessary to fix a part of geometric objects so that the solver would leave the position of the geometric object unchanged. For this purpose, the GCE API can call methods that can **freeze** or **fix** the geometric object. Only the application can change the state of frozen or fixed geometric objects by calling

GCE_SetPointXY and **GCE_SetCoordValue** method.

It is useful to freeze geometric objects when it is required to fix some part of the drawing. For example, in CAD systems a part of drawing geometry might be created by projecting a 3D model, and therefore it will be permanently linked with it by a unilateral associative link: 3D model → flat projection.

A geometric object can be frozen using

bool **GCE_FreezeGeom**(GCE_system **gSys**, geom_item **g**) method.

The method will return **true**, if the geometric object **g** became actually frozen. It should be noted that freezing is not considered a constraint, so the value returned by the **GCE_IsConstrainedGeom** method does not depend on whether the object was frozen. The degree of freedom of a frozen object is zero.

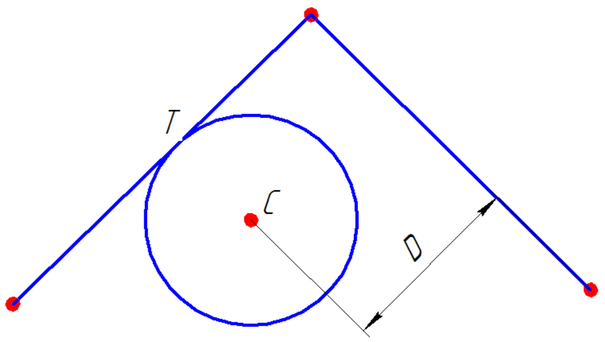
An alternative method used to fix a geometric object for the solver is to set object fixing constraint using

constraint_item **GCE_FixGeom**(GCE_system **gSys**, geom_item **g**) method.

The function fixes the object **g** belonging to the **gSys** constraint system and returns the descriptor of newly added "object fix" constraint. Unlike **GCE_FreezeGeom**, this method creates a constraint that can be deleted at any moment by the **GCE_RemoveConstraint** method used to release the object. The fixing constraint is also described in Item S.3.6. Unary Constraints: Horizontality/Verticality and Fixing Variants.

S.1.11. Geometric Object Control Points

Among the geometric types supported by the solver (see Item S.1.3. Supported Geometry Types), a **point** plays a special role. First of all, a point is the most elementary geometric object. Secondly, all other geometry types can be expressed using points that are called *control* points. For example, a circle is expressed by the <C,R> structure, where C is the center point, and R is a scalar that determines the numeric radius value. Thirdly, control points can independently take part in constraints as their arguments. For example, they permit to define constraints both for a circle and for each its separate point, for example, the distance from the circle center to the line (see *Fig. S.1.11.1.*).



C is the center control point
T is the tangency of the circle and the line
D is the distance between the center and the line

Fig. S.1.11.1.

In order to work with control point as an independent point, you need to request its descriptor from the geometric object using

geom_item **GCE_PointOf**(GCE_system **gSys**, geom_item **g**, point_type **pnt**) function.

The function returns a descriptor of the control point of the geometric object **g** that belongs to the **gSys** constraint system**.** The **pnt** parameter determines what control point of the object is requested and it takes one of the following values:

- GCE_FIRST_END is the starting point of the curve (the first end);
- GCE_SECOND_END is the end point of the curve (the second end);
- GCE_CENTRE is the center of circle, ellipse, or its arc (bounded curve);
- GCE_Q1 is an ellipse quadrantal point (3 hours);
- GCE_Q2 is an ellipse quadrantal point (12 hours);
- GCE_Q3 is an ellipse quadrantal point (6 hours);
- GCE_Q4 is an ellipse quadrantal point (9 hours).

Control points of spline curve can be requested using

geom_item **GCE_SplinePoint**(GCE_system **gSys**, geom_item **spl**, *size_t* **pntIdx**) method.

The function returns a descriptor of the **spl** spline control point with the **pntIdx** number, that takes values from 0 to N–1, where N is the number of the spline control points. Fig. S.1.11.2. shows a spline that has N = 7 control points (red circles) numbered starting from 0.

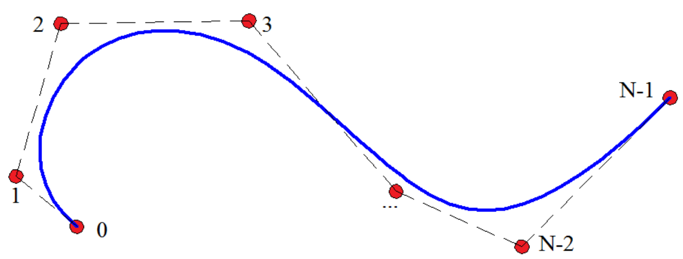


Fig. S.1.11.2.

S.1.12. Scalar Variables

The GCE module is mostly used to calculate 2D plane geometric objects, while constraints and dimensions are applied to them. These data types inherently have vector representation. Parametric drawing assumes that geometry is linked with driving or variational numeric parameters. So it is impossible to do without another type, a **numeric variable** or a **scalar**. Their values are usually expressed using distance or angle measurement units.

Variables of the following type can only be used in a parametric constraint system:

- A numeric variable associated with a dimension, it is a **dimensional variable**. When resolved, a dimensional variable is equal to measured dimension value. Dimensional variable can be both an independent driving parameter of a sketch, or variational/measurement parameter. The use of variables for creating driving and variational dimensions is discussed in Item S.2.2. Driving and Variational Dimensions.
- An auxiliary geometric constraint parameter, for example, a parameter that specifies where tangency points are for complex curves. You can find more details in **Auxiliary tangency parameter** Section, Item S.3.10. Tangency.
- A variable that is not associated with a geometric constraint or a dimension may be used in scalar linear or non-linear equations that establish relations between first values and other values.

In GCE module API, a variable is represented by the var_item data type. You can create variables by calling

var_item **GCE_AddVariable**(GCE_system **gSys**, double **val**) method.

The function returns a descriptor of a scalar variable with the initial value **val**. The variable with this descriptor can be further used to determine dimensions, constraints, and equations.

Any variable can be **dependent** or **independent.** A dependent variable is governed by measurement dimensions or constraints, and its value is calculated by the solver. An independent variable can't be modified by the solver, and its value is defined by the application. To make a variable independent, just call

constraint_item **GCE_FixVariable**(GCE_system **gSys**, var_item **var**) API method,

that will create a driving parameter constraint and return the constraint descriptor that fixes its value. An independent variable can be varied by calling **GCE_ChangeDrivingDimension** method. Driving parameter constraint together with other types of driving dimensions permits to create a parametric sketch; its geometry can be changed according to a user defined law by changing independent parameters. Creation of driving parameters and dimensions is described in Item S.2.2. Driving and Variational Dimensions.

S.1.13. Linear Equation

A linear equation is expressed by formula $\boldsymbol{a}_1 \cdot \boldsymbol{v}_1 + \boldsymbol{a}_2 \cdot \boldsymbol{v}_2 + \boldsymbol{a}_3 \cdot \boldsymbol{v}_3 + \ldots + \boldsymbol{a}_n \cdot \boldsymbol{v}_n + c = 0$, where $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3, \ldots, \boldsymbol{a}_n$, *and* $\boldsymbol{c}$ are constant coefficients, and $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \ldots, \boldsymbol{v}_n$ are scalar equation variables. You can create a linear equation by calling

constraint_item **GCE_AddLinearEquation**(GCE_system **gSys**, const double * **a**,
const var_item * **v**, *size_t* **n**, double **c**) method,

that returns a descriptor of the linear equation registered in the **gSys** system.

Initial method call data are given below:

- **gSys** is a constraint system;
- **a** is an array of constant coefficients for variables;
- **v** is an array of linear equation variables;
- **n** is a number of linear equation variables;
- **c** is a constant coefficient c that is not multiplied by any variable.

S.1.14. API Call Journalling

C3D Solver permits to record the entire API call log, including the original parameters and the parameters returned by called functions for any specific GCE_system constraint system. The result will be a log file that can be later used by C3D geometric core developers to debug and test C3D Solver functions.

Log structure. The log file is a text file, it uses simple S-expression syntax that permits to record any nested and list-based data structures. Any API function called for a specific constraint system is recorded in the log file in the following form:

(*GCE_Func* (*arg list*) (*returned value*)),

where *GCE_Func* is the name of the called GCE function, (*arg list*) is a list of function arguments, and (*returned value*) is the returned value.

For example, this is how a log of a call that creates a circle with a center in the coordinate system origin and radius 2.5 looks like:

(GCE_AddCircle ((0.0 0.0) 2.5) #1)

The circle has received symbolic identifier #1, which is the value returned by the GCE_AddCircle function.

The log is actually a session record for a specific constraint system written in a language that can be interpreted by special tools available to C3D Solver developers. You can manually edit or simplify a log, if not all calls are required to achieve a specific result.

Enabling logging mode. In order to enable the constraint system logging, you need to call

bool **GCE_SetJournal**(GCE_system **gSys**, const char * **fName**) function,

that will assign the **fName** log file to the **gSys** constraint system. If logging mode was successfully initialized, then the function returns **true**.

The method input parameters are:

- **gSys** is an empty constraint system created by the **GCE_CreateSystem** method,
- **fName** is a string with a complete file path. Example: ”..\\Journals\\sample.jrn”

If logging is enabled, then data about all API C3D Solver calls to the **gSys** constraint system are recorded.

Warning! The log file will be ready only when a session of work with the constraint system is finished, i.e. immediately after calling the **GCE_RemoveSystem** method.

Code example. An example of code that demonstrates how constraint system session data are recorded in the log is shown below.

```
GCE_system gSys = GCE_CreateSystem();
// Switch on journalling for the constraint system.
GCE_SetJournal( gSys, "C:\\Logs\\gce_sample.jrn" );
// Make some calls of C3D Solver …
GCE_point p1Val, p2Val; // Coordinates of two points = (1,1) and (2,2)
p1Val.x = p1Val.y = 1.0;
p2Val.x = p2Val.y = 2.0;
geom_item pnt[2] = { GCE_AddPoint(gSys, p1Val),GCE_AddPoint(gSys, p2Val) };
GCE_AddCoincidence( gSys, pnt );
// ...
// ...
GCE_Evaluate( gSys );
// Finalize the constraint system
GCE_RemoveSystem( gSys );
    // The journal file is ready!
```

Note. Logging is required only for debugging, so it is recommended to enable it only in debug mode. It is not recommended to leave logging enabled for application release version or production version, as it can result in extra time and memory overheads.

Debugging and testing. Logging permit geometric core developers to debug C3D Solver functions by recreating the call log created when the application worked. This method is based on the fact that the internal state of a constraint system is completely defined by API call sequence, therefore it is possible to repeat the same call sequence with the same data to fully recreate a problem arising when the application calls the solver.

An extensive collection of logs forms a test case base that is an essential tool for C3D Solver module development and quality control. Each C3D Solver revision is checked on a test base of logs to guarantee that none of previous patches and upgrades was lost.

Therefore, in order to receive consulting or resolve a problem related to embedding C3D Solver in an application, you need to enable logging mode for a while, execute the required sequence of actions, and send the resulting file with necessary comments or questions to C3D Labs technical support. When the error is fixed, the corresponding log is added in test case base.

S.2. TWO-DIMENSIONAL DIMENSIONS

Dimensions are constraints that link geometric objects with a numeric parameter (dimension value). Each dimension measures a specific numeric geometric property, for example, the distance between points, the angle between lines, or the circle radius. Logical constraints and dimensions differ as the former have no numeric arguments. Each dimension always has one numeric argument (scalar) besides geometric arguments. Driving sketch geometry by using parameters and dimensions is discussed in Item S.2.2. Driving and Variational Dimensions

S.2.1. Auxiliary Points of Distance Dimension

You can't uniquely define distance dimensions and also diameter dimensions for some object types without additional indication of the points used to measure the dimension. For example, a distance between a circle and a line segment can be measured using two methods: from the "farthest" circle point to the segment or from the "closest" point to the segment. See Fig. *S.2.1.1.* The terms "farthest" and "closest" are quoted because the situation can become different when the dimension is changed: then the "farthest" point will become closer to the segment than the "closest" point.

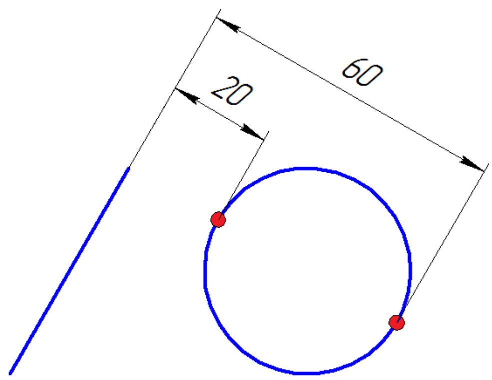


Fig. S.2.1.1.

To uniquely define the dimensions, you should call the **GCE_AddPoint** method to create an auxiliary point, in which the dimension extension line from the circle side will start. The descriptor of this point will be used as an auxiliary point when the distance dimension would be created. In the example shown in Fig. S.2.1.1., only one point on the circle side is required. In other cases, such as distance from circle to circle, two auxiliary points are needed, because there are four possible dimension measurement variants for a pair of circles. See Fig. S.2.1.2. Dimensions equal to 30, 50, 60 and 90 show variants of measuring distances between two circles, and, correspondingly, variants of selecting auxiliary point pairs on the circles.

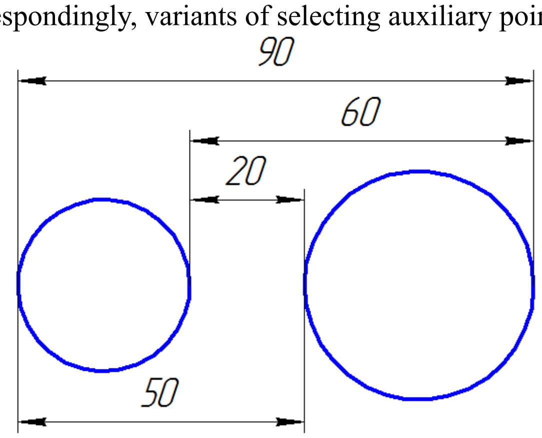


Fig. S.2.1.2.

S.2.2. Driving and Variational Dimensions

Dimensions differ from logical constraints (parallelism, perpendicularity, etc.) in that they are linked with a numeric value called *dimension value*. A dimension value can be set by two methods:

- **Driving dimension.** Dimension value is set by an independent user-defined numeric parameter that can be changed only from outside via solver API (by calling the **GCE_ChangeDrivingDimension** method). In this case the dimension does not depend on the geometry: on the contrary, the geometry depends on the dimension, as well as on the geometric constraint. The dimensions defined this way are required to control the geometry using dimensional parameters and they are called **driving** dimensions.
- **Variational dimension.** Dimension value is defined by a numeric variable that is calculated by the solver as well as the geometry. In fact, the current value will be associated with a variable specially created by calling the **GCE_AddVariable** method, and it can be reused in other dimensions or equations. The value of dimension variable can be changed by the geometry. This representation is used when it is required to measure the dimension rather than to set it. Dimensions linked with variables are called **variational** dimensions.

Sometimes it is required to create a *variational* dimension that behaves as a *driving* dimension, especially when you need to control two or more dimensions linked with one numeric parameter simultaneously. In this case, you can create one or more variational dimensions that use one variable that can be made a *driving parameter* by calling:

constraint_item **GCE_FixVariable**(GCE_system **gSys**, var_item **var**),

where **var** is the dimension variable descriptor.

The function returns the *"driving parameter"* constraint that fixes the current value of the **var** variable. After calling **GCE_FixVariable**, the variable virtually becomes an independent parameter of the constraint system and it can be changed only from the outside by calling the **GCE_ChangeDrivingDimension** method applied to the descriptor of the received constraint.

Driving dimensions and parameters can be varied using the GCE_result **GCE_ChangeDrivingDimension**(GCE_system **gSys**, constraint_item **dItem**, double **dVal**) method.

The method input parameters are:

- **gSys** is a parametric sketch constraint system;
- **dItem** is a descriptor of driving dimension or driving parameter;
- **dVal** is the required value of the driving dimension or the driving parameter;

If the operation is executed successfully, then the function would return the GCE_RESULT_OK code. If the function would return an error code (not GCE_RESULT_OK), then it can be caused by one of the following reasons:

- Invalid system descriptors or **gSys** and **dItem** constraints.
- The received constraint is not a driving dimension or driving parameter, for example, if the constraint is a variational dimension.

You should take into account that this function does not calculate anything, but just prepares the change of the driving dimension or parameter. In order to apply the change, please call the **GCE_Evaluate** calculation function. If it is required to change two and more driving dimensions simultaneously, then first make a series of preparatory **GCE_ChangeDrivingDimension** calls for each driving parameter or dimension, and then calculate the new positions of geometric elements determined by the new state of driving dimensions or parameters by calling **GCE_Evaluate** once.

S.2.3. Zero Dimensions and Signed Dimensions

All types of distance dimensions have a continuous domain of definition that includes zero, negative and positive values. Fig.S.2.3.1 shows a sketch with a dimension that positions the geometry in both positive and negative value domains. In the left image, the sketch dimension is negative, and the image on the right shows the same sketch with dimension sign changed from negative to positive.

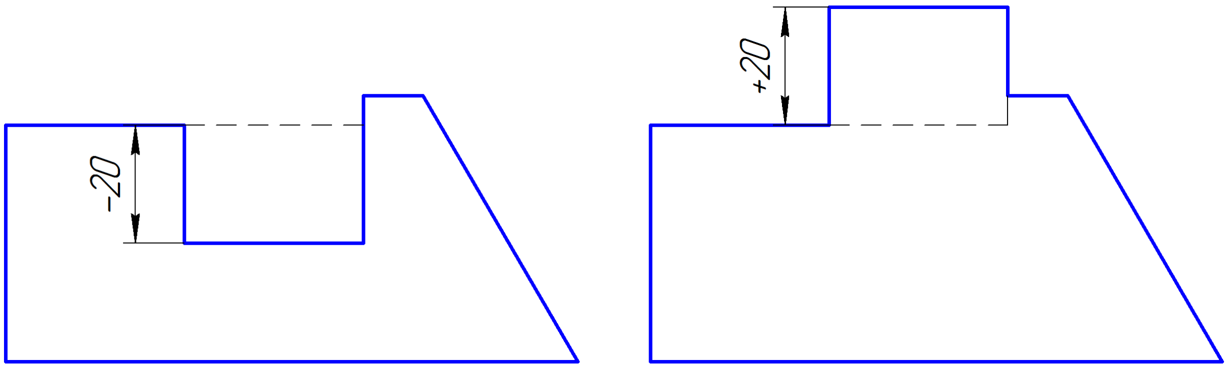


Fig. S.2.3.1.

Any type of distance and angular dimensions can have zero value. On the same Fig.S.2.3.1*,* a dashed line shows the geometry in a zero state, when the dimension value is equal to zero.

Alternating signs are most typical for dimensions set for an object oriented in space, for example, a line with a normal vectors. The line divides space into two semi-planes, so the distance dimension between a line and a point would have various signs depending on the side of the line where the point is located. A point lying on a line would correspond to zero dimension.

It is worth noting that the distance between the points is intrinsically non-negative; but a distance dimension set for two points can be negative or equal to zero. It has a practical sense when you work with a parametric drawing. For example, the dimension with alternating signs shown in the left and right sketch in Fig. S.2.3.1*,* can be defined as "distance from point to line" or "point-to-point distance" constraint. In both cases, its behavior would be the same.

S.2.4. Distance Dimension

Distance dimension is set for a pair of geometric objects. Table S.2.4.1 shows geometry type pairs supported for distance dimensions by the C3D Solver.

Table S.2.4.1. Geometric object pairs supported for distance dimensions

	Point	Line	Circle	Ellipse	Spline	Parametric curve
Point	√	√	√			
Line	√	√	√			
Circle	√	√	√			
Ellipse						
Spline						
Parametric curve						

Regardless of curve type associated with the distance dimension, it measures the distance between points lying on the curves. These are usually the points that provide the smallest distance between the objects. However, it is also possible to choose other point pairs to measure the dimension. *Fig. S.2.1.2.* in Item .2.1. Auxiliary Points of Distance Dimension. shows four examples of various distance dimensions for the same pair of circles.

Distance dimension for a couple of geometric objects is set by the

constraint_item **GCE_AddDistance**(GCE_system **gSys**, geom_item **g**[2], const GCE_ldim_pars & **dPars**) method.

The function returns a distance dimension descriptor set for the objects **g**[0] and **g**[1] belonging to the **gSys** constraint system.

The **dPars** parameter structure defines the following distance dimension settings:

- **dPars.dPars** sets the parameter value, it can be either an independent parameter or a variable;
- **dPars.hp**[2] is a pair of auxiliary points that set the measurement result when it is required (learn more in Item .2.1. Auxiliary Points of Distance Dimension).

The **dPars.dPars** parameter actually defines the source, where the dimension value is taken. If **dPars.dPars.var** = GCE_NULL_V, then the dimension value is determined by the independent numeric parameter **dPars.dPars.dimValue**, which makes this dimension a driving dimension. If **dPars.dPars.var** contains variable descriptor, then the dimension is variational. Driving and variational dimensions were discussed in more detail in Item S.2.2. Driving and Variational Dimensions.

A pair of auxiliary point descriptors **dPars.hp**[2] is not mandatory, if the dimension measurement variant is unique (for example, the distance between a point and a line) or if it can be selected by the solver that would choose the measurement variant closest to the current object position and the dimension value. In this case, **dPars.hp**[2] is by default equal to a pair of zero descriptors {GCE_NULL, GCE_NULL}. If one or both auxiliary points **dPars.hp**[0] and **dPars.hp**[1] are defined, then the coordinates of these points would specify the measurement option for the distance dimension. Measurement options are described in more detail in Item .2.1. Auxiliary Points of Distance Dimension.

S.2.5. Directed Distance Dimension

A simple distance dimension measures a distance between points lying on geometric objects. A directed distance dimension is also defined for a pair of points, but the distance is measured between the projections of points along the line having a particular direction. This dimension type is mostly used to create vertical and horizontal dimensions that define the distance between points projected on X or Y axis. Fig. S.2.5.1 shows the examples of vertical and horizontal dimensions for a point pair. Generally, the dimension direction can be arbitrary.

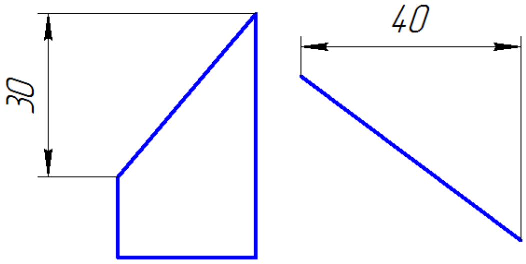


Fig. S.2.5.1.

Directed dimensions also behave in a specific way during a geometric transformation of the sketch by the GCE_Transform method. Directed dimension would rotate if the transformation matrix contains a rotation component.

Directed dimensions are created by the

constraint_item **GCE_AddDirectedDistance**(GCE_system **gSys**, geom_item **p**[2],
const GCE_ldim_pars & **dPars**) method.

The function returns the descriptor of directed dimension.

The method input parameters are:

- **gSys** is a parametric sketch constraint system;
- **p**[2] are the descriptors of point pair;
- **dPars** is a data structure, that besides standard settings inherent to distance dimensions (see similar settings for GCE_AddDistance), additionally sets the **dPars.dirAngle** angular direction. For instance, if it is required to define a *horizontal* dimension, then **dPars.dirAngle** = $\boldsymbol{0}$; if it is required to define a *vertical* dimension, then **dPars.dirAngle** = $\boldsymbol{\pi/2}$. Generally, **dPars.dirAngle** can range from $\boldsymbol{0}$ to $\boldsymbol{2\pi}$.

Note. In the current C3D Solver version, the directed dimension can be set only for points.

S.2.6. Distance From a Point to a Segment

Distance from a point to a segment is a distance dimension that includes three points, where two points define a line segment, and from it the distance to the third point is measured. This dimension is created by the

constraint_item **GCE_AddDistancePLs**(GCE_system **gSys**, geom_item **p**[3],
const GCE_dim_pars & **dPars**) method.

The specific feature of this constraint is that the segment length should be non-zero. It means that the distance dimension would stop working if the two points defining the segment would coincide. The dimension can be applied when it is required to define the distance dimension only based on points, without any linear object. However, when it is not needed, it is recommended to use the **GCE_AddDistance** function, which would be sufficient to create almost all variants of distance dimensions.

S.2.7. Angular Dimensions

An angular dimension is defined for two geometric objects that have vector guiding lines. The angular dimension value is specified in radians, its values range from $\boldsymbol{0}$ to $\boldsymbol{2\pi}$ and it is defined by an angle between the guiding vectors of the first and second objects measured counterclockwise. An angular dimension measured this way is shown in Fig. S.2.7.1. as two intersecting lines with given directions. The right part of Fig. S.2.7.2. shows a pair of lines that have the same angle between vectors, but with the dimension measures a reverse angle sector. The value of such dimension is determined by formula $\boldsymbol{D = 2\pi - \alpha}$, where $\boldsymbol{\alpha}$ is the measured angle between the vectors, and $\boldsymbol{D}$ ***is*** the dimension value**.** This dimension is also called a conjugate dimension. Another way to receive it is just to swap angular dimension arguments.

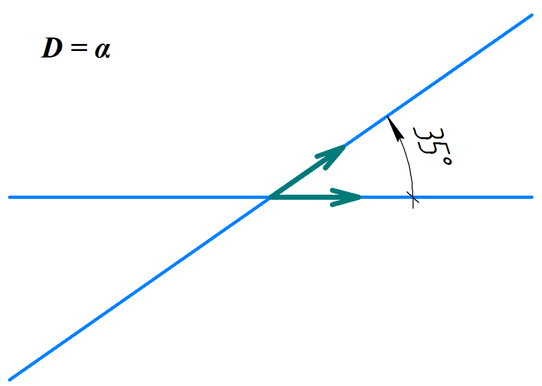


Fig. S.2.7.1.

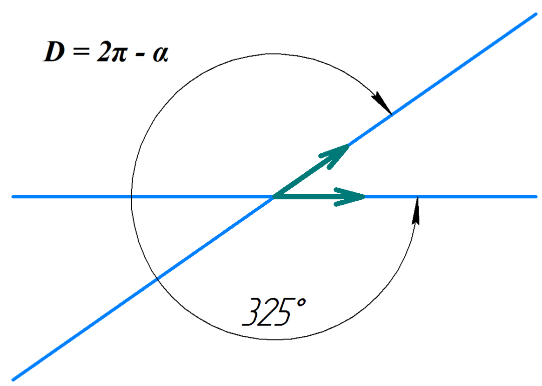
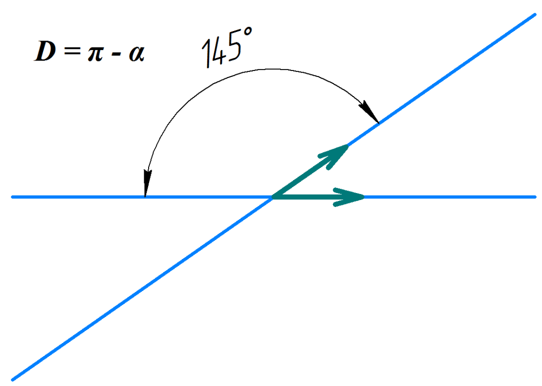


Fig. S.2.7.2.

There are four ways to define an angular dimension for one pair of directions, including the two ways described above. Fig. S.2.7.3. shows a dimension, its value is determined by adjacent angle formula: $\boldsymbol{D = \pi - \alpha}$. Fig. S.2.7.4. (right part) shows an alternate adjacent angle version that is defined by the formula $\boldsymbol{D = \pi + \alpha}$**.**



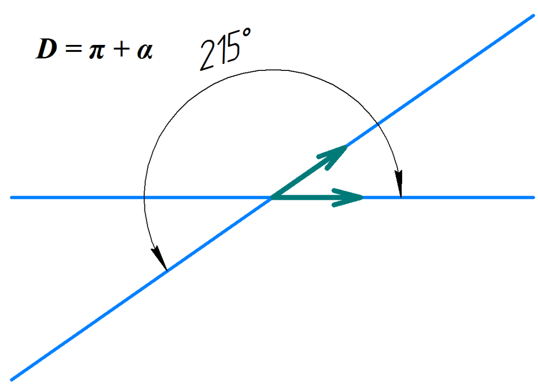


Fig. S.2.7.3. *Fig. S.2.7.4.*

One more pattern associated with four ways to set the angular dimension is that the dimension with the value $\boldsymbol{D = 2\pi - \alpha}$ can be produced from the dimension with the value $\boldsymbol{D = \alpha}$ after a simple swap of the first and second dimensional constraint objects. Similarly, the 4th variant ($\boldsymbol{D = \pi + \alpha}$) can be received from the 3rd one ($\boldsymbol{D = \pi - \alpha}$) by swapping the arguments.

Angular dimension for a pair of directed objects, such as segments, lines, or ellipses, is defined by the constraint_item **GCE_AddAngle**(GCE_system **gSys**,
geom_item **l1**, geom_item **l2**,
const GCE_adim_pars & **dPars**) method.

The function creates a new angular dimension defined for the **l1** and **l2** geometric objects in the **gSys** constraint system. The dimension value and the way to define it are determined by the **dPars** structure.

The function returns the descriptor of a new angular dimension.

The **dPars** structure determines the value of an angular dimension defined using the **dPars.dPars** structure (**GCE_dim_par**), where the dimension value is stored in **dPars.dPars.var** variable or if **dPars.dPars.var** = **GCE_NULL_V** then the dimension value is defined by **dPars.dPars.dimValue** independent numeric parameter, that makes it a driving dimension. Driving and variational dimensions are further described in Item S.2.2. Driving and Variational Dimensions.

If **dPars.adjacent** = **false**, then the dimension value would be defined by the formula $D=f\cdot\alpha$. The $\boldsymbol{f}$ factor in this formula is defined by the **dPars.factor** parameter, and it permits to set the angular dimension in other units. For example, if **dPars.factor** = **180/$\boldsymbol{\pi}$**, then the application can set the angular dimension in degrees. To set the angular dimension with value $D=f\cdot(2\pi-\alpha)$, it is required to swap **l1** and **l2**.

To define the angular dimensions with adjacent angle variant shown in Fig. S.2.7.3. and Fig. S.2.7.4., it is required to set the flag **dPars.adjacent** = **true**. In this case, the dimension value will be defined by formula $D=f\cdot(\pi-\alpha)$ or $D=f\cdot(\pi+\alpha)$. The second formula is recieved by swapping the arguments **l1** and **l2**. The $\boldsymbol{f}$ factor is also defined by the **dPars.factor** parameter.

S.2.8. Angular Dimension Based on 3 or 4 Points

C3D Solver provides one more method to determine an angular dimension, in this case the guiding vectors are defined by point pairs. The same method permits to set the angular dimension based on three points that determine the angle of the measuring triangle.

The angular dimension based on three or four points is received using the
constraint_item **GCE_AddAngle4P**(GCE_system **gSys**, geom_item **fPair**[2],
geom_item **sPair**[2], const GCE_adim_pars & **dPars**) method.

The point pairs defined by the **fPair** and **sPair** descriptors determine the angular dimension measurement vectors. The vector from the point **fPair**[0] to the point **fPair**[1] determines the first angle side, and the vector from **sPair**[0] to **sPair**[1] determines the second angle side. *Fig. 2.8.1.* shows the measurement angles of the dimension based on three points (at the left) and based on four points (at the right). In order to define a dimension based on three points, it is required to set the angle vertex point twice, in the first pair and the second pair, for example, **fPair**[0]=**sPair**[0].

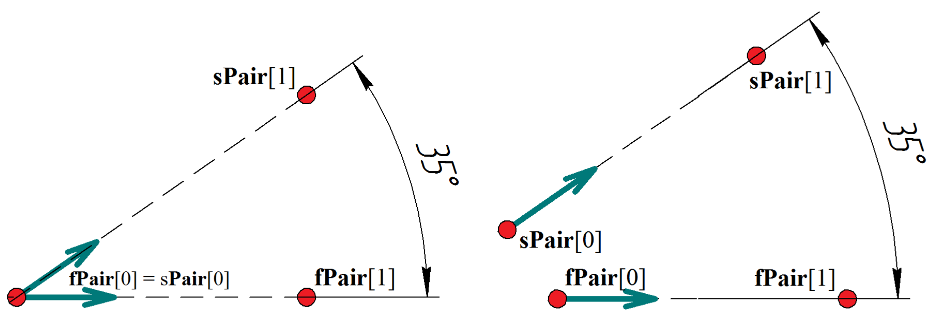


Fig. S.2.8.1

Other parameters of the **GCE_AddAngle4P** function are set in the same way as for the **GCE_AddAngle** function.

S.2.9. Radial and Diameter Dimensions

These types of dimensions are available for circles and arcs, they correspondingly specify circle radius or diameter that is twice the radius. As well as distance and angular dimensions, radial and diameter dimensions can be associated with an independent numeric parameter or a scalar variable, i.e. that can be *driving* or *variational*, this can be set by the GCE_dim_pars structure.

Radial dimensions are created using the following method:

constraint_item **GCE_AddRadiusDimension**(GCE_system **gSys,**
geom_item **cir,**
GCE_dim_pars **dPar**).

A diameter dimension is created by the following method:

constraint_item **GCE_AddDiameter**(GCE_system **gSys,**
geom_item **cir,**
GCE_dim_pars **dPar**).

Both methods have the same initial data:

- **gSys** is the geometric constraint system, where the new dimension is registered;
- **cir** is circle descriptor or circle arc descriptor;
- **dPars** is the structure that determines the dimension value in length units.

The dimension value is stored in the **dPars.var** variable, or, if **dPars.var = GCE_NULL_V** then the parameter value is determined by **dPars.dimValue** independent numeric parameter, that makes it a driving dimension. Driving and variational dimensions are described in more detail in Item S.2.2. Driving and Variational Dimensions.

S.3. TWO-DIMENSIONAL LOGICAL CONSTRAINTS

Logical constraints, unlike dimensions, do not have numeric parameters and they are determined for geometric objects only. C3D Solver permits to set the following logical constraints for geometric objects: horizontality/verticality, coincidence, parallelism/perpendicularity, tangency, equality of radii, etc.

S.3.1. Coincidence of a Point and Other Object

If two geometric objects have the same type, then coincidence constraint presumes equality of two geometric objects. However, in the current version of C3D Solver this constraint is valid only if one of the arguments is a point.

If one of the coincidence arguments is a point, and the other one is a curve, then the constraint would define the point lying on the curve.

The coincidence constraint is defined by the following method:

constraint_item **GCE_AddCoincidence**(GCE_system **gSys**, geom_item **g**[2]).

The function would add to **gSys** system the coincidence constraint for two geometric objects (**g**[0] and **g**[1]) and would return the constraint descriptor.

S.3.2. Alignment of Points

This type of constraint is mostly used to horizontally or vertically align point pairs. In general, this constraint would align points on an imaginary line with a given angular direction.

Point alignment is set up by calling the method

constraint_item **GCE_AddAlignPoints**(GCE_system **gSys**, geom_item **p**[2], double **ang**).

The function returns a descriptor of a new constraint registered in the **gSys** system.

p[0] and **p**[1] pair parameters set the descriptors of the aligned points, and the **ang** parameter is set in radians and it determines the direction, along this direction the points would be aligned. For example, the **ang**=***0*** angular direction would correspond to horizontal alignment of points, and if **ang**= ***π/2***, then the points would be aligned vertically.

S.3.3. Parallelism/Perpendicularity

Parallelism and perpendicularity constraints are set for pairs of lines or segments using the following methods:

constraint_item **GCE_AddParallel**(GCE_system **gSys**, geom_item **g**[2]),
constraint_item **GCE_AddPerpendicular**(GCE_system **gSys**, geom_item **g**[2]),

Methods accept the following initial data: **gSys** is the geometric constraint system, **g**[2] is a pair of straight-line curves. The methods return a descriptor of a geometric constraint registered in the **gSys** system.

S.3.4. Collinearity

Collinearity means that the geometric objects included in this constraint lie on the same line. Collinearity can be defined both for two geometrical objects, and for three points.

Collinearity constraint is created for a pair of geometric objects by calling the

constraint_item **GCE_AddColinear**(GCE_system **gSys**, geom_item **g**[2]) method,

it requires that one of the constraint arguments should be a straight-line, and the other one should be either a line or a point. The geometric arguments are set by the **g**[2] pair. The function returns a descriptor of a new constraint registered in the **gSys** system.

Collinearity constraint can be created for three points by calling the

constraint_item **GCE_AddColinear3Points**(GCE_system **gSys**, geom_item **p**[3]) method.

The descriptors of three points that should be placed on one line are passed via the **p**[3] parameter. The function returns a descriptor of a new constraint registered in the **gSys** system.

S.3.5. Equality of Lengths and Radii

Both functions that create constraints for equality of lengths and equality of radii accept the same data types: a constraint system and a pair of geometric objects.

The geometric constraint that sets the equality of two bounded curve lengths is created by the

constraint_item **GCE_AddEqualLength**(GCE_system **gSys**, geom_item **g1**, geom_item **g2**) method.

In the current version of C3D Solver, this method works for a pair of segments only. The constraint arguments are defined by **g1** and **g2** segment descriptors. The function returns a descriptor of a new constraint registered in the **gSys** system.

The equality of radii of two circles or arcs is defined by the

constraint_item **GCE_AddEqualRadius**(GCE_system **gSys**, geom_item **c1**, geom_item **c2**) method,

it accepts correspondingly the **gSys** constraint system, **c1** first circle or arc descriptor, and **c2** second circle or arc descriptor. The function would return a descriptor of a new constraint registered in the **gSys** system.

S.3.6. Unary Constraints: Horizontality/Verticality and Fixing Variants

The following constraints: horizontality, verticality, and object/line/direction fixing are applied to a single geometric object, i.e. they are *unary* constraints. In order to create one of these constraints, please call the following function:

constraint_item **GCE_AddUnaryConstraint**(GCE_system **gSys**, constraint_type **cType**, geom_item **geom**).

The function adds a unary constraint for the **geom** geometric object to the **gSys** system**.** The **cType** parameter defines the enumeration type of the unary constraint and it can take one of the values listed in Table S.3.6.1.

Table S.3.6.1. Unary constraint types

Constraint type	Description
GCE_FIX_GEOM	**Fixing a geometric object** makes this object immovable. The solver cannot change the state of this object during calculations. A fixed geometric object can be changed only if the application calls GCE_SetPointXY or

	GCE_SetCoordValue.
GCE_HORIZONTAL GCE_VERTICAL	**Horizontality** and **varticality** constraints are applicable to line and line segment types. These constraints align a linear object with sketch coordinate system: horizontality for the OX axis, and verticality for the OY axis, correspondingly. Please note that these constraints permit to flip the line guiding vector by 180 degrees.
GCE_ANGLE_OX	This constraint fixes the angular position of a line or a segment. The GCE_ANGLE_OX constraint is described in greater detail in Item S.3.9. Angular Position Constraint.
GCE_LENGTH	This constraint fixes the line segment length. It is described in more detail in Item S.3.7. Length Fixing.
GCE_RADIUS_DIM	This constraint is described in greater detail in Item S.3.8. Radius Fixing. For the GCE_UnaryConstraint call, this constraint type works only for a circle. If you need to apply it to an ellipse, please call the GCE_FixRadius method.

S.3.7. Length Fixing

It is suggested that this constraint fixes the length of a curve that has a start and an end (a spline or a bounded curve), but it is applicable to line segments only in the current version of C3D Solver. Length fixing is a dimensional constraint, so it permits to change the length using the GCE_ChangeDrivingDimension method.

The length is fixed by the

constraint_item **GCE_FixLength**(GCE_system **gSys**, geom_item **ls**) method.

S.3.8. Radius Fixing

Radius fixing constraint is applicable to circles and ellipses. C3D Solver is unable to change circle or ellipse radius if radius fixing constraint is set for the object. The application can work with this constraint as with a driving dimension that permits to change the radius using the **GCE_ChangeDrivingDimension** method. As for circles, this constraint is similar to a driving radial dimension that was described in Item S.2.9. Radial and Diameter Dimensions.

A radius can be fixed by the following method:

constraint_item **GCE_FixRadius**(GCE_system **gSys**, geom_item **g**, coord_name **cName** = GCE_RADIUS).

Initial method call data are given below:

- **gSys** is the geometric constraint system;
- **g** is a circle/ellipse descriptor;
- **cName** is the radius type that takes the following values: GCE_RADIUS is the default value for a circle; GCE_MAJOR_RADIUS or GCE_MINOR_RADIUS mean fixing the radius at the major or minor ellipse axis.

The method returns the constraint descriptor that fixes circle/ellipse radius.

S.3.9. Angular Position Constraint

Angular position constraint is applicable to line and line segment types. In fact, this is a dimensional constraint that creates a driving dimension fixing the angular position of a linear object in its initial state relative to the OX axis in general sketch coordinate system. The angular position ranges from 0 to 2π. In order to fix a direction, call **GCE_AddUnaryConstraint** with GCE_ANGLE_OX constraint type. Unlike GCE_HORIZONTAL and GCE_VERTICAL, this constraint type does not permit to flip the line guiding vector by 180 degrees. As this is a dimensional constraint, the line angular position can be changed by calling GCE_ChangeDrivingDimension.

S.3.10. Tangency

Tangency is a geometric constraint that positions a pair of curves so that they would be tangent in one point. Table S.2.15.1 shows supported tangency curve pairs.

Table S.2.15.1. Curve pairs supported by tangency constraint

	Line	Circle	Ellipse	Spline	Parametric curve
Line		√	√	√	√
Circle	√	√	√	√	√
Ellipse	√	√		√	
Spline	√	√	√	√	
Parametric curve	√	√			

Selecting the tangency variant. Fig. S.3.10.1 shows two tangency examples for circle-circle and circle-line pairs. Dashed lines indicate the circles with alternate tangency variants. The figure clearly demonstrates that any tangency constraint has two solutions. For instance, circles may be tangent while remaining one outside or inside another; in other example in Fig. S.3.10.1. the right side shows that there are two mutual positioning variants for circle and line tangency: the circle can be "at the left" or "at the right" from the line.

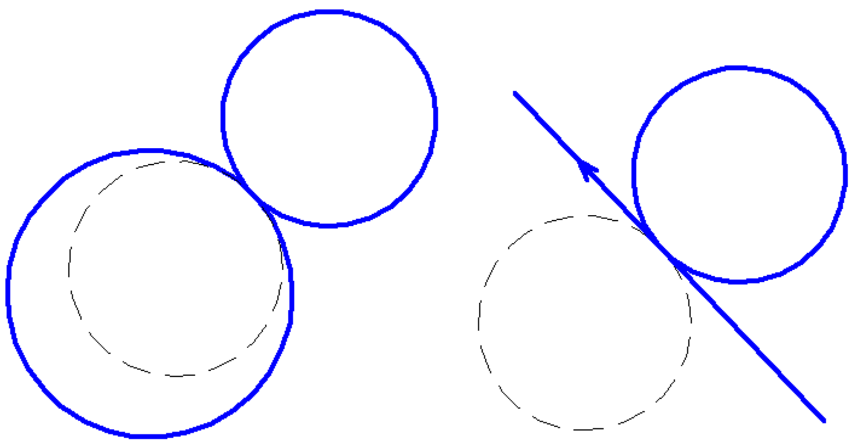


Fig. S.3.10.1.

The GCE geometric solver selects one of two tangency variants based on proximity to the preferred solution. In fact, the tangent objects themselves indicate which of the two mutual positioning variants would be chosen. When the constraint is formed, the solver "remembers" the current mutual position of the objects and saves it when the sketch geometry would be further modified (see Item S.3.15. Mutual Object Positioning).

Auxiliary tangency parameters. The above approach to the tangency variant selection is sufficient if you are dealing with analytical curves (lines, circles, ellipses) only. For a curve with arbitrarily shapes, such as a spline or a parametric curve, it is required to specify the tangency point location along the curve.

Fig.S.3.10.2. shows an example, where the mutual position of a line and a spline is already chosen (the curves are co-directional in the tangency point). However, this variant also has its variants with other possible tangency point locations. The t1 and t2 variables taken in parametric spline domain indicate two alternate tangency points for a spline and a segment with one fixed end.

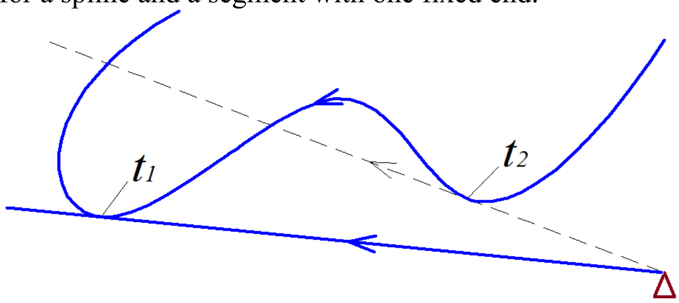


Fig. S.3.10.2.

To specify the locations of the tangency points on splines or parametric curves, it is required to add auxiliary variables specifying suggested tangency point in the parametric domain of the curve. The most common implementation of this process in end-user application is as follows. The user selects the tangency object with a pointer, and pointer coordinates are considered as an approximation of the suggested tangency point. If pointer coordinates are known, it is possible to approximately assess the tangency parameter (the parameter of the closest point on the curve).

Creating a tangency constraint. A tangency constraint is defined by calling the following method:

constraint_item **GCE_AddTangent**(GCE_system **gSys**, geom_item **g**[2]**,** var_item **tPar**[2]).

The method input parameters are:

- **gSys** is the parametric sketch constraint system;
- **g**[2] are the descriptors of two curves;
- **tPar**[2] are the variable descriptors, these are auxiliary tangency parameters for the first curve and the second curve.

The function returns a description of a new tangency constraint created for the given pair of curves. The values of auxiliary parameters **tPar**[0] and **tPar**[1] are taken into account only if the first or second curve correspondingly is a spline or a parametric curve. GCE_NULL_V empty descriptor should be passed for analytical curves (line/circle/ellipse). For example, if both curves are analytical, then you can pass {GCE_NULL_V,GCE_NULL_V} array to **tPar**. If the tangency is defined for a spline or a parametric curve, then an auxiliary variable is created for the constraint by calling the **GCE_AddVariable** method. The initial value of the variable would indicate the suggested tangency point in the parametric curve domain. It should be noted that any variable, as well as geometric objects, is a subject of calculations, so its value would be updated in all calculation requests of the solver (GCE_Evaluate, see Item S.4.1. Calculating the Constraint System), i.e. the solver would actually track the current value of the tangency parameter.

If **tPar**[idx] = GCE_NULL_V for a spline or for parametric curve, then the tangency point location would be determined automatically based on proximity to the initial position of objects.

S.3.11. Multiple and End Tangencies

The GCE geometric solver supports multiple tangencies, i.e. it is possible to set two or more tangency constraints for one pair of curves if these curves include splines or parametric curves. Fig. S.3.16.1 shows a horizontal line and a spline with two set tangency constraints that have various tangency points.

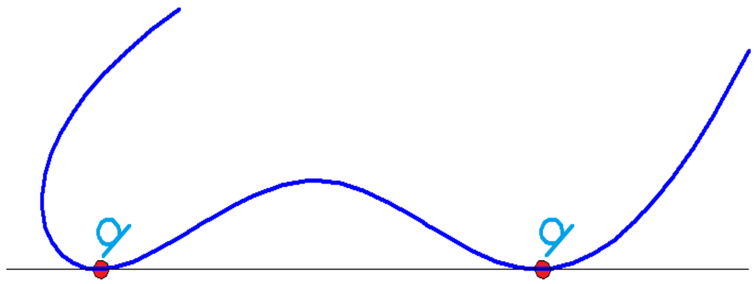


Fig.S.3.11.1.

Please note that in the case, if the tangency pair contains only analytical curves (line, circle, ellipse), it is possible to set only one tangency constraint for the same pair.

End tangency with G1 continuity is a specific example of tangency when a spline is involved, if the tangency point coincides with one of the spline ends. In parametric drawing, the end tangencies create smooth joints of two curves with **G1** type continuity. Fig. S.3.11.2 shows smooth joints of a spline with other curves.

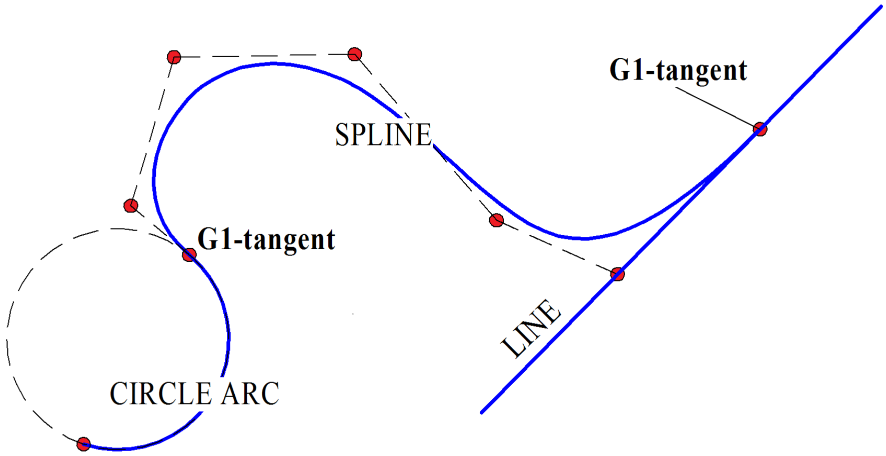


Fig. S.3.11.2.

The **GCE_AddTangent** method considers an end tangency of a non-closed spline, as a special case, and automatically adds to the tangency a condition that spline end point belongs to other tangency curve. Therefore, in general a tangency point can freely move along a spline, but in end tangency case the tangency point is attached to one of the spline ends.

You can create end tangency using one of the following methods:

- Pass spline auxiliary parameter **tPar**[splineIdx] equal to the parameter of spline start or end to the **GCE_AddTangent** method.
- Call the **GCE_AddTangent** method without an auxiliary tangency parameter, i.e. **tPar**[splineIdx]=GCE_NULL_V, but spline tangency endpoint should lie on other tangency curve. This condition can be provided at the moment when the spline is created, or preliminarily provide coincidence of the spline end with other curve by calling **GCE_AddIncidence** and then call **GCE_Evaluate**.

S.3.12. Mirror Symmetry

Mirror symmetry constraint can be set for three geometric objects. The first and second objects are mirror reflections of each other relative to the third object, it is a line that determines the symmetry axis.

A symmetry constraint can be created by the

constraint_item **GCE_AddSymmetry**(GCE_system **gSys**, geom_item **g**[2], geom_item **lObj**) method.

S.3.13. Bisector

Bisector constraint can be applied to three linear objects, if the first two linear objects divide the plane into four sectors, and the third object is an axis that divides the selected sector in half. A bisector constraint can be created by the
constraint_item **GCE_AddAngleBisector**(GCE_system **gSys**,

geom_item **l1**, geom_item **l2**,
, geom_item **bl**,
GCE_bisec_variant **variant**) method.

The method has the following initial data:

- **gSys** is the geometric constraint system;
- **l1, l2** are a pair of straight-line objects that divide the plane into sectors;
- **bl** is a bisector, line or line segment that divides the sector in halves between **l1** and **l2** lines;
- **variant** is one of three variants that determine the sector of the bisector:
  - GCE_BISEC_CLOSEST selects one of the bisector sectors based on the relative position of objects in the initial state (see Item S.4.3. Initial Approximation);
  - GCE_BISEC_PLUS means that the bisector divides a sector indicated by the vector sum of the directions **l1** and **l2**;
  - GCE_BISEC_MINUS means that the bisector divides a sector indicated by a vector difference of the directions **l1** and **l2**;

The function returns the descriptor of a constraint declaring that **bl** is a bisector for a sector formed by the **l1** and **l2** lines.

Please note that this constraint does not lose its meaning if the lines **l1** and **l2** do not intersect. In this case, the bisector divides a part of the surface between the lines **l1** and **l2** in two halves, and it lies on equal distances from these two lines.

S.3.14. Middle Point

Middle point constraint can be applied to three points linked by the fact that the third point lies in the middle of the segment formed by the first and second points. This constraint is created by calling the

constraint_item **GCE_AddMiddlePoint**(GCE_system **gcSys**, geom_item **pnt**[3]) method.

S.3.15. Mutual Object Positioning

The Item S.3.10. Tangency shows an example of tangency of a circle and a line that can have two mutual positioning variants (Fig. S.3.10.1.). C3D Solver remembers the mutual orientation when it is created and strives to maintain it during all further changes. A similar selection can be enabled for linear and angular dimensions. For example, a linear dimension for a point and a line presumes that the point would always stay on the same side of the line (relative to the normal) if the sign remains the same.

Mutual position of objects can be kept for such constraints, as tangency, linear and angular dimensions. Other constraints, such as parallelism, perpendicularity, collinearity, etc., permit to change mutual positions. For example, perpendicularity of two lines may transform a left vector pair into a right vector pair by changing their mutual position. Therefore, the angle between their directions can be either 90 or 270 degrees.

S.4. CALCULATION OF TWO-DIMENSIONAL CONSTRAINTS

S.4.1. Calculating the Constraint System

The main task of C3D Solver is to satisfy the sketch constraint system, it is implemented using the

- GCE_result **GCE_Evaluate**(GCE_system **gSys**) API function, the function returns the **gSys** constraint system calculation
- result code.
- The function does not execute any calculation operations if all constraints are satisfied at the time when it is called.

Returned values:

GCE_RESULT_Ok means that the constraint system is successfully resolved, and the solver assigned a new state meeting all given system constraints to all geometric objects.

- **GCE_RESULT_Overconstrained** means that the function has discovered contradictory constraints during calculation, i.e. the constraint system has a subset, in which the constraints can't be satisfied at the same time. Geometry state remains unchanged.
- **GCE_RESULT_Not_Satisfied** means that the function was not able to find a solution that meets all specified constraints. Geometry state remains unchanged. This diagnostic code is returned in the following situations:
  - A dimension is out of boundaries of the solution existence domain.
  - Positions of fixed or frozen points are out of boundaries of solution existence domain.
  - In rare cases it is caused by poor initial approximation. Meeting the constraints requires considerable changes in parametric coordinates.
  - High mutual interference. For example, for such conditions if an angle is changed by one degree, then the object would offset to a considerable distance.
  - In rare cases, this results from lack of numeric robustness caused by dramatic difference between the dimensions of objects in one drawing, for example, when the smallest and the largest sketch objects differ by more than 7 orders of magnitude (as when one ellipse is 1 mm long and the other one is 10 km long).
  - In general case, it can be caused by other situations when the solution doesn't exist or can't be calculated.
- **GCE_RESULT_InvalidGeometry.** The found solution leads to degradation of geometry objects (for example, a segment/arc has zero length, a circle/ellipse has zero radius). Geometry state remains unchanged.

S.4.2. Changing or Rquesting the Geometry State

When the application works with geometric constraints, it can request the geometry state calculated by the solver, and the solver, in its turn, can pass the state of object coordinates to the application.

The following functions request the geometry state:

```
GCE_point GCE_GetPointXY( GCE_system gSys,
                          geom_item g,
                          point_type pName = GCE_PROPER_POINT );
double GCE_GetCoordValue( GCE_system gSys, geom_item g, coord_name cName );
double GCE_GetVarValue( GCE_system gSys, var_item var )
```

The following functions are used to pass the state of object coordinates from the application to the solver:

```
bool      GCE_SetPointXY( GCE_system gSys,
                          geom_item g,
```

point_type **pName**,
GCE_point **xyVal**);
bool GCE_SetCoordValue(GCE_system **gSys**,
geom_item **g**,
coord_name **cName**, double **crdVal**);
bool GCE_SetVarValue(GCE_system **gSys**, var_item **var**, double **val**);

S.4.3. Initial Approximation

Creating a parametric drawing involves incremental changes, adding or deleting objects and constraints, as well as varying dimensions. Each new request for calculation of constraints system is based on the initial state of geometric objects remaining as a result of the previous **GCE_Evaluate** request.

It should be kept in mind that an initial geometry state is an indispensable preliminary condition required to satisfy the constraints. Two sketches that have the same set of geometric objects and constraints, but various initial states, can have various solutions. C3D Solver takes into account the initial state and it tries to satisfy all the given constraints with minimum deviations from the initial approximation and to change the minimum number of objects.

The initial approximation plays the key role not only when the calculation is requested, but also when the constraints are created. Mutual arrangement of geometric objects shows a constraint definition variant. For example, if one circle is located inside another one, then when a tangency is added for the two circles (**GCE_AddTangent**), then the solver would create such tangency when one circle would always remain inside the other one. Mutual arrangement of geometric objects is described in more detail in Item S.3.15. Mutual Object Positioning

S.4.4. Overdefined Consistent And Inconsistent Constraint Systems

In practice, many sketches are overdefined, as they contain redundant constraints. In the simplest case, the constraints can be duplicated, for example, there can be two tangencies for one pair of circles. In more general cases, an extra constraint is completely satisfied by other constraints that define the same geometric condition. For example, if "perpendicularity" is set between two lines, "horizontality" is set for one line, and "verticality" is set for another line at the same time, then any of these three constraints can be considered as an extra one. C3D Solver excludes extra constraints from calculation. This overdefinition type is related to **consistent** systems that permit extra constraints if they do not contradict each other.

Overdefinition may also cause the constraint system become **inconsistent**, if the redundant constraint contradicts other constraints, i.e. it can't be satisfied together with other constraints. To correct this situation, it is required to delete one of the constraints in the overdefined group. If an extra constraint causes the system to become inconsistent, then the **GCE_Evaluate** method returns the **GCE_RESULT_Overconstrained** error code.

Blocked dimensions. It is recommended to avoid extra constraints even if it does not result in contradictions. In particular, extra constraints may block the controlling dimensions created to add constraints to the sketch. For instance, the left part of Fig. S.4.4.1 shows a parametric rectangle defined by three driving dimensions (they are equal to 40, 40, and 60) and paired verticalities and horizontalities (V and H symbols). In this example, the constraint system is overdefined, as it contains two V constraints, two H constraints, and two dimensions equal to 40. The constraint chain is satisfied, but neither of two driving dimensions equal to 40 can be changed separately without adequate change of the other one. To release the dimension, it is required either to delete one of the dimensions equal to 40 or delete one of horizontality/verticality constraints. The sketch on the right shows the solution that deletes verticality of the upper segment. Then all dimensions can be changed. It should be noted that the dimension equal to 60 is not included into the chain of extra constraints. It means that every overdefinition covers a group of constraints, not always the whole sketch, but rather a part of it. Any dimension in overdefined group becomes blocked. Diagnostics of blocked dimensions is described in Item S.4.10. Redundancy Test.

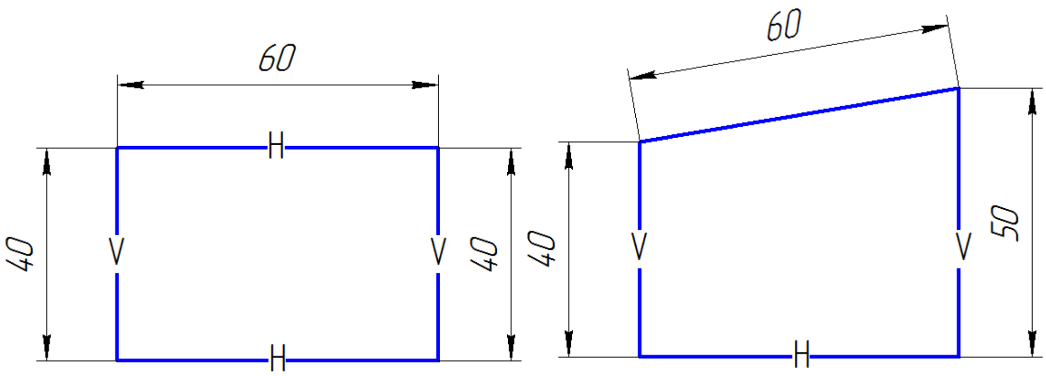


Fig. S.4.4.1.

S.4.5. Underdefined Constraint Systems

The item above (S.4.4.) discussed the overdefined state, when the constraint system has more constraints than needed. This item is dedicated to other situation natural to drawing process: in the sketch there are geometric objects not completely defined by constraints. In the beginning of the drawing process, the sketch may have no constraints at all, and all geometric objects may have full degree of freedom. The more logical constraints and dimensions are added to the sketch, the less degrees of freedom remains for the objects. Completely defined sketch has no degrees of freedom at all. We should note that both situations can coexist in one drawing; some part of the sketch can be overdefined, the rest can be underdefined and at the same time some part of geometry may be completely defined.

Most solution methods available in C3D Solver were designed specifically for underdefined constraint systems. Besides that C3D Solver uses some benefits of underdefined cases to split the whole constraint system into a sequence of subsystems of smaller size calculated one after another. In these or other underdefined cases, the solver selects one of the optimal solutions based on the following principles:

- The number of geometric objects changed to satisfy the constraints should be as low as possible.
- Offset distances of the geometric objects modified by the solution from their initial approximation should be minimal (see Item S.4.3. Initial Approximation).

S.4.6. Degree of Freedom Analysis

In the course of parametric drawing process, it is important to control, which parameters of the sketch geometric objects are completely defined, and which parameters require additional constraints. The Item S.4.5. Underdefined Constraint Systems described the principles used to select the solution for geometric objects that do not have enough constraints to explicitly define their state. Degree of freedom analysis functions show (visualize) underdefined geometric objects to the user. One of them calculates the degree of freedom for a point:

GCE_point_dof **GCE_PointDOF**(GCE_system **gSys**, geom_item **pnt**).

This method permits to define the current degree of freedom for a point that can be a control point of any geometric object, for example, circle/ellipse center, spline control point, one of segment ends, etc.

S.4.7. Information Requests

This item describes a number of functions used by the application to get various information on the objects registered in the constraint system, or diagnostic and calculation results. These functions were designed for information requests, and their call does not change the state of the constraint system or its particular data.

Geometry type. You can request the type of the geometric object registered in the constraint system by calling the

geom_type **GCE_GeomType**(GCE_system **gSys**, geom_item **g**) function,

that returns the enumerative type of the **g** geometric object registered in the **gSys** system. Returned values:

- GCE_POINT is a 2D point;
- GCE_LINE is a 2D line;
- GCE_CIRCLE is a 2D circle;
- GCE_ELLIPSE is a 2D ellipse;
- GCE_SPLINE is a 2D spline;
- GCE_PARAMETRIC_CURVE is a general 2D parametric curve;
- GCE_BOUNDED_CURVE is a curve bounded by end points.

The list of data request functions is given below.

The base curve type is returned by the

geom_type **GCE_BaseCurveType**(GCE_system gSys, geom_item crv) function.

Connectivity with constraints is evaluated by the

bool **GCE_IsConstrainedGeom**(GCE_system gSys, geom_item g) function.

Constraint satisfaction is evaluated by the

bool **GCE_IsSatisfied**(GCE_system gSys, constraint_item cItem) function.

Degenerated geometric objects are returned by the

std::*vector*<geom_item> **GCE_DiagnoseGeometr**y(GCE_system gcSys) function.

Currently set dimension value (not a measured value) is returned by the

double **GCE_DimensionParameter**(GCE_system gSys, constraint_item dItem) function.

Compliance of the control point with the given coordinates is assessed by the

bool **GCE_CheckPointSatisfaction**(GCE_system gSys, geom_item pnt, point_type cp, double px, double py) function.

The current constraint solution state is returned by the

GcConState **GCE_GetConstraintState**(GCE_system, constraint_item gc_item) function.

The state of the constraint system according to the latest solution results is returned by the

GcConstraintStatus **GCE_GetConstraintStatus**(GCE_system gSys) function.

S.4.8. Dragging of Geometric Objects

This item describes the functions that implement interactive underdefined sketch manipulation mode that is called dragging. Initialize dragging mode for an object control point:

GCE_result **GCE_PrepareDraggingPoint**(GCE_system gSys,
GCE_dragging_point drgPnt,
GCE_point curXY).

Initialize control point dragging mode for a set of objects:

GCE_result **GCE_PrepareDraggingPoint**(GCE_system gSys,
const std::*vector*<GCE_dragging_point> & cPntArr,
GCE_point curXY).

Initialize dragging mode for a set of objects:

GCE_result **GCE_PrepareMovingGeoms**(GCE_system gSys,
std::*vector*<geom_item> & geoms,
GCE_point curXY).

Move the dragging point:

GCE_result **GCE_MovePoint**(GCE_system gcSys,
GCE_point curXY).

S.4.9. Geometric Transformation

C3D Solver permits to execute geometric transformation of parametric sketch based on a matrix that can contain shift, rotation, and scaling. In order to execute the specified transformation, call the following method for sketch constraint system:

GCE_result **GCE_Transform**(GCE_system **gSys**, const MbMatrix & **mat**)

Please take into account that this method transforms both the geometry and the constraints. For example, linear dimensions can change their values if the transformation matrix contains scaling.

One should note that due to transformation result a part of constraints can become unsatisfied. You can call the **GCE_IsSatisfied** function to check this. The constraints that remained satisfied are called *invariant* constraints relative to this transformation.

S.4.10. Redundancy Test

C3D Solver provides various ways to evaluate the redundancy of constraints.

The methods

GCE_result **GCE_DeviateDimension**(GCE_system **gSys**, constraint_item **dItem**, double **delta**)

GCE_result **GCE_DeviationTest**(GCE_system **gSys**, constraint_item **dItem**, double **delta**)

evaluate the redundancy of constraints based on dimension constraint deviations.

S.5. TWO-DIMENSIONAL SPLINES AND PARAMETRIC CURVES

This chapter describes work with such curves as splines, general parametric curves and ellipses. The common feature of all these curves is that C3D solver takes into account not only geometry of these curves, but also their parametrization.

S.5.1. Spline Curves

C3D Solver uses NURBS curves as a mathematical basis for spline curves. NURBS presentation has de-facto become an industry standard used to present free-form curves, and it is supported by almost all CAD systems. The NURBS Book by Les Piegl and Wayne Tiller (Springer Publishing Company) is a widely recognized canonical book in this subject domain.

C3D Solver offers two spline definition methods:

- A spline constructed based on a set of characteristic points defined by their coordinate values;
- A spline that passes through a set of interpolation points defined by descriptors of previously registered geometric objects.

A spline can be created from a NURBS curve based on

geom_item **GCE_AddSpline**(GCE_system **gSys**, GCE_spline **spl**) method.

The function accepts GCE_spline data structure that contains data defining the spline: the starting values of control points, the interpolation point array (if required), weights, closedness indicator, etc.

A special constraint that fixes the spline derivative vector is created by

constraint_item **GCE_FixSplineDerivative**(GCE_system gSys, geom_item spline,
double par, uint derOrder, GCE_vec2d * fixVal = NULL) method;

S.5.2. General Parametric Curves

How to use parametric curves. One useful application of parametric curves is the case when the sketch contains fixed curves with unsupported geometry type. For example, a planar sketch can be created in the context of 3D CAD model, and then the sketch can receive 3D curve projections of arbitrary form from it. The projection curves that are included in the constraint system as parametric curves permit to link sketch geometry to projection objects. Parametric curves can also be used to layout drawings with the fragments produced by projecting the geometry from 3D models that keep an associative link with it.

S.5.3. Constraints Based on Parametric Curves

The following methods are used to create special constraints for parametric curves, splines, and ellipses:

constraint_item **GCE_AddPointOnPercent**(GCE_system **gSys**,
geom_item **curve**,
geom_item **pnt**[3],
double **k**),

constraint_item **GCE_AddPointByMetricPercent**(GCE_system **gSys**,
geom_item **curve**,
geom_item **pnt**[3],
double **k**),

constraint_item **GCE_AddFixCurvePoint**(GCE_system **gSys**,
geom_item **curve**,
geom_item **pnt**),

constraint_item **GCE_AddPointOnParEllipse**(GCE_system **gSys**,
geom_item **pnt**,

geom_item **ellipse**, double **t**).

S.6. THREE-DIMENSIONAL GEOMETRIC SOLVER

This section describes another software component of the C3D Solver module — the three-dimensional geometric solver. The three-dimensional geometric solver is applied for geometric modeling when you need to arrange 3D-dimensional objects as a diagram of mutual relations and make them subject to geometric constraints and dimensions. You may need this for the following tasks:

- Merging solid body parts into assemblies;
- Kinematic analysis, including an inverse kinematic task;
- Solid body assembly animation;
- Modeling 3D wireframes.

In combination with the collision detection component (see Chapter R.4. Body collision detecting) the geometric solver helps CAD users assess the motion boundaries of mechanism parts.

S.6.1. Terms and Definitions

Local coordinate system (LCS) — a coordinate system that determines a 3D object position with respect to the *World Coordinate System*. LCS is specified by the **origin point** and three vectors — Z, X, Y axes that are always orthonormal for C3D Solver tasks. The LCS definition is related to Transformation.

Geometric object — the main subject of the C3D Solver calculations. It is a 3D object, such as a point, curve, surface, or LCS. C3D Solver supports a specific set of 3D geometry types listed in section S.6.4. Supported geometry types. Note that solid body abstraction for C3D Solver is presented as a group of geometric objects combined in a Cluster with a common LCS rather than a topological structure (C3D Modeler). See section S.6.12. Geometric scene clustering, assembly modeling.

Standard position — LCS which has the same origin point and axes as *the World Coordinate System*.

Transformation — with respect to C3D solver tasks, transformation of a geometric object using a combination of translations and rotations only. Mathematically, the transformation is presented in the form of a square 4x4 matrix — the Transformation Matrix. It is convenient to use the transformation matrix for LCS representation, bearing in mind that any LCS can be the result of transforming from the *standard position* (see Standard position). Note that C3D Solver is used only for transformations that retain the distance between any points of a geometric object. It means that C3D Solver does not support any transformations that distort the orthogonality or normality of LCS vectors, for example, scaling.

Rigid set or Cluster — a subset of geometric objects within a common LCS that C3D Solver sees as a geometric object being calculated. Clusters are used in a geometric model to represent solids and geometrically rigid subassemblies. Any geometric object of the LCS type can be a cluster in the Solver.

Geometric constraint — a relation that determines a link between two, three or more geometric objects that are called *constraint arguments*. Examples of geometric constraints: "line L is parallel to plane P", "cylinder C is tangent to plane P", "distance between planes P1 and P2 is 10.0", etc.

Constraint argument — a geometric object or a numeric parameter connected by a geometric constraint or dimension.

Geometric constraint system — a set of mutually related geometric objects. The geometric constraint system formulates a GCSP (geometric constraint satisfaction problem) the purpose of which is to find positions of 3D geometric objects that satisfy all the given constraints.

Initial approximation — coordinate values of all geometric objects and constraint system variables that the solver considers a state close to the target solution. The initial approximation generally satisfies the majority of constraints and is suggested as the starting condition of GCSP**.** The initial approximation also defines the solution which C3D Solver selects for tasks with a non-numerable set of solutions (see Under-defined constraint systems). C3D Solver may offer different solutions for the same system of constraints with different initial approximations.

Geometric constraint satisfaction problem (GCSP) — a task to transform a set of geometric objects with preliminary evaluations (**Initial approximation**) into a position that satisfies all the given constraints.

In addition to the list of objects and constraints, the GCSP definition also includes the **Initial approximation** that affects the solution result.

Three-dimensional geometric solver — a software component that solves the problem of satisfying constraints for 3D objects. As part of the C3D Solver module, this component is technically referred to as GCM (***G****eometric* ***C****onstraint* ***M****anager*). Another component, GCE calculates planar objects (see S1.Two-dimensional geometric solver).

Well-defined constraint system — a geometric constraint system that has a single possible solution or a finite range of solutions.

Inconsistent constraint system — a constraint system with no solution. A constraint system that has at least one solution is, accordingly, referred to as **Consistent**.

Under-defined constraint systems — a geometric constraint system with infinite solutions.

Over-defined constraint system — a constraint system that includes constraints which, if removed, do not affect the set of solutions or transform an inconsistent system into a consistent system. Redefined constraint systems usually include unnecessary constraints, the removal of which either does not affect anything or eliminates inconsistencies.

Dimensional constraint or Dimensions — a geometric constraint that includes a numeric parameter argument that measures distance or angle between two geometric objects. The numeric value of this parameter is called the **Dimension value**.

Distance dimension — a dimension constraint that measures the distance between points belonging to two geometric objects. For instance, the distance between a point and a plane is measured as the distance between the point and its nearest projection on the plane. Distance dimension value is always set using distance measurement units (meter, millimeter, inch, etc.) and may have both positive and negative values, including zero.

Angular dimension — a dimension constraint that measures the angle between two directions taken from a pair of geometric objects. In other words, the angular dimension value is the angle of rotating the first object vector to align it with the second object vector. For example, the angle between a line and a plane is determined by the angle of the line rotation to be aligned with the plane using the shortest path. Angular dimension values are set in radians.

Logical constraint — a geometric constraint that measures a boolean value: **true** if the constraint is satisfied and **false** if the constraint is not satisfied. Examples of logical constraints: parallelism of two lines, tangency between a cylinder and a plane, coaxiality of two cones, etc. Note that dimensions are not logical constraints.

Parametric model — a *geometric model* making it possible to create to different instances of this model by modifying independent numeric parameters (control parameters). The geometric constraint system may be a parametric model if at least one *driving dimension* is set for it (see S.6.17. Driving dimensions).

S.6.2. Assigning the GCE geometric solver

Within the C3D geometric kernel, the three-dimensional geometric solver has an internal technical designation — **GCM** (***G****eometric* ***C****onstraint* ***M****anager*). The **GCM_** prefix in the names of functions and data structures means that they are related to the three-dimensional solver API. The GCM component calculates the position and coordinates of geometric objects that satisfy a predefined set of constraints and dimensions, ensuring the integrity of the geometric model of an application. The GCM geometric solver is used with such geometric objects as a point, line, plane, circle, sphere, cylinder, cone, toroid, or their combination in rigid sets (Cluster). Positions of geometric objects may be subject to dependencies in the form of constraints from a predefined set of types, including logical constraints: parallelism, perpendicularity, tangency, coincidence, coaxiality, symmetry and various dimensions in distance and angle measurement units.

The core functionality of the geometric solver GCM is available via the **gcm_api.h** header file. The application interacts with the constraint solver based on simple C++ data structures; all these structures are declared in the **gcm_types.h** file.

S.6.3. Embedding the GCM component into an application

The GCM geometric solver is created as a multi-purpose 3D parametric modeling component. Therefore, it can be embedded into any application where the functionality of dimensions and constraints should be added to a 3D geometric model. The key feature is that integration with the C3D Solver does not require any modifications in data structures of the application.

The software interface of the C3D Solver has its own abstract system of data types which is not related to data types of the C3D Modeler module. Therefore, the application may either use or not use data C3D Modeler data structures when working with the solver. Connecting the geometric solver to manage a geometric model requires that the application interacts with the solver as follows:

1. **Create a geometric constraint system** (see **GCM_CreateSystem**). This is the first thing to be done to begin using C3D Solver.
2. **Add objects to the constraint system.** Add geometric objects to the system using **GCM_AddGeom**, **GCM_SubGeom** calls (see S.6.9. Adding and deleting geometric objects). Descriptors (see Table S.6.6.1. Descriptor data types) that provide these functions are used for storing the links of the application objects to C3D Solver's geometric objects. For optimal memory use, we do not recommend adding objects unless they are connected by constraints.
3. **Add geometric constraints to the system.** Add constraints using the calls mentioned in S.6.10. Adding and deleting geometric objects. Application constraints refer to C3D Solver constraints using special descriptors (see Table S.6.6.1. Descriptor data types).
4. **Solve the system of constraints.** C3D Solver uses the **GCM_Evaluate** call to calculate a new state of geometric objects that satisfy the previously declared constraints.
5. **Apply the calculation results.** Apply the calculated coordinates of objects from the C3D Solver constraint system to geometric objects of the application. The calculated result is stored in the C3D Solver. So, the application should update the state of its objects by requesting new coordinate values from the solver.
6. **Deleting objects and constraints.** Once you have finished a constraint system session, the application calls the **GCM_RemoveSystem** method. Also, during the life cycle of the constraint system, you can individually delete constraints and objects when you no longer need them (see S.6.9. Adding and deleting geometric objects

To adapt the geometric solver to native data types of the application, organize an interface between the application and the GCM component. You can see a sample diagram of interaction between the constraint solver and the application in Figure *S.6.3.1.* The diagram suggests creating a special Constraint Manager to solve the following tasks:

- "Hides" the API C3D Solver behind it and provides an interface which is more convenient and expressed in the native data of the application;
- Loads data on 3D model objects and constraints to the solver (geometric problem definition);
- Processes command queries such as adding/deleting data of the constraint system, and calculation requests;
- Ensures feedback from the solver, applies the results of C3D Solver calculations to the geometric model of the application;
- Updates the solver data for synchronization with the application model.

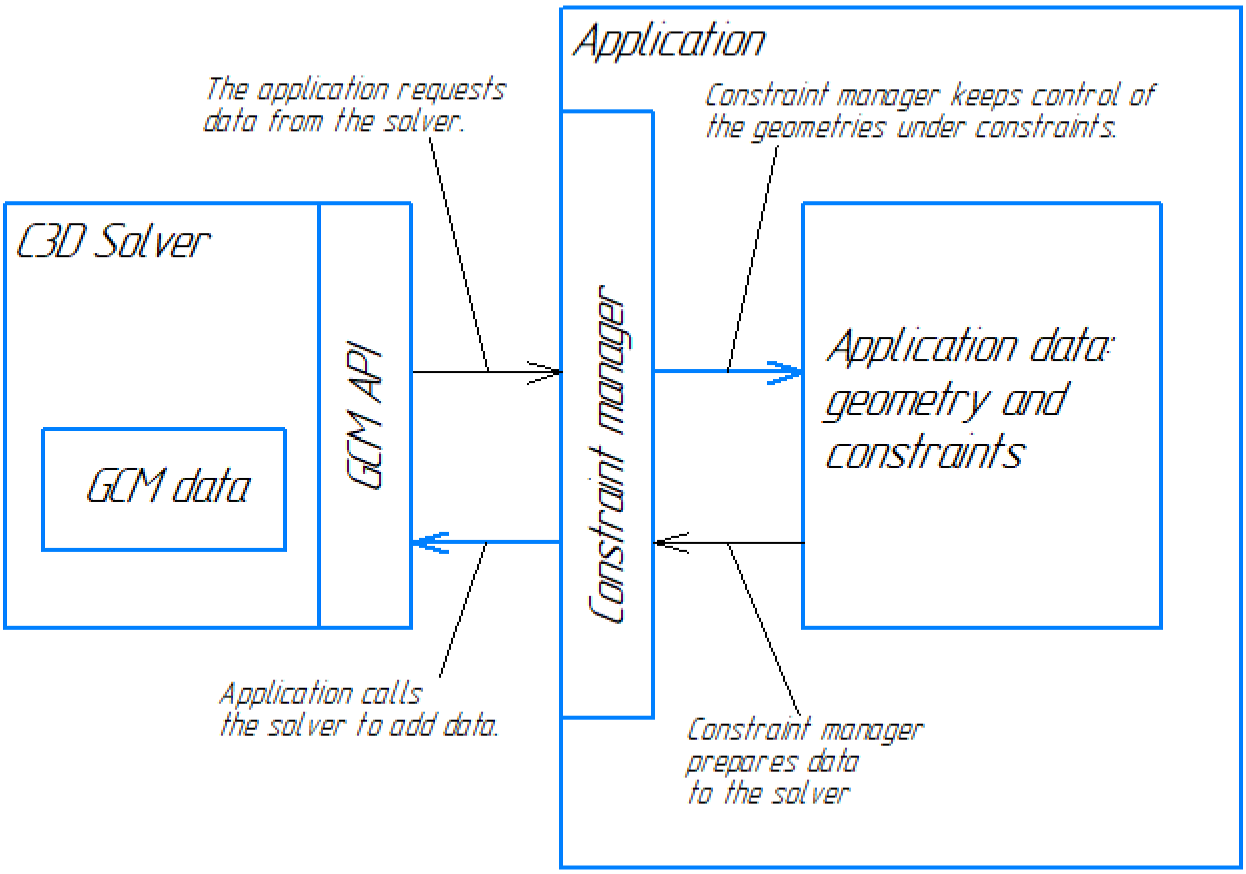


Fig.S.6.3.1.

The diagram shown in *Fig. S.6.3.1.* is not mandatory. It simply demonstrates one of the options to integrate C3D Solver with the application.

The GCM component does not permit saving user data in a document file, so an application developer should take care of read/write permissions for the 3D model constraint system in the application document.

S.6.4. Supported geometry types

A geometric scene controlled by GCM is created as a set of geometric objects that belong to one of the following types:

- Point
- Line
- Plane
- Cylinder or Cone
- Sphere
- Toroid
- Circle
- Local Coordinate System (LCS)

The above-mentioned geometry types can be grouped to create geometrically rigid sets — clusters that usually perform the role of a solid or subassembly in the application. In the constraint system, clusters are registered using the LCS type (enum `GCM_LCS` value).

You can request the type of a geometric object registered in the constraint system by calling the GCM_g_type **GCM_GeomType** function (GCM_system **gSys**, GCM_geom **g**)

with the following input parameters:

- **gSys** — descriptor of the geometric constraint system;
- **g** — descriptor of the geometric object that belongs to the **gSys** system.

The function will return one of the following values specified by enum GCM_g_type:

GCM_NULL_GTYPE	for an empty object (**g** = GCE_NULL);
GCM_POINT	for a point;
GCM_LINE	for a line;
GCM_PLANE	for a plane;
GCM_CYLINDER	for a cylinder;
GCM_CONE	for a cone;
GCM_SPHERE	for a sphere;
GCM_TORUS	for a toroid;
GCM_CIRCLE	for a circle;
GCM_LCS	for a local coordinate system;
GCM_MARKER	for marker in the form of <O,Z,X>;
GCM_SPLINE	for a spline (not yet supported).

S.6.5. Supported constraint types

Any geometric constraint links geometric objects that are called constraint *arguments*. Any object listed as a geometry type in section S.6.4. Supported geometry types can be used as a constraint argument. For instance, a cylinder can be a **tangency** constraint argument. Constraints are classified as *unary*, *binary* and *ternary* based on the number of geometry arguments. They link one, two, or three objects correspondingly. **Symmetry** is an example of the ternary constraint — it includes two objects reflecting one another and the reflection symmetry plane. Generally speaking, a constraint may have N arguments. *Logical constraints* assume dependencies between geometric objects only. *Dimensional constraints* establish a link between geometric objects and a numeric parameter (Dimension value) that measures the distance or angle. Therefore, the last argument of dimensional constraints is always a numeric (scalar) argument. Types of constraints supported by the GCM geometric solver can be found in Table S.6.5.1. Geometric constraint types.

Table S.6.5.1. Geometric constraint types (enum GCM_c_type).

Constraint	Arity (number of arguments)	Designation in API
	Logical constraints	
Coincidence	2	GCM_COINCIDENT
Concentricity	2	GCM_CONCENTRIC
Parallelism	2	GCM_PARALLEL
Perpendicularity	2	GCM_PERPENDICULAR
Tangency	2	GCM_TANGENT
Mirror symmetry	3	GCM_SYMMETRIC
User-defined dependency	N	GCM_DEPENDENT
	Dimensional constraints	
Distance	2+1	GCM_DISTANCE
Angle	3+1	GCM_ANGLE
Radius	1+1	GCM_RADIUS
	Pattern constraints	
Dependency on a pattern	3+1	GCM_PATTERNED
Linear pattern (translational)	3	GCM_LINEAR_PATTERN

Angular pattern (rotational)	3	GCM_ANGULAR_PATTERN
Mechanical transmissions		
Mechanical transmission	2+2	GCM_TRANSMITTION
Cam mechanism	2+2	GCM_CAM_MECHANISM

S.6.6. Basic data types of GCM solver API

The application interacts with the constraint solver based on simple C++ data structures; all these structures are declared in the **gcm_types.h** file. The key role is played by descriptor data types that identify any objects controlled by the solver (see Table S.6.6.1. Descriptor data types).

Table S.6.6.1. Descriptor data types

Solver data type	Implementation	Interpretation
GCM_system	void *	constraint system descriptor
GCM_object	struct { size_t id; }	descriptor of computational object
GCM_geom	GCM_object	geometric object descriptor
GCM_constraint	GCM_object	constraint descriptor
GCM_pattern	GCM_object	pattern descriptor

Data types with finite enumerations of possible values are listed in Table S.6.6.2.

Table S.6.6.2. Enumeration data types

Solver data type	Interpretation
GCM_g_type	geometric object type
GCM_c_type	constraint type
GCM_alignment	alignment variant
GCM_tan_choice	tangency variant
GCM_result	diagnostic code
GCE_scale	pattern scalability option

Data structures listed below (Table S.6.6.3.) are used to transfer the set of parameters when API functions are called. For example, the GCM_g_record structure is used to transfer the geometric object type and its positioning LCS to the solver.

Table S.6.6.3. Data structures

Solver data type	Interpretation
GCM_vec3d	3D vector coordinates
GCM_POINT	3D point coordinates
GCM_g_record	geometric object record
GCM_extra_param	GCM_dependent_func call parameter
GCM_c_arg	geometric or numeric constraint argument
GCM_c_record	constraint record, its type and arguments

S.6.7. Geometric constraint system

The geometric constraint system is a set of geometric objects mutually related by constraints and dimensions. Types of supported geometric objects and constraints can be found in paragraphs S.6.4. Supported geometry types, S.6.5. Supported constraint types, respectively. The geometric model created in an application is expected to have its own constraint system. From a software engineering viewpoint, this means that each model is associated with the constraint system using the GCM_system descriptor type, and each geometric object and constraint is represented by its unique descriptors of types GCM_geom and GCM_constraint.

Before proceeding to work with geometric constraints, declare a constraint system for the model by calling:

GCM_system **GCM_CreateSystem**().

The function returns the empty constraint system as the GCM_system descriptor, which is a pointer to internal data structure instance in the geometric solver. All further manipulations with the constraint system will be performed using this descriptor. For example, if you want to declare a point, call the following function for the created constraint system:

GCM_geom **GCM_AddPoint**(GCM_system **gSys**, MbCartPoint3D **pVal**)

The function returns a descriptor of the geometric point that belongs to the geometric constraint system **gSys.** The **pVal** parameter defines the start <x,y,z> coordinates of this point.
When you finish work with the geometric model, always call the function that deletes its constraint system:

void **GCM_RemoveSystem**(GCM_system **gSys**)

This function completely releases the memory occupied by the constraint system with all object and constraint data. After calling the **GCM_RemoveSystem** function, the constraint system descriptor becomes invalid, i.e. if you try to use the descriptor then the application might crash.

Function

void **GCM_ClearSystem**(GCM_system **gSys**)

clears the constraint system; it deletes objects and constraints only from the memory, but it keeps the constraint system valid for further work.

Also, by calling

bool **GCM_ReadSystem**(GCM_system **gSys**, reader & **in**)

you can save the constraint system to the file stream while, calling

bool **GCM_WriteSystem**(GCM_system **gSys**, writer & **out**),

you can restore the complete system state. The functions for saving and recovering from the stream ensure safe storage of descriptor values of all constraints and objects.

Warning! Read/write functions of the constraint system save only the data on objects and constraints that are loaded onto C3D Solver. All source data on objects and model constraints are stored on the application side. The application developer must take care to ensure the integrity of the data. **GCM_ReadSystem**, **GCM_WriteSystem** functions may be useful in the following cases:

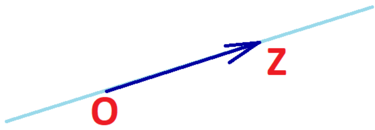
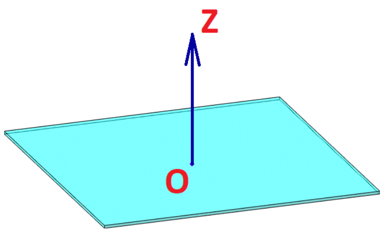
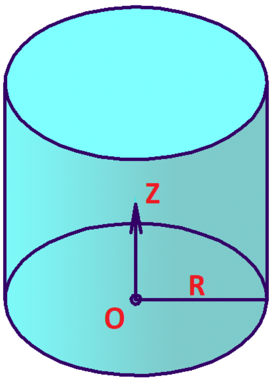
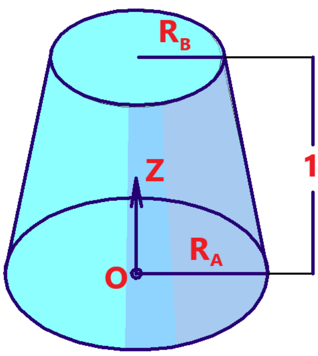
- Transfer of data from the application developer to the C3D Solver developer for debugging during technical maintenance. See also paragraph S.6.18. Journalling API calls of the GCM solver where journalling GCM calls for debugging and technical maintenance purposes is described;
- Converting native formats into a C3D file with a certain data loss;
- Temporary saving the constraint system state for various purposes (partial loss of data for such

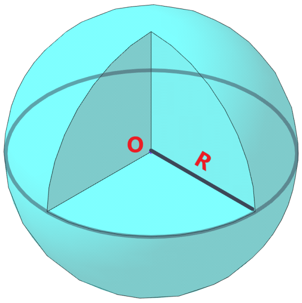
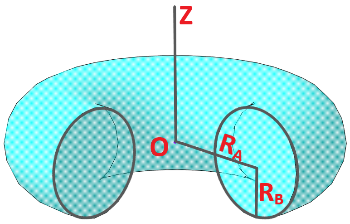
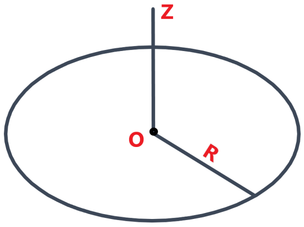
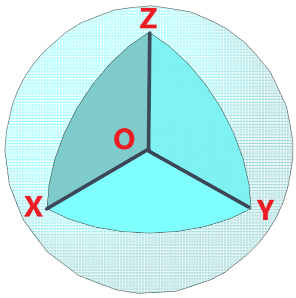
constraints is possible, such as mechanical transmissions and GCM_DEPENDENT dependencies).

S.6.8. Reresentation of geometric objects

GCM three-dimensional geometric solver uses a certain representation of data of geometric objects which is illustrated in Table S.6.8.1.. All objects are expressed in coordinates of points, vectors and numbers (scalars) written in a form specific for each type.

Table S.6.8.1. Representation of geometric objects

Geometry type	Symbol	Tuple and its values
Point		<x y z> — Cartesian coordinates of the point
Line		<O Z> O — point on line; Z – directing vector of the line.
Plane		<O Z> O — plane point; Z – normal vector of the plane.
Cylinder		<O Z R> O — point on the cone axis; Z – directing vector of the axis. R — cylinder radius.
Cone		<O Z R_A R_B> O — center of the lower cone section; Z – directing vector of the axis. R_A, R_B — radii of the lower and upper sections of the cone whose height is considered equal to 1.

Sphere		<O Z R> O — sphere center; R — sphere radius,
Toroid		<O Z R_A R_B>
Circle		<O Z R> O — circle center; Z – direction of the circle axis; R — circle radius.
Local Coordinate System		<O Z X Y> O — LCS origin point; Z X Y – three unit vectors of the LCS.

Records of all these geometric objects can be unified in the form of the tuple of values:

<O Z X Y R_A R_B>,

where four values O Z X Y define the object location in space, namely, its LCS determined by CS start point and three axes Z, X, Y, while two scalar values R_A, R_B define the parameters of radial objects, such as circle, cylinder, cone, toroid, and sphere.

A geometric object of any type has its associated local system of coordinates. Even a point or sphere can be redundantly represented in LCS for which the directions of Z, X, Y axes do not matter, but coordinates of the origin point O are important. Similarly, a plane has its own LCS, the origin point of which is defined by the plane position, while the Z axis defines the plane normal. Here, the values of axes X, Y for the plane are ignored.

In the software interface of the 3D solver, the unified form of recording <O Z X Y R_A R_B> is used in the GCM_g_record structure, whose data fields can be found in Table S.6.8.2.:

Table S.6.8.2. Description of data fields of the GCM_g_record structure.

Data type	Data field	Interpretation (geometrical meaning is specified in table S.6.8.1.)
`GCM_g_type`	`type`	Geometry type.

GCM_POINT	origin	Point of the geometric object positioning (origin of LCS, center of a circle or sphere, etc.).
GCM_vec3d	axisZ	Z axis of the local coordinate system (for example, axisZ is a directing vector for a straight line).
	axisX	X axis of the local coordinate system.
	axisY	Y axis of the local coordinate system.
double	radiusA	Sets the radius of a circle, sphere or cylinder; sets the base radius for a cone, and the greater radius for a toroid.
	radiusB	Sets the lesser radius for a toroid or cone.

GCM_g_record summarizes any geometry record. Therefore, some GCM_g_record data fields may remain unused, depending on the geometry type. For instance, axisX, axisY fields are required only when type = GCM_LCS, while they are irrelevant for such objects as plane, cylinder and toroid. The axisZ field is not used for points and spheres. In other cases, however, it sets the orientation of an object in space.

The GCM solver's API supports the auxiliary functions **GCM_Point**, **GCM_Line**, **GCM_Plane**, **GCM_Cone**, **GCM_Cylinder**, **GCM_Circle**, **GCM_Torus**, **GCM_Sphere**, **GCM_SolidLCS** making it possible to fill the GCM_g_record structure correctly. Details of these functions are given in section S.6.9. Adding and deleting geometric objects

S.6.9. Adding and deleting geometric objects

The geometric solver mainly works with geometric objects, so, in starting to create the constraint system, the application declares the geometric objects that will become constraint arguments in the constraint system.

Every geometric object declared in the constraint system will have its unique identifier – a GCM_geom descriptor. Its geometric type (GCM_g_type) remains unchanged throughout the object's life. API C3D Solver calls that add geometric objects to the system are described below.

A point is added by calling

GCM_geom **GCM_AddPoint**(GCM_system **gSys**, const MbCartPoint3D & **pVal**)

The function returns a descriptor of the geometric point of the **GCM_POINT** type that belongs to the **gSys** geometric constraint system**.** The **pVal** parameter sets the start values of <x,y,z> point coordinates.

To create geometric objects of other types, including a point, you should fill the GCM_g_record data structure which is a unified record for any geometry type supported by the solver. This structure value is the input value of function

GCM_geom **GCM_AddGeom**(GCM_system **gSys**, const GCM_g_record & **gRec**)

The result of the call is the descriptor of the geometric object registered in the **gSys** system**.** Geometric object parameters are specified by fields of the GCM_g_record structure:

- **gRec.type** sets the geometric type of the object (GCM_g_type);
- **gRec.origin** sets the object's position and its **LCS** origin;
- **gRec.axisZ** sets the Z axis vector of LCS;
- **gRec.axisX** sets the X axis vector of LCS;
- **gRec.axisY** sets the Y axis vector of LCS;
- **gRec.radiusA** and **gRec.radiusB** set scalar parameters of radial objects.

See S.6.8. Reresentation of geometric objects for details of data representation in the GCM_g_record structure.

To conveniently and correctly fill the GCM_g_record structure, the GCM component API supports the auxiliary functions listed below:

Function

GCM_g_record **GCM_Point**(const MbCartPoint3D &)

performs actual conversion of the value of 3D point coordinates from the MbCartPoint3D data type into

GCM_g_record, perceived by the **GCM_AddGeom** function.

GCM_g_record **GCM_Line**(const MbCartPoint3D & **org**, const MbVector3D & **axisZ**)
returns the line record based on a point and directing vector.

GCM_g_record **GCM_Plane**(const MbCartPoint3D & **org**, const MbVector3D & **axisZ**);
returns the record of a plane defined by a normal point and vector.

Function for creating a cone record:

GCM_g_record **GCM_Cone** (const MbCartPoint3D & **centre**, const MbVector3D & **axis**,
double **radiusA**, double **radiusB**)

Input parameters:

- **centre** — center of the base circle of the cone,
- **axis** — direction vector of the cone axis,
- **radiusA** — cone base radius,
- **radiusB** — cone section radius (lesser radius).

Cone parameters are assumed to describe a virtual truncated cone whose height is always equal to a length unit, i.e. there is a unit distance between the cross section of **radiusA** and the **radiusB** section (see figure in Table S.6.8.1.).

Function for creating a cylinder record:

GCM_g_record **GCM_Cylinder** (const MbCartPoint3D & **centre**, const MbVector3D & **axis**,
double **radius**)

Input parameters:

- **centre** — a point on the cylinder axis;
- **axis** — direction vector of the cone axis;
- **radius** — cylinder radius.

Function for creating a circle record:

GCM_g_record **GCM_Circle** (const MbCartPoint3D & **centre**, const MbVector3D & **axis**, double **radius**)

Input parameters:

- **centre** — circle center coordinates;
- **axis** — direction of the circle axis;
- **radius** — circle radius.

Function for creating a toroid record:

GCM_g_record **GCM_Torus** (const MbCartPoint3D & **centre**, const MbVector3D & **axis**,
double **majorR**, double **minorR**)

Input parameters:

- **centre** — toroid center coordinates;
- **axis** — direction of the toroid revolution axis,
- **majorR** — distance from the toroid center to the center of the generating circle of the toroid;
- **minorR** — radius of the generating circle of the toroid.

Function for creating a sphere record:

GCM_g_record **GCM_Sphere** (const MbCartPoint3D & **centre**, double **radius**).

Input parameters:

- **centre** — sphere center;
- **radius** — sphere radius.

Function for creating a rectangular right-handed coordinate system**LCS**:

GCM_g_record **GCM_SolidLCS**(const MbCartPoint3D & **org**, const MbVector3D & **axisZ**,
const MbVector3D & **axisX**).

Input parameters:

- **org** — coordinate system origin;
- **axisZ** — Z unit vector;
- **axisX** — X unit vector;

The result of the function is the **LCS** record in the form of <O Z X Y>, where axis Z is directed by vector product X∘Y that corresponds to the right-handed orientation of X-Y-Z . The geometric type of the LCS record is **GCM_LCS**.

Another method for creating the LCS record has the same name but its argument is MbPlacement3D:

GCM_g_record **GCM_SolidLCS** (const MbPlacement3D &).

This method allows creating both right-handed and left-handed CS.

The result returned by the **GCM_SolidLCS** function is used to create a solid body (cluster) in the constraint system by calling **GCM_AddGeom** or GCM_SubGeom.

You can use the following call to create an empty GCM_g_record record that can be used to create a default value:

GCM_g_record **GCM_NullGeom**(void).

Below is the code fragment that demonstrates creating a system that includes the **sph** sphere with its center in the CS origin and the cylinder **cyl** with its center coordinates <10, 0, 0> and oriented along the Z axis.

```
GCM_system gSys = GCM_CreateSystem();

const MbCartPoint3D cylPos( 10.0, 0.0, 0.0 );
/*
  Register two geoms, sphere and cylinder.
*/
GCM_geom sph = GCM_AddGeom( gSys, GCM_Sphere(MbCartPoint3D::origin, 10.0/*radius*/) );
GCM_geom cyl = GCM_AddGeom( gSys, GCM_Cylinder(cylPos, MbVector3D::zAxis, 10.0/*radius*/) );
   ...
   ...
/*
  Finalize the constraint system to free its located memory.
*/
GCM_RemoveSystem( gSys );
```

In addition to the GCM_AddGeom call, the solver provides for another method to register geometric objects providing for solid body abstraction — geometrically rigid sets with their own **LCS**. Such subsets are called *clusters*. To create a *cluster*, declare its **LCS** using the following query:

```
GCM_geom lcs = GCM_AddGeom( gSys, GCM_SolidLCS(org,axisZ,axisX) );
```

Values `org`, `axisZ`, `axisX` set the cluster position and orientation (its **LCS**). The resulting **lcs** descriptor will define the **LCS** of the geometrically rigid set whose elements are declared by the following call:

GCM_geom **GCM_SubGeom**(GCM_system **gSys**, GCM_geom **lcs**, const GCM_g_record & **gRec**).

Input parameters:

- **gSys** – geometric constraint system;
- **lcs** — cluster LCS descriptor;
- **gRec** — record of geometric parameters of the object defined in the cluster LCS.

The function returns the descriptor of the geometric object that belongs to the cluster. A specific feature of the object that belongs to the cluster is that its location in space depends on the cluster's LCS. Thus, the dependent object changes its location together with the cluster's LCS. This is its key difference from objects created using the **GCM_AddGeom** method. Handling clusters is described in more detail in S.6.12. Geometric scene clustering, assembly modeling.

Geometric objects returned by **GCM_SubGeom** and **GCM_AddGeom** functions may participate in constraints and dimensions, i.e. they can be used as Constraint argument.

Geometric objects are deleted by the following call:

void **GCM_RemoveGeom**(GCM_system **gSys**, GCM_geom **g**).

This method deletes the geometric object from the system and renders its descriptor **g** invalid. If a geometric object remains an argument of one constraint at the moment when the system is called, it actually remains in the system, but it will be inevitably deleted after deleting the last constraint associated with this object. Therefore, the GCM API user can release a geometric object when it is no longer needed for the application without paying attention to any binding constraints.

S.6.10. Adding and deleting geometric objects

You can use the following call to add the majority of geometric or dimension constraints:

GCM_constraint **GCM_AddConstraint**(GCM_system **gSys**, const GCM_g_record & **cRec**).

It requires completion of the **cRec** data structure that stores the data on the **cRec.type** type and the argument array of the **cRec.args** constraint. The function returns a descriptor of a new constraint registered in the **gSys** system.

As an example, let us add a constraint right now — coincidence of point and plane:

```
GCM_system gSys = GCM_CreateSystem();

GCM_c_record cRec;  // The record of point and plane coincidence.
cRec.type = GCM_COINCIDENT;

/*
  Some point and plane as arguments of the coincidence.
*/
cRec.args[0] = GCM_AddPoint( gSys, MbCartPoint3D::origin );
cRec.args[1] = GCM_AddGeom( gSys, GCM_Plane(MbCartPoint3D(0,0,1), MbVector3D::zAxis) );
cRec.args[2] = GCM_NO_ALIGNMENT; // The alignment option doesn't matter in the point-plane case.

/*
  Add coincidence to the system.
*/
GCM_constraint cItem = GCM_AddConstraint( gSys, cRec );

/*
  Finalize the constraint system.
*/
GCM_RemoveConstraint( gSys, cItem );
GCM_RemoveSystem( gSys );
```

There is a certain sequence of completing **args** arguments in the GCM_c_record structure for each constraint type. In the above example, we have demonstrated that the first two elements `cRec.args[0]` and `cRec.args[1]` of the `GCM_COINCIDENT` constraint are descriptors of the Coincidence constraint, while the third element `cRec.args[2]` defines the alignment option.

GCM_c_record structure. The table below contains the template for completing the GCM_c_record structure which is sent to the **GCM_AddConstraint** method for various constraint types.

Table S.6.10.1. Data argument types for various constraints (GCM_c_record completion)

Constraint type	Argument types				
type	args[0]	args[1]	args[2]	args[3]	args[4]
GCM_COINCIDENT	GCM_geom	GCM_geom	GCM_alignment		
GCM_CONCENTRIC					
GCM_PARALLEL					
GCM_PERPENDICULAR					
GCM_TANGENT	GCM_geom	GCM_geom	GCM_alignment	GCM_tan_choice	
GCM_SYMMETRIC	GCM_geom	GCM_geom	GCM_geom	GCM_alignment	
GCM_DISTANCE	GCM_geom	GCM_geom	double	GCM_alignment	
GCM_ANGLE (planar angle)	GCM_geom	GCM_geom	GCM_geom (axis of rotation)	double	GCM_alignment
GCM_ANGLE (3D angle)	GCM_geom	GCM_geom	GCM_NULL (no axis)	double	GCM_alignment

GCM_RADIUS	GCM_geom	double			

GCM_alignment is an option value considered in S.6.11. GCM_alignment option. GCM_tan_choice values will be described in sections dedicated to the tangency constraint (GCM_TANGENT). Table S.6.10.1. lists only interpretation rules for data types accepted by the GCM_c_arg element type in GCM_c_record::args array for a certain constraint type.

Therefore, the GCM_c_record data structure is a unified method for setting a range of constraints using only one method — **GCM_AddConstraint**. However, there are alternative ways to add constraints for convenient development in GCM API that do not require completing the GCM_c_record structure.

You can use the following function to add any **binary** constraints that bind a pair of geometric objects:

GCM_constraint **GCM_AddBinConstraint**(GCM_system **gSys**, GCM_c_type **cType**,
GCM_geom **g1**, GCM_geom **g2**, GCM_alignment **aVal**, GCM_tan_choice **tVar**);

Input parameters of the function:

- **gSys** — descriptor of the constraint system;
- **cType** – type value from the following list: GCM_COINCIDENT, GCM_PARALLEL, GCM_PERPENDICULAR, GCM_TANGENT, GCM_CONCENTRIC;
- **g1**, **g2** – a pair of geometric object descriptors, binary constraint arguments;
- **aVal** – alignment option making it possible to select mutual orientation for objects (see S.6.11. GCM_alignment option);
- **tVar** – tangency selection option for selecting a method for executing constraints of the GCM_TANGENT type.

The result of the call is the binary constraint descriptor binding **g1** and **g2** objects with the selected alignment/tangency parameters **aVal**, **tVar**.

The dimension constraint of the GCM_DISTANCE type that defines the distance between objects is created using the following call:

GCM_constraint **GCM_AddDistance**(GCM_system **gSys**, GCM_geom **g1**, GCM_geom **g2**,
double **dVal**, GCM_alignment **aVal**).

The call uses input data **g1**, **g2**, **aVal** — the same as for the **GCM_AddBinConstraint** function; numeric value **dVal** – sets the dimension value.

The same algorithm is used for calls that define the angular dimension, mirror symmetry, and radial dimension: **GCM_AddAngle**, **GCM_AddSymmeric**, **GCM_FixRadius**. They also do not require completing the GCM_c_record transit structure.

S.6.11. GCM_alignment option

The constraint type (GCM_c_type) is the main feature of a constraint, which is sufficient to state if object positions satisfy this constraint type or not. However, participation of orientable objects, such as a plane or line, in the constraint allow for different variants of the same GCM_c_type type. For example, based on two faces of a body within the same plane, we can state that they satisfy the coincidence constraint (GCM_COINCIDENT). However, this statement allows for two coincidence variants, when normal face vectors have the same direction or opposite directions. Figures S.6.11.1., S.6.11.2. demonstrate alternative alignment variants for two bodies using the GCM_COINCIDENT constraint that binds a pair of colored faces.

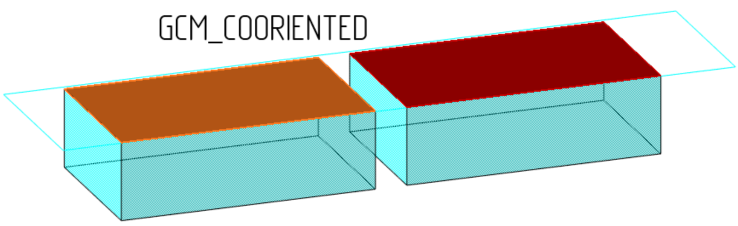


Fig. S.6.11.1.

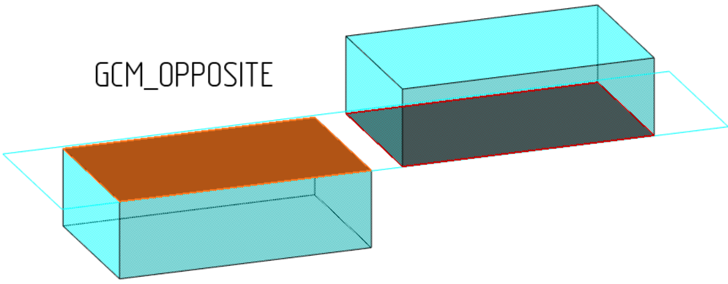


Fig. S.6.11.2.

By default, the solver fulfill the coincidence and keeps the orientation of geometric objects as close to their original state as possible (see Initial approximation). Therefore, one of the two variants is selected automatically based on the minimum rotation principle. However, if necessary, a certain mutual orientation of vectors can be implemented. For this purpose, **GCM_AddBinConstraint**, **GCM_AddConstraint**, **GCM_AddDistance** methods support the GCM_alignment option, whose values can be found in the table.

GCM_alignment adds a condition to the constraint type info (GCM_c_type) that excludes any ambiguity when selecting a solution for the constraint. `GCM_COORIENTED` and `GCM_OPPOSITE` values affect the constraints with parallel Z vectors of their arguments. For example, perpendicularity of a plane and a line means that normal vectors of the plane and the line are parallel. Another example: two coincident planes mean that normals of their faces have either the same direction (Fig. S.6.11.1.) or the opposite direction (Fig. S.6.11.2.). Other alignment option values are applied in more complicated cases together with such constraints as tangency, linear and angular patterns, where the selection range includes more than 2 solutions. For example, tangency of two toroids by their circles means eight solutions which, in addition to codirected vectors, vary in the tangency side with respect to Z axes, and the location of one toroid outside or inside the other.

Table S.6.11.3. GCM_alignment option values.

Name	Value
`GCM_NO_ALIGNMENT`	Unspecified orientation. The `GCM_NO_ALIGNMENT` value is applied when a constraint is not affected by orientation of geometric objects (for example, coincidence of two points) or when object orientation should remain free. For example, if you specify `GCM_NO_ALIGNMENT` in angular or linear patterns, the "sample" orientation will not affect the "copied instance".
`GCM_CLOSEST`	The solution is selected automatically. It should be as close to the initial position of geometric objects as possible.
`GCM_COORIENTED`	Sets cooriented Z vectors of constraint arguments.
`GCM_OPPOSITE`	Sets opposite Z vectors of constraint arguments.

`GCM_ALIGNED_0`
`GCM_ALIGNED_1`
`GCM_ALIGNED_2`
`GCM_ALIGNED_3`
	Solution variants for constraints with cooriented Z vectors of constraint arguments (`GCM_COORIENTED` subvariants).

`GCM_REVERSE_0`
`GCM_REVERSE_1`
`GCM_REVERSE_2`
`GCM_REVERSE_3`
	Solution variants for constraints with opposite Z vectors of constraint arguments (`GCM_OPPOSITE` subvariants).

`GCM_ALIGNED`	Creates the same orientation of LCS objects (coorientation along Z and X axes) for such constraints as `GCM_SYMMETRIC` and `GCM_PATTERNED`. This means that such constraints as symmetry, angular and linear patterns with this option will orient the "copied instance" in the same way as the "sample", and their positioning will follow the respective rules for copying.
`GCM_ROTATED`	It is applied for the `GCM_PATTERNED` constraint of the

	GCM_ANGULAR_PATTERN angular pattern. This means that LCS of the "copied instance" and the "sample" become fully coincident by rotating the pattern around its axis.
GCM_ALIGN_WITH_AXIAL_GEOM	It is applied as an additional option for angular and linear patterns in **GCM_AddLinearPattern**, **GCM_AddAngularPattern** functions.

In summary, GCM_alignment is an additional condition when GCM_c_type allows for more than one variant of a constraint execution.

S.6.12. Geometric scene clustering, assembly modeling

The principle of hierarchy is the basis of the geometric model of the constraint system. This principle means that geometric objects may be grouped as geometrically rigid sets called clusters. A Cluster is a similar to such CAD abstraction of a geometric model as a solid or subassembly. Clusters may be grouped in tree-structured hierarchies whose dependencies can be traced using the following function:

GCM_geom **GCM_Parent**(GCM_system **gSys**, GCM_geom **subGeom**).

This function returns the cluster of the **subGeom** object registered in the **gSys** constraint system. If the function returns zero descriptor **GCM_NULL**, it means that the requested object is not contained in any clusters and was registered in the constraint using the **GCM_AddGeom** call. Geometric objects subordinate to a cluster are created using another call — **GCM_SubGeom** (see S.6.9. Adding and deleting geometric objects).

Constraints within clusters. Geometric constraints and dimensions, as well as geometric objects, are parts of the hierarchy of clusters. There is no need to expressly specify the level, at which a certain constraint is calculated. The level is selected depending on which clusters its arguments belong to. When adding a constraint, the solver selects the minimum-sized subsystem that includes all of its arguments. In other words, the geometric constraint level is determined by the minimum subtree of the hierarchy that includes all of its arguments. *Fig. S.6.12.1.* illustrates an assembly example with two dimensions to be calculated at various hierarchy levels. The assembly consists of two parallelepipeds represented by Cluster1 and Cluster2, and each of them combines the planes, edges, and vertices of each parallelepiped.

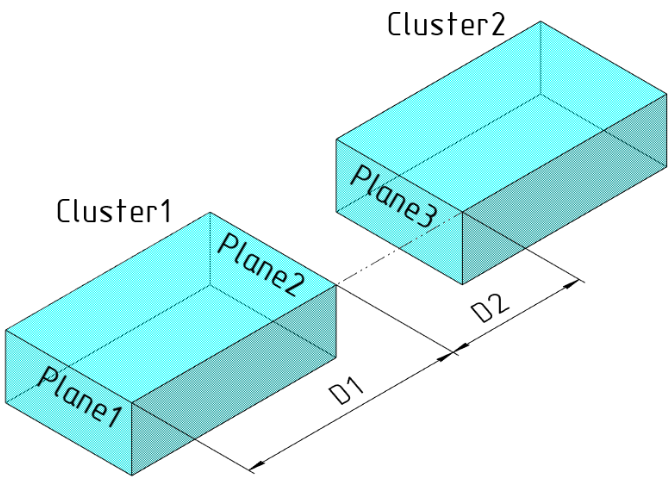


Fig. S.6.12.1.

The entire hierarchy consists of three constraint subsystems, calculated independently from the bottom up (*see Fig.* S.6.12.2). There are two dimensions in the assembly: D1 – distance between Plane1 and Plane2 that belong to the same Cluster1, D2 – distance between Plane2 and Plane3 that belong to different clusters (Cluster1, Cluster2). In this case, D1 will be calculated in Cluster1 of the cluster hierarchy, while dimensions D2 will be calculated after D2 together with all constraints of the root assembly level. Therefore, the D1 calculation results will determine the size of one parallelepiped only, while D2 will affect the positioning of two parallelepipeds with respect to one another.

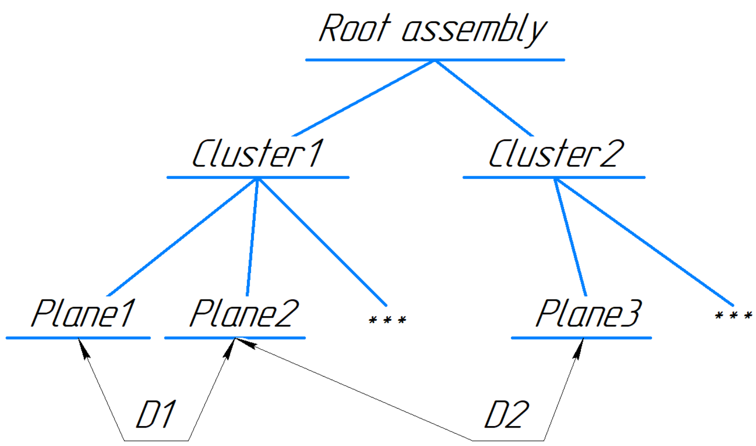


Fig. S.6.12.2.

Thus, the constraint level is determined by the minimum subtree of clusters that includes all of its arguments. In particular, if the constraint arguments are included in the same cluster, the constraint is part of the same cluster. If the constraint arguments are included in different clusters, the level is selected that constitutes the minimum subtree including all the constraint arguments.

The following is the source code that reproduces the case illustrated in *Fig. S.6.12.1*. This code is an example demonstrating how the solver creates the cluster hierarchy shown in *Fig. S.6.12.2*.

```
GCM_system gSys = GCM_CreateSystem();
/*
  Add LCS of the first cluster; it represents the first parallelepiped.
  Also its two planes will be added.
*/
GCM_geom cluster1 = GCM_AddGeom( gSys, GCM_SolidLCS(MbCartPoint3D::origin) );
GCM_FixGeom( gSys, cluster1 );  // First cluster will have a fixed LCS.
const MbCartPoint3D org1( 50.0, 0, 0 ); // Position of "Plane1" in LCS of the first cluster.
GCM_geom plane1 = GCM_SubGeom( gSys, cluster1, GCM_Plane(org1, MbVector3D::xAxis) );
const MbCartPoint3D org2( -112.0, 0, 0 ); // Position of "Plane2" in LCS of the first cluster.
GCM_geom plane2 = GCM_SubGeom( gSys, cluster1, GCM_Plane(org2, -MbVector3D::xAxis) );

/*
  Set a distance dimension D1 between "Plane1" and "Plane2".
  (opposite sides of the parallelepiped, see figure S.6.12.1)
*/
GCM_constraint D1 = GCM_AddDistance( gSys, plane1, plane2, -162.0 );

/*
  Add LCS of "Cluster2" (the second box).
*/
const MbCartPoint3D pos2( -262.0, 0, 0 ); // Position of the second cluster.
GCM_geom cluster2 = GCM_AddGeom( gSys, GCM_SolidLCS(pos2) );
const MbCartPoint3D org3( 50.0, 0, 0 ); // Position of "Plane3" in LCS of "Cluster2".
GCM_geom plane3 = GCM_SubGeom( gSys, cluster2, GCM_Plane(org3, MbVector3D::xAxis) );

/*
  Set distance dimension D2 between "Plane2" and "Plane3".
  (D2 defines a distance between the boxes, see figure S.6.12.1)
*/

GCM_constraint D2 = GCM_AddDistance( gSys, plane2, plane3, 100.0 );
/*
  Dimension D2 can change the position of "Cluster2" relative to "Cluster1".
*/
GCM_ChangeDrivingDimension( gSys, D2, 120.0 );
GCM_Evaluate( gSys );

GCM_RemoveSystem( gSys );
```

Cluster calculation order. Note that the cluster structure created as a result of the combination of **GCM_AddGeom** and **GCM_SubGeom** calls strictly determines the order of constraint calculation. Constraints of a single cluster are calculated at the same time but not before constraints of nested clusters have been calculated. Thus, all clusters are calculated from the bottom up, from subordinate to superior, including the root level.

S.6.13. Layout geometry (GCM_GROUND)

By default, the GCM solver has the Ground geometric object which is available using the constant GCM_GROUND identifier. It is a static object with the following properties:

- Fixed LCS in a standard position (see Standard position), i.e. coincident with the World Coordinate System;
- It is a geometrically rigid cluster where you can add sub-objects, as well as constraints and dimensions.

In the Ground object, you can conveniently position the entire fixed geometry of a scene. For example, when modeling assemblies, the Ground object includes the entire geometry accepted as a fixed part, with respect to which other assembly parts are positioned — they can be fixed or movable depending on dimensions and constraints related to the Ground object. Normally, a 3D CAD model provides for O-X, O-Y, O-Z base axes and planes of the World Coordinate System OXY, OYZ, OZX, that participate in constraints and dimensions together with other objects. Such objects can be conveniently declared as part of the Ground object.

For example, you can request the solver to display the O-X-Y plane of the World Coordinate System:

```
GCM_geom worldXY = GCM_SubGeom( gSys, GCM_GROUND, GCM_Plane(MbCartPoint3D::origin, MbVector3D::zAxis) )
```

where **gSys** – the assembly constraint system, **worldXY** – the descriptor of the O-X-Y plane of the World Coordinate System. The **worldXY** plane now can be used as an argument for any constraint where a plane can be used.

Thus, the Ground special object is an anchor which the basic geometry of a scene can be attached.

S.6.14. Fixing and freezing 3D geometric objects

Any geometric object created in the constraint system is initially free, i.e. it has all the degrees of freedom inherent to a particular object type. During calculations, the C3D Solver may change the state of geometric objects when it is needed to satisfy the constraints. Sometimes it is necessary to fix a part of geometric objects so that the solver would leave the position of the geometric object unchanged. For this purpose, API GCM supports two methods: **GCM_FreezeGeom**, **GCM_FixGeom_**, **freezing** and **fixing**. Both calls fix a geometric object using different methods. Only the application can change the state of frozen and fixed geometric objects using the **GCM_SetPlacement** call.

Freezing is convenient when you need to fix a geometric object for its entire lifetime or temporarily. For example, when adding a new part to an assembly, it is convenient to bind it to other assembly elements from the very beginning. You can position a part using constraints, but other assembly objects are better remaining in the same places. Then, when adding an object we can freeze all assembly elements to which the new part is being bound. Once the positioning is finished, the freezing is disabled. A geometric object can be frozen using the following method:

void **GCM_FreezeGeom**(GCM_system **gSys**, GCM_geom **g**).

Note that freezing is not a constraint. It makes the respective object fixed for the solver within the coordinate system where the object was defined using **GCM_AddGeom** and **GCM_SubGeom** methods. The result of the **GCM_FreezeGeom** call can be cancelled by calling

void **GCM_FreeGeom**(GCM_system **gSys**, GCM_geom **g**),

which releases the **g** object and returns its degree of freedom.

An alternative method to fix a geometric object for the solver is to create a constraint that fixes the object within the global coordinate system using the following function:

GCM_constraint **GCM_FixGeom_**(GCM_system **gSys**, GCM_geom **g**).

The function returns the descriptor of the new constraint which totally fixes the **g** object within the World Coordinate System. The result of this call is cancelled by the **GCM_RemoveConstraint** call, i.e. removed as a normal constraint.

So, there are two aspects that differ **freezing** and fixing using **GCM_FixGeom_**:

- Freezing **GCM_FreezeGeom** removes degrees of freedom of the object without creating a new constraint;
- If a frozen geometric object is added using **GCM_SubGeom**, the solver leaves its position fixed but only within the LCS of the cluster where it had been declared; the Global degree of freedom of a frozen sub-object is determined by the degree of freedom of its cluster.

S.6.15. Evaluating the constraint system

The main task of the three-dimensional geometric solver (see Geometric constraint satisfaction problem) is performed by the following function:

GCM_result **GCM_Evaluate** (GCM_system **gSys**).

The function calculates the new state of geometry and distributes error codes to those constraints that cannot be satisfied. The function does not execute any calculation operations if all constraints are satisfied at the time when it is called.

GCM_Evaluate will return the `GCM_RESULT_Ok` code, if an efficient solution is found for all input constraints. Any other returned value means that the constraint system cannot be entirely satisfied. So, the returned **GCM_Evaluate** value means either a success or a diagnostic code of one of the unsolved constraints or another code indicating the reason for the failed solution. Diagnostic codes are described in S.6.16. Diagnostic codes of a solution.

After the **GCM_Evaluate** call, each geometric constraint is assigned a diagnostic status. You can retrieve it using the following call:

GCM_result **GCM_EvaluationResult**(GCM_system **gSys**, GCM_constraint **cItem**).

The call will output one of the following values for each constraint:

- `GCM_RESULT_None` means that the constraint status is not defined, i.e. the constraint has not been calculated yet.
- `GCM_RESULT_Satisfied` means that the constraint has been solved;
- `GCM_RESULT_Overconstrained` means that the constraint is part of a group of conflicting constraints that cannot be all satisfied simultaneously. Generally, you can fix the situation by removing one of the constraints;
- `GCM_RESULT_Not_Satisfied` means that the constraint is not satisfied (in the event when the cause of a failed solution cannot be identified);
- `GCM_RESULT_Unsolvable` means that the constraint cannot be solved. Possible reasons: conflicting constraints, no required constraints to position objects (for example, in a cam mechanism), poor initial approximation;
- `GCM_RESULT_IncompatibleArguments` means that the constraint is not applicable to objects with these geometry types. For example, it is impossible to implement the coincidence of a plane and a cylinder;

See S.6.16. Diagnostic codes of a solution for all GCM_result values.

Calling **GCM_Evaluate** changes the state of geometric objects. To make it possible for the application to apply the results of calculations, you need to use the **GCM_Placement** function to request LCS values that position objects.

Function

MbPlacement3D **GCM_Placement**(GCM_system **gSys**, GCM_geom **g**) returns the position (LCS) of a geometric object **g** based on the current state of the geometric model. We already mentioned in S.6.8. Reresentation of geometric objects that each geometric object has its own LCS, even if it is a primitive like a point. LCS is convenient, since we can use it to express positions of all types of objects, including not only solid bodies, but also simple objects, such as a point, line, and plane. Also a call **GCM_Origin** can be used to request a point of the LCS. It is convenient when we deal with non-orientable geometries such as point and sphere.

Another function, if applied to radial objects, such as a circle, sphere, or cylinder, will return the radius value:

double **GCM_Radius**(GCM_system **gSys**, GCM_geom **g**).

Also, by calling the following functions:

double **GCM_RadiusA**(GCM_system **gSys**, GCM_geom **g**),
double **GCM_RadiusB**(GCM_system **gSys**, GCM_geom **g**)

you can retrieve the major and the minor radii of a toroid or cone.

The application is capable not only of requesting the current geometry state but also changing it in the event of any modifications on the CAD model side that require synchronization of the geometric solver. For this purpose, the following call is used:

void **GCM_SetPlacement**(GCM_system **gSys**, GCM_geom **g**, const MbPlacement3D & **place**).
The call changes the current LCS value of the object.
Source call data:

- **gSys** — geometric constraint system;
- **g** – geometric object descriptor;
- **place** — new position (LCS) within the World Coordinate System.

Note that this function changes the object state without revaluating the constraint system. Using the **GCM_Evaluate** call, you can change the defined state if there are any constraints that cannot be satisfied.

At any moment, you can request each constraint if it is satisfied or not using the following function:

bool **GCM_IsSatisfied**(GCM_system **gSys**, GCM_constraint **cItem**),
which will return **true**, if the **cItem** constraint is satisfied in the current geometry state. The result of the **GCM_IsSatisfied** call depends only on the current state of the constraint's geometric arguments and is not related to its diagnostic status, returned using the **GCM_EvaluationResult** query.

S.6.16. Diagnostic codes of a solution

The table below lists the codes that output calculation results or results of executing the GCM API component function. These values are results of such calls as **GCM_Evaluate**, **GCM_EvaluationResult**, **GCM_SolveReposition**.

Table S.6.16.1. Enumeration type values GCM_result

Resulting codes of executing the API solver functions	
GCM_RESULT_None	**No result.** For the **GCM_EvaluationResult** call, it means that the constraint has not been calculated yet.
GCM_RESULT_Ok	**Executed successfully.** The result of successful execution of an operation or successful calculation of constraints.
GCM_RESULT_ItsNotDrivingDimension	**Not driving dimension.** Failed attempt of the **GCM_ChangeDrivingDimension** call for a constraint which is not a driving dimension.
GCM_RESULT_Unregistered	**Invalid descriptor of an object or constraint.** Case: Calling API with a descriptor that does not belong to the constraint system.
GCM_RESULT_Aborted	**Calculations aborted by the user request.**

GCM_RESULT_InternalError GCM_RESULT_Error	**Program error** (not mathematic). For situations related to incorrect completion of an operation or incorrect data.

Results of the constraint system calculation	
GCM_RESULT_Satisfied	**The constraint or constraint system has been satisfied** (= GCM_RESULT_Ok).
GCM_RESULT_Overconstrained	**The constraint redefines the model.** The error occurs in the event of conflicting constraints (**Inconsistent constraint system**) within an over-defined constraint system (**defined constraint system**).
GCM_RESULT_Unsolvable	**Unresolvable constraints.** The error occurs in various situations when no solution can be found. Generally, it is caused by a set of constraints that cannot be satisfied simultaneously (**Inconsistent constraint system**) or when dimensions are outside the range of acceptable values.
GCM_RESULT_Not_Satisfied	**The constraint is not satisfied.** The error code is returned when constraints cannot be solved. Probable causes of the failure: • A dimension is outside the range of values making a solution possible; • Positions of fixed or frozen points are outside the range of values making a solution possible; • In rare cases, it is caused by poor initial approximation. Satisfying the constraints requires considerable changes in parametric coordinates; • Strong mutual influences. For example, for such conditions if an angle is changed by one degree, then the object would be offset to a considerable distance. • In rare cases, numeric instability due to wide scattering of the order of coordinates being calculated. For example, a geometric model includes both microscopic objects (1e-3) and large objects (1e+6). • In the general case, this can be caused by other situations when the solution does not exist or cannot be calculated.
GCM_RESULT_MatedFixation	**A geometric constraint is added between fixed or frozen objects**.
GCM_RESULT_InvalidArguments	**Constraint arguments are not defined.** Incorrectly defined constraint when its arguments are empty (argument descriptor = GCM_NULL).
GCM_RESULT_IncompatibleArguments	**Incompatible arguments for this constraint type.** Incorrectly defined constraint when its arguments are incompatible or this combination of arguments is not supported. For example, an attempt to set the coincidence of a plane and a cylinder.
GCM_RESULT_InappropriateArgument	**The argument type cannot be used for this constraint.** An incorrectly defined constraint when one of its arguments cannot be used for this constraint type. For example, an attempt to make points and spheres parallel makes no sense.
GCM_RESULT_Duplicated	**Duplicate constraint.** The code indicates that the

	constraint system contains two identical constraints. Generally, this situation does not make it impossible to calculate constraints.
GCM_RESULT_DraggingFailed GCM_RESULT_InappropriateAlignment GCM_RESULT_InconsistentAngleType GCM_RESULT_InconsistentAlignment mtResCode_UnsupportedTangencyChoice mtResCode_IsNoPossibleForCircTanChoice GCM_RESULT_InconsistentPlanarAngle	

Diagnostic cases for the GCM_DEPENDED constraint	
GCM_RESULT_DependentConstraintUnsolved GCM_RESULT_CyclicDependence GCM_RESULT_MultiDependedGeom GCM_RESULT_OverconstrainingDependedGeoms mtResCode_InvalidDependenceForFixGeom	

Diagnostic codes for mechanical transmissions	
mtResCode_CoaxialMtGearTransmissionIsNotAvalable mtResCode_NoSeparatedSolutionForCamGear mtResCode_CyclicDependenceForTwoOrMoreCamGears mtResCode_InconsistentFollowerAxis	

S.6.17. Driving dimensions

Driving dimensions are dimension constraints that determine a distance or angle between two geometric objects and, therefore, add numeric parameters to a geometric scene that control the positions of its objects. By modifying the dimension parameters, you can control the model state and create different variants of parametric model instances.

Driving dimensions can be created using the following calls: **GCM_AddDistance**, **GCM_AddAngle**, and **GCM_FixRadius**. The main purpose of driving dimensions is to control the positions of geometric objects using the numeric parameter of length, radius, or angle. Note, that constraints of patterns created using the **GCM_AddGeomToPattern** method also relate to driving dimensions, since the pattern elements are also characterized by numeric parameters that determine their linear or angular positions.

Function

GCM_result **GCM_ChangeDrivingDimension**(GCM_system **gSys**, GCM_constraint **dItem**
, double **dVal**)

facilitates modification of any value of the **dItem** dimension and, therefore, control of the state of the parametric model.

Input parameters of the call:

- **gSys** — constraint system of the parametric model;
- **dItem** – dimension constraint descriptor;
- **dVal** – new diameter value set in length units for distance dimensions or in radians for angle dimensions.

Returned values:

- GCM_RESULT_Ok for successful operations;
- GCM_RESULT_ItsNotDrivingDimension means that the returned constraint is not driving dimensions or a driving parameter, for example, if the constraint is a variational dimension.
- GCM_RESULT_None or GCM_RESULT_Unregistered are returned if the delivered **gSys** or **dItem** descriptors are invalid.

Note, that this function does not calculate anything, just prepares the change of the driving dimension. You should call **GCM_Evaluate** for the change to come into effect. If you need to modify two or more driving dimensions at the same time, first make a series of **GCM_ChangeDrivingDimension** calls to each of them and then call **GCM_Evaluate** once.

S.6.18. Journalling API calls of the GCM solver

C3D Solver enables recording of the history of API GCM calls for their further recreation during debugging and testing. For example, if you need a consultation or encounter an application issue when using C3D Solver, you can enable the journalling mode for a while and execute the scenario of interest. The recorded journal can be forwarded to C3D Labs technical support with a description of your situation. When the error is fixed, the corresponding log is added to the test case base.

API calls for the three-dimensional geometric solver are logged in the same manner as journalling for the 2D solver (**S.1.14. API call journalling**). When the journalling mode is enabled using the following call:

bool **GCM_SetJournal**(GCM_system **gSys**, const char * **fName**),

the solver records the history of API calls with respect to the **gSys** constraint system. Then, the recorded journal can be used by C3D Solver developers to analyze or update the test database.

Warning! The journal file will be ready only when a session of work with the constraint system is finished, i.e. immediately after calling the **GCE_RemoveSystem** method. To correctly record the journal, the **GCM_SetJournal** call must immediately follow **GCM_CreateSystem**. Example:

```
 GCM_system gSys = GCM_CreateSystem();

#ifdef C3D_DEBUG
  GCM_SetJournal( gSys, "d:\\Logs\\gcm_sample2.jrn" );
#endif // C3D_DEBUG

  /*
    Vertexes of the triangle A-B-C.
  */
  GCM_geom A = GCM_AddPoint( gSys, MbCartPoint3D(138.0, -38.0, -31.0) );
  GCM_geom B = GCM_AddPoint( gSys, MbCartPoint3D(67.0,  -44.0,  51.0) );
  GCM_geom C = GCM_AddPoint( gSys, MbCartPoint3D(20,30,10) );

  /*
    Set distance constraints of the sides of the triangle.
  */
  GCM_constraint d1 = GCM_AddDistance (gSys, A, B, 100.0, GCM_CLOSEST );
  GCM_constraint d2 = GCM_AddDistance (gSys, B, C, 100.0, GCM_CLOSEST );
  GCM_constraint d3 = GCM_AddDistance (gSys, C, A, 100.0, GCM_CLOSEST );
  GCM_Evaluate( gSys );

  /*
    Finalize the constraint system. The journal file is written.
  */
  GCM_RemoveSystem( gSys );
```

Warning! Journalling is required only for debugging, so it is recommended to enable it only in debug mode. It is not recommended to leave journalling enabled for the application release version or product version, as it can result in extra time and memory overheads.

T.1. DATA EXCHANGE WITH OTHER SYSTEMS

C3D Geometric Kernel Converters provide data exchange with other systems. In its turn, data exchange process includes two tasks.

The first task is import. It means reading a third-party geometric model having one of exchange formats from a file or a stream, and constructing a C3D geometric model from the received data.

The second task is export. It means representing the C3D geometric model in one of exchange formats and recording this representation in the specified stream or file.

The most comprehensive data set that can be passed via converter contains the following:

1. Information about the shape that can be described by bodies, surfaces, wireframes and groups of points.
2. Information about the model structure: specifying the components that can be reused in the model.
3. Item information: item ID, comments, information about its authors, etc.
4. Attributes: visual properties and elementary attributes.
5. Annotation elements: dimensions, technical specifications, and designations.

The C3D Geometric Kernel supports six text formats and two binary formats. Four text formats (ACIS, IGES, STEP, and X_T) and X_B binary format pass information about geometric shape of the simulated object using a boundary representation. Two text formats (STL and VRML) and STL binary format use polygonal representation to pass the geometric shape of the simulated object. The JT binary format provides both boundary and polygonal representations.

T.1.1. Converter Operation Principles

In its work, the converter can use interfaces that can be divided into two groups. The first group includes the interfaces implemented on the C3D side, and the second group contains the interfaces that should be implemented by the user. Furthermore, the C3D geometric kernel provides a standard implementation of all interfaces required to exchange data.

The interfaces correspond to the following concepts:

1. The converter (the IConvertor3D interface) is a special object used to transfer data. It is implemented on the C3D side.
2. Converter properties (IConvertorProperty3D interface) serve to pass the exchange settings to the converter. The interface has a standard implementation.
3. The document (ItModelDocument interface) is an object that contains the whole passed model. The interface has a standard implementation.
4. Part(ItModelPart interface) and assembly (ItModelAssembly interface) are the model parts that can be presented as details or as assembly units. The interfaces have a standard implementation.
5. Instance (ItModelInstance interface) is an object that defines how a component is positioned or reused in the model. The interface has a standard implementation.

A component is a generic concept for a detail and for an assembly. It is assumed that a component can be a minimal unit corresponding to a separate file handled by the end application. The use of components and instances permits to form a model tree, and its components can be reused.

T.1.2. How to Work With the Converter

Converters provide means for export and import of models presented as both MbModel and user-implemented documents Ошибка: источник перекрёстной ссылки не найден.

Group of functions for MbModel exchange using files is located in conv_i_converter.h and declared in the c3d namespace.

MbeConvResType **ImportFromFile** (MbModel & **model**,
const c3d::path_string& **fileName**,
IConvertorProperty3D * *property = 0*,
Ошибка: источник перекрёстной ссылки не найден * *indicator* = 0),

MbeConvResType **ExportIntoFile** (MbModel & **model**,
const c3d::path_string& **fileName**,
IConvertorProperty3D * *property = 0*,
Ошибка: источник перекрёстной ссылки не найден * *indicator* = 0).

Arguments of functions are:

- **model** model which is exported or replaced by imported,
- **fileName** is a full file path,
- *property* are the converter properties,
- *indicator* is read/write progress indicator.

The exchange format's choice is based on file name's extension. The possible disagreement between the **fileName** argument and value of the *property*.FullFilePath method is resolved in the following way. The value of the *property*.FullFilePath is used only if **fileName** is empty and *property* is not null pointer.

The similar functions provide MbModel exchange using memory buffer.

MbeConvResType **ImportFromBuffer** (MbModel & **model**,
const char* **data**,
size_t **length**,
MbeModelExchangeFormat **modelFormat**,
IConvertorProperty3D * *property = 0*,
Ошибка: источник перекрёстной ссылки не найден * *indicator* = 0),

MbeConvResType **ExportIntoBuffer** (MbModel & **model**,
MbeModelExchangeFormat **modelFormat**,
char*& **data**,
size_t& **length**,
IConvertorProperty3D * *property = 0*,
Ошибка: источник перекрёстной ссылки не найден * *indicator* = 0).

Arguments of functions are:

- **model** model which is exported or replaced by imported,
- **modelFormat** – формат обменного файла,
- **data** is a buffer address,
- **length** is a buffer size,
- *property* are the converter properties,
- *indicator* is read/write progress indicator.

The value of *property*.FullFilePath() is used to detect the exchange format only in case when modelFormat is mxf_autodetect and *property* is not null pointer.

To prevent memory leakage the buffer created and filled by **ExportIntoBuffer** must be deallocated by user. The delete[] operator should be for deallocation.

The next function automatically detects file format by extension and uses model document ItModelDocument as a source of an imported model.

MbeConvResType **ImportFromFile** (Ошибка: источник перекрёстной ссылки не найден & **document**,
const c3d::path_string& **fileName**,
IConvertorProperty3D * *property = 0*,
Ошибка: источник перекрёстной ссылки не найден * *indicator* = 0).

Arguments of function are:

- **document** is a document that passes the geometric model and other information,
- **fileName** is a full file path,
- *property* are the converter properties,
- *indicator* is read/write progress indicator.

The Converter functionality is based on methods of the IConvertor3D class.
To get started with the converter, first receive its instance.
The
IConvertor3D * **GetConvertor3D** () function
is defined in global scope and it creates a converter that can import/export models using any supported format. The function has no parameters. If successful, the function returns the converter, otherwise it returns a null pointer.
The function is declared in the conv_i_converter.h file.
If the converter was created, then data are exchanged by calling converter methods. The following converter methods:

MbeConvResType **SATRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
std::iostream * **stream**,
IProgressIndicator * *indicator* = 0),
MbeConvResType **SATWrite** (IConvertorProperty3D& **property**,
ItModelDocument & **document**,
std::iostream * **stream**,
IProgressIndicator * *indicator* = 0),
MbeConvResType **SATRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator* = 0,
MbRefItem * *qeuryStitch* = 0),
MbeConvResType **SATWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator* = 0,
MbRefItem * *qeuryStitch* = 0),
MbeConvResType **IGSRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator* = 0,
MbRefItem * *qeuryStitch* = 0),
MbeConvResType **IGSWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator* = 0,
MbRefItem * *qeuryStitch* = 0),
MbeConvResType **JTRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,

```
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType  JTWrite   ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType  XTRead    ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType  XTWrite   ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType STEPRead   ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType STEPWrite  ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType  STLRead   ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType  STLWrite  ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType VRMLRead   ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
MbeConvResType VRMLWrite  ( IConvertorProperty3D & property,
                          ItModelDocument &          document,
                          IProgressIndicator *       indicator = 0,
                          MbRefItem *                qeuryStitch = 0 ),
```

correspondingly import and export models using SAT, IGES, JT, X_T, STEP, STL, and VRML formats.

The method input parameters are:

- **property** are the converter properties,
- **stream** is a stream that receives or records an exchange format model,
- **document** is a document that passes the geometric model and other information,
- *indicator* is read/write progress indicator,
- *qeuryStitch* is a requestor of surface stitching or measurement units during import (this is an obsolete parameter, it is recommended not to use it; it is better to control stitching using the converter properties).

If successful, the method returns cnv_Success, otherwise the method returns an error code from the MbeConvResType enumeration.

When the work is finished, it is required to delete received converter instance.

The converter inherits the IConvertor3D class interface that is declared in the conv_i_converter.h file.

T.1.3. IConvertorProperty3D Converter Property

A converter property (or simply a property) is one of data exchange method parameters. As single IConvertorProperty3D interface is used to control converters of all formats, it contains both universal methods for all formats and methods specific for some formats. Furthermore, data sets that control export and import processes differ.

The universal methods can be used to get the exchange file path (if a stream is not explicitly set), configure both filtration of sent objects by types and data exchange log.

const c3d::path_string **FullFilePath** () const;
bool **GetIoPermission**(MbeIOPermiss **nPermission**);
void **GetIoPermissions**(std::vector<bool>& **ioPermissions**);
void **LogReport**(ptrdiff_t **id**, eMsgType **msgType**, eMsgDetail **msgText**);

The methods called during export to any format provide information about the document as a whole, about the coordinate system where the geometric model should be oriented, and also request whether it is required to force transform exchange format objects to right coordinate systems while keeping their shape and mutual positions, and length units scale factor all linear dimensions are multiplied by.

bool **GetPropertyString** (MbeConverterStrings **nString**, std::string & **propertyString**);
MbPlacement3D **GetOriginLocation**();
bool **ReplaceLocationsToRight**();
double **LengthUnitsFactor**( );

The methods called during import of files of all formats and corresponds to length unit scale factor all linear dimensions are multiplied by:
double **AppLengthUnitsFactor**( );

The methods called during import of SAT, X_T, X_B, JT, STEP and IGES formats permit or prohibit automatic stitching of separate surfaces to bodies and indicate stitching precision:

bool **EnableAutoStitch**(double& **stitchPrecision**);

During import from IGES and SAT formats, a method is called that permits or prohibits the converter to call the requester:

bool **CanShowMessages**();

Other methods are either specific for export to various formats or were introduced for debugging.

During export to Parasolid (X_T or X_B selection) or STL formats a method is called that determines whether the data would be represented in binary or text format:

bool **IsFileAscii** ();

For export to STEP, the methods with default implementation are called, they define text representation in annotation elements and STEP application protocol that would be used:

eTextForm **GetAnnotationTextRepresentation** ();
MbeImpExpFormat **GetFormat** ();

For export to IGES format a method is called that determines whether the topology information would be exported:

bool **IsOutOnlySurfaces**();

For export to polygonal formats (STL and VRML) the methods with default implementation are called that control triangulation calculation parameters:
MbStepData **TesselationParameters**();
bool **DualSeams**();
void **DualSeams**(bool **dSeams**);

During export into STEP and SAT formats the following method is called to specify the version of format. It's default implementation returns the code of the last supported version.
long int **GetFormatVersion** ();
The following methods are used for debugging:
std::string **GetDocumentName**() const;
bool **IsAssembling** () const;
void **SetIoPermission**(MbeIOPermiss **nPermission**, bool **setF**);
void **SetPropertyString** (MbeConverterStrings **nString**, const std::string & **propertyString**);
bool **ExportComponentsSeparately**().

The C3D geometric kernel provides a standard implementation of the IConvertorProperty3D interface.
The ConvConvertorProperty3D class is a standard implementation of the converter property, Fig. T.1.3.1. The property inherits the IConvertorProperty3D class interface.

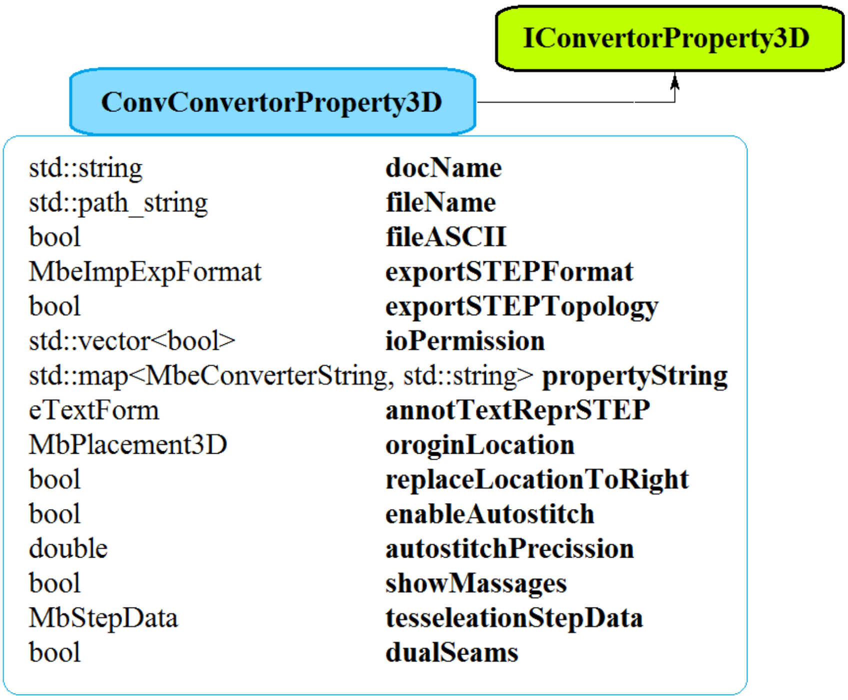


Fig. T.1.3.1

The ConvConvertorProperty3D class stores the following information in its accessible fields: **fileName** is the path to the exchange file; **docName** is the document name; **fileASCII** is a text file export indicator; **exportSTEPFormat** is a STEP export format protocol; **exportIGESTopology** is a flag indicating topology export to IGES format; **ioPermission** is the object type filter; **propertyStrings** is special information about the document; **annotTextReprSTEP** is annotation element text representation; **originLocation** is the local coordinate system of the model; **enableAutostitch** is a face stitching flag; **autostitchPrecision** is a face stitching precision; **tesseleationStepData** are triangulation parameters; **showMessages** is the control parameter for requestor calls; **dualSeams** is seam stitching flag and **logRecords** is a data exchange log code container.
The ConvConvertorProperty3D class fields have the following default values:

- **docName** and **fileName** are empty lines,
- **fileASCII is** true,
- **formatVersion** format version on export (default value is EXPORT_DEFAULT),
- **exportIGESTopology** is true,
- **ioPermission** permits to export/import objects of any type,
- **propertyStrings** is an empty container,
- **annotTextReprSTEP** is exf_TextOnly,
- **originLocation** is a default value,
- **enableAutostitch** is true,
- **autostitchPrecision** is 0.3,
- **tesseleationStepData** is a default value,
- **showMessages** is false,
- **dualSeams** is true,
- **logRecords** is an empty container.

The IConvertorProperty3D interface is declared in the conv_i_converter.h file.
The ConvConvertorProperty3D class is declared in the conv_model_properties.h file.

T.1.4. ItModelDocument Model Document

The ItModelDocument interface is one of the interfaces that permit to send data with complex tree-like structure and many components, and its components are reusable. The document provides two operation modes for a model. In the first one, the model generated by the C3D Geometric Kernel objects is used directly. In the second one, the user implements the ItModelPart, ItModelAssembly, ItModelInstance interfaces and the methods that work with implementations. In its turn, the converter calls interface methods of the returned objects.

The interface methods can be grouped as described below.

The following group of methods was designed for direct work with the model and annotation element container:

void **SetContent**(MbItem* **content**);
MbItem * **GetContent**();
map_of_visual_items **GetAnnotationItems**(eTextForm **form**);
void **SetAnnotationItems**(const map_of_visual_items& **visItems**);

Interface-oriented methods are correspondingly divided into the following subgroups.

The group of methods was designed to generate the model during import:
SPtr<ItModelAssembly> **CreateAssembly**(const std::vector< SPtr<MbItem> > & **componentItems**, const c3d::path_string& **fileName**);
SPtr<ItModelPart> **CreatePart**(const std::vector< SPtr<MbItem> > & **componentItems**, const c3d::path_string& **fileName**);
bool **FinishImport**(IProgressIndicator * **indicator**);

The following group of methods was designed to get linked model components during export:
bool **IsAssembly**();
bool **IsEmpty**();
SPtr<ItModelAssembly> **GetInstanceAssembly**( );
SPtr<ItModelPart> **GetInstancePart**( );

The following methos is used for debugging:
void **OpenDocument**();

It is not known in advance which particular implementation would be used to send data, so both the **SetContent** method and one of the **CreateAssembly** or **CreatePart** methods (depending on import result) are called during import. Export process starts with call of the **GetInstanceAssembly** or **GetInstancePart** method (depending on the result of **IsAssembly** call), and the **GetContent** method is called only if the converter is unable to generate the model.

In the model generated using the C3D Geometric Kernel objects, some MbItem objects correspond to the components. This defines the structure and the content of annotation element container, the latter is handled by **GetAnnotationItems** and **SetAnnotationItems** methods.

There is a standard implementation of the ItModelDocument in the C3D geometric kernel. The C3DModelDocument class implements the interface, and the corresponding implementations of ItModelPart, ItModelAssembly, ItModelInstance interfaces are hidden. For export the model can be set directly using the **SetContent** method or generated in a standard way using ItModelAssembly or ItModelPart calls. After import user can call the **GetContent** to get the generated model.

The ItModelDocument interface is declared in the conv_i_converter.h file. The C3DModelDocument implementation and it's aliases RegularModelDocument and ConvModelDocument are declared in the conv_model_properties.h file.

T.1.5. IProgressIndicator Progress Indicator

The progress indicator (or simply the indicator) provides feedback from the converter to the user. The converter calls various methods when it exchanges data.

The following method sets step range, step size and information displayed about the process:
bool **Initialize**(size_t **range**, size_t **delta**, IStrData & **massage**).

The following method informs that a certain number of steps were completed:

bool **Progress**(size_t **n**).

The method returns a flag indicating whether it is required to continue the operation.

The following method determines whether the operation should be terminated on user request:

bool **IsCancel**().

The method below informs the user that the operation is completed:

void **Success**().

The following method informs the user on abnormal process termination:

void **Stop**().

The user can specify the method that sends information on the need of forced process termination. The user is suggested to implement the following method as a standard one:
void **SetCancel**(bool **cancel**).

The IProgressIndicator interface is declared in the conv_i_converter.h file.

In the test application, the ProgressIndicatorImp class is one of possible indicator implementations.

The ProgressIndicatorImp class is declared and implemented in the test_converter.cpp file of C3D Geometric Kernel test application.

T.1.6. Model Tree Architecture

The following conventions are accepted for a model generated using the C3D Geometric Kernel objects.

- A document is equivalent to a root component (top-level component).
- If a boundary representation format is used, then any component should have a corresponding

MbAssembly type object. If a polygonal representation format is chosen, then MbMesh object can be used as a component. It is expected that any component was assigned certain properties, so each component should have the MbProductInfo and MbPersonOrganizationInfo type attributes.

- The own objects of the component can belong to the MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance and MbPlaneInstance types for boundary representation formats, and to MbMesh type for the polygonal representation formats stored in MbAssembly.
- Component nesting and positioning is implemented using MbInstance objects. MbInstance is stored in the top-level component and it refers to a bottom-level component. Cyclic dependencies are not permitted.

The approach based on generating data exchange model by the C3D Geometric Kernel components has several drawbacks. The most important drawback is that the use of specific objects and configurations impedes the functionality development: for any changes, all users should implement corresponding changes. If model, component, and instance interfaces are implemented on the user side, then the users no longer depend on specific implementation.

In terms of interfaces, the architecture can be described as follows.

- The ItModelDocument contains a top-level component.
- There are two component types: ItModelPart details and ItModelAssembly assemblies. They differ only in interpretation, both types can refer to lower-level components.
- The MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance, and MbPlaneInstance objects can be own objects of the components.
- Component nesting and positioning is implemented by means of ItModelInstance objects. Cyclic dependencies are not permitted.

T.1.7. ItModelInstanceProperties Model Element Properties

ItModelInstanceProperties is a basic class; it is an ancestor of ItModelPart, ItModelAssembly, and ItModelInstance interfaces. The methods declared in this interface can be grouped as follows.

The first group of methods sends information about the component.

The following methods send item ID:

std::string **Name**(),
bool **SetName**(const std::string& **name**).

The following methods send item designations:

std::string **Marking**(),
bool **SetMarking**(const std::string& **marking**).

The following methods send information about item author:

std::string **Author**(),
bool **SetAuthor**(const std::string& **author**).

The following methods send information about the organization:

std::string **Organization**(),
bool **SetOrganization**(const std::string& **organization**).

The following methods send comments:

std::vector< std::string> **GetComments**(),
bool **SetComments**(const std::vector< std::string>& **comments**).

The second group of methods sends visual properties.

The following methods are designed to send own visual properties of component or instance:

bool **GetColor**(MbAttributeContainer& **visual**),
bool **SetColor**(const MbAttributeContainer& **visual**).

The methods below send the visual properties of objects belonging to any element, for example, body faces:

bool **GetColor**(MbAttributeContainer& **visual**, const MbName& **name**),
bool **SetColor**(const MbAttributeContainer& **visual**, const MbName& **name**).

The following methods send visual properties of objects belonging to the component (for example, a body with a given index):

bool **SetColor**(const MbAttributeContainer& **visual**, size_t **index**).

The third group of methods sends technical specifications applied to components. The following methods are included in this group:

void **GetRequirements**(vector_of_annotation& **annot**, eTextForm **textRepr**),
void **SetRequirements**(const vector_of_annotation& **annot**).

T.1.8. ItModelPart and ItModelAssembly Components

The ItModelPart and ItModelAssembly interfaces have a similar set of methods, so you can consider them identical, the only distinction is the interpretation. Component properties, visual properties, and technical specifications are assigned using ItModelInstanceProperties basic class methods.

The **PureFileName**() method is called during export and it returns a line that indicates component file name.

Own objects are handled by the following method group:

void **GetItems**(std::vector< SPtr<MbItem> > & **items**,
MbeGettingItemType **itemType**,
bool **includeInvisible**),
void **AddItems**(const std::vector< SPtr<MbItem> > & **items**).

The **GetItems** method was designed to deliver elements to the container using type and scope filters that are generated on the basis of values returned by the IConvertorProperty3D::**GetIoPermission** method. The **AddItems** method is used for import and it was designed to add elements in the component.

The following methods were designed to work with instances:

SPtr<ItModelInstance> **PrepareInstance**();
SPtr<ItModelInstance> **NextInstance**(bool **includeInvisible**).

The **PrepareInstance** method was designed to insert a component during import. The **NextInstance** method is an instance iterator during export. It is assumed that the iterator is ready to deliver instances from the start. The cycle ends when a null pointer is returned. During the walk-through, the scope filter value corresponds to the result of calling the IConvertorProperty3D::**GetIoPermission** (iop_wInvisible) method.

The following pair of methods gets and sets annotation elements during export and import correspondingly:

vector_of_annotation **GetAnnotationItems**(eTextForm **textForm**,
bool **includeInvisible**),
void **SetAnnotationItems**(const vector_of_annotation & **annot**).

The **GetAnnotationItems**(eTextForm **textForm**) method is not called in converters.
The ItModelPart and ItModelAssembly interfaces are declared in the conv_model_properties.h file.

T.1.9. ItModelInstance Instances

For historical reasons, the ItModelInstance instance interface was inherited from ItModelInstanceProperties. It was preserved for compatibility. According to this approach, only its own visual properties can be assigned to an instance.

The void * **GetId**() method permits to differentiate the instances of equivalent components. If various instances return the same value, then the equivalence is determined based on content analysis results. During export, the converter is eager to optimize the model by reuse of equivalent components. In order to reduce calculation costs during export, a user can also provide information which components are definitely non-equivalent.

The following methods are used during export to get such instance data, as local instance coordinate system, instance contents, and instance emptiness:

bool **GetPlacement**(MbPlacement3D & **place**),
bool **IsAssembly**(),
bool **IsEmpty**().

If the instance is non-empty, then one of the following methods is called based on the result of the **IsAssembly**() method:

SPtr<ItModelAssembly> **GetInstanceAssembly**( ),
SPtr<ItModelPart> **GetInstancePart**( ).

These methods return the interface of the component referenced by the instance.
The following methods create components during import:

SPtr<ItModelAssembly> **CreateAssembly**(const MbPlacement3D &**place**,
const std::vector<SPtr<MbItem> > & **componentItems**,
const c3d::path_string& **fileName**);
SPtr<ItModelPart> **CreatePart**(const MbPlacement3D &**place**,
const std::vector< SPtr<MbItem> > & **componentItems**,
const c3d::path_string& **fileName**);

As a result, these methods return a component interface, its location in top-level component is described by **place** local coordinate system, the component is filled with elements from the **componentItems** container, and **fileName** contains a corresponding file name.

The methods listed below add previously created components to an instance:

bool **SetAssembly**(const MbPlacement3D & **place**, const ItModelAssembly * **existing**),
bool **SetPart**(const MbPlacement3D & **place**, const ItModelPart * **existing**).

The location of the **existing** component is defined by the **place** local coordinate system.
ItModelInstance interfaces are declared in the conv_model_properties.h file.

T.2. BOUNDARY REPRESENTATION CONVERTERS

The C3D Geometric Kernel can read ACIS, IGES, STEP, X_T, and X_B files, construct an internal model on their basis, and write the internal model using the listed formats. The ACIS, IGES, STEP, and X_T text formats and the X_B binary format pass information about the geometric shape of the simulated object using boundary representation. The bodies represented by faces, edges, and vertexes are used in the boundary representation to describe the geometric shape.

T.2.1. General Description of the Boundary Representation Converter Functions

Special functions defined in the global scope differ from the converter methods (described in Item T.1.2) in that they have the following limitations that are minor in most cases: they do not work with streams and they do not accept surface stitching requester as an argument.

All functions have the same signature type: they accept both the IConvertorProperty3D converter properties (described in Item T.1.3. IConvertorProperty3D Converter Property) and ItModelDocument model document (described in Item T.1.4. ItModelDocument Model Document) as arguments, and also the IProgressIndicator progress indicator (described in Item T.1.5. IProgressIndicator Progress Indicator) as an optional parameter. The behavior of all functions is also similar: all functions receive a converter instance, call one of its methods, and delete the converter when they finish. If successful, the functions return cnv_Success, otherwise they return an error code from the MbeConvResType enumeration.

T.2.2. General Information About Boundary Representation Converter Parameters

When the converter sends data, it calls the **FullFilePath**, **GetIoPermission**, **GetIoPermissions**, **LogReport** methods of the IConvertorProperty3D interface.

For import the converter calls the **EnableAutoStitch** method of the IConvertorProperty3D interface.

For export the converter calls the **GetPropertyString**, **GetOriginLocation**, **ReplaceLocationsToRight** methods of the IConvertorProperty3D interface.

In standard implementation of the ConvConvertorProperty3D interface (it is described in Item T.1.3. IConvertorProperty3D Converter Property), the **fileName** field should contain correct full path to the exchange file. Default values of other fields guarantee that the methods work correctly. For export the file will be created or automatically rewritten if there are no limitations from the file system.

A standard RegularModelDocument or ConvModelDocument implementation can be selected as a model document. It is preferable to use the latter one if you need to pass item data using the STEP exchange format with as much details as possible.

It is permitted to pass a null pointer as a progress indicator.

T.2.3. Importing Models in SAT Format

The function

MbeConvResType **SATRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

imports SAT geometric model (up to version 22.0).

The input and output method parameters are:

- **property** are the converter properties,

- **document** is the model document,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters. Though the SAT converter calls the IConvertorProperty3D::**CanShowMessages** method, the returned value does not affect anything as there is no scale requester.

The geometric model is either a single component containing MbSolid objects, or one-level assembly having components that contain bodies. Designations of non-root components, visual properties and elementary attributes of faces, edges, and vertexes are imported.

SATRead function example use is absent. To demonstrate import from SAT, the test application uses IConvertor3D converter method, where a stream is explicitly defined.

T.2.4. Exporting a Model to SAT Format

The function

MbeConvResType **SATWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

exports the geometric model to the SAT 2.0 format.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters.

For export, models with several components are transformed in one-level assembly that keeps its shape. Reused elements are duplicated. MbSolid objects only are exported to SAT format. Designations of non-root components, visual properties and elementary attributes of faces, edges, and vertexes are exported.

SATWrite function example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.2.5. Importing a IGES Model

The function

MbeConvResType **IGSRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

imports IGES 5.3 geometric models.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters. Though the IGES converter calls the IConvertorProperty3D::**CanShowMessages** method, the returned value does not affect anything as there is no surface stitching requester.

The imported model can have arbitrary nesting degree and it can contain MbSolid, MbWireFrame, MbPointFrame type objects. Information about root component author and organization, designations of non-root components, visual properties of body faces and wireframe structure edges are imported.

IGSRead method example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.2.6. Exporting a Model to IGES Format

The function

MbeConvResType **IGSWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

exports the geometric model to the IGES 5.3 format.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters.

The exported model can have arbitrary nesting degree and it can contain MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance, MbPlaneInstance objects. Information about root component author and organization, designations of non-root components, visual properties of body faces and wireframe structure edges are exported.

IGSWrite method example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.2.7. Importing X_T and X_B Models

The function

MbeConvResType **XTRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

imports the geometric model in Parasolid formats (text-based X_T format and binary X_B format). Versions up to 25.0 are supported.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters.

The imported model can have arbitrary nesting degree and it can contain MbSolid type objects. Component designations, visual properties, and elementary attributes of body faces, edges, and vertexes are imported.

XTRead method example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.2.8. Exporting Models to X_T and X_B Formats

The function

MbeConvResType **XTWrite** (IConvertorProperty3D & **property**,

ItModelDocument & **document**,
IProgressIndicator * *indicator*)

exports the geometric model to X_T text format or X_B binary format (version 10.0).

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General export settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters. The converter generates either text (*.X_T) or binary (*.X_B) file depending on the value returned by the IConvertorProperty3D::**IsFileAscii** method.

The exported model can have arbitrary nesting degree and it can contain MbSolid type objects. Component designations, as well as visual properties of faces, edges, and body vertexes are exported.

XTWrite method example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.2.9. Importing STEP Models

The function

MbeConvResType **STEPRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

imports geometric models that have STEP format. 203 and 214 application protocols are supported.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters.

The imported model can have arbitrary nesting degree and it can contain MbSolid, MbWireFrame, MbPointFrame, and MbSpaceInstance objects. Component properties, visual properties and elementary attributes of body faces, edges, and vertexes, material densities, as well as dimensions and technical specifications are imported.

STEPRead method example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.2.10. Exporting Model to STEP Format

The function

MbeConvResType **STEPWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

exports the geometric model to STEP format. 203 and 214 application protocols are supported.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is a document that passes the geometric model and other information,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.2.2. General Information About Boundary Representation Converter Parameters. The converter generates 203 or 214 application protocol data based on the value returned by the IConvertorProperty3D::**GetFormat** method. The value returned by the IConvertorProperty3D::**GetAnnotationTextRepresentation** method is passed as an argument when ItModelDetail::**GetAnnotationItems**, ItModelAssembly::**GetAnnotationItems**, ItModelDocument::**GetAnnotationItems**, and ItModelInstanceProperties::**GetRequirements** methods are called.

The exported model can have arbitrary nesting degree and it can contain MbSolid, MbWireFrame, MbPointFrame, and MbSpaceInstance type objects. Component properties, visual properties and elementary attributes of body faces, edges, and vertexes as well as material densities are exported. Component dimensions and their technical specifications are also exported.

STEPWrite method example use is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.3. POLYGONAL REPRESENTATION CONVERTERS

The C3D Geometric Kernel can read text and binary STL files and VRML files, construct an internal model on their basis, and write the internal model using these formats. The STL and VRML text formats and the STL binary format use polygonal representation to pass the geometric shape of the simulated object. Triangles and polygons are used in polygonal representation to describe the geometric shape.

T.3.1. General Description of Polygonal Representation Converter Functions

The special functions described in a global scope have only one distinction from converter methods (described in Section T.1.2): they do not accept surface stitching requester as an argument.

All functions have the same signature type: they accept both the IConvertorProperty3D converter properties (described in Item T.1.3. IConvertorProperty3D Converter Property) and ItModelDocument model document (described in Item T.1.4. ItModelDocument Model Document) as arguments, and also the IProgressIndicator progress indicator (described in Item T.1.5. IProgressIndicator Progress Indicator) as an optional parameter. The behavior of all functions is also similar: all functions receive a converter instance, call one of its methods, and delete the converter when they finish. If successful, the functions return cnv_Success, otherwise they return an error code from the MbeConvResType enumeration.

T.3.2. General Information About Polygonal Representation Converter Parameters

When the converter sends data, it calls the **FullFilePath**, **GetIoPermission**, **GetIoPermissions**, **LogReport** methods of the IConvertorProperty3D interface.

For export, the converter calls the **ReplaceLocationsToRight**, **TesselationParameters**, **DualSeams** methods of the IConvertorProperty3D interface. Triangulation calculation parameters are used to construct a polygonal representation of all the exported objects except MbMesh.

In standard implementation of the ConvConvertorProperty3D interface (it is described in Item T.1.3. IConvertorProperty3D Converter Property), the **fileName** field should contain correct full path to the exchange file. Default values of other fields guarantee that the methods would work correctly. For export the file would be created or automatically rewritten if there are no limitations from the file system.

If you plan to use one of standard model document implementation, then you should make a choice based on whether you plan to use either interface methods or a model generated using the C3D geometric kernel objects. The RegularModelDocument implementation should be used in the first case, and ConvModelDocument implementation should be used in the second case.

It is permitted to pass a null pointer as a progress indicator.

T.3.3. Importing STL Models

The

MbeConvResType **STLRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

function imports binary or text geometric model in the STL format.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is the model document,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the

MbeConvResType enumeration.

General import settings are described in Item T.3.2. General Information About Polygonal Representation Converter Parameters. The imported model has one component that contains one MbMesh type object. STL format does not support passing polygons.

STLRead method usage example is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.3.4. Exporting the Model to STL Format

The

MbeConvResType **STLWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

function exports binary or text geometric model to STL format.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is the model document,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General export settings are described in Item T.3.2. General Information About Polygonal Representation Converter Parameters. Based on the value returned by the IConvertorProperty3D::**IsFileAscii** method, the converter generates text or binary STL file. The STL converter also calls the IConvertorProperty3D::**GetOriginLocation** method.

The model shape is saved during export (accurate up to conversion to polygonal representation), but both the polygons and the information about its structure are completely lost. Reused elements are duplicated.

STLWrite method usage example is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.3.5. Importing VRML Model

The

MbeConvResType **VRMLRead** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)

function imports VRML 2.0 geometric models.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is the model document,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General import settings are described in Item T.3.2. General Information About Polygonal Representation Converter Parameters.

The imported model can have any nesting degree and it can contain MbMesh objects. The visual properties of grids are imported.

An example of **VRMLRead** method is given in the test_converter.cpp file of the C3D Geometric Kernel test application.

T.3.6. Exporting a Model to VRML Format

The
MbeConvResType **VRMLWrite** (IConvertorProperty3D & **property**,
ItModelDocument & **document**,
IProgressIndicator * *indicator*)
function exports the geometric model to the VRML 2.0 format.

The input and output method parameters are:

- **property** are the converter properties,
- **document** is the model document,
- *indicator* is a read/write progress indicator.

If successful, the function returns cnv_Success, otherwise it returns an error code from the MbeConvResType enumeration.

General export settings are described in Item T.3.2. General Information About Polygonal Representation Converter Parameters.

The model shape (accurate up to conversion to polygonal representation), information about its structure, and the visual properties of grids are preserved during export.

VRMLWrite method usage example is given in the test_converter.cpp file of the C3D Geometric Kernel test application.