Developer manual

C3D Labs

2026

Contents

1 General_Information **1**
- 1.1 C3D Toolkit . . . 1
- 1.2 Structure and Distinctive Features . . . 1
- 1.3 C3D API . . . 2
- 1.4 Functionality . . . 3
- 1.5 Theoretical Foundations . . . 4

2 Quick Start **5**
- 2.1 Package . . . 5
- 2.2 Test Application . . . 5
- 2.3 Development in the .NET Environment . . . 7

3 C3D Modeler **9**
- 3.1 METHODS USED TO BODIES CONSTRUCTING . . . 9
- 3.2 OPERATIONS ON BODIES . . . 103
- 3.3 TWO-DIMENSIONAL CURVE CONSTRUCTION METHODS . . . 239
- 3.4 CURVE CONSTRUCTION METHODS . . . 256
- 3.5 SURFACE CONSTRUCTION METHODS . . . 270
- 3.6 DIRECT MODELING METHODS . . . 286
- 3.7 SHEET METAL BODY CONSTRUCTION METHODS . . . 316
- 3.8 AUXILIARY METHODS . . . 328
- 3.9 ELEMENTARY OBJECTS . . . 337
- 3.10 GEOMETRICAL OBJECTS . . . 345
- 3.11 TWO-DIMENSIONAL CURVES . . . 352
- 3.12 CURVES . . . 356
- 3.13 SURFACES . . . 390
- 3.14 SPECIAL OBJECTS . . . 431
- 3.15 TOPOLOGICAL OBJECTS . . . 443
- 3.16 OBJECTS OF GEOMETRIC MODEL . . . 462
- 3.17 MULTITHREADING . . . 483
- 3.18 FORMAT C3D . . . 490
- 3.19 ATTRIBUTES . . . 507
- 3.20 CONSTRUCTING TRIANGULATION . . . 512
- 3.21 CONSTRUCTING FLAT PROJECTIONS . . . 523
- 3.22 CALCULATION OF INERTIAL CHARACTERISTICS . . . 530

4 C3D B-Shaper **539**

4.1 Automatic Shell Recognition Mode by Polygon Mesh . . . 540
4.2 MbMeshProcessor Class – Shell Creation . . . 541
4.3 Recognition Tolerance . . . 542
4.4 Polygon Mesh Segmentation Editing . . . 543
4.5 Surface Reconstruction on a Segment . . . 544

5 C3D PolyShaper 545
5.1 Features of Implementing a Doubly Linked List of Edges . . . 545
5.2 Mesh Iterators . . . 547
5.3 Segmentation and Segmentation Iterators . . . 552
5.4 The Topology of the MbPolymeshTopo Polygon Object . . . 556
5.5 Mesh and Segmentation Attributes . . . 572
5.6 Edge-Dependent Attributes . . . 573

6 C3D Collision Detection 577
6.1 Detecting the Intersection of Two Bodies . . . 577
6.2 Determining Collisions in the Set of Bodies . . . 578
6.3 Collision Detection Queries . . . 581
6.4 Configuring a Collision Detection Query . . . 582
6.5 Grouping Bodies Included in the Control Set . . . 584

7 C3D Solver 585
7.1 TWO-DIMENSIONAL GEOMETRIC SOLVER . . . 585
7.2 TWO-DIMENSIONAL DIMENSIONS . . . 601
7.3 TWO-DIMENSIONAL LOGICAL CONSTRAINTS . . . 612
7.4 CALCULATION OF TWO-DIMENSIONAL CONSTRAINTS . . . 621
7.5 TWO-DIMENSIONAL SPLINES AND PARAMETRIC CURVES . . . 626
7.6 THREE-DIMENSIONAL GEOMETRIC SOLVER . . . 627

8 C3D Converter 653
8.1 Converter Operation Principles . . . 654
8.2 How to Work with the Converter . . . 654
8.3 Exchange Format Features . . . 661
8.4 IConvertorProperty3D Converter Property . . . 661
8.5 ItModelDocument Model Document . . . 664
8.6 IProgressIndicator Progress Indicator . . . 666
8.7 Model Tree Architecture . . . 667
8.8 Product Attributes . . . 672
8.9 Model Configuration Import . . . 673
8.10 C3D Converter Plugins . . . 674
8.11 IFC Import . . . 674
8.12 Import of Construction Format Metadata . . . 675
8.13 Auxiliary Converter Methods . . . 675
8.14 Creating a Solid from User Objects . . . 676

9 Reference Materials 679
9.1 Operation Result Codes . . . 679
9.2 Blend Surface Construction Methods (Fillet or Chamfer) . . . 687

CHAPTER 1

General_Information

1.1 C3D Toolkit

C3D Toolkit is a software product that includes a geometric kernel, a parametric kernel, a converter, and modules for visualization and transformation of geometric models.

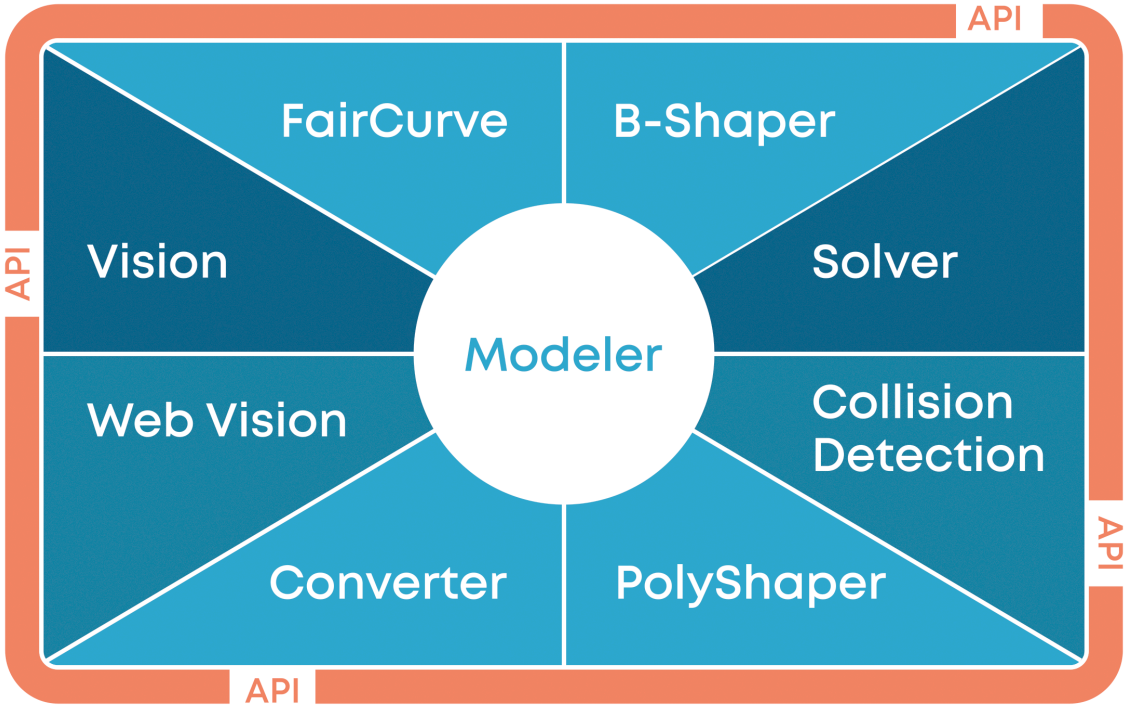
C3D Toolkit is intended for use in computer-aided design systems as a software component.

C3D Toolkit provides a software implementation of mathematical methods for constructing numerical models of geometry of real and imaginary objects, as well as mathematical methods for maintaining these models. Numerical models are used in systems that perform design (Computer-Aided Design), calculating (Computer-Aided Engineering) and manufacturing (Computer-Aided Manufacturing) of modeled objects. Numerical models of geometry of real and imaginary objects are called geometric models.

A geometric model contains a description of the shape of the modeled object and a description of the relationships between the elements of the model. In addition, a geometric model includes its construction history, which stores the methods and the sequence of constructing the model, and the elements of the geometric model are provided with attributes that carry information about physical, technological and other properties of the elements.

1.2 Structure and Distinctive Features

C3D Toolkit consists of five modules shown in the figure: the C3D Modeler geometric kernel, the C3D Solver parametric kernel, the C3D Converter data exchange module, the C3D Vision visualization module, and C3D B-Shaper, which converts polygonal models into solid models with boundary representation (B-Rep).



The C3D Modeler geometric kernel constructs a geometric model, edits the model by modifying its internal data, builds triangulation, computes the model's inertial properties, generates flat projections of the model, and detects collisions between model elements.

The C3D Solver parametric kernel maintains relationships between elements of a geometric model, which makes it possible to edit the model by synchronously modifying its elements, to construct similar models, and to simulate mechanisms.

C3D Converter provides data exchange of geometric models with other systems.

The C3D Vision visualization module provides high-quality visualization of a geometric model and supports interaction with the user interface of engineering software.

C3D B-Shaper converts polygonal models into solid models with boundary representation.

Having a geometric kernel, a parametric kernel, and a converter in a single software product is one of the distinctive features of C3D Toolkit.

1.3 C3D API

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

The key features of the C3D API are openness and extensibility. C3D provides direct access to objects, which allows developers 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 the API should not change significantly over the longest possible period of time. Backward compatibility means that code written using a previous version works functionally the same with the next version of the API.

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 ease of use of the API for users.

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.

1.4 Functionality

The C3D Modeler geometric kernel constructs a geometric model that contains a description of the shape of the modeled object, a description of relations between elements of the geometric model, the model construction history, and attributes of geometric model elements.

To describe the shape of the modeled object, the C3D Modeler geometric kernel uses boundary representation (Boundary Representation). The geometric kernel also supports polygonal representation (Polygonal Representation). To construct a geometric model, it uses solid modeling methods (Solid Modeling), surface modeling methods (Surface Modeling), direct modeling methods (Direct Modeling), and sheet metal modeling methods (Sheet Metal Modeling).

The methods used to construct a model, their sequence, and the required source data are stored in the construction log. The construction log makes it possible to edit a geometric model by completely rebuilding the model with new parameters.

Additional information about geometric model elements is stored in attributes. Objects of a geometric model as well as individual elements of these objects are supplied with attributes.

The C3D Modeler geometric kernel constructs a polygonal model based on its boundary representation. The polygonal model is constructed by triangulating geometric model elements and is used for visualization and computations. In addition, the geometric kernel constructs planar projections of a geometric model (Mapping), computes inertia properties of a geometric model (Inertia Properties), and detects collisions between model elements (Collision Detection).

The C3D Solver parametric kernel imposes variational relations on geometric model elements. These relations are called geometric constraints (Geometric Constraints). Geometric constraints work with three-dimensional and two-dimensional objects of a geometric model. Geometric constraints represent conditions applied to model elements and expressed by equations. Geometric constraints make it possible to create assemblies by applying dependencies to their component parts. Dimensions of one or several parts can be related by equations, which makes it possible to edit geometric model elements synchronously. If almost all degrees of freedom of a geometric model are constrained and a law of variation is defined for the remaining degrees of freedom, then the geometric model can be moved and a mechanism can be simulated.

The C3D Converter module provides data exchange with other systems by reading and writing a geometric model in ACIS, IGES, STEP, X_T, VRML, OBJ, SAT, X_B, STL, JT, 3MF, and SAB formats. ACIS, IGES, STEP, X_T, SAT, X_B, and SAB formats transfer information about the geometric shape of a modeled object using boundary representation. STL, VRML, OBJ, and 3MF formats transfer the geometric shape of a modeled object using polygonal representation. JT format can store a model both in boundary representation and in polygonal representation.

The C3D Vision visualization module is responsible for visual representation of geometric models and for operation of a graphical user interface of an engineering application. The component controls the rendering quality of geometric models using mathematical, software, and

hardware tools, which increases the performance of software when it works with large assemblies. Using C3D Vision in development opens up a number of new capabilities for managing three-dimensional scenes, makes it possible to use an existing construction tree of 3D models, animation, interactive tools for manipulating a scene in the “user–computer” mode, as well as virtual devices that are an integral part of a modern engineering software interface.

The C3D B-Shaper module converts polygonal models into solid models with boundary representation. As a result of executing the algorithms, a user can edit the obtained model with familiar CAD system tools while significantly reducing data complexity and size.

1.5 Theoretical Foundations

The boundary representation used by C3D provides an exact description of the geometric shape of a modeled object. The geometric shape in C3D is described by a set of faces that lie on the boundary separating the interior of the modeled object from the rest of space. Faces are curved surfaces that are joined to each other along their edges. The edges of faces can have a complex shape. The formation and joining of faces are performed during model construction. These operations are provided by the model construction methods and the data organization in C3D.

Geometric constraints that describe relationships between model elements and other conditions are formulated as equations. To find a solution that satisfies these equations, the C3D Solver parametric kernel uses a variational approach. The variational approach ensures that all geometric constraints are treated equally.

Using triangulation, the boundary representation makes it possible to construct a polygonal representation of the model, which is used for visualization and geometric computations. Polygonal objects consist of triangular and quadrilateral patches that approximate faces, and polylines that approximate edges. In the C3D Modeler geometric kernel, triangulation is performed according to the Delaunay principle in the parameter space of surfaces.

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

The mathematical objects, methods, and algorithms used in C3D are described in the book by N. N. Golovanov, “Geometric Modeling”, 2nd edition, Moscow: DMK Press, 2024.

CHAPTER 2

Quick Start

2.1 Package

The C3D distribution package includes the library files *c3d.lib*, *c3d.dll*, *libc3d.so*, *libc3d.dylib* and a set of header files *Include*.h*. On the Windows operating system, the library files are built in the 32bit/64bit, Unicode/Multibyte, Debug/Release configurations for the Visual Studio 2019 and Visual Studio 2022 development environments. On the Linux operating system, the library files are built with the GCC and Clang compilers in the 64bit, Unicode, Debug/Release configurations. On the macOS operating system, the library files are built with the Clang compiler in the 64bit, Unicode, Debug/Release configurations. On the FreeBSD operating system, the library files are built with the Clang compiler in the amd64, Unicode/Multibyte, Release/Debug configurations.

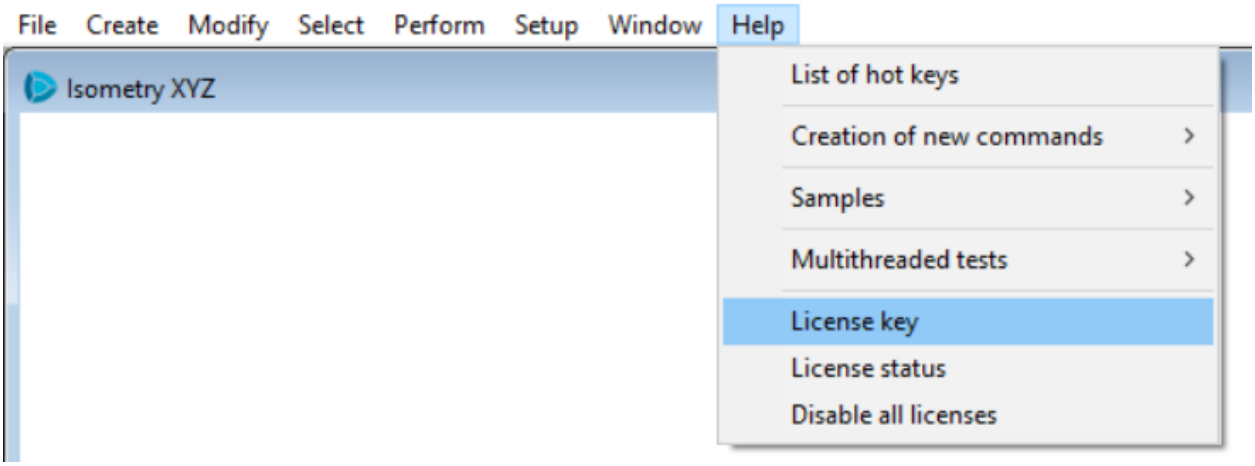
The C3D documentation is generated from the *Include*.h* header files. The header files contain descriptions of C3D objects and methods in Russian and English. Descriptions of C3D objects and methods are also provided in this manual. The *Changes.txt* file contains information about interface changes of the software product.

The distribution package includes a wrapper for C3D that allows using the .NET technology when developing applications in the C# language.

The C3D distribution also includes the *test.exe* application for the Windows system that demonstrates the C3D capabilities, the source code of this application, the *CMakeLists.txt* file for generating the application project, and a set of model files.

2.2 Test Application

The *Example\Demo* directory contains a Windows-ready test application *test.exe* that demonstrates the capabilities of the geometric kernel *c3d.dll*. After starting *test.exe*, you must enter the license key and signature by selecting the “Help -> License key” menu item:



The *Example\Models* directory contains sample models as *.c3d*, *.stp*, *.x_t*, and *.stl* files that can be used by the *test.exe* test application.

The *Example\Source* directory contains the source code of the *test.exe* test application, as well as the *CMakeLists.txt* file for generating the project/solution of the test application.

To compile the C3D geometric kernel test application, follow these steps:

1. Create a new directory in a convenient location, for example “TestApp”.
2. In the <C3D> directory, select an archive that matches your development environment.
3. Copy the <Include>, <Debug>, and <Release> directories from the selected archive to the new directory.
4. Copy the <Source> directory from the <Example> archive to the new directory.
5. Make sure that the new directory contains four folders: <Debug>, <Include>, <Release>, and <Source>.
6. Install CMake, selecting the “Add CMake to the system PATH for all users” option during installation.
7. Create a test application project/solution by performing the following steps:
   - Run CMake version 3.24 or later.
   - For “Where is the source code”, specify the <path_to_testapp>\TestApp\Source directory.
   - For “Where to build the binaries”, specify the <path_to_testapp>\TestApp\Build directory.
   - Click the “Configure” button to configure the project/solution.
   - When prompted “Create Directory”, confirm creation of the <path_to_testapp>\TestApp\Build directory.
   - When prompted “Specify the generator for this project”, select the configuration of the development environment that corresponds to your C3D version.
   - Click the “Generate” button to generate the project/solution files.

Note: if the directory with the Qt5 distribution has been added to PATH, you can enable the “MathTest_QT” option in CMake to create a project that uses the Qt5 library instead of the Win32 API to create the application window.

8. On Windows, open the generated TestApp\Build\Test.sln test application solution in your development environment.
9. On Linux, you can use the following commands:

```
cmake -S Source -B Build -DCMAKE_BUILD_TYPE=Release  # optional, equivalent to␣
↪step 7
cmake --build Build -- -j4 # Qt5 packages must be preinstalled, for example on␣
↪Ubuntu with "sudo apt install qtbase5-dev qt5-qmake"
```

10. If you encounter issues when using CMake, you can use the ready-made Test.vcxproj project located in the Example directory (the project is prepared for MS Visual Studio, Unicode configuration, MS Windows). To build the project, copy the Test.vcxproj and Test.vcxproj.filters project files to the directory created in step 1, and open the Test.vcxproj test application project in your development environment.
11. To activate the C3D geometric kernel, you must write the received license key to the *c3d.lic* file. On Windows, the file is located in the USERPROFILE directory (C:\Users\user_name). On UNIX systems, the file is located in the HOME directory (/home/user_name).
12. After compilation, run the generated *test.exe* test application from the \Debug or \Release directory.
13. If you need to build the test application using Conan packages of the kernel and its examples (the c3d and c3d-examples packages), you can use the short instruction in the *conanfile-examples-usage.py* file in the \Source directory. To build the project, you must specify the kernel package version number.

2.3 Development in the .NET Environment

C3D can run in the .NET environment. To develop applications in the .NET environment, you must use the wrapper supplied with C3D.

The C3D wrapper is a dll file *NetC3D.dll* built on the .NET Framework 4.5.2 platform in 32bit/64bit, Debug/Release configurations, and for the Visual Studio 2019 and Visual Studio 2022 development environments. The library is compiled with “Strong Name” signing enabled.

To use C3D in C# applications, follow these steps:

1. Select the *NetC3D.dll* file of the required configuration from the C3D distribution: 32bit/64bit, Debug/Release for one of the development environments: Visual Studio 2019, Visual Studio 2022.
2. Place the *c3d.dll* file from the same distribution, with the same configuration and development environment (32bit/64bit, Debug/Release, Visual Studio 2019, Visual Studio 2022), into the same directory as *NetC3D.dll*.
3. Add the *NetC3D.dll* file to the project under development: *References -> Add Reference -> Browse...*, then select the *NetC3D.dll* file.
4. Before calling functions from *NetC3D.dll*, you must enter the license key and signature. One possible way to input the key and signature is as follows:

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

where *C3Dkey* and *C3Dsignature* are environment variables that store the key and signature.

CHAPTER 3

C3D Modeler

The C3D Modeler module performs all geometric computations required to construct 2D sketches and 3D models. To describe the shape of a modeled object, C3D Modeler uses a boundary representation of geometry, where the model is built from three-dimensional solids created with surfaces and curves. The solids are then grouped into assemblies, and these assemblies are used to form assemblies of the next level.

Along with the boundary representation of the model, C3D Modeler supports a polygonal representation of geometry. Elements of polygonal models are approximation objects composed of plates and polygons. Such models are used for calculations and visualization.

Objects of a geometric model, such as B-Rep solids, polygonal objects, and wireframe models, are provided with construction histories that store the methods, their order, and the input data required to construct the objects.

A construction history allows you to edit a geometric model and rebuild the model with new parameters. Information about the model geometry in the form of surfaces, curves, and points is stored in topological elements of model objects such as faces, edges, and vertices.

Additional information for each element of a model object and for the object as a whole can be stored in them as attributes.

For license activation of the module, call the EnableMathModules() function with the appropriate parameters.

3.1 METHODS USED TO BODIES CONSTRUCTING

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

3.1.1 Constructing an Elementary Body

The function

```
MbResultType ElementarySolid( const MbElementarySolidParams  & params,
                                    MbElementarySolidResults & results );
```

constructs an elementary body based on specified points and the type of body being created (see Table 1), as well as on an elementary surface (sphere, torus, cone, cylinder).

The input parameter is an object **params** of the MbElementarySolidParams class (see the definition in *op_shell_parameter.h*), which contains the source data for the body creation operation.

The output parameter of the function is the **results** object of the MbElementarySolidResults class, derived from the MbOperationResults class (see the definition in *op_shell_parameter.h*), which contains the constructed body and generalized results of the function.

The function returns the result code of the operation from the enumeration MbResultType.

The function is declared in the *action_solid.h* file.

Constructing an Elementary Body Based on Control Points

For constructing solids based on control points, the MbElementarySolidParams class defines the following constructor to initialize the **params** object with operation parameters (see *op_shell_parameter.h*):

```
MbElementarySolidParams( const ElementaryShellType      & solidType,
                         const c3d::SpacePointsVector   & points,
                         const MbSNameMaker             & operNames );
```

with parameters:

- **solidType** — the type of body being created;
- **points** — the set of points;
- **operNames** — the operation's namer.

Different types of bodies require different numbers of control points. Table 1 shows the number of control points required to create a body of **solidType** type:

Table 1 — Number of control points for various types of elementary bodies

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_Cone	truncated cone	4 points
et_Block	block	4 points
et_Wedge	wedge	4 points
et_Prism	prism	the number of base vertices + 1 point
et_Pyramid	pyramid	the number of base vertices + 1 point
et_Plate	plate	4 points
et_Icosahedron	icosahedron	3 points
et_Polyhedron	polyhedron	3 points
et_Tetrapipe	tetra-pipe	3 points
et_Octapipe	octa-pipe	3 points

For constructing a sphere, **points**[0] defines the center of the sphere, **points**[1] defines the direction of the **axisZ** axis in the sphere's local coordinate system, **points**[2] together with the previous points defines the plane of the **axisX** and **axisZ** axes in the sphere's local coordinate system. The distance between **points**[0] and **points**[2] defines the radius of the sphere (see Fig. 1):

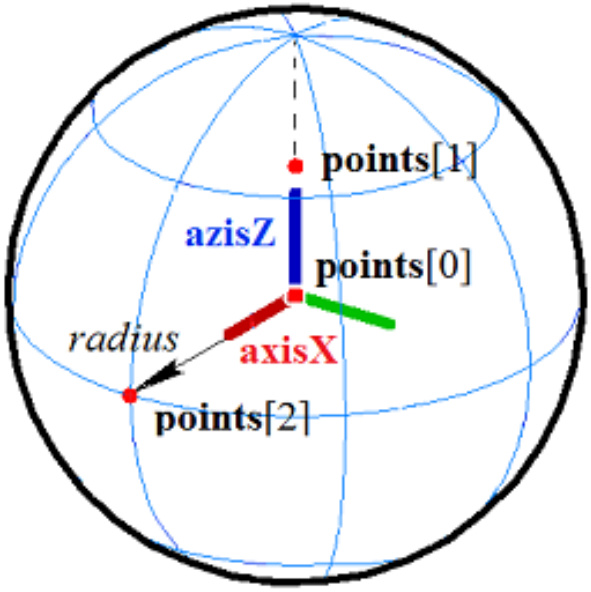


Fig. 1

For constructing a torus, **points**[0] defines the center of the torus, **points**[1] defines the direction of the **axisX** axis in the torus's local coordinate system, **points**[2] together with the previous points defines the plane of the **axisX** and **axisZ** axes in the torus's local coordinate system. The distance between **points**[0] and **points**[1] defines the major torus radius; the distance between **points**[1] and **points**[2] defines the minor torus radius (Fig. 2):

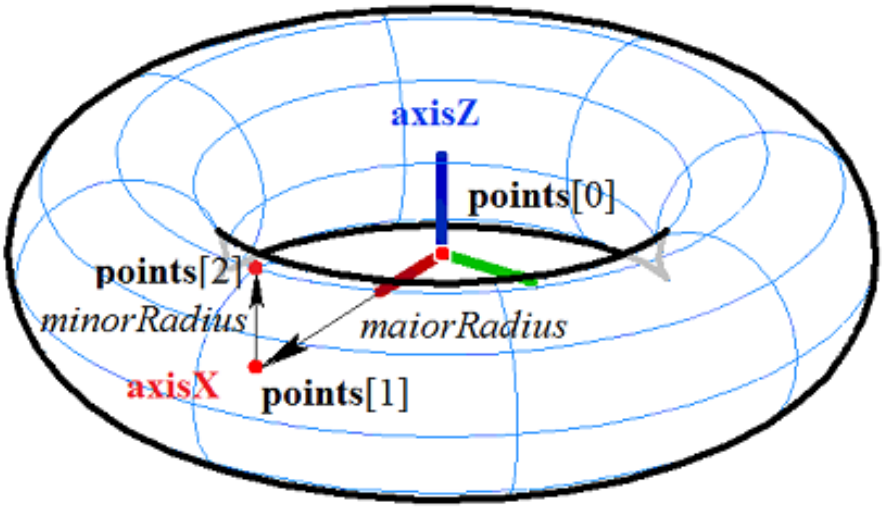


Fig. 2

For constructing a cylinder, **points**[0] defines the center of the cylinder's lower base, **points**[1] defines the center of the cylinder's upper base and the direction of the **axisZ** axis in the cylinder's local coordinate system, **points**[2] together with the previous points defines the plane of the **axisX** and **axisZ** axes in the cylinder's local coordinate system. The distance between **points**[0] and **points**[1] defines the cylinder height; the distance from the **axisZ** axis to **points**[2] defines the cylinder radius (Fig. 3):

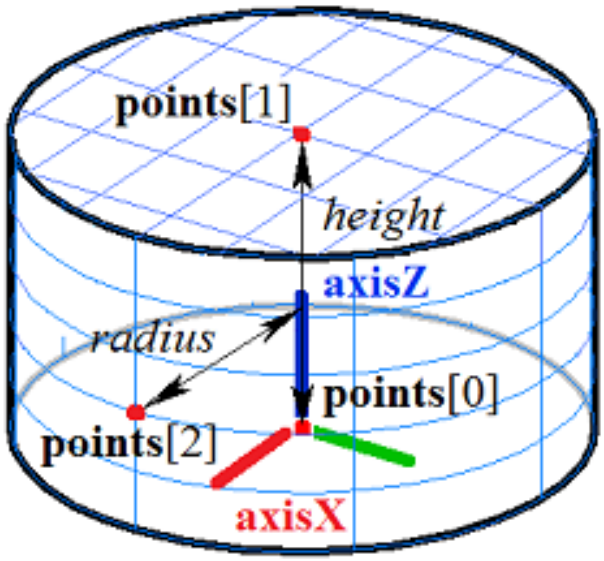


Fig. 3

For constructing a cone, **points**[0] defines the cone vertex, **points**[1] defines the center of the cone base and the direction of the **axisZ** axis in the cone's local coordinate system, **points**[2] together with the previous points defines the plane of the **axisX** and **axisZ** axes in the cone's local coordinate system. The distance between **points**[0] and **points**[1] defines the cone height. The cone angle is determined from the condition that **points**[2] lies on the cone's side surface (Fig. 4):

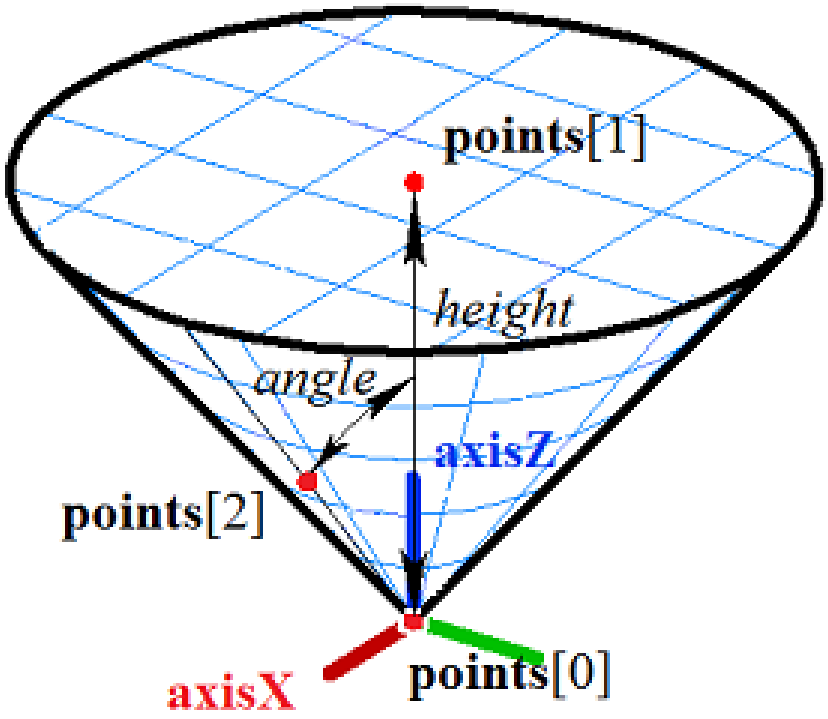


Fig. 4

For constructing a truncated cone, **points**[0] defines the cone vertex, **points**[1] defines the center of the larger cone base and the direction of the **axisZ** axis in the cone's local coordinate system, **points**[2] defines the center of the smaller cone base, **points**[3] together with the previous points defines the plane of the **axisX** and **axisZ** axes in the cone's local coordinate system. The distance between **points**[1] and **points**[2] defines the height of the truncated cone. The cone angle is determined from the condition that **points**[3] lies on the cone's side surface (Fig. 5):

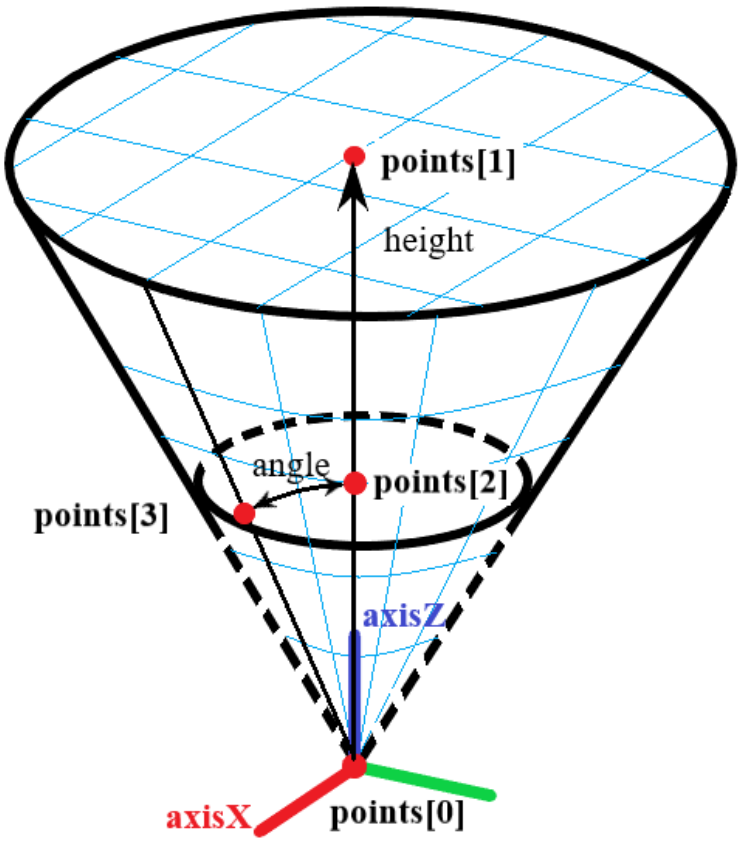


Fig. 5

For constructing a rectangular block, **points**[0] and **points**[1] define an edge and two vertices of the block, **points**[2] together with the previous points defines the plane of the block's lower base. The block edge parallel to the **points**[0]– **points**[1] edge passes through **points**[2]. **points**[3] defines the plane of the block's upper base (Fig. 6):

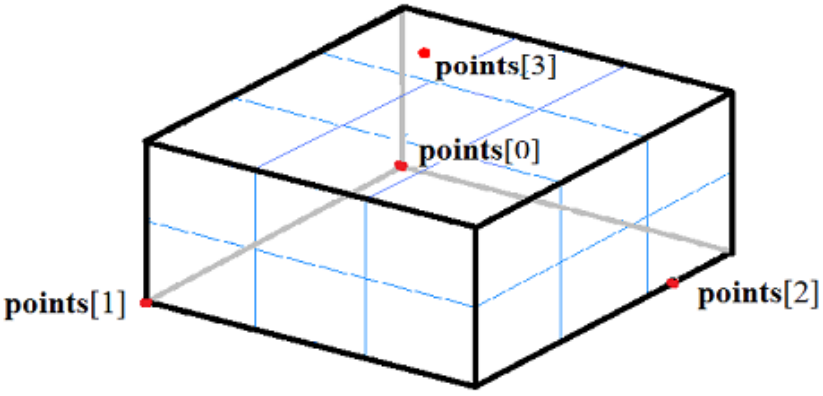


Fig. 6

For constructing a rectangular wedge, **points**[0] and **points**[1] define an edge and two vertices of the wedge, **points**[2] together with the previous points defines the plane of the wedge's lower

base and its vertex. The wedge edge parallel to the **points**[0]– **points**[1] edge passes through **points**[2]. **points**[3] defines the plane of the wedge's upper base (Fig. 7):

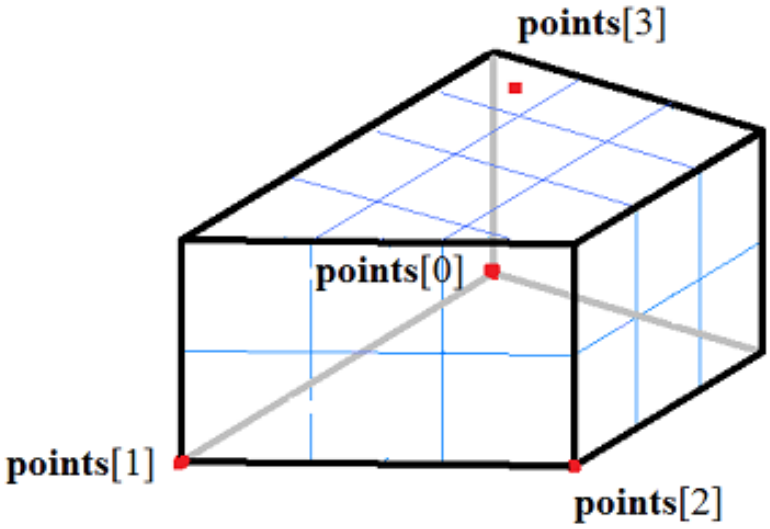


Fig. 7

For constructing a plate with cylindrical ends, **points**[0] and **points**[1] define an edge and two vertices of the plate, **points**[2] together with the previous points defines the plane of the plate's lower base. The plate edge parallel to the **points**[0]– **points**[1] edge passes through **points**[2]. **points**[3] defines the plane of the plate's upper base (Fig. 8):

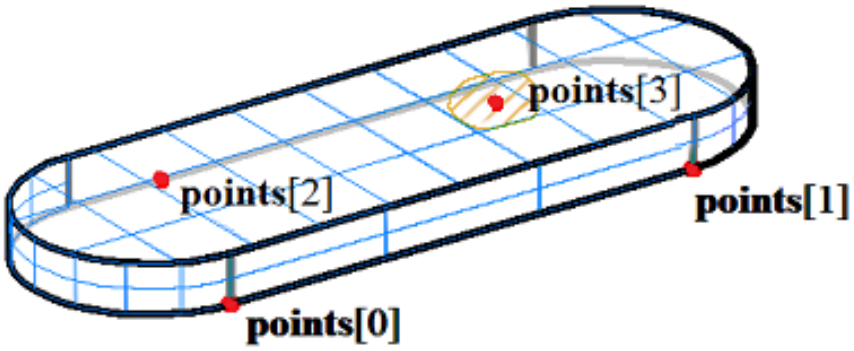


Fig. 8

For constructing a straight prism, **points**[0], **points**[1], ..., **points**[n-1] define the base polygon. The distance from the plane of the lower base to the last point **points**[n] defines the prism height. Fig. 9 shows a straight prism with a pentagonal base:

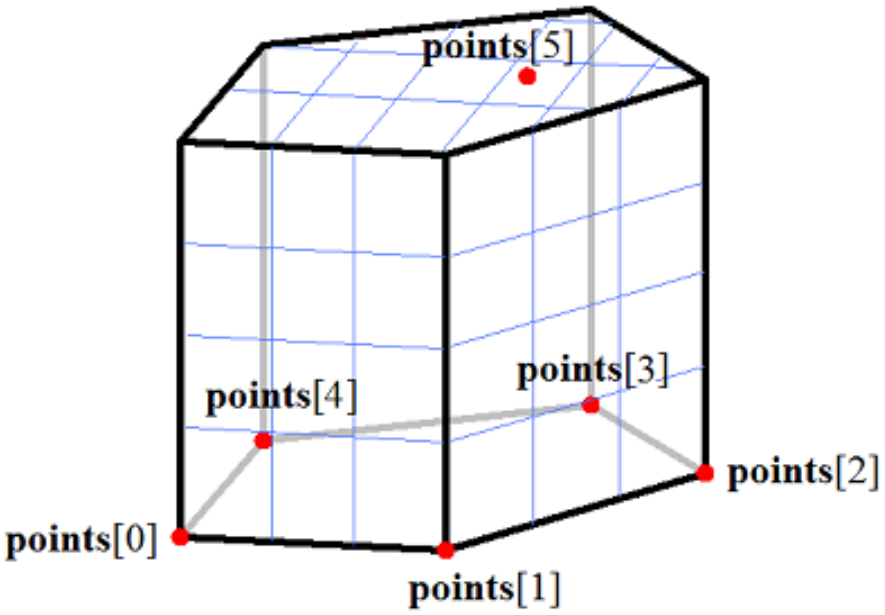


Fig. 9

For constructing a pyramid with a polygon at the base, **points**[0], **points**[1], ..., **points**[n-1] define the base polygon; the last point **points**[n] defines the pyramid vertex. Fig. 10 shows a pyramid with a pentagonal base:

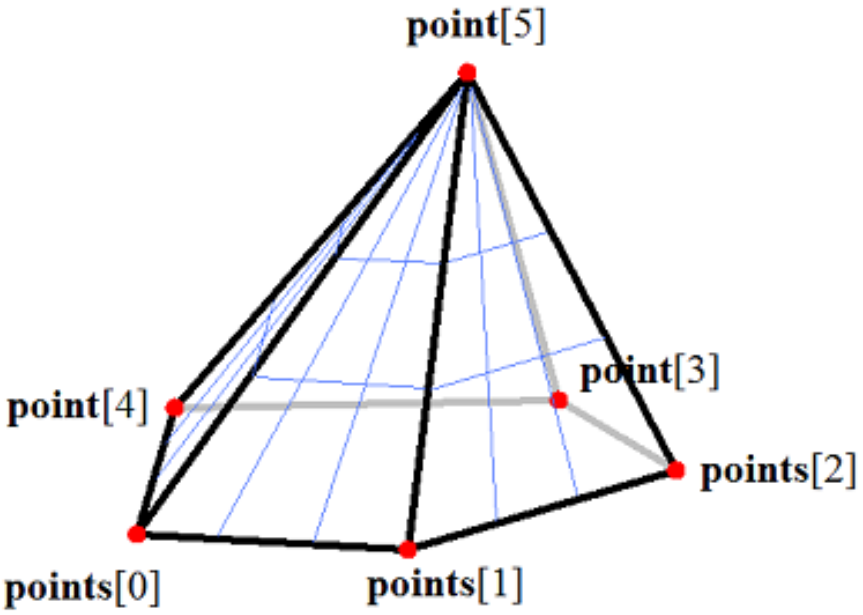


Fig. 10

The base polygon of a prism or pyramid can be arbitrary.

For constructing an icosahedron, **points**[0], **points**[1], **points**[2] define the vertices and planes of all faces of the shape (Fig. 11):

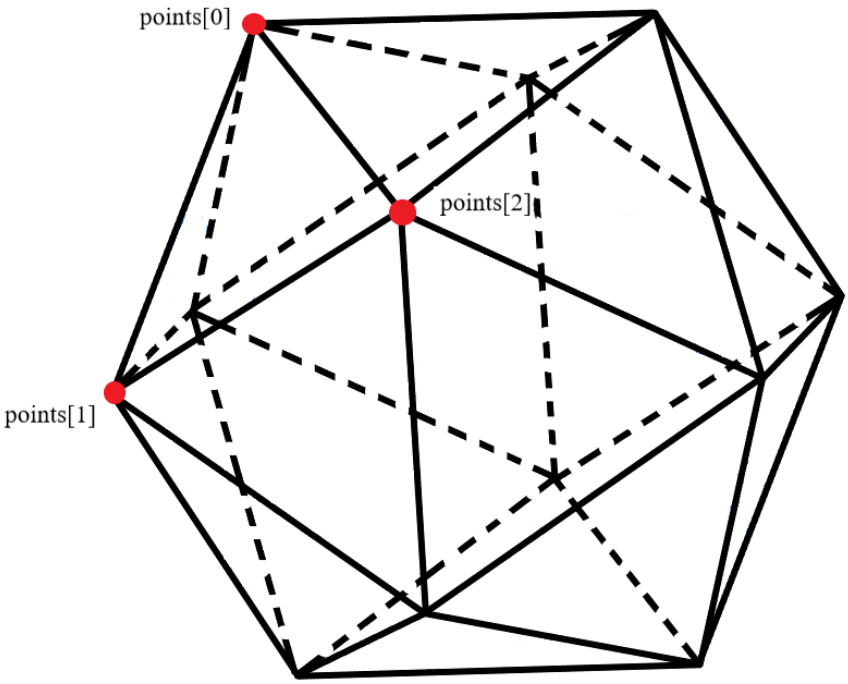


Fig. 11

Constructing an Elementary Body by a Given Surface

For constructing solids based on an elementary surface, use the *ElementarySolid()* function with the following constructor to initialize the **params** object with operation parameters (see *op_shell_parameter.h*):

```
MbElementarySolidParams( const SPtr<const MbElementarySurface> & surface,
                         const MbSNameMaker                    & operNames );
```

with parameters:

- **surface** — the elementary surface (sphere, torus, cylinder, cone);
- **operNames** — the operation's namer.

An elementary surface can be a sphere MbSphere, a torus surface MbTorus, a cylindrical surface MbCylinder, or a conical surface MbCone. Fig. 12 shows a spherical surface and a body constructed on it:

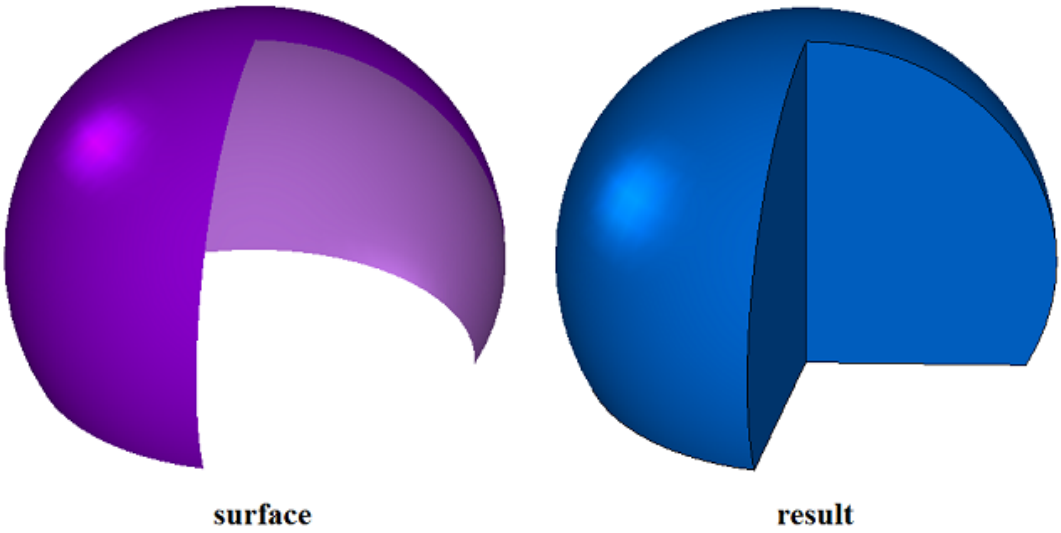


Fig. 12

Fig. 13 shows a torus surface and a body constructed on it:

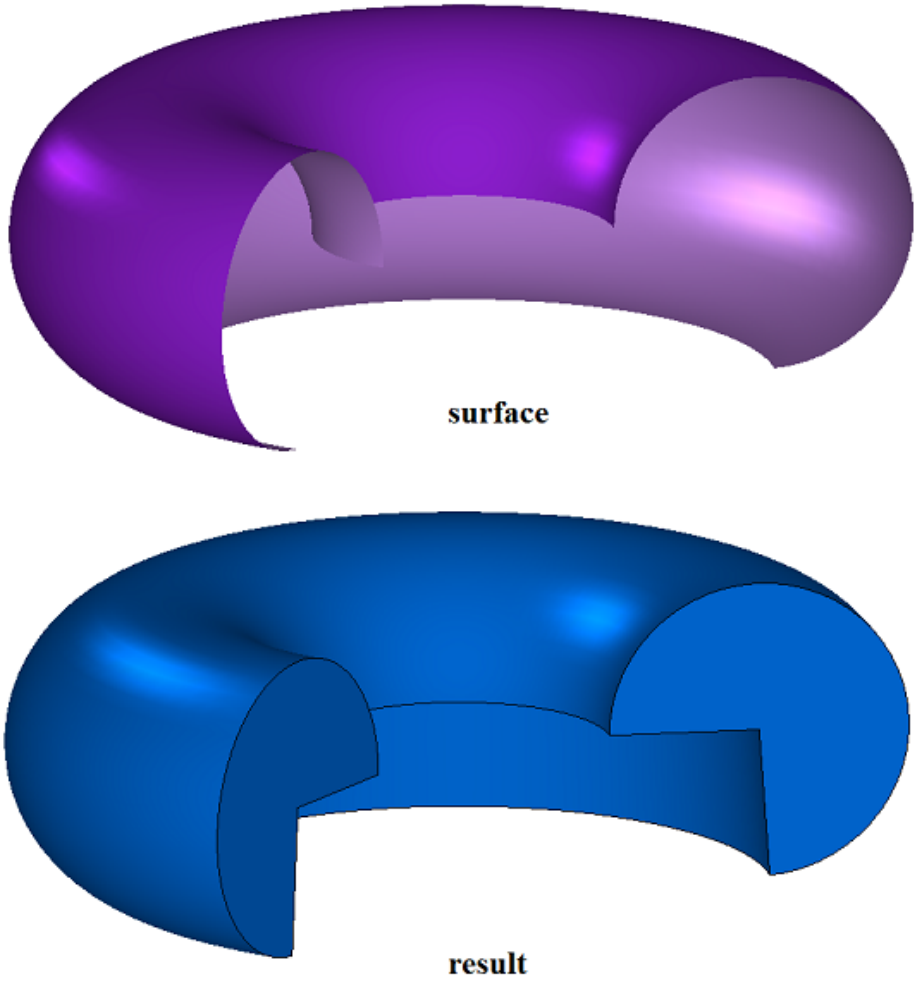


Fig. 13

Fig. 14 shows a cylindrical surface and a body constructed on it:

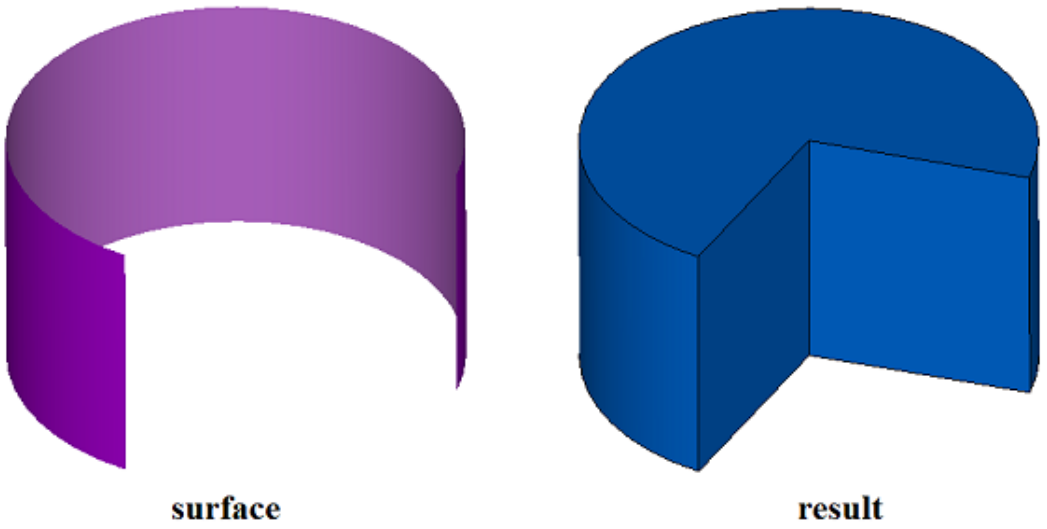


Fig. 14

Fig. 15 shows a conical surface and a body constructed on it:

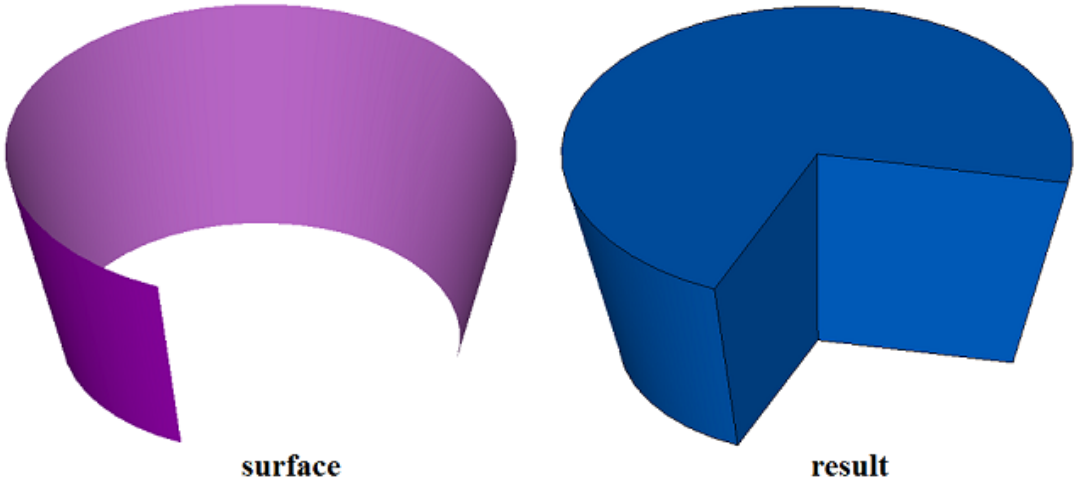


Fig. 15

If the listed surfaces are cyclically closed, the bodies constructed on them will have the corresponding shape. If the surface is not one of the listed elementary surfaces, the function returns the error code *rt_Error*.

The ElementarySolid() function adds the MbElementarySolid builder to the log of the constructed body (declared in *cr_elementary_solid.h*).

In the *test.exe* test application, constructing an elementary body is performed via the «Create→Body→By Points» and «Create→Body→Based on the surface→On an elementary surface» menu commands.

3.1.2 Constructing an Extrusion Body

The function

```
MbSweptData ExtrusionSolid( const MbSweptData &           sweptData,
                            const MbVector3D &            direction,
                            const MbSolid *               solid1,
                            const MbSolid *               solid2,
                                  bool                    checkIntersection,
                            const ExtrusionValues &       params,
                            const MbSNameMaker &          operNames,
                                  PArray<MbSNameMaker> &  contoursNames,
                                  MbSolid *&              result );
```

constructs an extrusion body.

The input parameters of the function are:

- **sweptData** – a parameter that contains data about a generatrix curve.
- **direction** – the extrusion direction.
- **solid1** – used when the MbSweptWay::sw_shell option is selected, for extrusion up to the nearest faces of a given body in the forward direction.
- **solid2** – used when the MbSweptWay::sw_shell option is selected, for extrusion up to the nearest faces of a given body in the reverse direction.
- **checkIntersection** – a flag for uniting the **solid1** and **solid2** bodies with intersection checking.
- **params** – an object of the ExtrusionValues class that contains information on how the extrusion magnitude in each direction is defined (by distance, up to a surface, up to a body).
- **operNames** – the operation name maker.
- **contoursNames** – the name makers of the segments of the generatrix contour.

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

The function returns an operation result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

An extrusion body belongs to a type of swept bodies obtained by moving a generatrix curve along a guide curve. The guide curve of an extrusion body is a line segment. The extrusion body is constructed by moving one or several curves along a segment whose direction is defined by the **direction** vector.

The **sweptData** parameter contains information about the generatrix curves. The MbSweptData and ExtrusionValues classes are described in the *op_swept_parameter.h* file.

The generatrix curve of an extrusion, revolution, or kinematic operation may include a set of two-dimensional contours, a set of three-dimensional contours, a wireframe, or a body.

Two-dimensional contours on a surface may intersect. Closed intersecting contours are united using a Boolean operation on regions (Fig. 1).

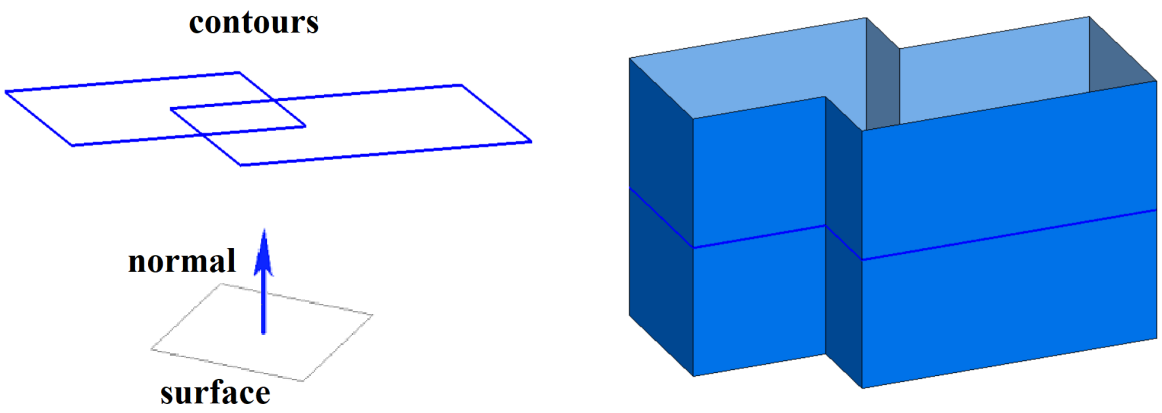


Fig. 1

In the case of constructing an open shell, if an open contour is among the intersecting contours, then a shell with intersecting parts will be constructed and the function will return an error.

In the case of constructing a closed shell, intersections of shells obtained from open contours are processed by a Boolean operation on shells.

For a set of two-dimensional contours on a surface, the following constraints apply:

- There may be one or several contours.
- If there are several contours, they must be either all closed or all open.
- If the contours are closed, they may be nested inside each other, and the nesting depth is not limited.

For two-dimensional contours on a non-planar surface, there is an additional constraint: all contours must be closed.

Construction of the operation based on two-dimensional contours on a non-planar surface assumes that the user specifies a body face as a generatrix. In this case, data for the generatrix can be obtained using the MbFace::GetSurfaceCurvesData() face method. This method returns a copy of the base surface of the face and a set of two-dimensional contours created from edge loops.

The constraint for three-dimensional contours is that constructing a closed body by extruding a three-dimensional contour is possible only for a thin-walled variant.

When both a body and a surface are specified at the same time, it is assumed that a kinematic operation is performed on the body along a curve on this surface, with the motion being aligned with the normal.

The body construction may be performed in the forward direction relative to the **direction** vector, in the reverse direction relative to the **direction** vector, or in both directions. The construction parameters in each direction are defined by MbSweptSide objects.

The **params** parameter contains information about the extrusion method in the forward direction MbSweptSide **side1** and about the extrusion method in the reverse direction MbSweptSide **side2**. Extrusion in each direction can be carried out in three ways:

1. If **way** = *sw_scalarValue*, the extrusion is performed over the length *scalarValue* in the **side1** or **side2** direction.
2. If **way** = *sw_shell*, the extrusion is performed up to the nearest **solid1** or **solid2** object.

3. If **way** = *sw_surface*, the extrusion is performed up to the **side1**.*surface* or **side2**.*surface* surface if **side1**.*distance* equals zero or **side2**.*distance* equals zero, respectively.

If **way** = *sw_surface* and **distance** is not equal to zero, the extrusion is performed up to a surface equidistant from **side1**.*surface* or up to a surface equidistant from **side2**.*surface*, respectively.

If **side1**.*rake* is not equal to zero or **side2**.*rake* is not equal to zero, the extrusion is performed with a draft **side1**.*rake* or **side2**.*rake* in the corresponding direction.

The **params**.*thickness1* parameter defines the offset outward from the generatrix curve, and the **params**.*thickness2* parameter defines the offset inward from the generatrix curve.

The **params**.*shellClosed* parameter controls whether the constructed body is closed.

The **params**.*checkSelfInt* parameter indicates the need to check the construction result for self-intersection. By default, **params**.*checkSelfInt* = *false*, and the check is not performed.

The **mergeFlags** control flags manage the merging of similar faces or edges (*mergeFaces* = *true* – merge similar faces; *mergeEdges* = *true* – merge similar edges).

Fig. 2 shows the data used in construction and the inheritance scheme of extrusion body construction parameters ExtrusionValues & **params**.

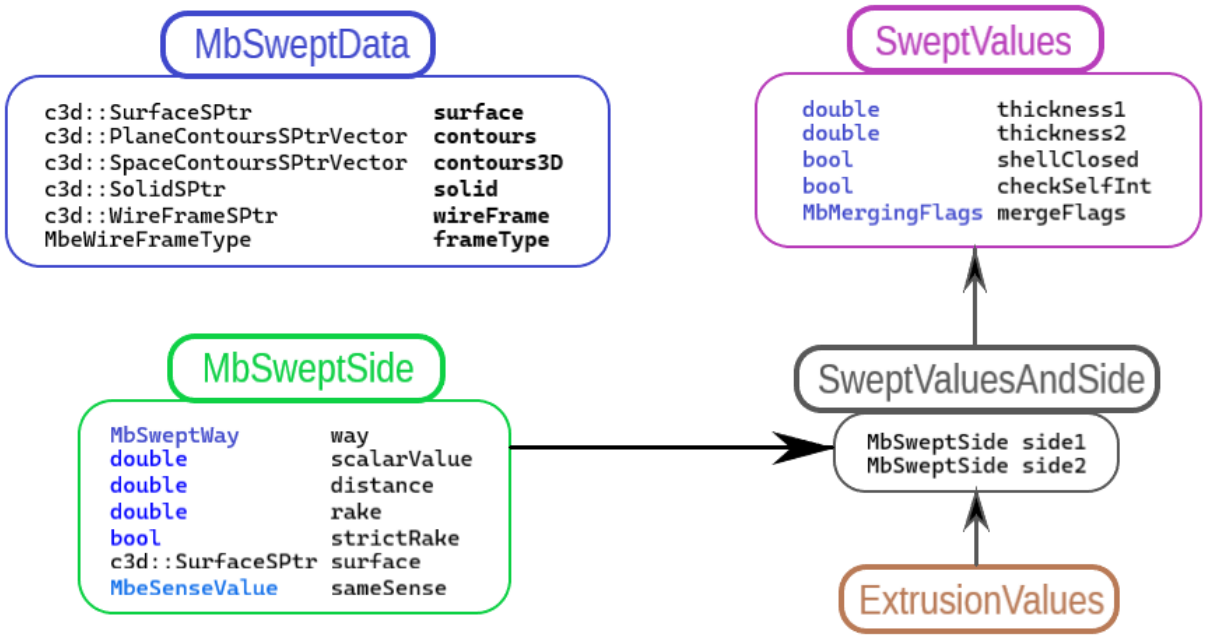


Fig. 2

The **operNames** parameter provides naming of faces, edges, and vertices of the constructed body.

Fig. 3 shows a two-dimensional contour **contour** and a planar surface **surface** (MbPlane).

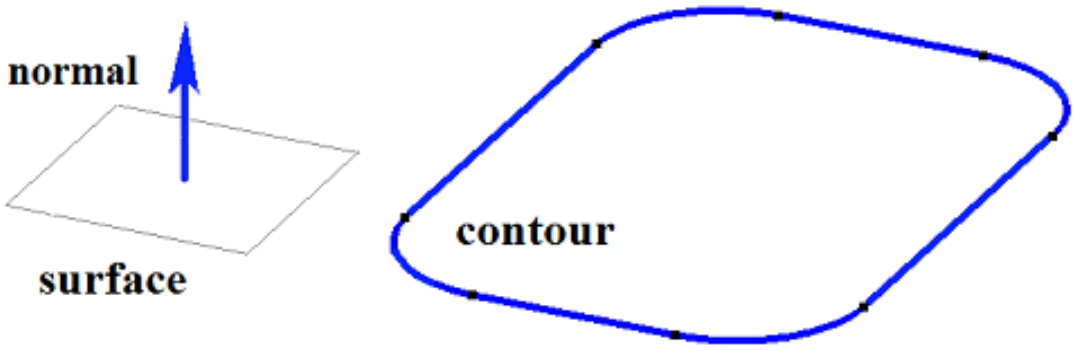


Fig. 3

Fig. 4 shows a closed body constructed by extrusion using the specified parameters of the contour shown in Fig. 3. Each segment of the contour corresponds to a face of the body, whose name is taken from the respective element of the name generator **contoursNames**[0].

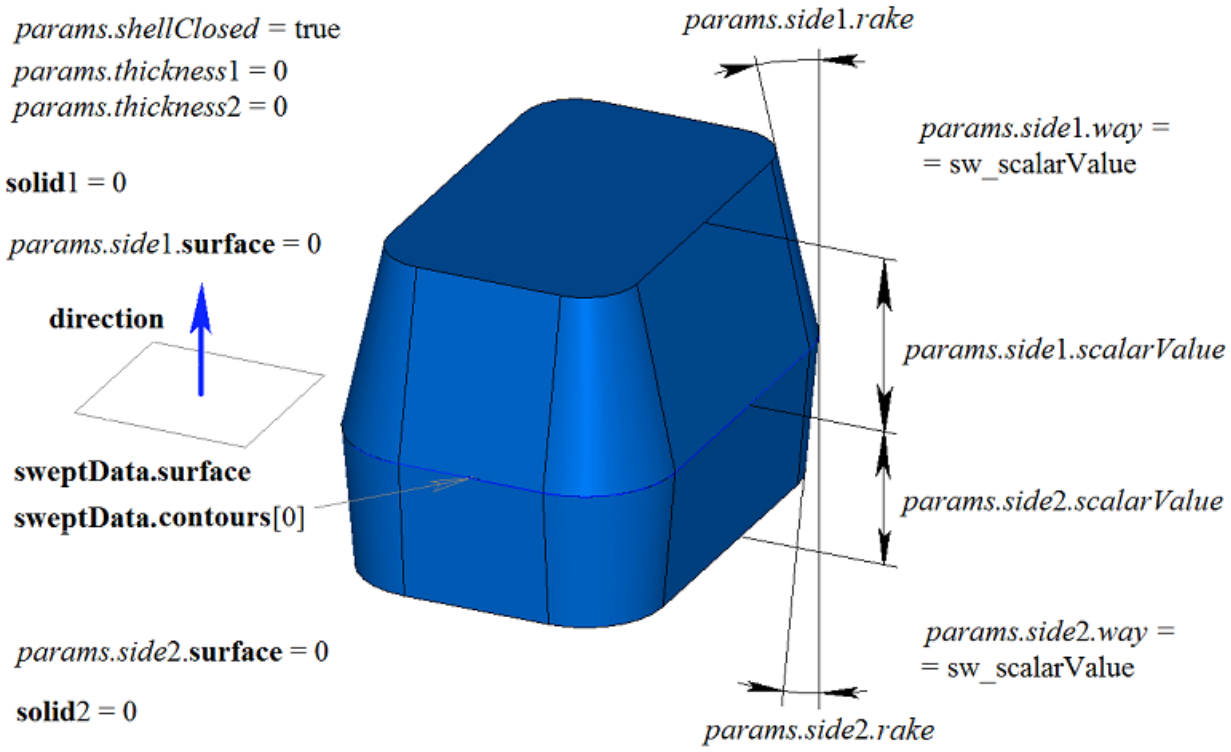


Fig. 4

Fig. 5 shows a closed thin-walled body constructed by extrusion using the specified parameters of the contour shown in Fig. 3.

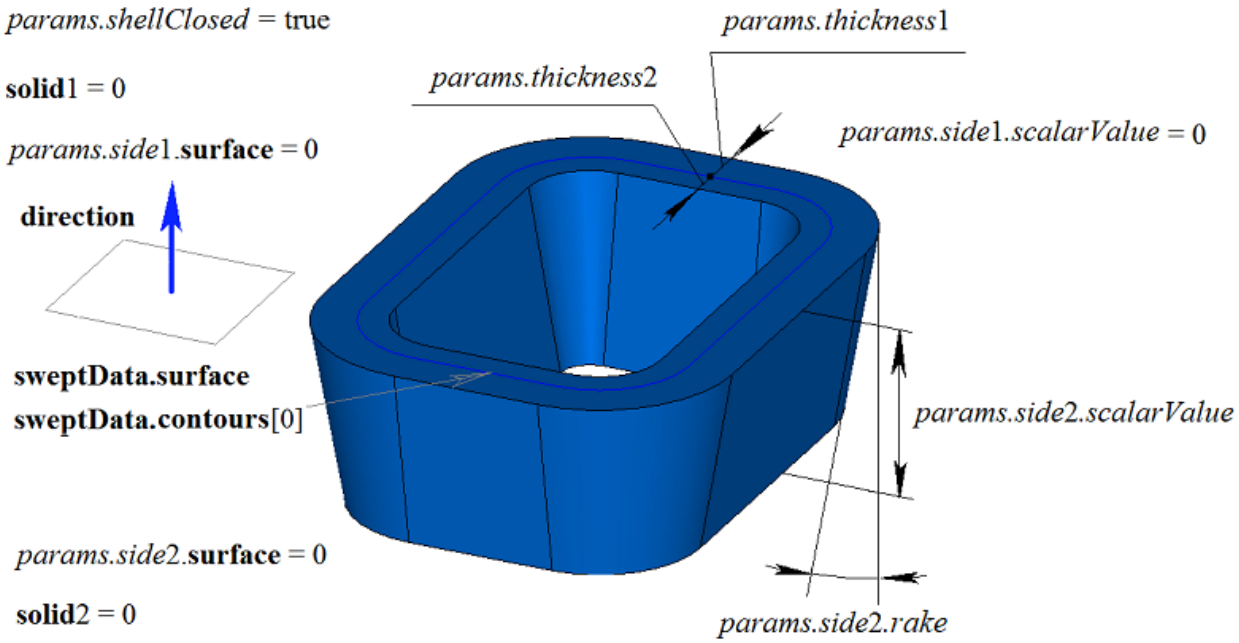


Fig. 5

Fig. 6 shows an open body constructed by extrusion using the specified parameters of the contour shown in Fig. 3. The construction parameters of the body shown in Fig. 4 differ from those of the body shown in Fig. 6 only by the **params**.*shellClosed* value.

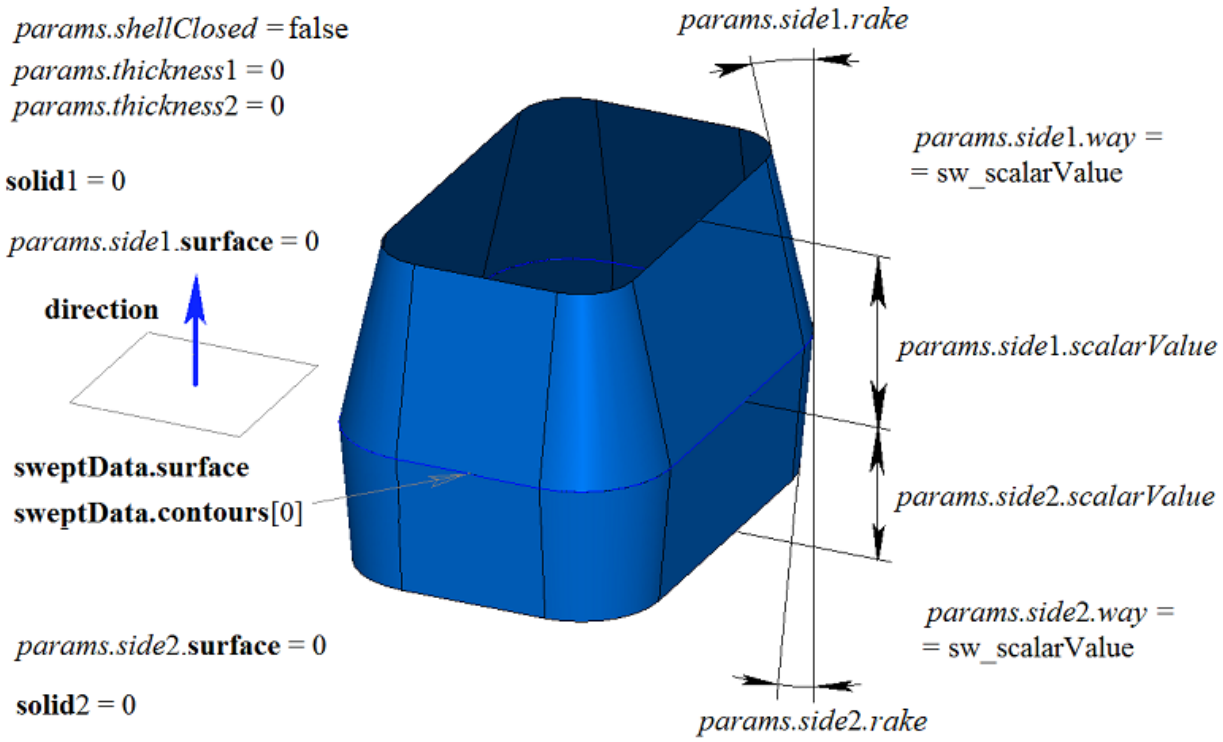


Fig. 6

Fig. 7 shows a two-dimensional contour **contour**, a planar surface **surface** (MbPlane), and two bodies **solid1** and **solid2** that will be used in the extrusion body construction. For the construction, the **solid1** and **solid2** bodies must completely cover the path of contour motion in the corresponding direction. At the same time, the **params**.*side1.rake*, **params**.*side2.rake*, **params**.*thickness1*, and **params**.*thickness2* parameters must be taken into account.

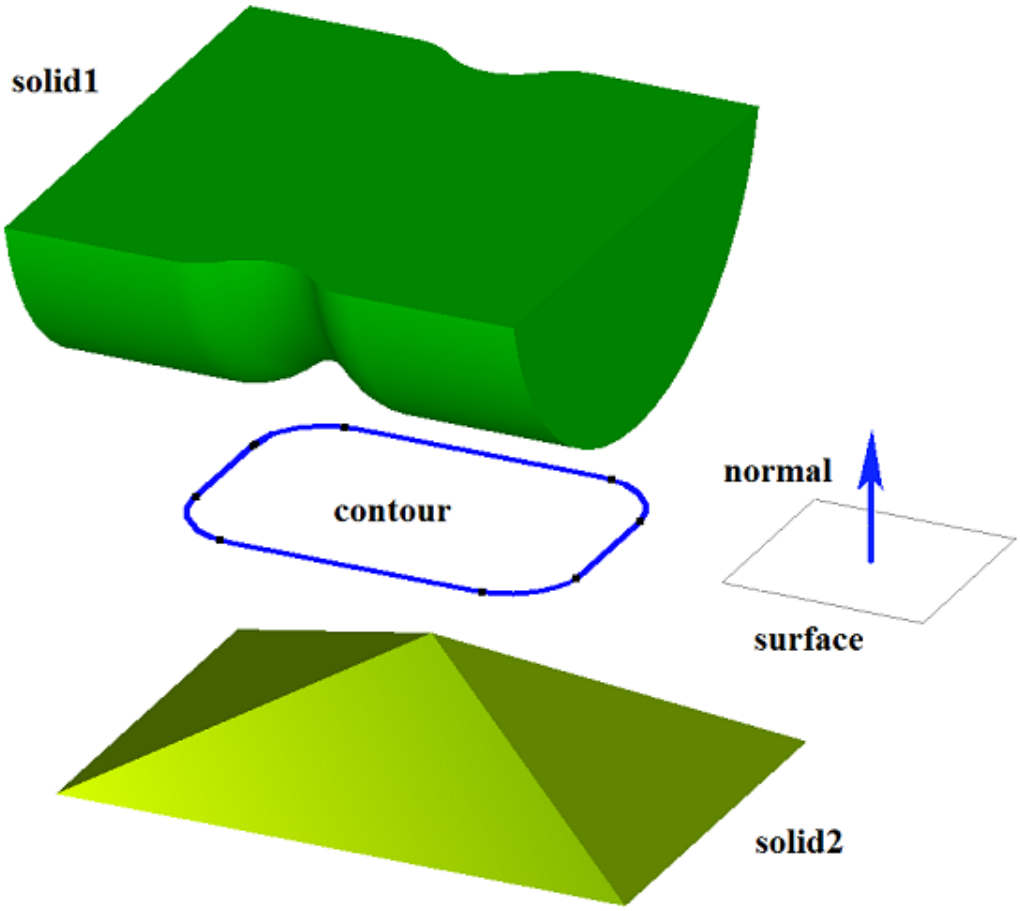


Fig. 7

Such a construction is performed by extruding the contour to a length exceeding the maximum distance to the specified body, followed by subtracting the specified body from the constructed body.

Fig. 8 shows a body constructed by extruding the contour shown in Fig. 7, with the “Up to nearest objects” options, where the **solid1** and **solid2** bodies are specified as such objects.

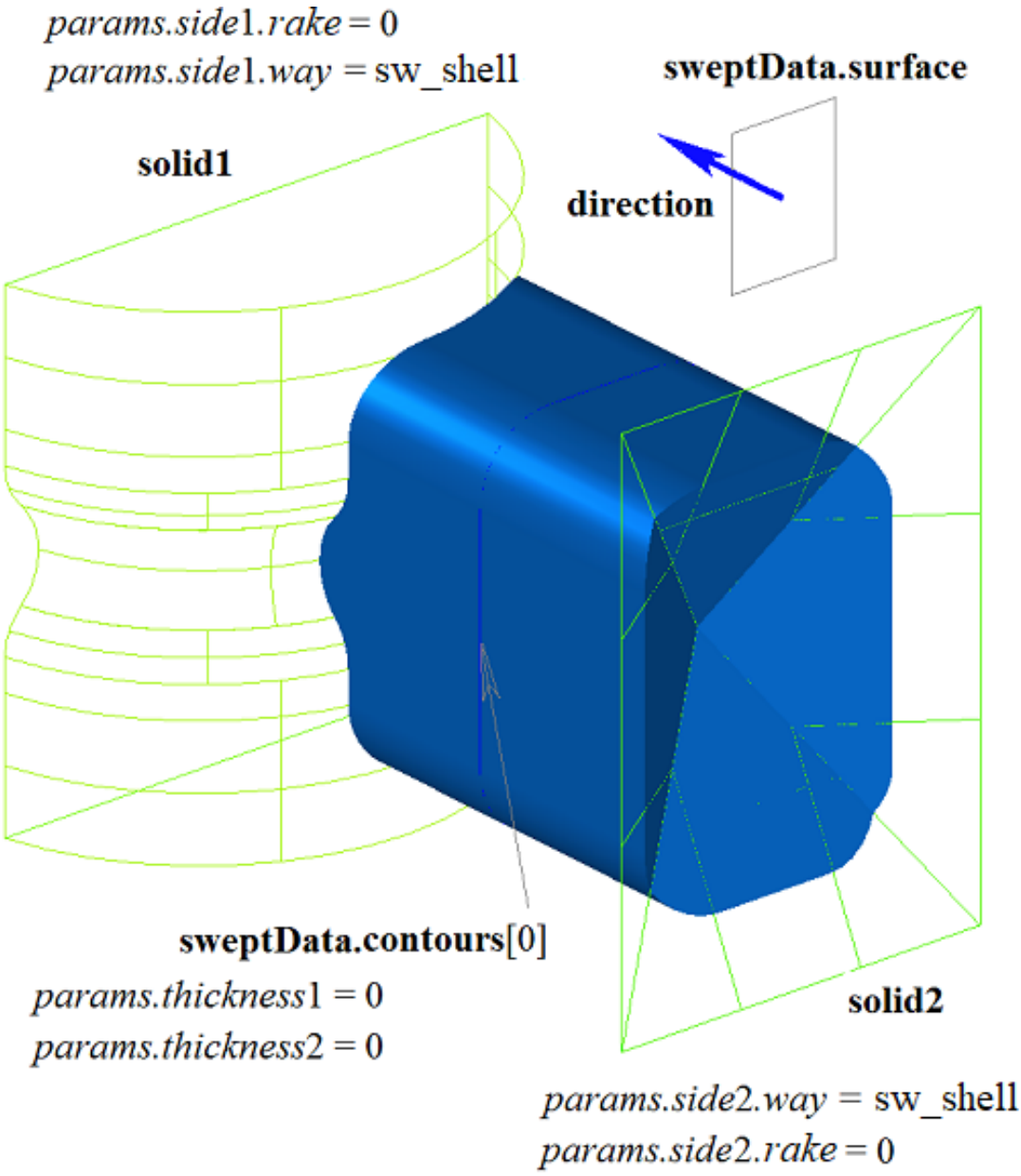


Fig. 8

Fig. 9 shows a thin-walled body with drafted faces constructed by extruding the contour shown in Fig. 7, with the “Up to nearest objects” options, where the **solid1** and **solid2** bodies are specified as such objects.

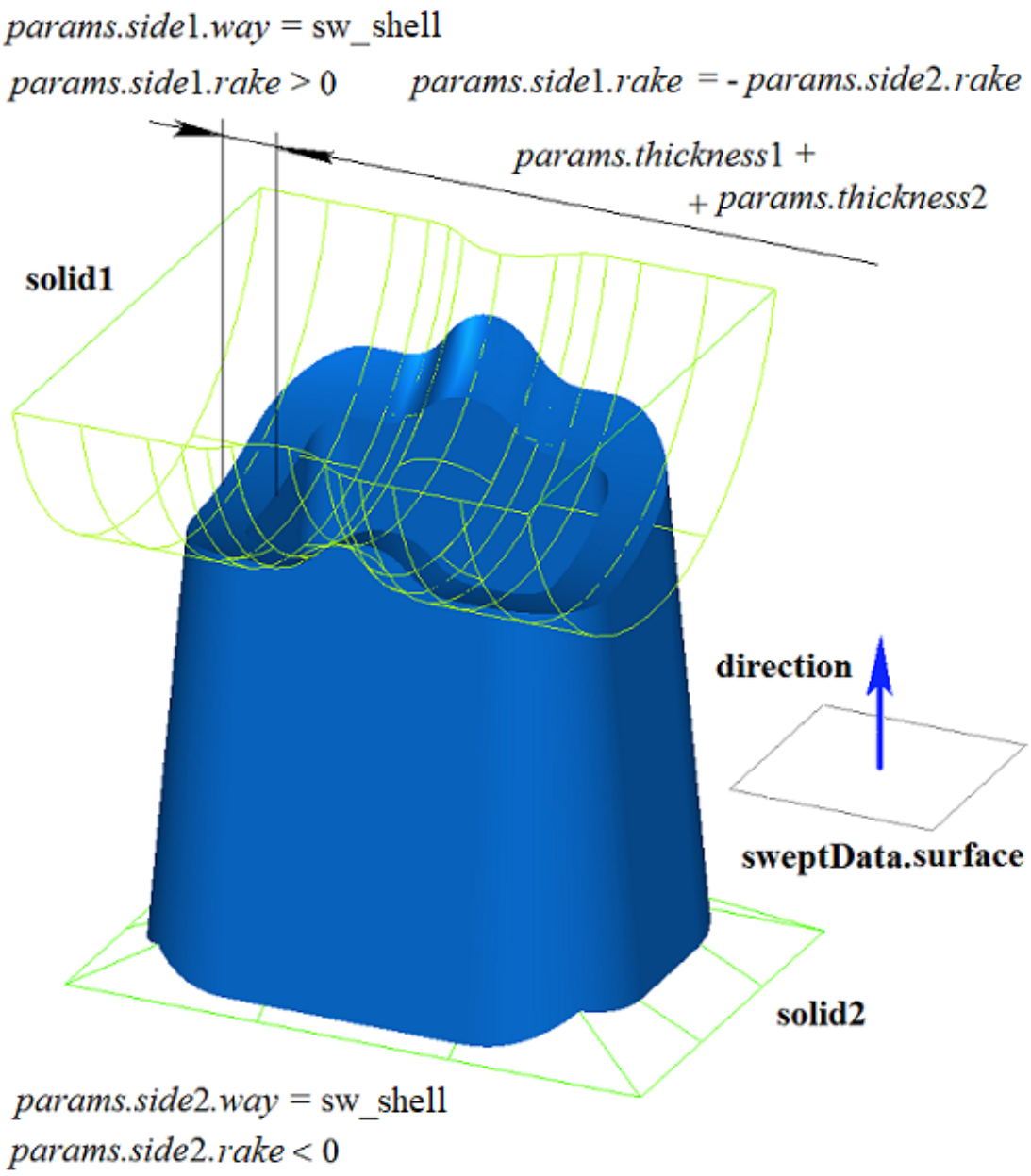


Fig. 9

Fig. 10 shows a two-dimensional contour **contour**, a planar surface **surface** (*MbPlane*) and two surfaces **surface1** and **surface2** that will be used in the extrusion body construction. For the construction, the **surface1** and **surface2** surfaces must completely cover the path of contour motion in the corresponding direction. At the same time, the **params**.*side1.rake*, **params**.*side2.rake*, **params**.*thickness1*, and **params**.*thickness2* parameters must be taken into account. The extrusion body is cut by the specified surfaces or by surfaces equidistant from them if **params**.*side1.distance* or **params**.*side2.distance* are not equal to zero.

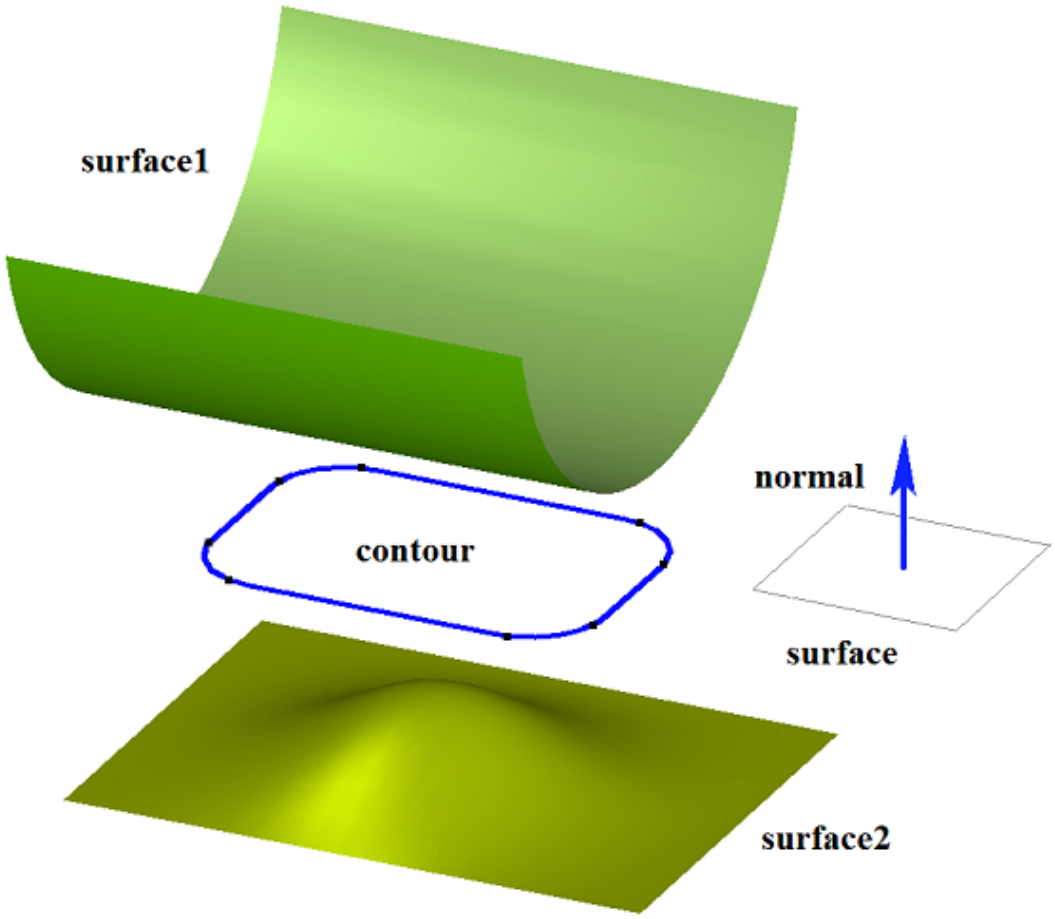


Fig. 10

Fig. 11 shows a body constructed by extruding the contour shown in Fig. 10, with the “Up to surface” options, where **surface1** and **surface2** are specified.

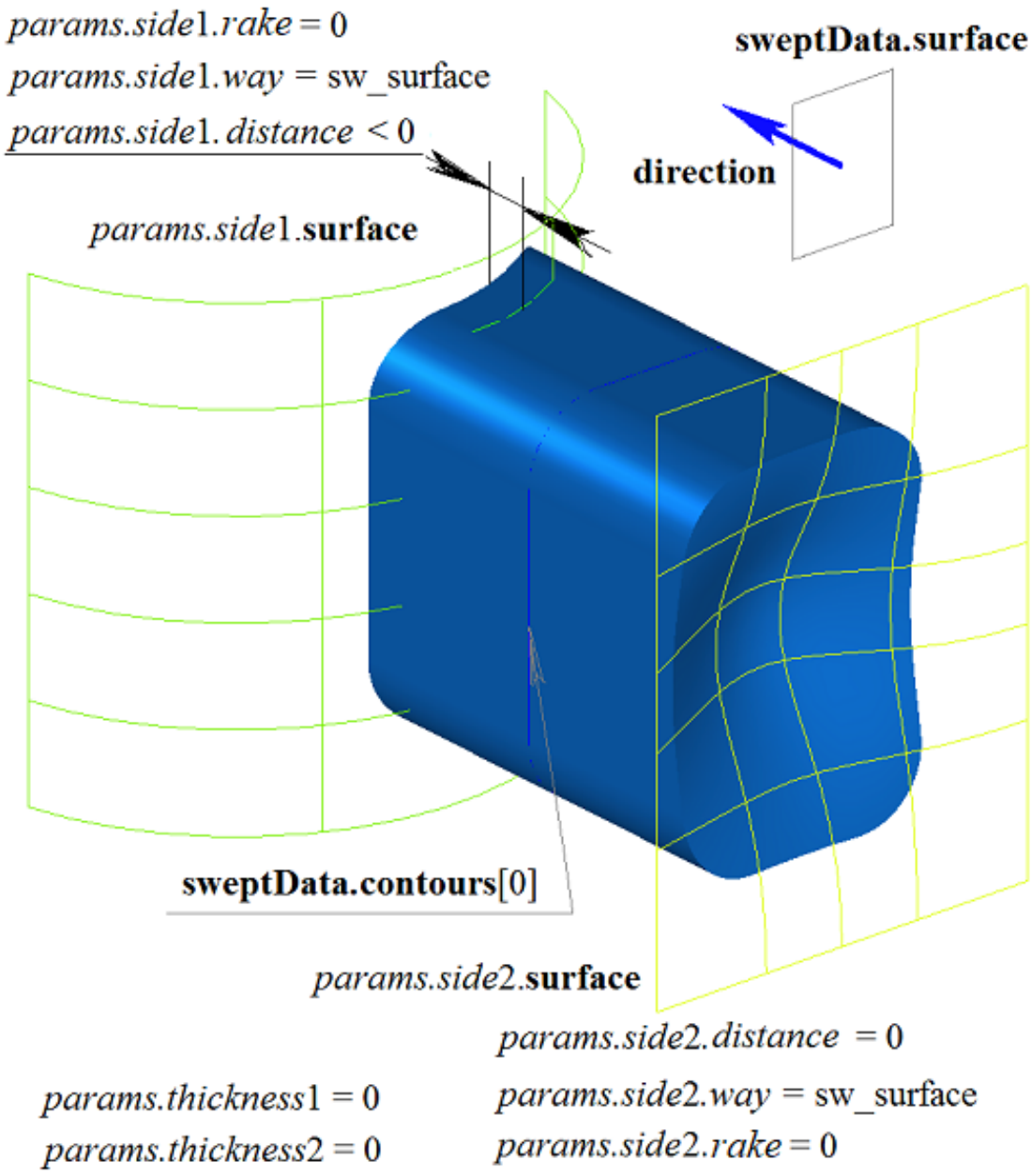


Fig. 11

Fig. 12 shows a thin-walled body with drafted faces constructed by extruding the contour shown in Fig. 10, with the “Up to surface” options, where **surface1** and **surface2** are specified.

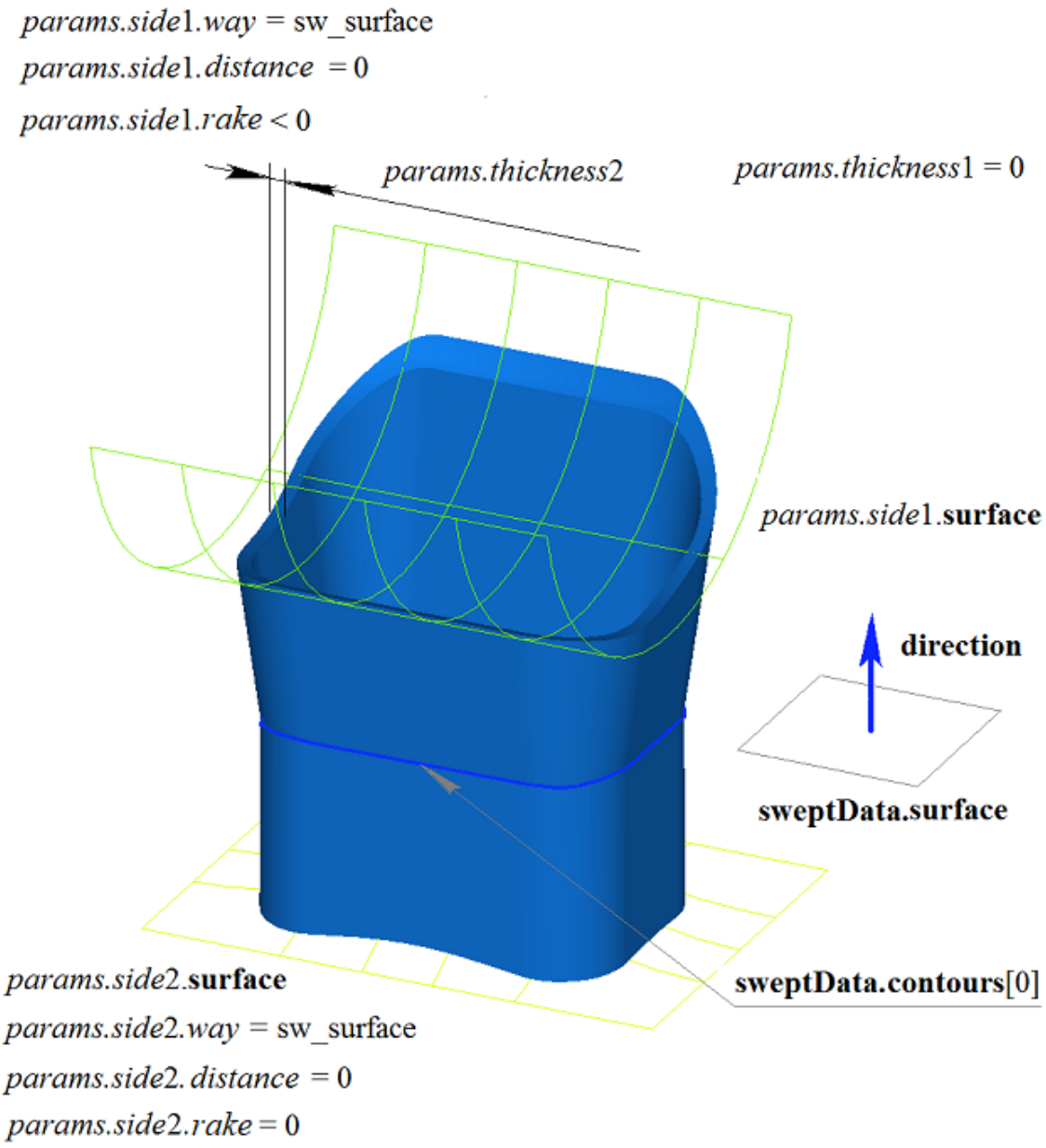


Fig. 12

A two-dimensional contour may lie on a planar or a curved surface. For example, a body can be constructed by extruding a contour on a curved surface obtained from a loop of one of the faces of a solid body shown in Fig. 13.

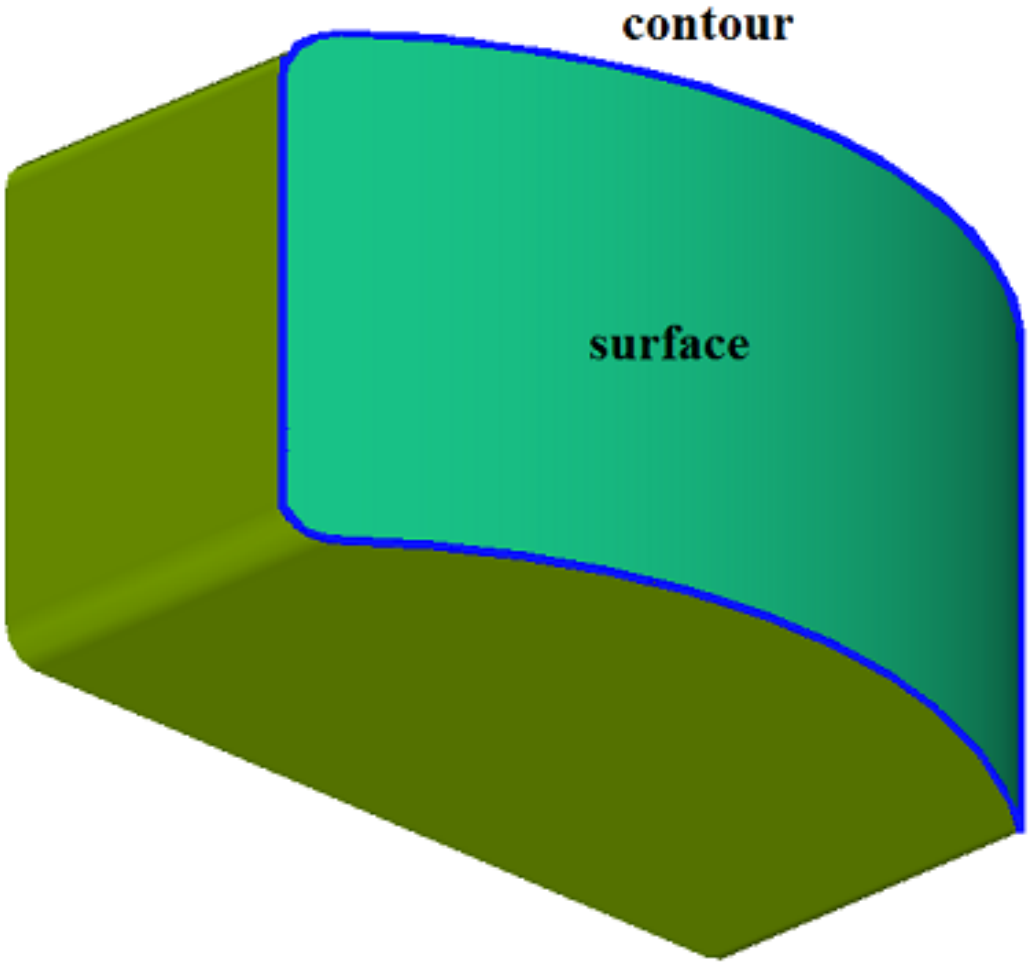


Fig. 13

Fig. 14 shows a body obtained by extruding the contour on the curved surface shown in Fig. 13.

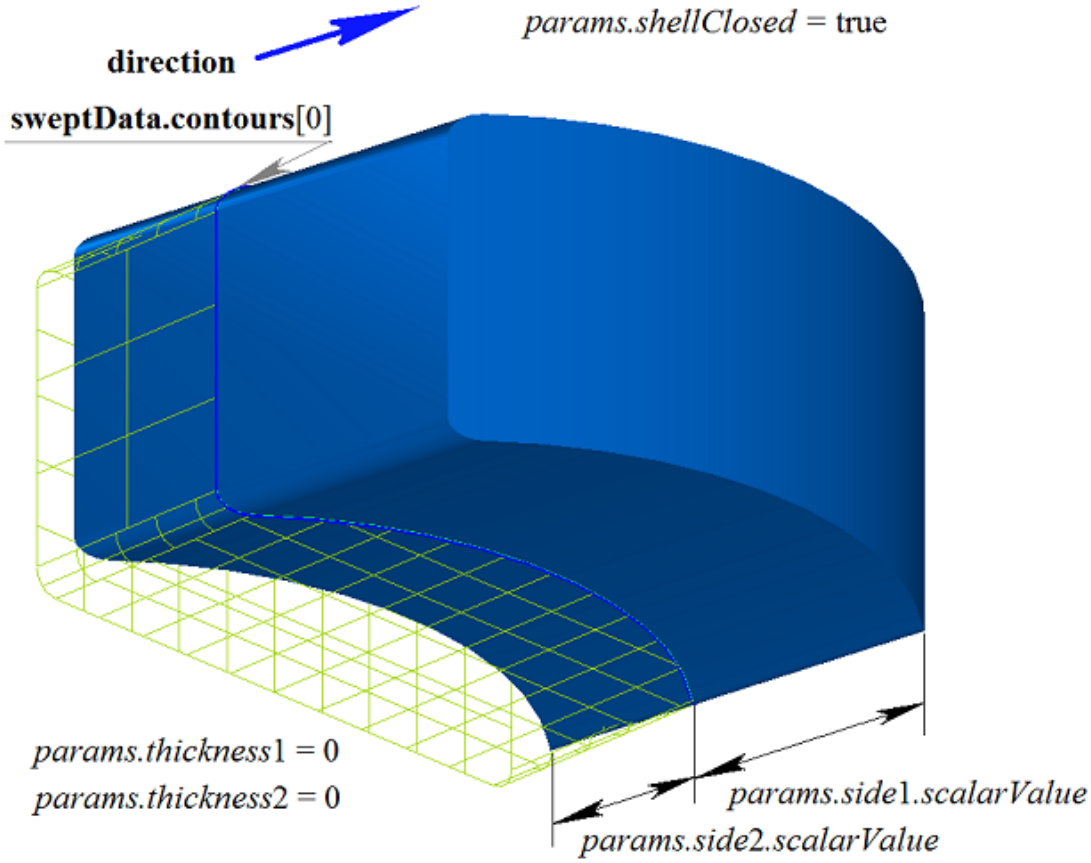


Fig. 14

Fig. 15 shows a thin-walled body obtained by extruding the contour on the curved surface shown in Fig. 13.

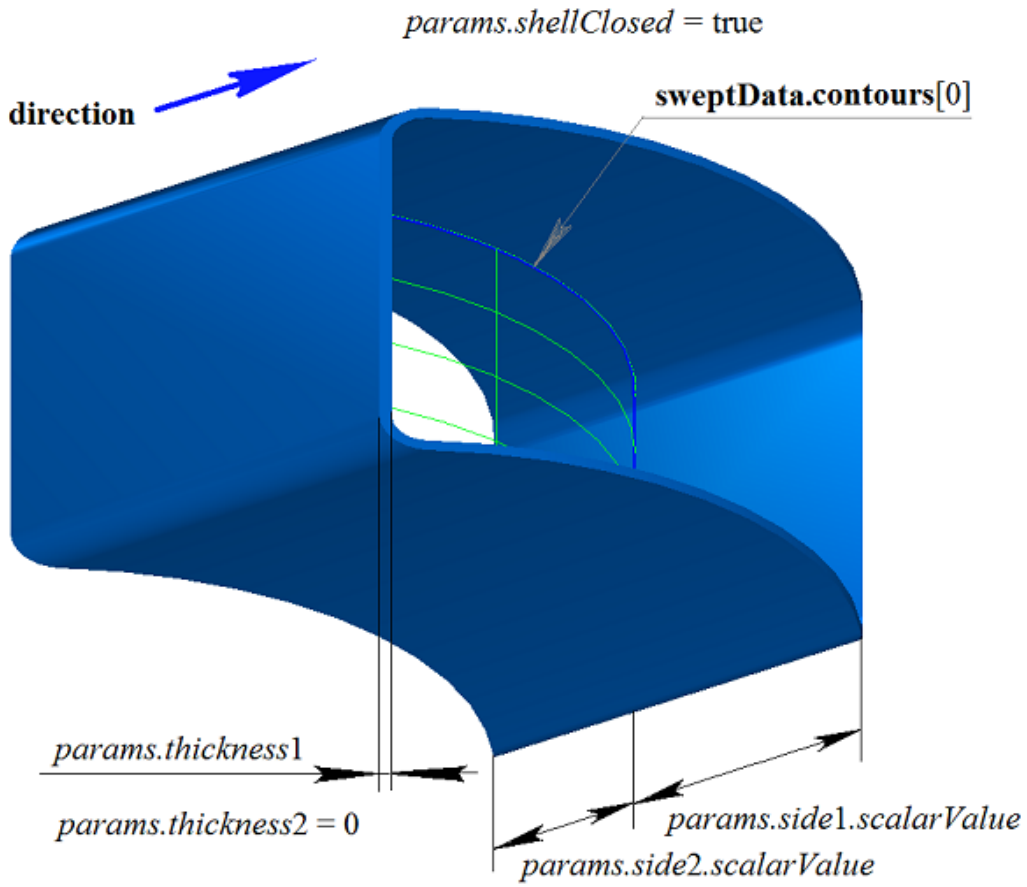


Fig. 15

Fig. 16 shows an open body obtained by extruding the contour on the curved surface shown in Fig. 13.

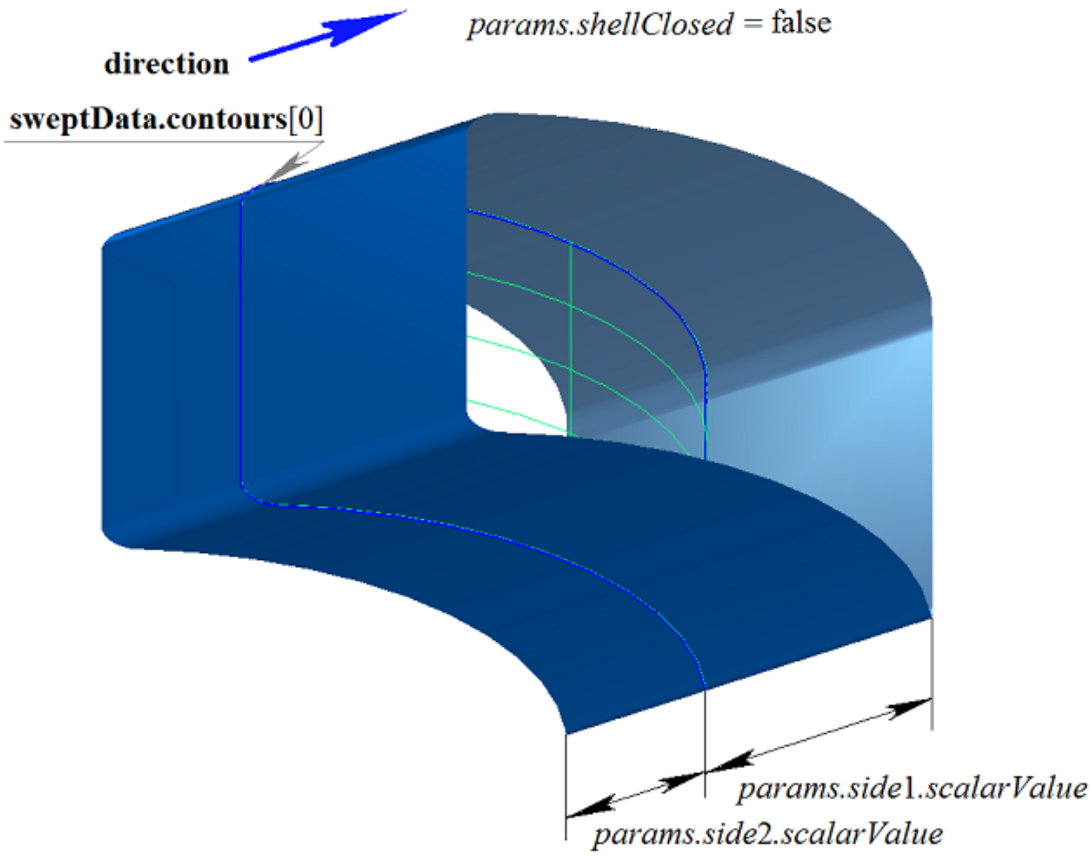


Fig. 16

If multiple non-intersecting two-dimensional contours lie on one surface, the considered function determines outer contours and inner contours nested in them, with potentially multiple nesting levels. Fig. 17 shows multiple non-intersecting two-dimensional contours **contours** and a planar surface **surface** (MbPlane).

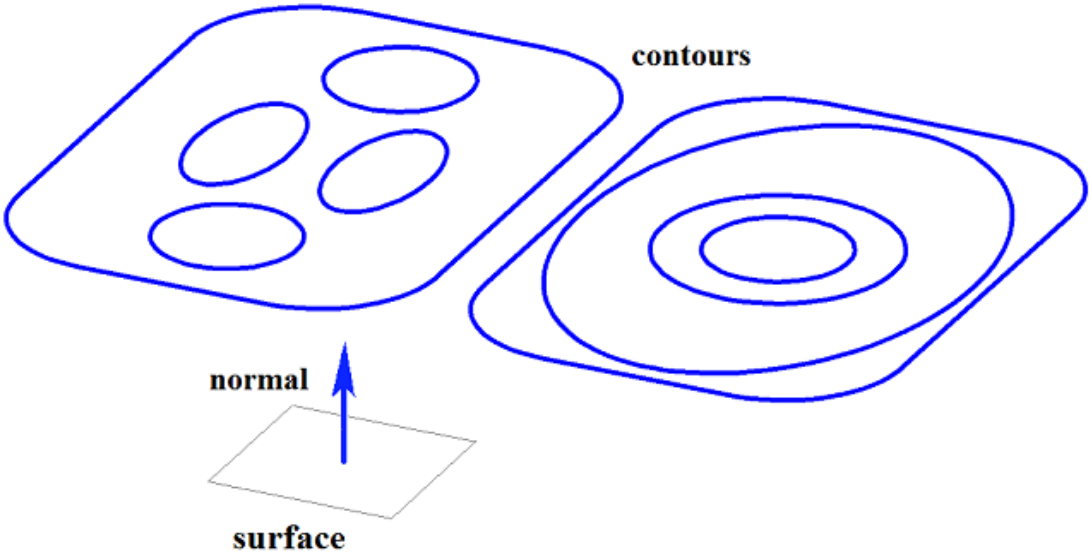


Fig. 17

Fig. 18 shows a closed body consisting of several parts constructed by extruding the multiple contours shown in Fig. 17.

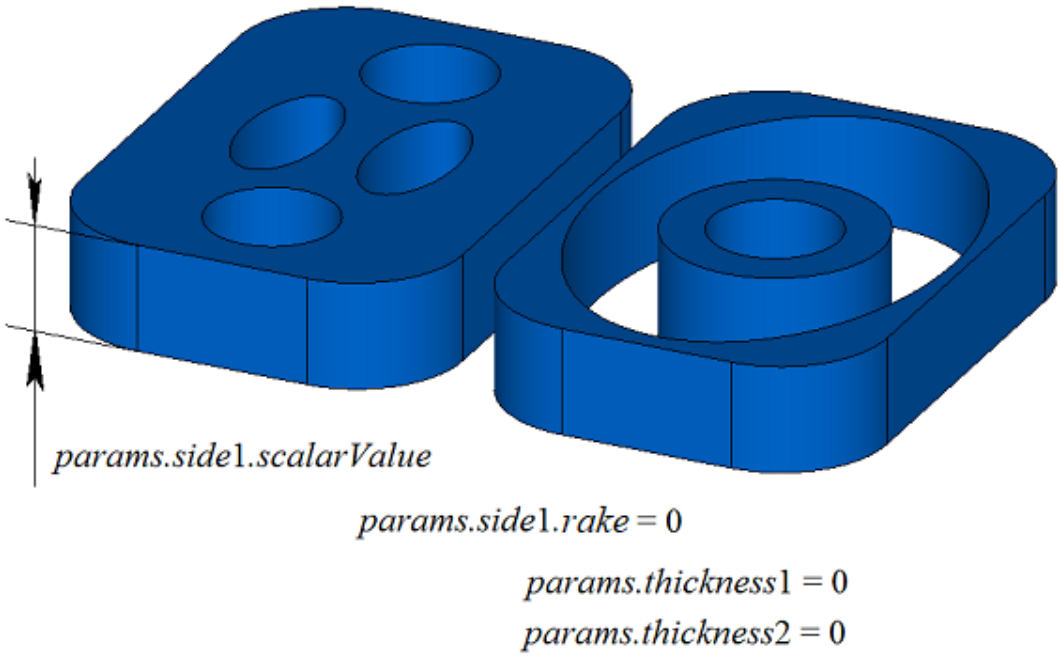


Fig. 18

Fig. 19 shows a closed body consisting of several parts constructed by extruding with a draft the multiple contours shown in Fig. 17.

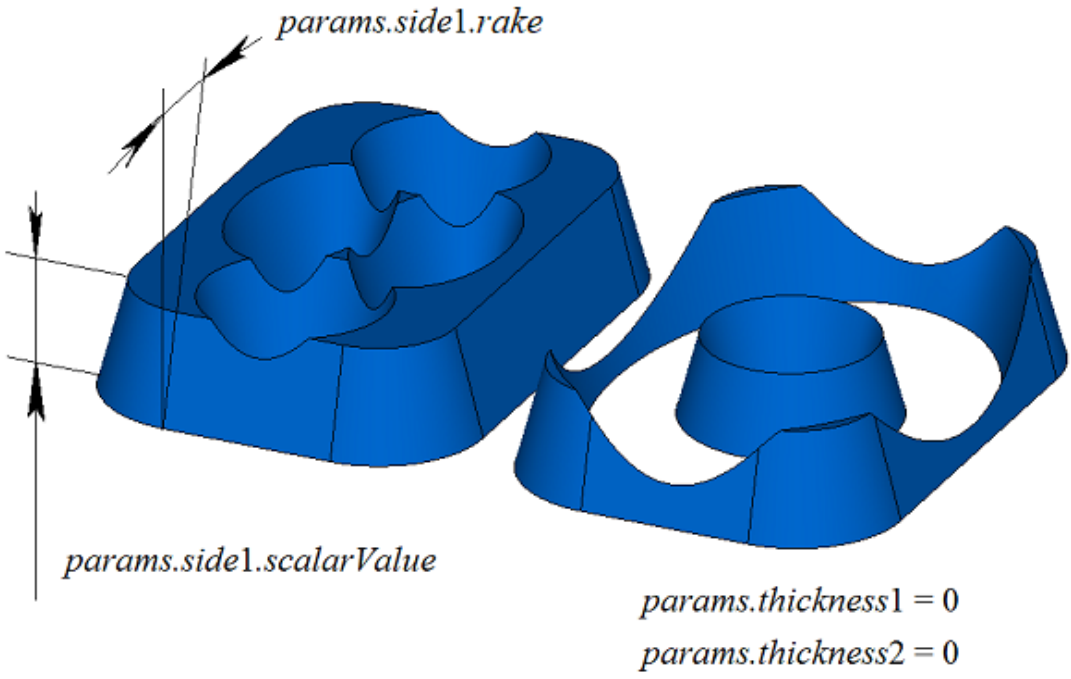


Fig. 19

Fig. 20 shows a closed thin-walled body consisting of several parts constructed by extruding the multiple contours shown in Fig. 17.

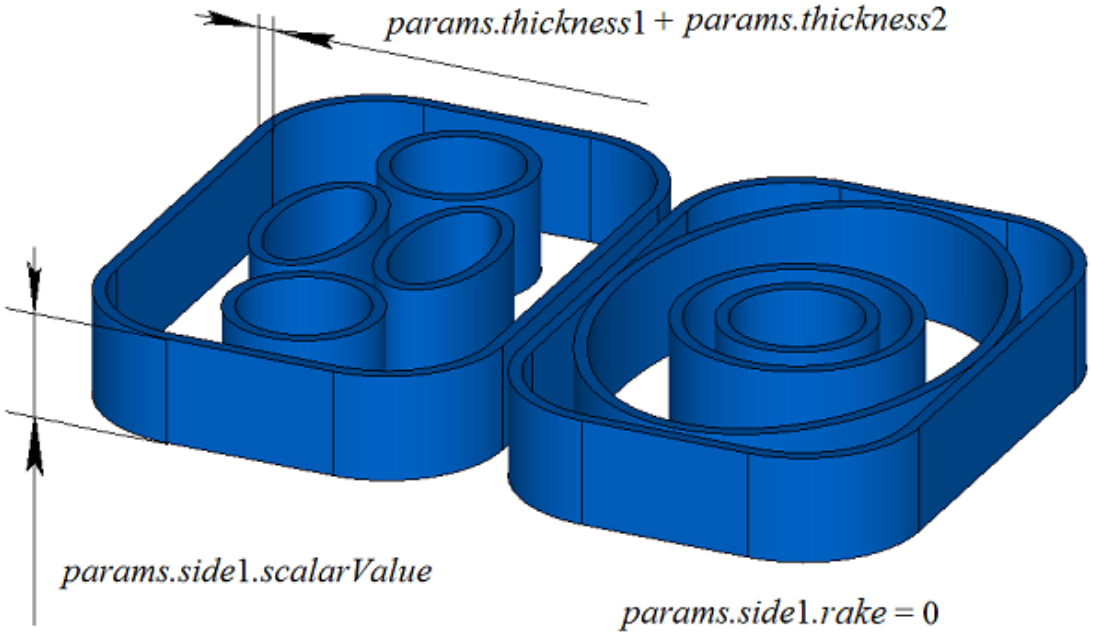


Fig. 20

Fig. 21 shows two three-dimensional contours:

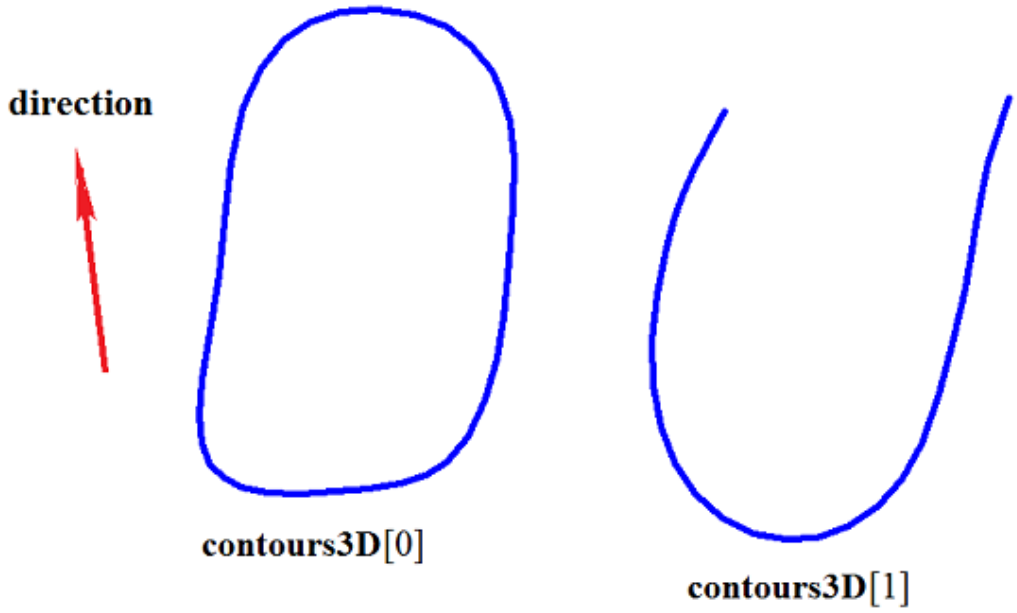


Fig. 21

Fig. 22 shows a doubly connected thin-walled closed body obtained by extruding the three-dimensional contours shown in Fig. 21.

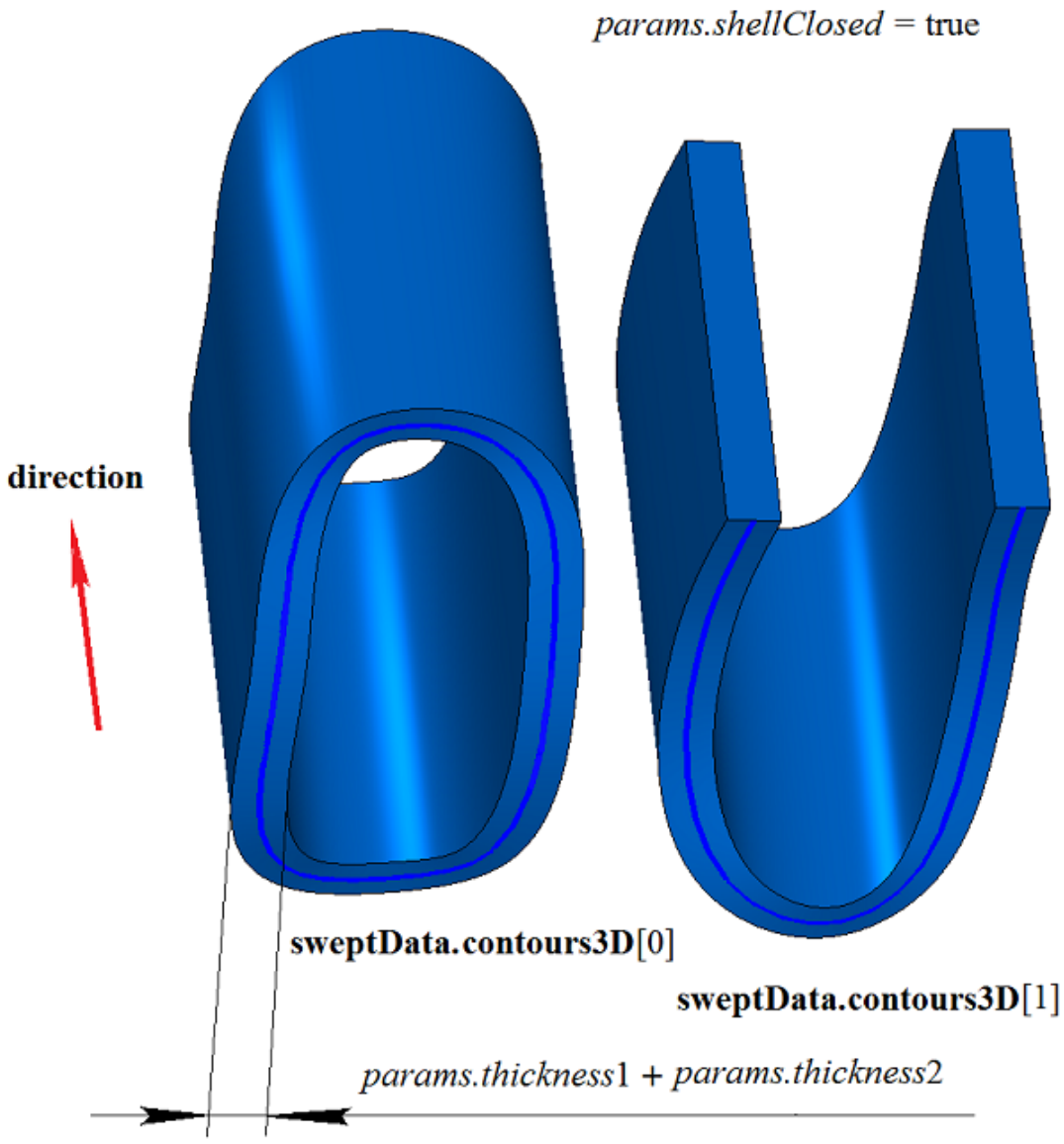


Fig. 22

Fig. 23 shows two open bodies obtained by extruding the three-dimensional contours shown in Fig. 21. The construction is performed separately for each contour.

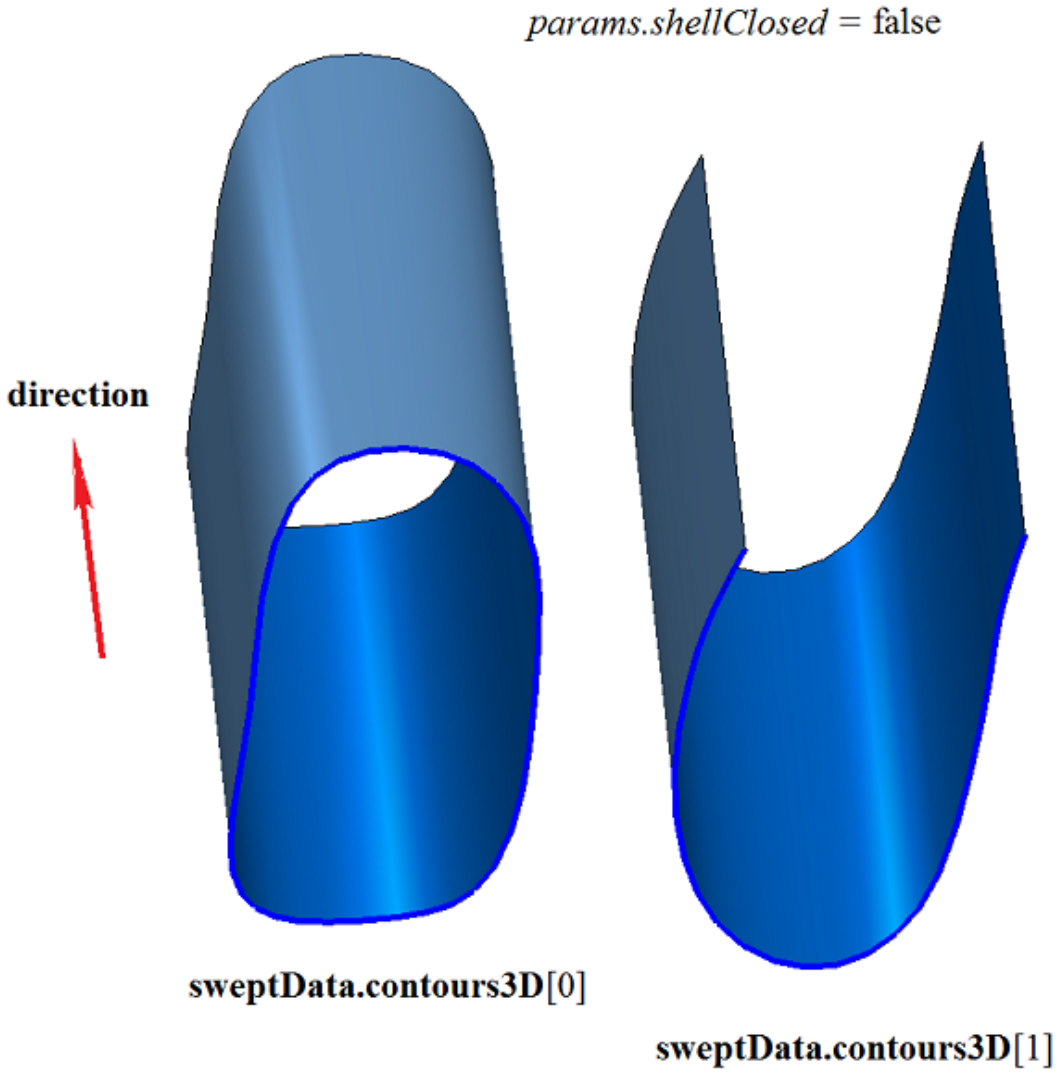


Fig. 23

The extrusion body construction function ExtrusionSolid() adds the MbCurveExtrusionSolid builder to the history of the constructed body, which contains all the data required for constructing the body. The MbCurveExtrusionSolid builder is declared in the *cr_extrusion_solid.h* file.

In the *test.exe* application, extrusion body construction is performed by the "Create->Body->Based on curves->By surface curve extrusion" and "Create->Body->Based on curves->By three-dimensional curve extrusion" menu commands.

3.1.3 Constructing a Revolution Body

The function

```
MbResultType RevolutionSolid( const MbSweptData &           sweptData,
                              const MbAxis3D &              axis,
```

(continues on next page)

(continued from previous page)

```
                                const RevolutionValues &      params,
                                const MbSNameMaker &          operNames,
                                      RPArray<MbSNameMaker> & contoursNames,
                                      MbSolid *&              result );
```

constructs a revolution body.

The input parameters of the function are:

- **sweptData** – data on generatrix curves;
- **axis** – axis of revolution;
- **params** – an object that contains revolution parameters;
- **operNames** – the operation name maker;
- **contoursNames** – the name makers of the segments of the generatrix contour.

The output parameter of the function is the constructed body (shell) **result**.

The function returns an operation result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

A revolution body belongs to a type of swept bodies obtained by moving a generatrix curve along a guide curve. The guide curve of a revolution body is a circle or its arc. The revolution body is constructed by rotating one or several curves around the **axis**.

The **sweptData** parameter contains information about the generatrix curves. The MbSweptData and RevolutionValues classes are described in the *op_swept_parameter.h* file. The generatrix curves may represent two-dimensional **contours** on a **surface** or **contours3D** in space. In a special case, the **surface** may represent a plane. The orientation of the **contours** may be arbitrary. The **contours** may be nested inside each other.

Rotation of curves may be performed in the forward direction relative to the **axis** axis, in the reverse direction relative to the **axis** axis, and in both directions. Rotation in the forward direction is performed counterclockwise when looking towards the **axis** axis. The construction parameters in each direction are defined by MbSweptSide objects.

The **params** parameter contains two MbSweptSide class objects – **side1** and **side2**, which store revolution parameters in the forward and reverse directions.

The revolution method is defined in the **way** field of the MbSweptSide class object:

- If **way** = *sw_scalarValue*, the rotation is performed through the angle **scalarValue** in the **side1** or **side2** direction. The rotation angle is defined in radians.
- If **way** = *sw_surface*, the rotation is performed up to the **side1**.*surface* or **side2**.*surface* surface if **side1**.*distance* equals zero or **side2**.*distance* equals zero, respectively.
- If **way** = *sw_surface* and **distance** is not equal to zero, the rotation is performed up to a surface equidistant from **side1**.*surface* or up to a surface equidistant from **side2**.*surface*, respectively.
- If **way** = *sw_shell*, the rotation is performed up to the nearest object (body) in the **side1** or **side2** direction.

The **params**.*thickness1* and **params**.*thickness2* parameters define the wall thickness of a thin-walled body. The **params**.*thickness1* parameter specifies the offset outward from the generatrix

curve, and the **params**.*thickness2* parameter specifies the offset inward from the generatrix curve.

The **params**.*shellClosed* parameter controls whether the constructed body is closed.

The **params**.*checkSelfInt* parameter indicates the need to check the construction result for self-intersection. By default, **params**.*checkSelfInt* = *false*, and the check is not performed.

The **params**.*shape* parameter controls the shape of the constructed body. If **params**.*shape* = 1, the constructed body has torus topology; if **params**.*shape* = 0, the constructed body has sphere topology. If the generatrix is an open planar curve and the axis of revolution lies in the plane of the curve, then it is possible to construct a revolution body with sphere-type topology. In this case, the generatrix is extended up to the axis of revolution.

Fig. 1 shows the data used in the construction and the inheritance scheme of revolution body construction parameters RevolutionValues & **params**.

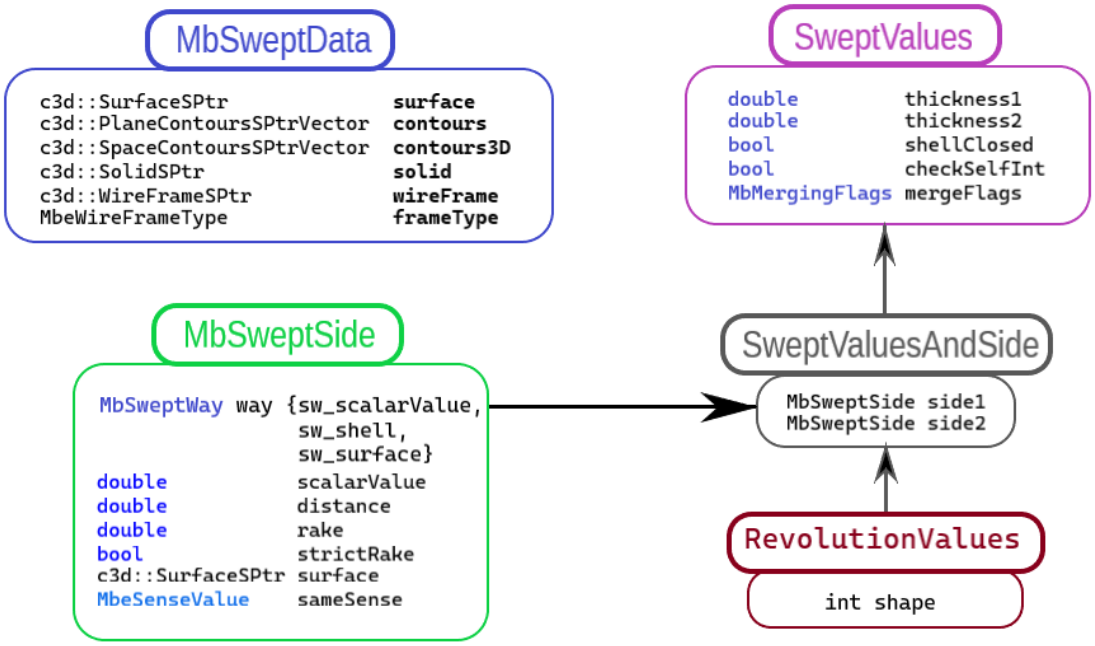


Fig. 1

The **operNames** and **contoursNames** parameters provide naming of faces, edges, and vertices of the constructed body.

Fig. 2 shows a two-dimensional contour **contour**, a planar surface **surface** (MbPlane), and a revolution axis **axis**.

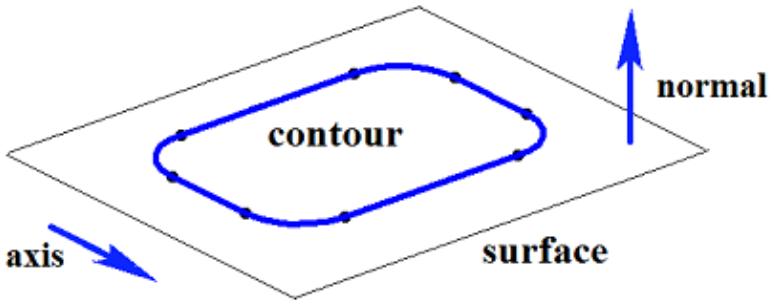


Fig. 2

Fig. 3 shows a closed body constructed by rotation using the specified parameters of the contour shown in Fig. 2.

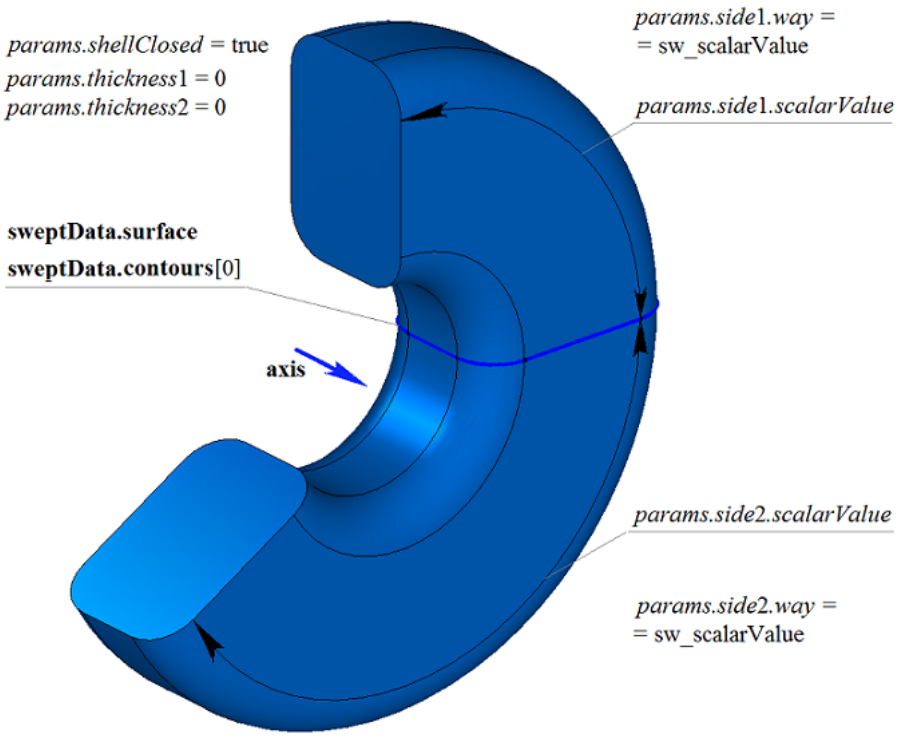


Fig. 3

Fig. 4 shows a closed thin-walled body constructed by rotation using the specified parameters of the contour shown in Fig. 2.

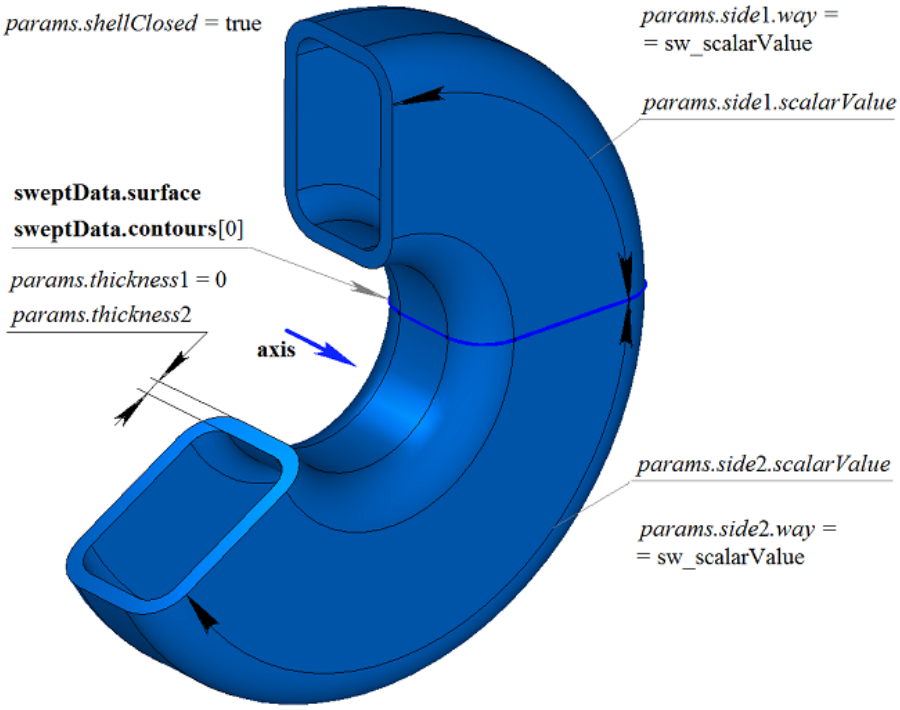


Fig. 4

Fig. 5 shows an open body constructed by rotation using the specified parameters of the contour shown in Fig. 2. The construction parameters of the body shown in Fig. 3 differ from those of the body shown in Fig. 5 only by the **params**.*shellClosed* value.

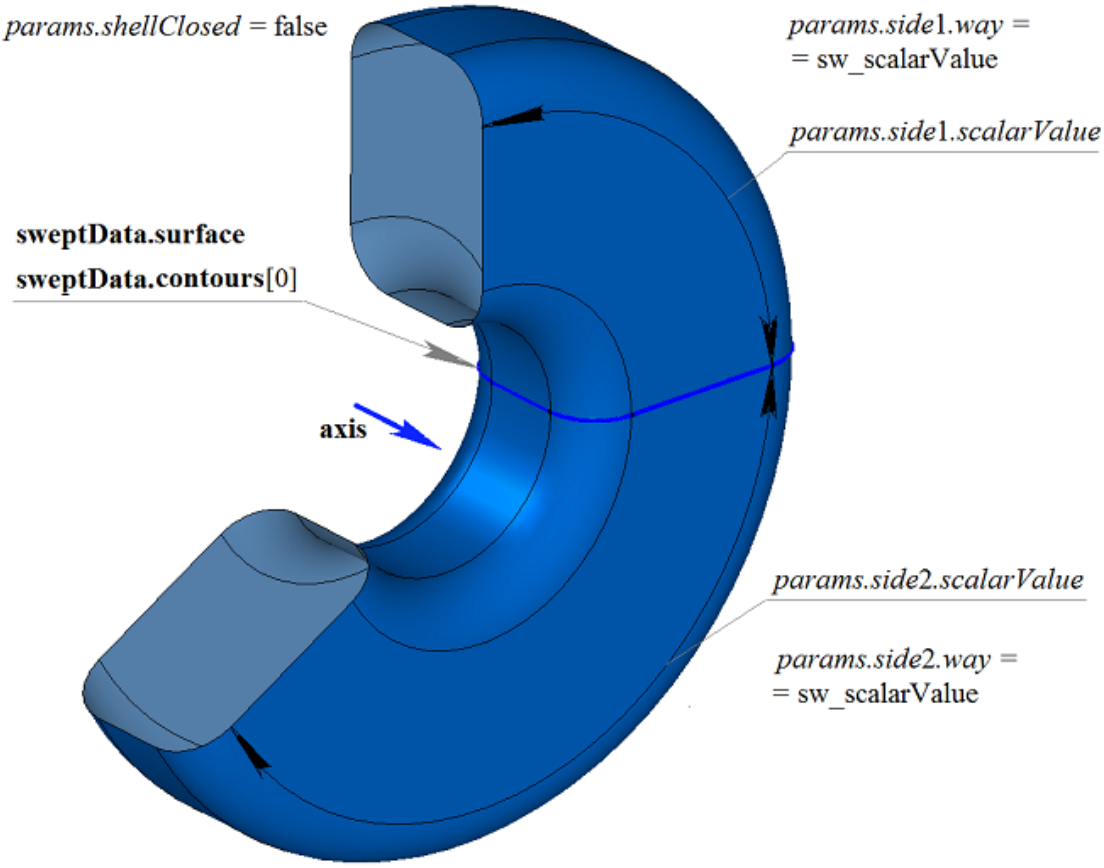


Fig. 5

Fig. 6 shows a two-dimensional contour **contour**, a planar surface **surface** (MbPlane) and two surfaces **surface1** and **surface2** that will be used in the revolution body construction. For the construction, the **surface1** and **surface2** surfaces must completely cover the path of contour motion in the corresponding direction. At the same time, the **params**.*thickness1* and **params**.*thickness2* parameters must be taken into account. The revolution body is cut by the specified surfaces (if **params**.*side1.distance* or **params**.*side2.distance* equal zero) or by surfaces equidistant from them (if **params**.*side1.distance* or **params**.*side2.distance* are not equal to zero).

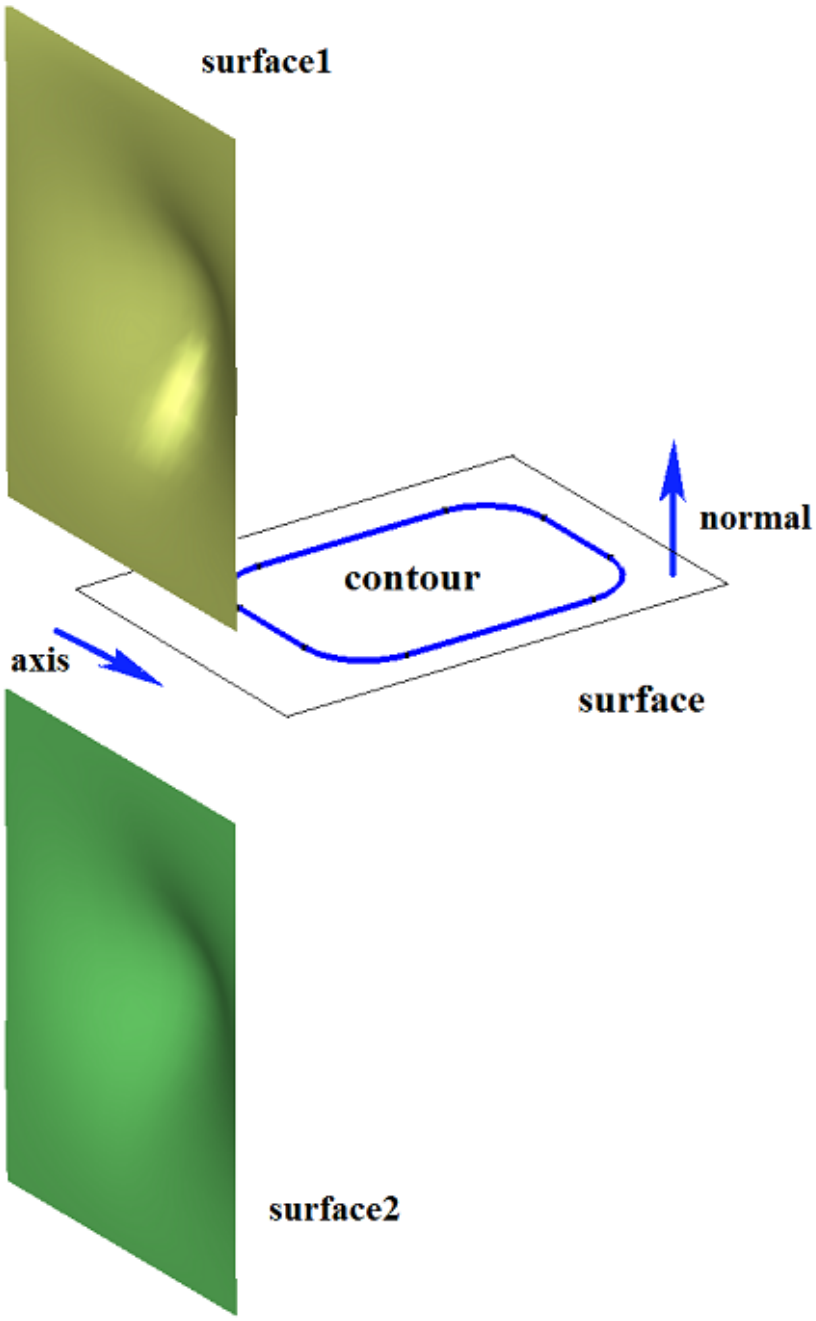


Fig. 6

Fig. 7 shows a body constructed by rotation of the contour shown in Fig. 6, with the "Up to surface" options (**params**.*side1(2).way* = *sw_surface*), where **surface1** and **surface2** are specified.

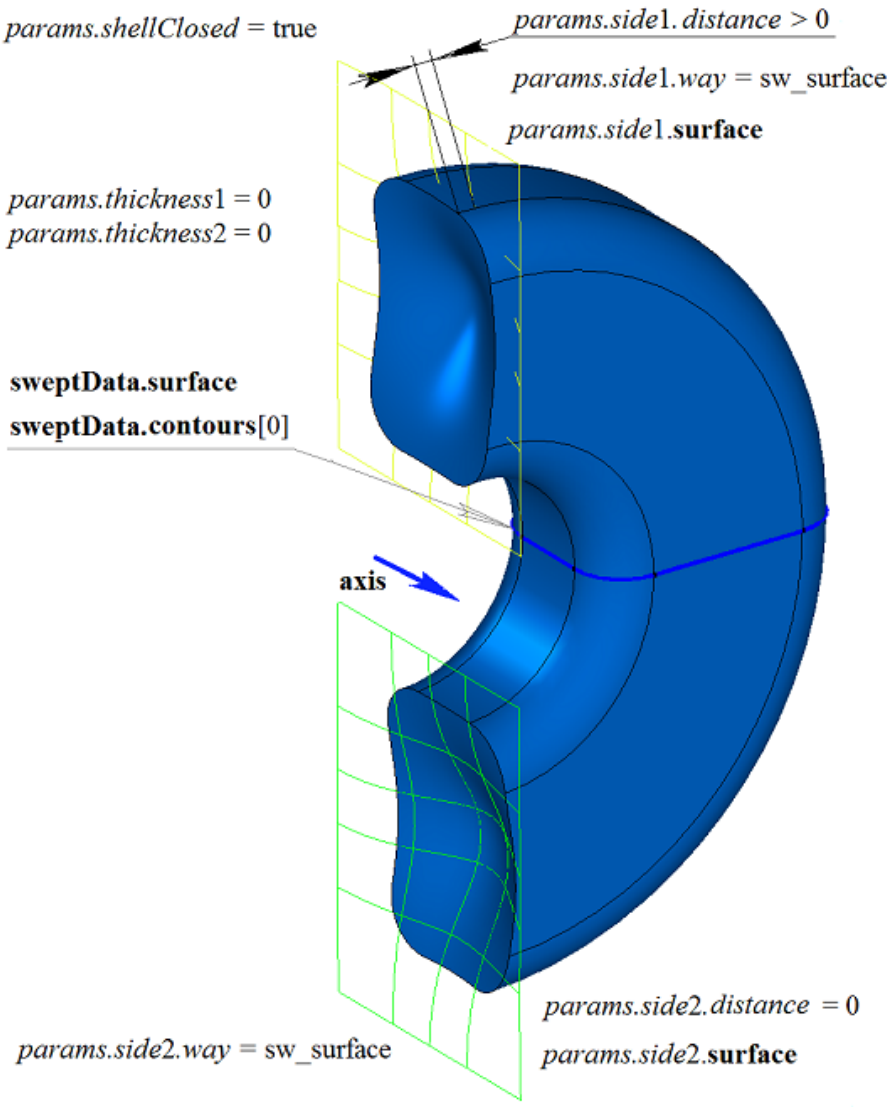


Fig. 7

Fig. 8 shows a thin-walled body constructed by rotation of the contour shown in Fig. 6, with the “Up to surface” options, where **surface1** and **surface2** are specified. The revolution body is cut by the specified surfaces.

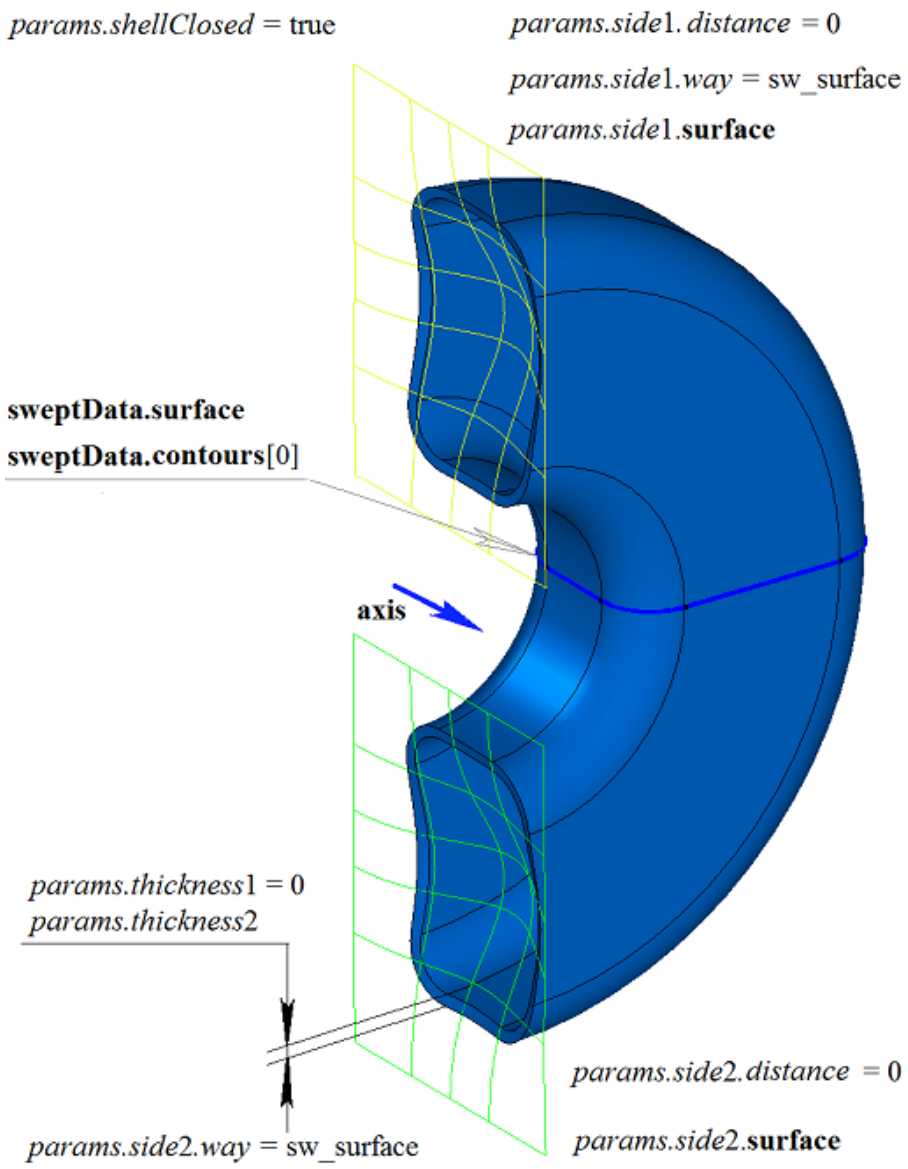


Fig. 8

A two-dimensional contour may lie on a planar or a curved surface. For example, a body can be constructed by rotation of a contour on a curved surface obtained from a loop of one of the faces of a solid body shown in Fig. 9.

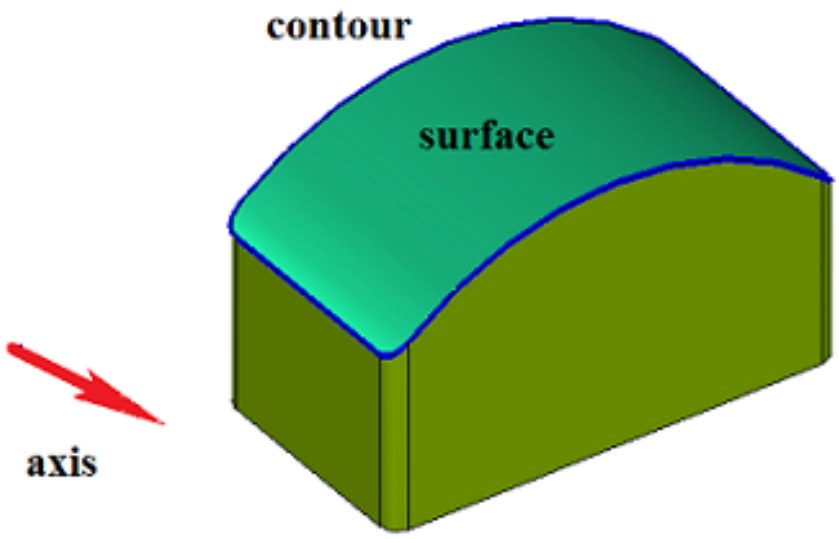


Fig. 9

Fig. 10 shows a closed body obtained by rotation of the contour on the curved surface shown in Fig. 9.

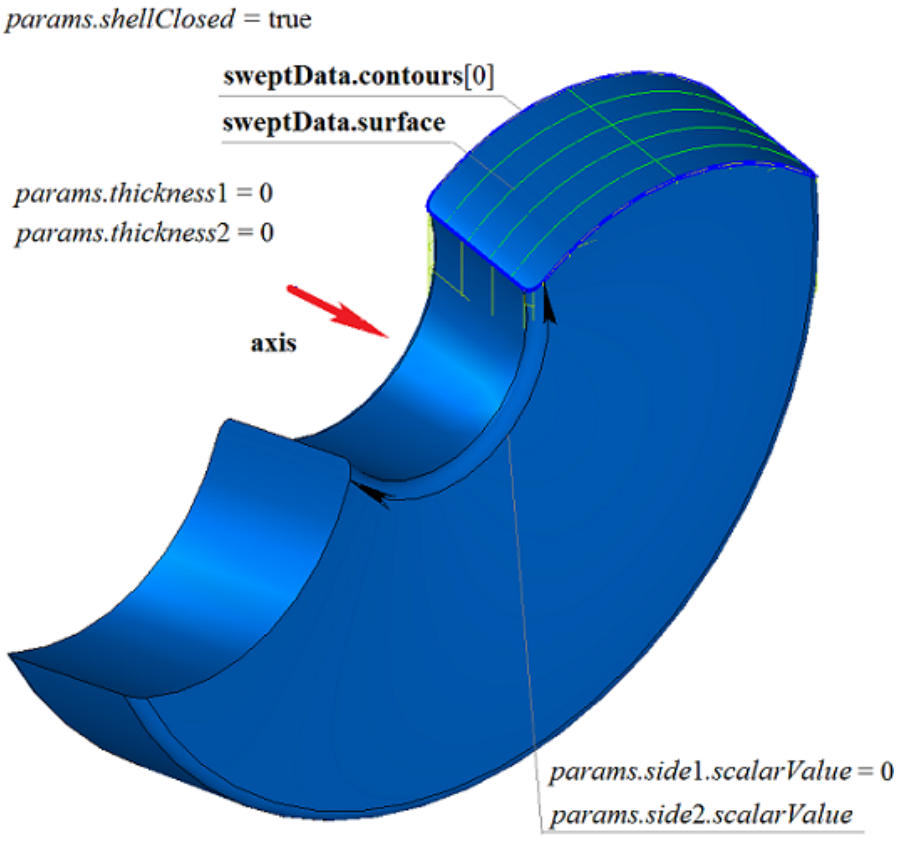


Fig. 10

Fig. 11 shows a closed thin-walled body obtained by rotation of the contour on the curved sur-

face shown in Fig. 9.

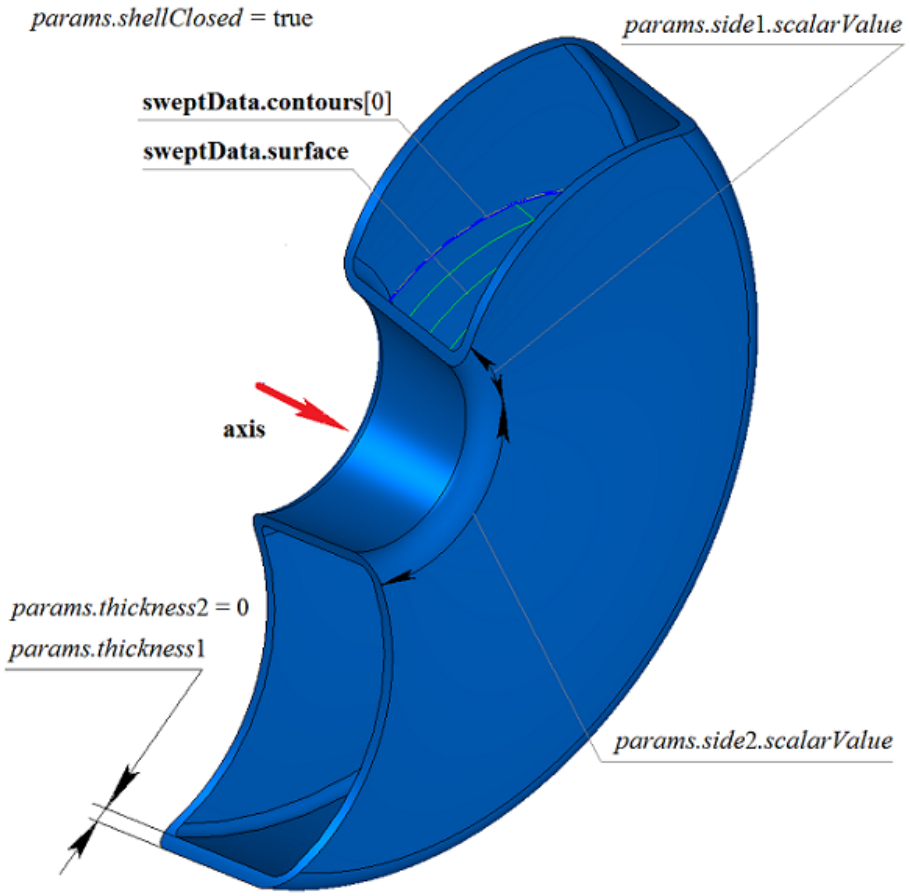


Fig. 11

Fig. 12 shows an open body obtained by rotation of the contour on the curved surface shown in Fig. 9.

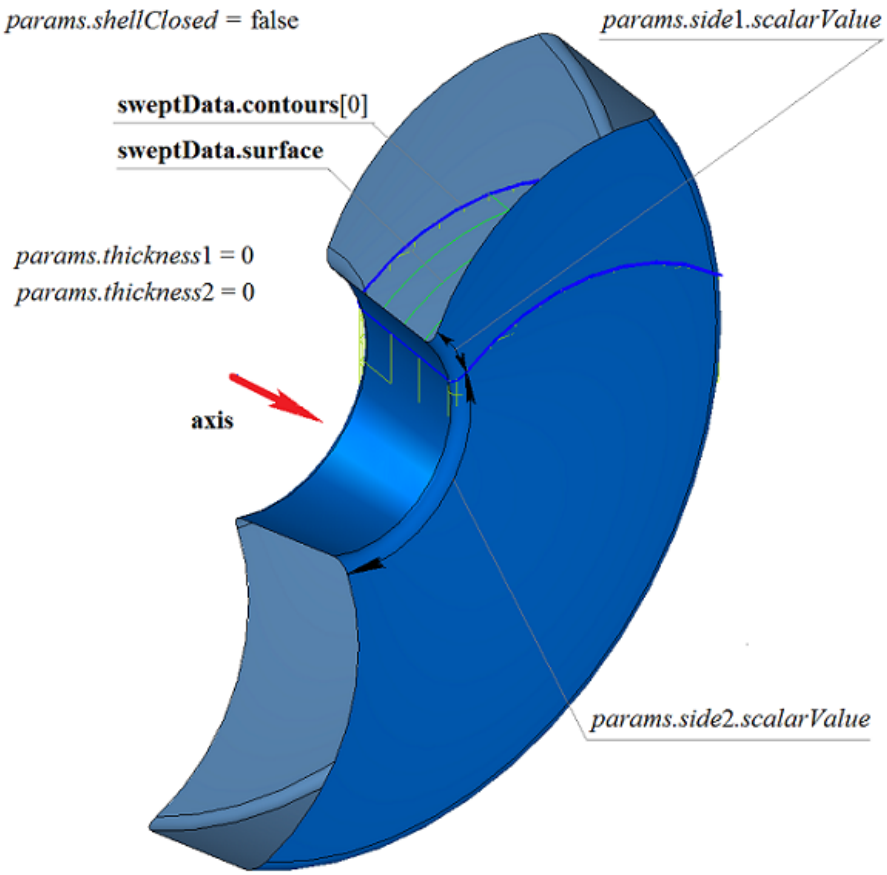


Fig. 12

If multiple non-intersecting two-dimensional contours lie on one surface, the considered function determines outer contours and inner contours nested in them, with potentially multiple nesting levels. Fig. 13 shows multiple non-intersecting two-dimensional **contours** and a planar surface **surface** (MbPlane).

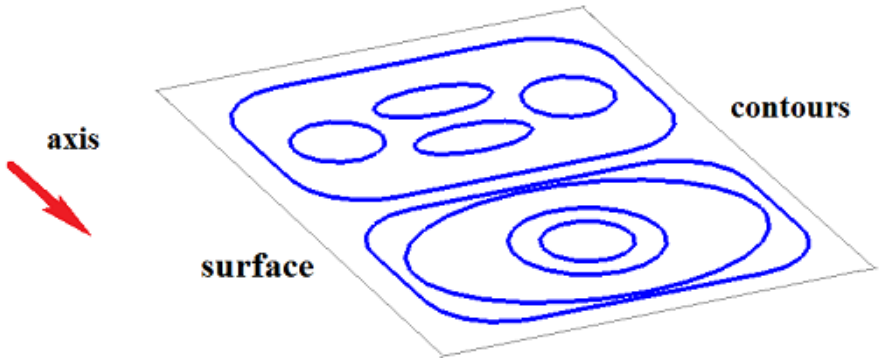


Fig. 13

Fig. 14 shows a closed body consisting of several parts constructed by rotation of the multiple contours shown in Fig. 13.

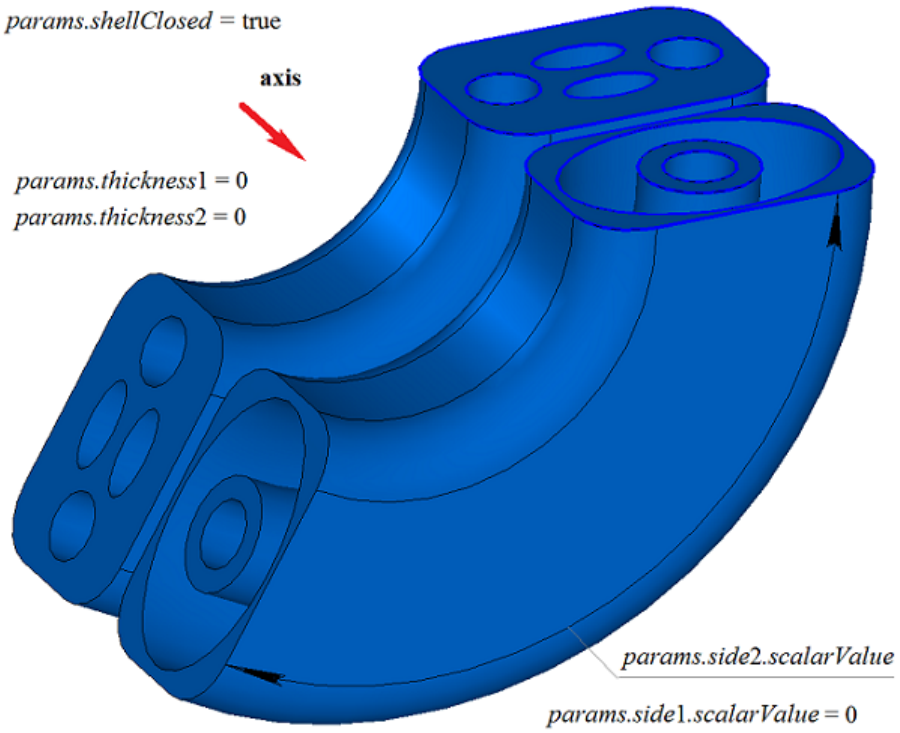


Fig. 14

Fig. 15 shows a closed thin-walled body consisting of several parts constructed by rotation of the multiple contours shown in Fig. 13.

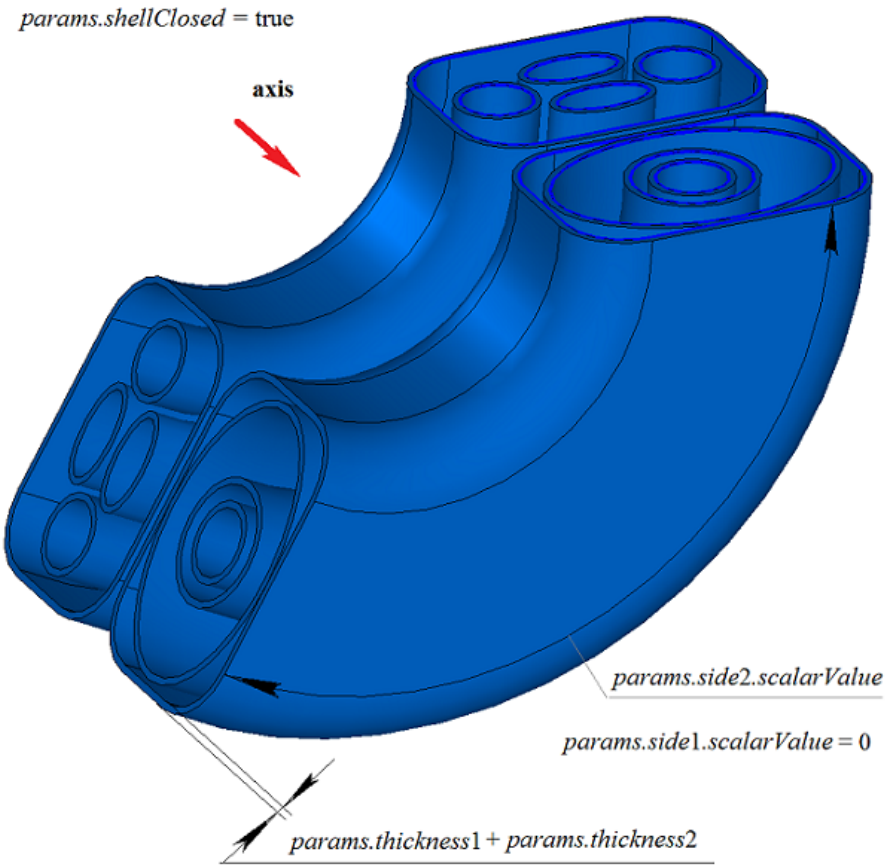


Fig. 15

Fig. 16 shows two three-dimensional contours.

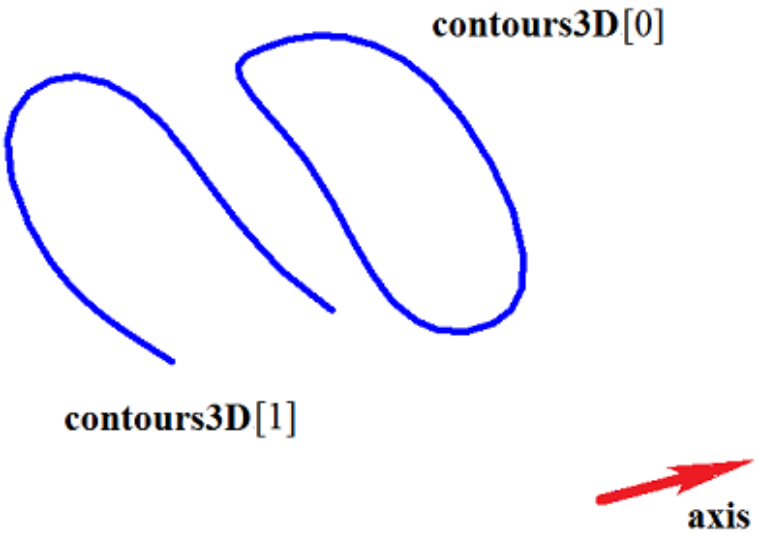


Fig. 16

Fig. 17 shows a doubly connected thin-walled closed body obtained by rotation of the three-dimensional contours shown in Fig. 16.

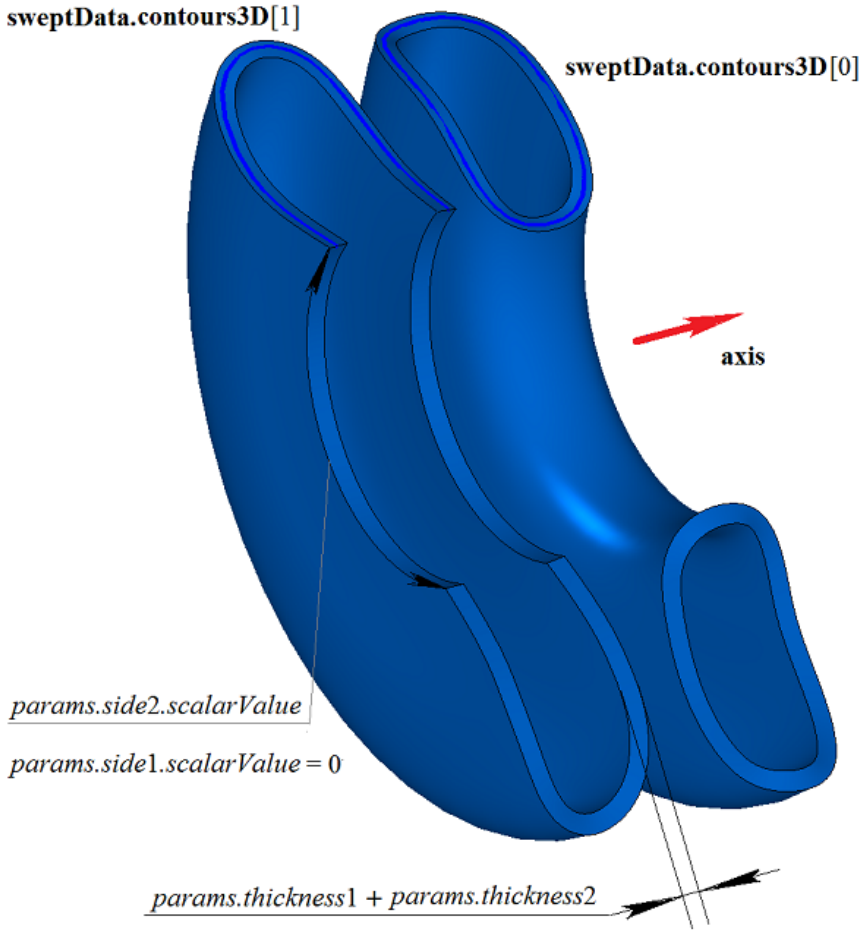


Fig. 17

Fig. 18 shows two open bodies obtained by rotation of the three-dimensional contours shown in Fig. 16. The construction is performed separately for each contour.

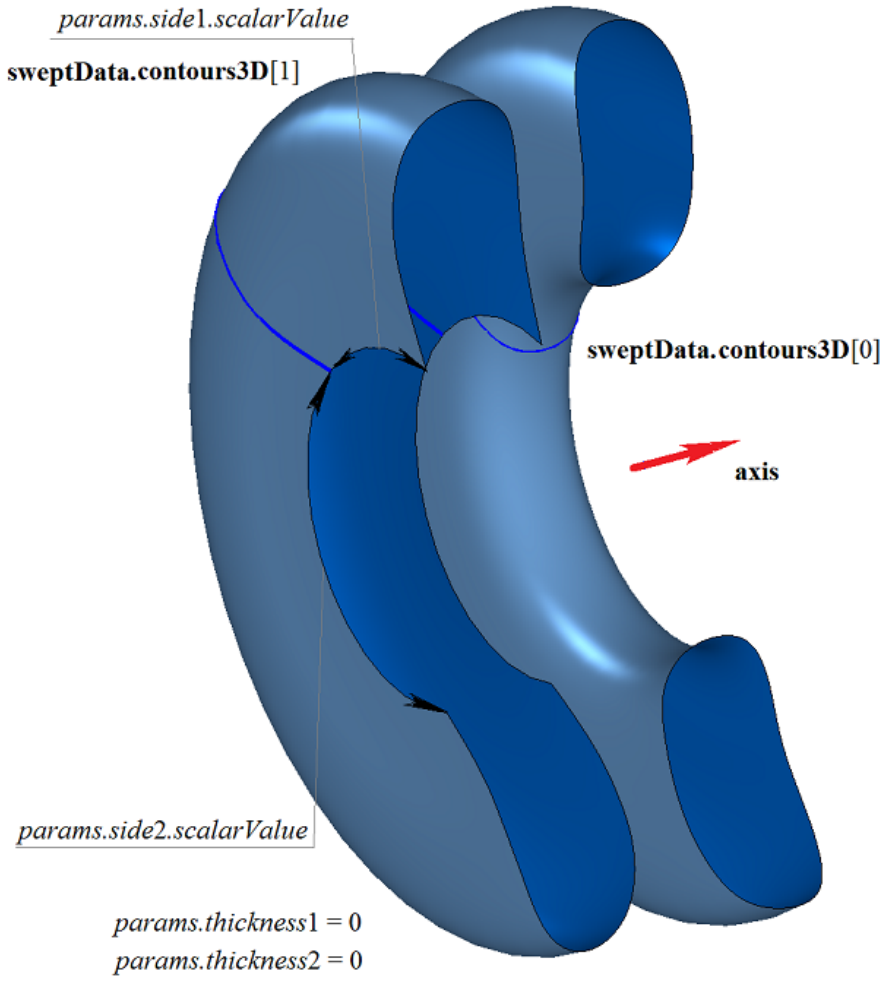


Fig. 18

When constructing a revolution operation up to a construction plane with a degenerate definition domain, the shell is constructed up to the nearest intersection with the specified plane.

If the plane has non-degenerate boundaries, the part of the plane up to which the rotation is constructed is determined by the position of the center of gravity of the contour bounding the plane relative to the projection of the revolution axis onto the plane.

The revolution body construction function RevolutionSolid() adds the MbCurveRevolutionSolid builder to the history of the constructed body, which contains all the data required for constructing the body. The MbCurveRevolutionSolid builder is declared in the *cr_revolution_solid.h* file.

In the *test.exe* test application, revolution body construction is performed by the “Create->Solid->Based on curves->Revolution of a curve lying on surface” and “Create->Solid->Based on curves->Revolution of a spatial curve” menu commands.

3.1.4 Constructing a Body by Sections

The function

```
MbResultType LoftedSolid( const MbLoftedSolidParams & loftSolParams,
                                c3d::SolidSPtr      & result );
```

constructs a body using spatial (including planar) sections.

The input parameter of the function is the **loftSolParams** object containing the initial data for the operation (definition of the MbLoftedSolidParams class, see *op_shell_parameter.h*).

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

The function returns a result code of the operation from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

The following constructors of the MbLoftedSolidParams class initialize the **loftSolParams** object with body construction parameters (see *op_shell_parameter.h*):

```
MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );

MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );

MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const OperationType                   oType,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );

MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
```

(continues on next page)

(continued from previous page)

```
                    const LoftedValues &                  params,
                    const OperationType                   oType,
                    const c3d::SpaceCurvesSPtrVector *    guideCurves,
                    const c3d::SpacePointsVector *        guidePoints,
                    const MbSNameMaker &                  operName,
                    const c3d::SNamesMakerSPtrVector &    contourNames,
                          IProgressIndicator *            prog = nullptr );
```

where

- **places** – a set of local coordinate systems for generatrix contours.
- **planeConours** – a set of generatrix contours.
- **surfaces** – a set of surfaces of generatrix contours.
- **oType** – Boolean operation type.
- **spine** – spine curve (may be nullptr).
- **params** – operation parameters.
- **guideCurves** – a set of guide curves defining trajectories for corresponding contour points.
- **guidePoints** – a set of points on generatrix contours defining corresponding points (point chains).
- **operName** – operation namer.
- **contourNames** – generatrix contour namers.
- **prog** – operation progress indicator.

Constructing a Body by Planar Sections

The surface of a body constructed by planar sections passes through all planar curves defining the body. The **places** set contains local coordinate systems whose XY planes contain the 2D contours **planeConours**.

The **places** and **planeConours** sets are index-aligned: **planeConours**[*i*] lies in the XY plane of the local coordinate system **places**[*i*]. The orientation of **planeConours** can be arbitrary. If all **planeConours** are closed, contour start positions are adjusted to prevent surface twisting by placing the starts as close to each other as possible. The **guidePoints** control points allow changing the correspondence of curve junction points between contours. If the **guidePoints** set is non-empty, it must be coordinated with the **places** and **planeConours** sets. Guide curves **guideCurves** can be used to control the body shape between sections. Any curve can serve as the body spine curve.

The **params** parameter contains information about the movement method, presence and wall thickness of the body, and whether the constructed body is closed.

The **params**.*thickness1* and **params**.*thickness2* parameters define the wall thickness of the constructed thin-walled body. The **params**.*thickness1* parameter specifies the outward offset from the generatrix curve, and the **params**.*thickness2* parameter specifies the inward offset from the generatrix curve.

The **params**.*shellClosed* parameter controls whether the constructed body is closed (*true* – closed body; *false* – non-closed body).

The **params**.*checkSelfInt* parameter indicates whether the construction result should be checked for self-intersection. By default, **params**.*checkSelfInt* = *false*, and the check is not performed.

The **mergeFlags** control flags manage merging of similar faces or edges (**mergeFaces** = *true* – merge similar faces; **mergeEdges** = *true* – merge similar edges).

The **params**.*closed* parameter controls the closedness of the section tube (presence of end caps). When **params**.*closed* = *true*, end caps are absent and the body has torus topology. By default, **params**.*closed* = *false*.

The **params**.*vector1* and **params**.*vector2* vectors define the body direction near the initial and final end caps. For example, they allow setting the body direction near end caps orthogonal to end cap planes. By default, **params**.*vector1* and **params**.*vector2* are zero.

The **params**.*setNormal1* and **params**.*setNormal2* parameters set normals at the initial and final end caps respectively if the initial section is a point (by default *false*).

The **params**.*derFactor1* and **params**.*derFactor2* parameters set derivative magnitude multipliers when setting normals at the initial and final end caps respectively.

The **params**.*directSurf1* and **params**.*directSurf2* vectors define the surface movement direction axis at the initial and final end caps respectively when setting normals.

Fig. 1 shows the input data used for construction and the parameter inheritance scheme for constructing a body by planar sections MbLoftedSolidParams & **loftSolParams**:

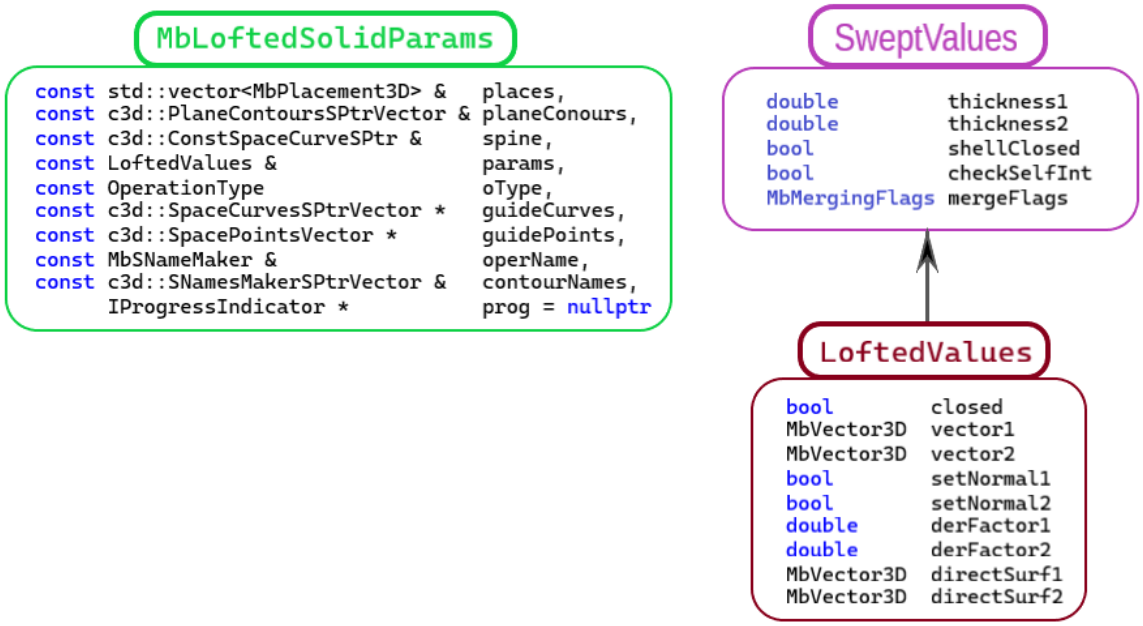


Fig. 1

The **contourNames** parameter provides naming for the faces of the constructed body.

Fig. 2 shows a set of 2D contours **planeConours** and their local coordinate systems **places** (arrows indicate the normal directions of the local coordinate systems):

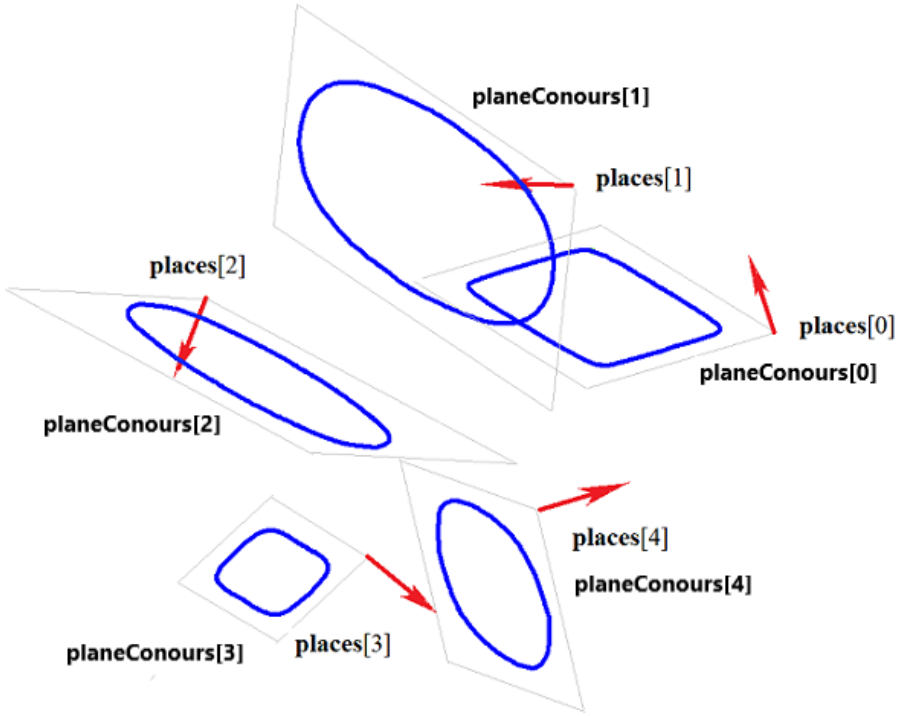


Fig. 2

Fig. 3 shows a closed body constructed by planar sections shown in Fig. 2, with specified normal directions at end caps for **params**.*closed* = *false*.

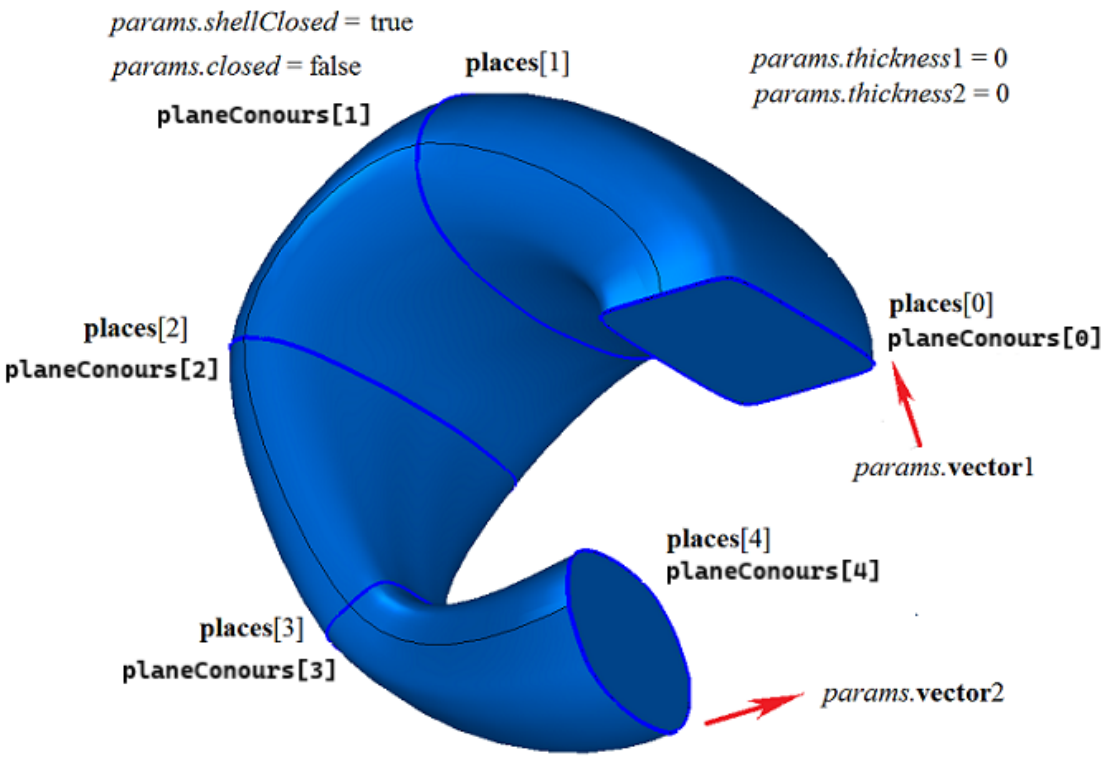


Fig. 3

Fig. 4 shows a closed body constructed by planar sections shown in Fig. 2, with **params**.*closed* = *true*. The constructed body has torus topology and lacks end caps.

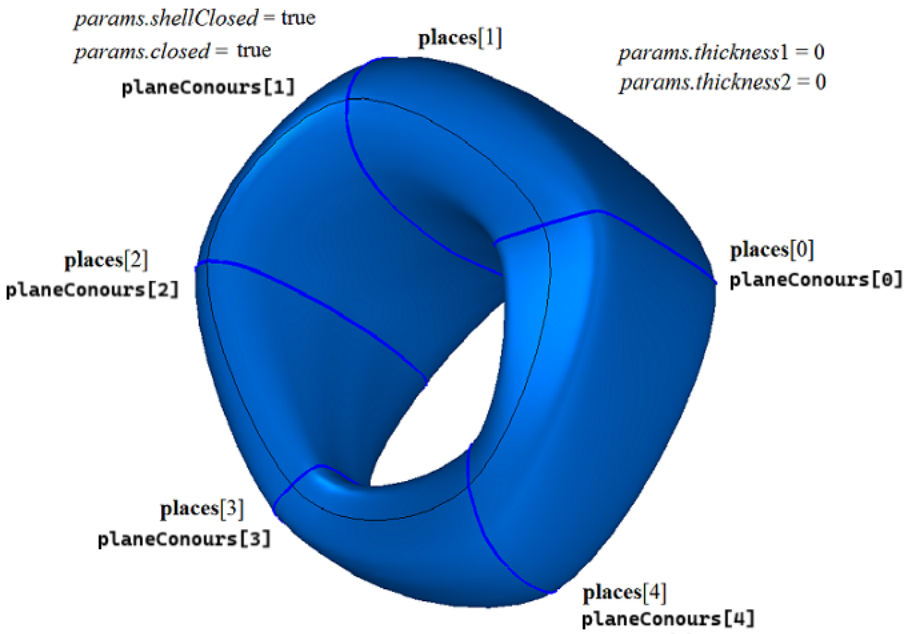


Fig. 4

Fig. 5 shows a closed thin-walled body constructed by planar sections shown in Fig. 2, without specified normals at end caps for **params**.*closed* = *false*:

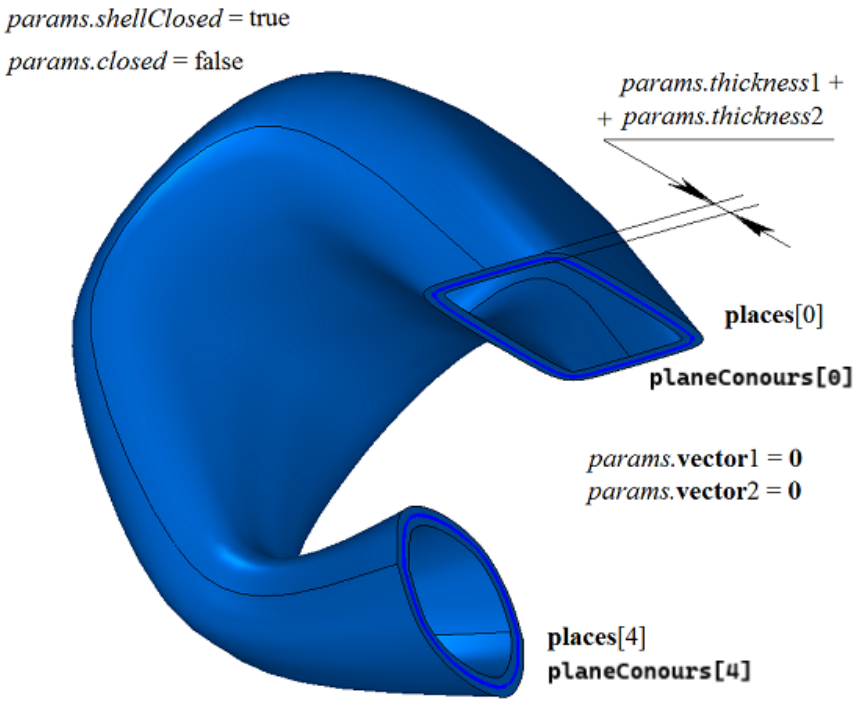


Fig. 5

Fig. 6 shows a closed thin-walled body constructed by planar sections shown in Fig. 2, with specified normals at end caps for **params**.*closed* = *false*:

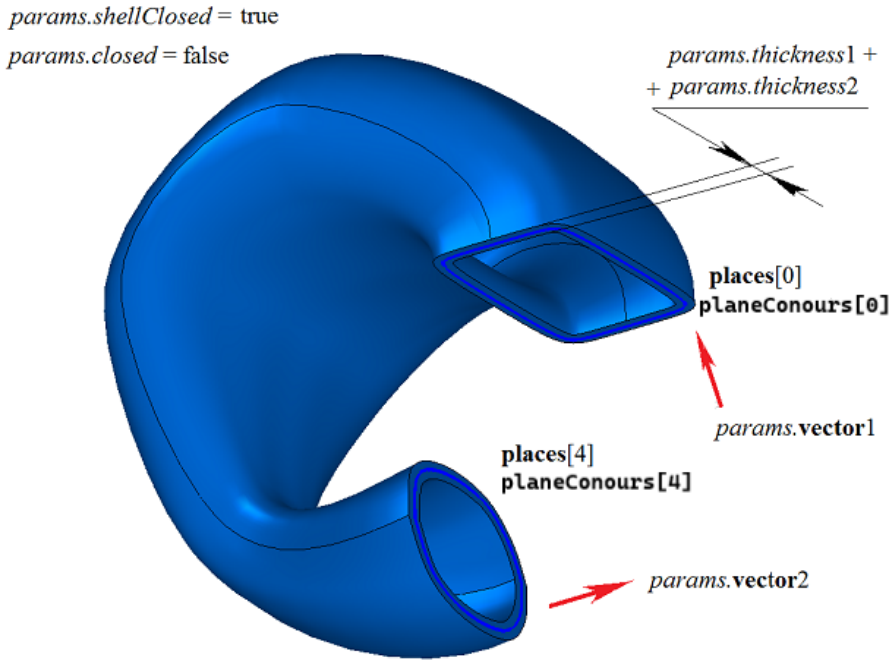


Fig. 6

Fig. 7 shows a closed thin-walled body constructed by planar non-closed contours without specified normals at end caps for **params**.*closed* = *false*. The **params**.*thickness1* and **params**.*thickness2* parameters must be non-zero.

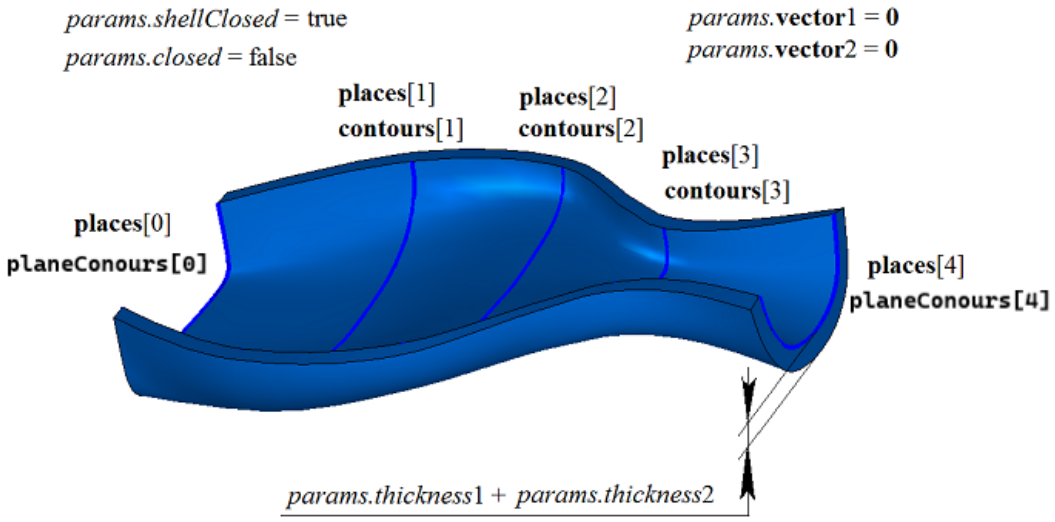


Fig. 7

Fig. 8 shows a non-closed body constructed by planar non-closed contours without specified normals at end caps for **params**.*closed* = *false*. The **params**.*thickness1* and **params**.*thickness2* parameters can be zero.

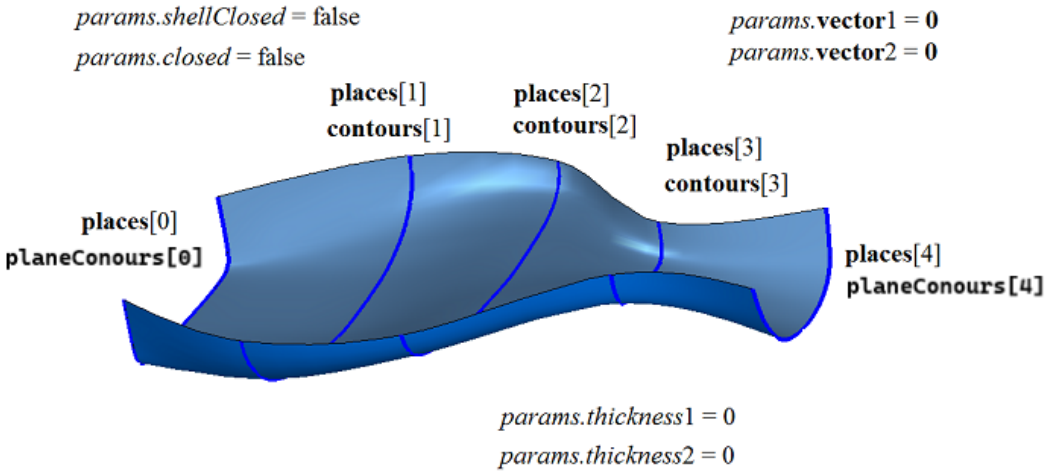


Fig. 8

When planar contours have different numbers of segments, some segments are subdivided so that each contour in the **planeConours** set has the same number of segments. Fig. 9 shows three contours with different numbers of segments.

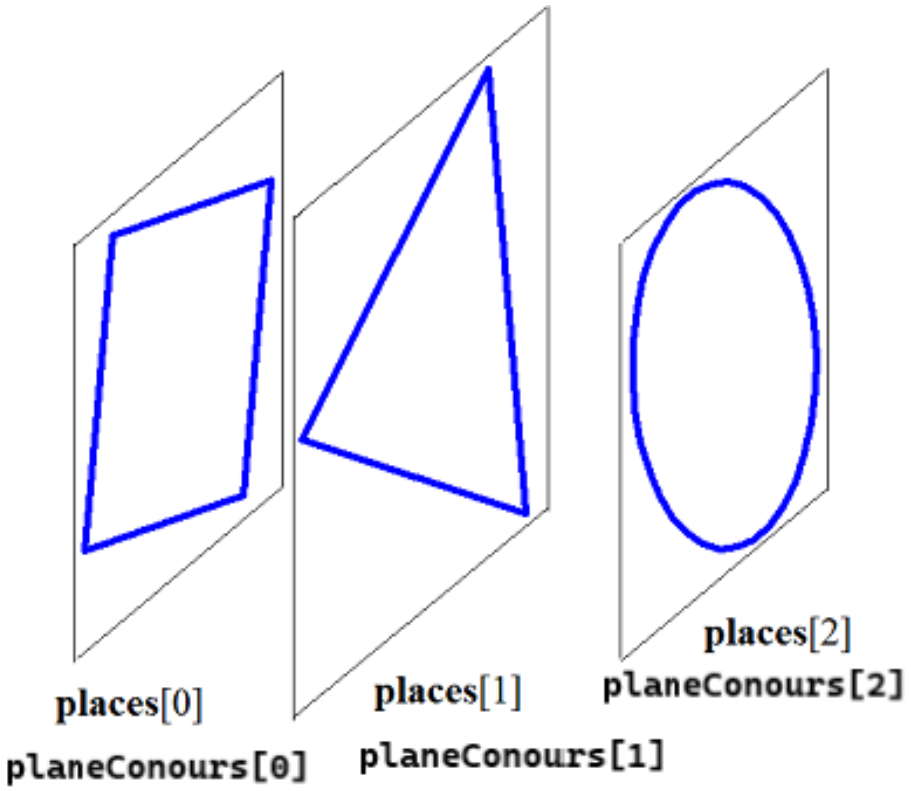


Fig. 9

Fig. 10 shows a closed body constructed from these contours, where one segment of the triangular contour is subdivided into two segments, and the circle is subdivided into four arcs:

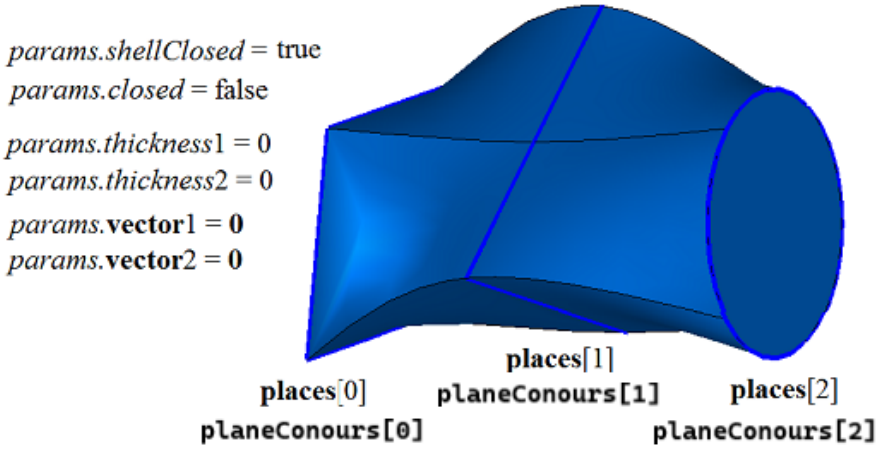


Fig. 10

The position of edges connecting vertices of different contours can be controlled using **guidePoints** control points. The **guidePoints**[*i*] points indicate segment junction positions between different contours that should be connected by edges. To demonstrate the use of **guidePoints** control points, consider constructing a body using planar sections shown in Fig. 11:

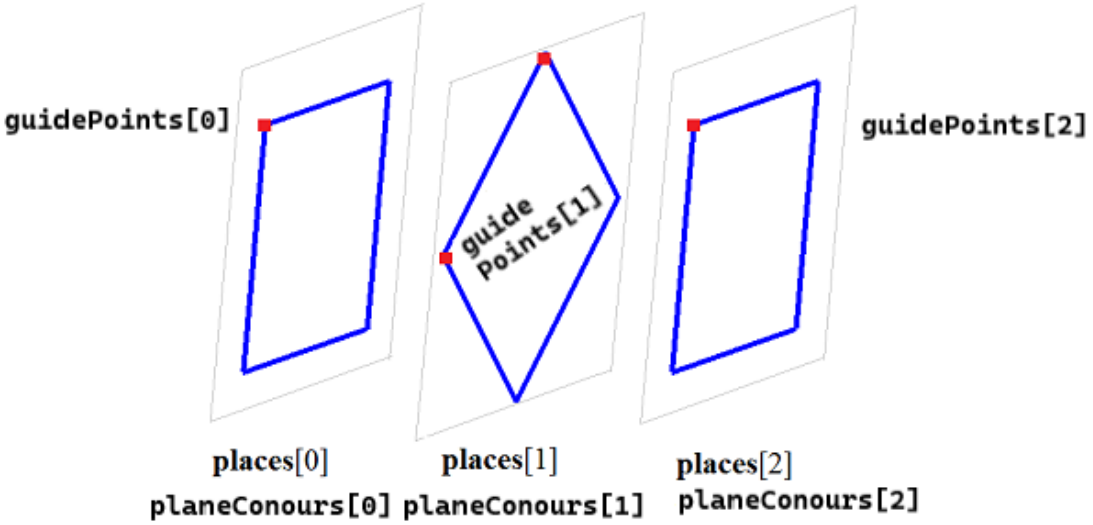


Fig. 11

Fig. 12 and 13 show bodies constructed using planar sections from Fig. 11 with different positions of **guidePoints** control points.

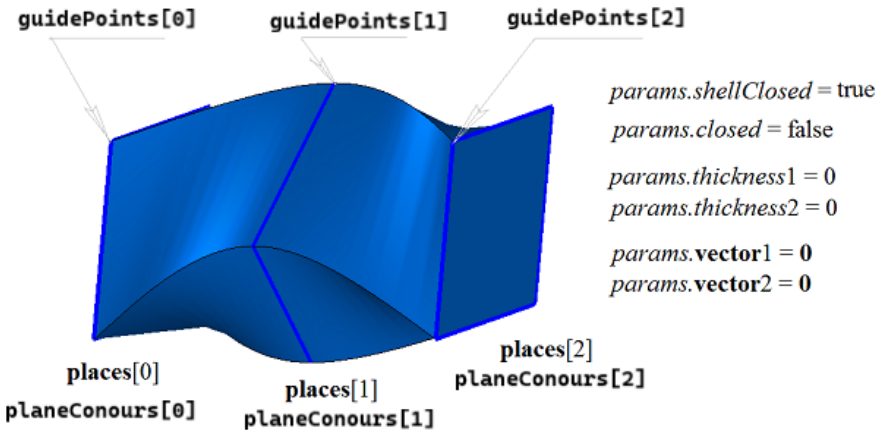


Fig. 12

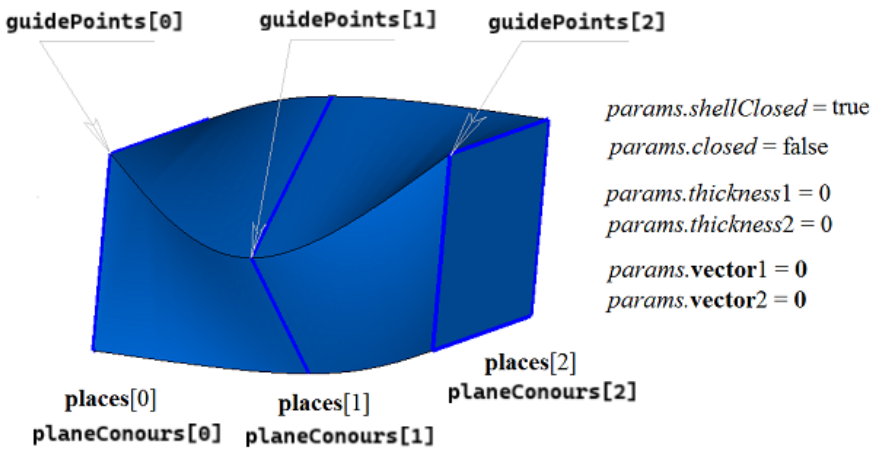


Fig. 13

Fig. 14 shows two 2D contours and a spine curve **spine** that will serve as a guide for constructing a body by planar sections:

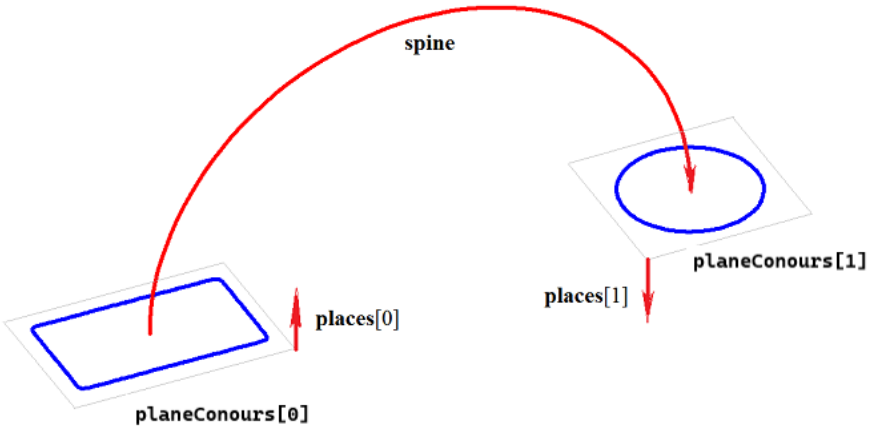


Fig. 14

Fig. 15 shows a closed body constructed by planar sections and spine curve shown in Fig. 14:

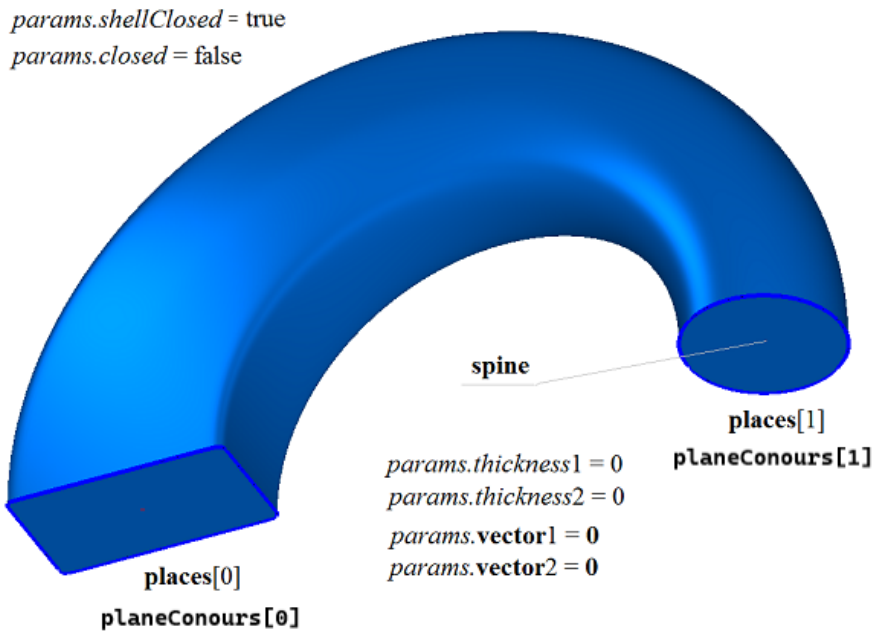


Fig. 15

Fig. 16 shows a closed thin-walled body constructed by planar sections and spine curve shown in Fig. 14:

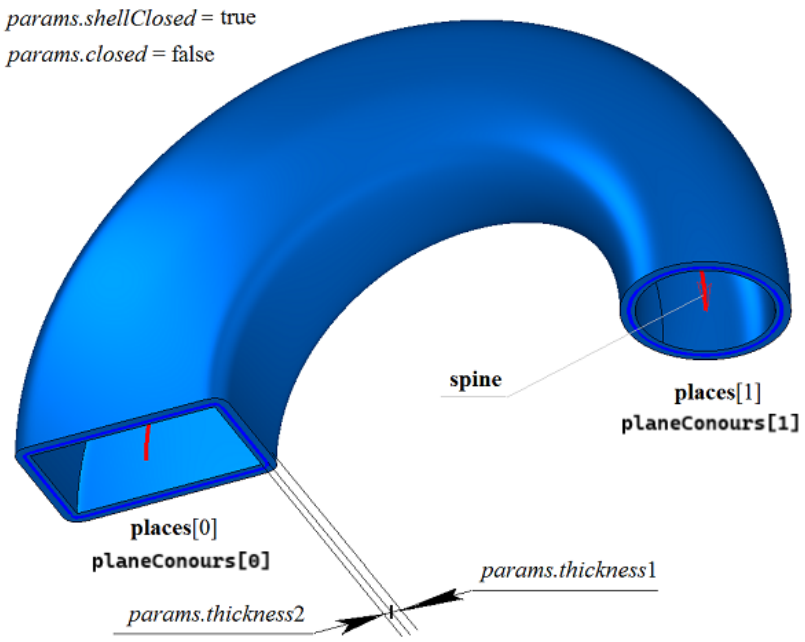


Fig. 16

Fig. 17 shows a non-closed body constructed by planar sections and spine curve shown in Fig. 14:

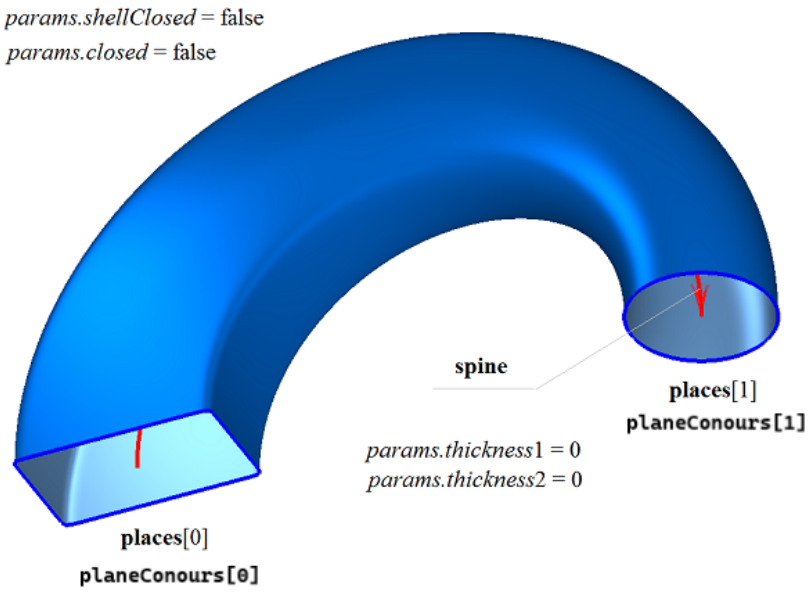


Fig. 17

The body construction function by sections LoftedSolid() adds the MbCurveLoftedSolid builder to the history of the constructed body, which contains all the data required for constructing the body. The MbCurveLoftedSolid builder is declared in the *cr_lofted_solid.h* file.

In the *test.exe* test application, body construction by planar sections is performed by the "Create→Solid→Based on curves→Lofting" and "Create→Solid→Based on curves→Lofting with a spine and guide curves" menu commands.

3.1.5 Creating a Body by a Specified Set of Faces or a Shell

The function

```
MbSolid * CreateSolid(       MbFaceShell &  shell,
                       const MbSNameMaker &  names );
```

creates a body with a specified set of faces or a shell without construction history.

The function takes an object **shell** of the MbFaceShell class as an input parameter (see the definition in *topology_faceset.h*), which contains the options of a specified set of faces or a shell.

If successful, the function returns a constructed body with no history, otherwise it returns zero.

This function is declared in the *action_solid.h* file.

The **names** parameter provides naming of faces, edges, and vertices of the constructed body.

A shell is a collection of MbFace objects that adjoin one another along MbCurveEdge edges.

In the general case, a shell can be multiply-connected and represent several mutually disconnected sets of faces.

A shell is called **closed** if it has no boundary edges; otherwise, the shell is called **open**.

The shell closure indicates the possibility of using many of its internal points in MbSolid body operations.

Formally, a set of faces imposes no restrictions, but in practice, the faces in the set satisfy certain conditions.

To emphasize this feature of shells, the concept of a “homogeneous shell” is introduced. Shells that satisfy the following requirements are called **homogeneous**:

1. Shells are finite.
2. Shells do not self-intersect.
3. Shells are two-sided (orientable).
4. No more than two faces meet at each edge of a shell. The two faces of the shell are joined such that the outer side of one face transitions continuously to the outer side of the other face.
5. Any closed shell traversal around any vertex along its surface must visit all faces adjacent to that vertex and cross all edges emanating from it.

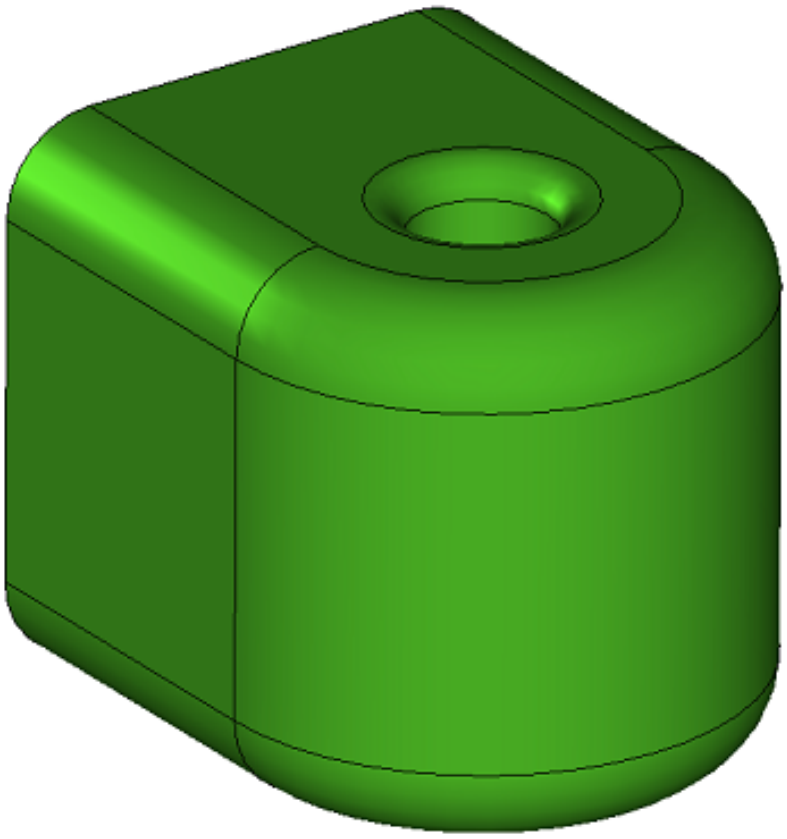
Since the outer side of a face transitions to the outer side of a neighboring face, a homogeneous shell possesses a well-defined outer side and inner side.

A closed homogeneous shell divides three-dimensional space into two parts, one of which is located inside the shell.

A closed homogeneous shell is called **external** if its outer side faces outward toward the region of space outside the shell-bounded region.

A closed homogeneous shell is called **internal** if its outer side faces inward toward the region of space bounded by the shell.

The figure below shows a body that was constructed by a set of faces or a shell:



shell

The function names unnamed faces, edges, and vertices and creates a solid from the specified shell. The function performs no checks or constructions. If the set of faces contains boundary edges, the function constructs an open solid. The CreateSolid() function adds a simple builder MbSimpleCreator to the log of the created solid, which is declared in the *cr_simple_creator.h* file.

3.1.6 Constructing an Open Body Based on a Surface

The function

```
MbResultType SurfaceShell( const MbSurfaceShellParams & params,
                           c3d::SolidSPtr &       result );
```

constructs a body consisting of one face based on the original surface. The surface must be without self-intersections, with the correct orientation of bounding curves in the case of the surface MbCurveBoundedSurface.

A **params** object of the class MbSurfaceShellParams is the input parameter of the function and contains the following parameters for a transmitted surface (see the definition in *op_shell_parameter.h*):

- **_surface** – is an original surface;
- **_operNames** – is the operation's namer;

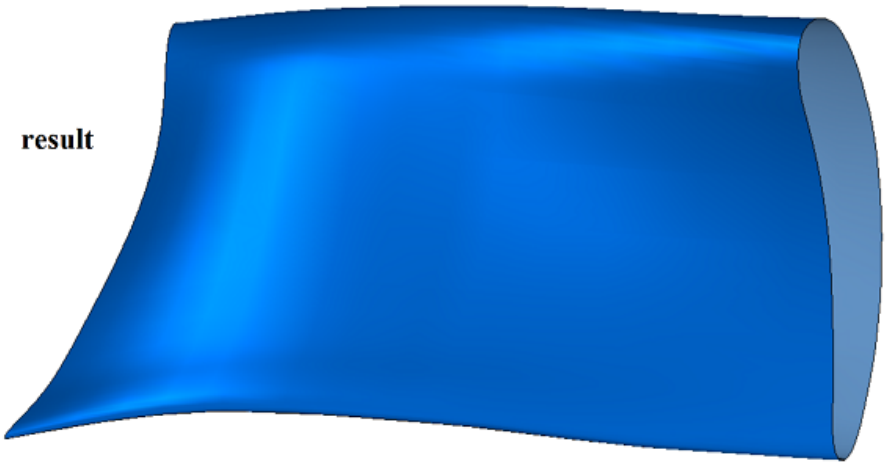
- **_sameSurface** – is the flag for using a surface (true) or a copy of it (false).

The function output parameter is the **result** parameter, which contains the constructed body.

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

This function is declared in *action_shell.h* file.

Fig. 1 shows a body constructed from a surface cyclically closed in the first parametric direction:



The function doesn't check or calculate anything. The constructed body has only one face. The face differs from the original surface in that it has edges on the borders and seams, and it has vertexes in edge mating points.

If the original 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 original surface has borders, then the function will construct a non-closed body. The function adds MbSimpleCreator simple constructor declared in *cr_simple_creator.h* file in newly constructed body log.

In the *test.exe* application you can create a body based on a surface using the menu command «Create→Body→Based on a surface».

3.1.7 Constructing a Ruled Body

The function

```
MbResultType RuledShell( const MbRuledShellParams & ruledParams,
                               MbRuledShellResult & outParams );
```

constructs a non-closed ruled body from two parametrically defined curves.

A **ruledParams** object of the MbRuledShellParams class is the input parameter of the function (see *op_shell_parameter.h*) and contains the construction parameters:

- **pars** – is an object containing operation parameters (geometric information needed to construct the body);
- **operNames** – is the operation namer;
- **isPhantom** – is the phantom creation mode.

The function output parameter is the **result** parameter, which contains a constructed body.

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

This function is declared in *action_shell.h* file.

A ruled body is constructed from two curves – **pars**.*curve0* and **pars**.*curve1*. If the curves are composite, 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.

The **pars**.*curve0* and **pars**.*curve1* curves can be additionally split into segments using **pars**.*breaks0* and **pars**.*breaks1* parameters.

The **pars**.*joinByVertices* parameter indicates whether it is required to connect the contours having the same number of segments through the vertexes. By default, **pars**.*joinByVertices* = *false*.

The **pars**.*checkSelfInt* parameter indicates whether it is necessary to check the curves for self-intersection. By default, **pars**.*checkSelfInt* = *false*.

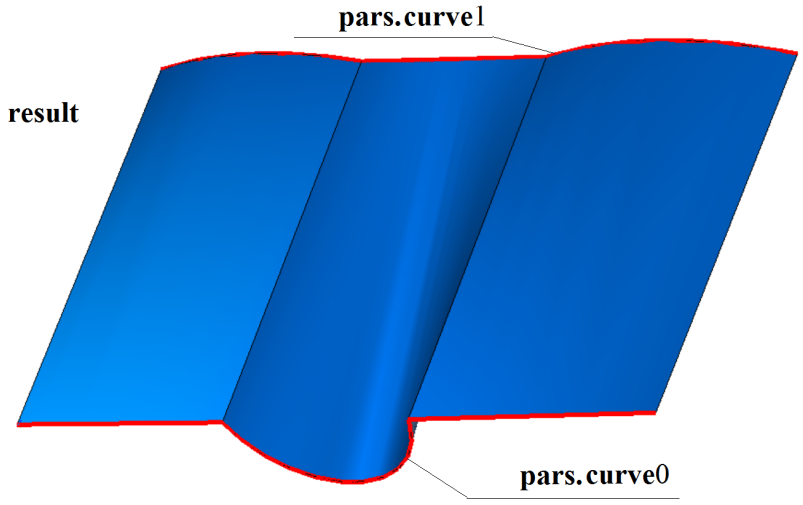
The **pars**.*simplifyFaces* parameter specifies whether to simplify edges of a constructed body.

The **pars**.*proportional* parameter initiates the proportional natural curve parametrization.

The **pars**.*mismatchMax* parameter defines the acceptable mismatch of opposite segment connection points.

The **pars**.*segmentSplit* parameter indicates whether to divide a shell into faces by contour segments.

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



A constructed body has one or several faces. A ruled surface always has borders formed by the initial curves. This function can check the faces for self-intersection. **pars**.*curve0* and **pars**.*curve1* curves can be cyclically closed. The function adds MbRuledShell constructor declared in *cr_ruled_shell.h* file in a newly constructed body log.

3.1.8 Constructing a Body from a Curve Grid

The function

```
MbResultType MeshShell( const MbMeshShellParameters & params,
                              MbMeshShellResults &    result );
```

constructs a body from a curve grid.

The function accepts as an input parameter an object **params** of the MbMeshShellParameters class (see the definition in *op_shell_parameter.h*), which contains the parameters:

- **_params** – an object, which contains construction parameters;
- **_operNames** – the operation namer;
- **_isFantom** – the flag for creating a phantom body (*true* – phantom mode).

The output parameter of the function is an object **result** of the MbMeshShellResults class, which contains the constructed body and the directions of the connections.

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in *action_shell.h* file.

The parameters **_params** contain the geometric information necessary for body construction, Fig. 1:

MeshSurfaceValues

```
RPArray<MbCurve3D> curvesU;
RPArray<MbCurve3D> curvesV;
RPArray<MbPolyline3D> chainsU;
RPArray<MbPolyline3D> chainsV;
bool                  uClosed;
bool                  vClosed;
bool                  checkSelfInt;
MbeMatingType         type0;
MbeMatingType         type1;
MbeMatingType         type2;
MbeMatingType         type3;

c3d::SurfacesVector surface0;
c3d::SurfacesVector surface1;
c3d::SurfacesVector surface2;
c3d::SurfacesVector surface3;

bool                  defaultDir0;
bool                  defaultDir1;
bool                  defaultDir2;
bool                  defaultDir3;
bool                  autoDirection;
bool                  tesselate;
bool                  g2Cont;
bool                  useProport;
std::map<size_t, bool> splitU;
std::map<size_t, bool> splitV;
```

Fig. 1

The body is constructed from two sets of curves – **_params**.*curveU* and **_params**.*curveV*. The curves **_params.curvesU** are located along the first parametric direction of the body faces. The curves **_params.curvesV** are located along the second parametric direction of the body faces.

Curves along each direction are specified in the order of their sequence in the network. Each curve of one direction must share exactly one common point with each curve of the other direction.

One or both extreme curves of any single direction may be specified as point curves. In this case, all curves of the other direction must pass through the point(s) specified for that direction.

Closing the surface along a given direction is possible if all curves of that direction are closed.

Curves may have breaks, and in this case, a polyhedral shell will be constructed with edges at the break locations.

Sets of chains along parametric directions **_params.chainsU**, **_params.chainsV** are specified as polyline curves and are used for aligning the surface parameterization. Each chain defines additional vertices through which additional curves will be automatically constructed without user intervention.

It is possible to create a surface using only one set of curves. In this case, the **defaultDir** flags specify the surface twisting direction during connection, since two twisting variants are possible for each boundary.

The **_params**.*uClosed* parameter indicates whether the body surface is closed along the first parametric direction. For this, all curves in the set **_params.curvesU** must be cyclically closed.

The **_params**.*vClosed* parameter indicates whether the body surface is closed along the second parametric direction. For this, all curves in the set **_params.curvesV** must be cyclically closed.

Using the parameters **_params.***uClosed* and **_params.***vClosed*, it is possible to construct open surfaces from cyclically closed curves.

The parameter **_params.***checkSelfInt* specifies whether to check the constructed body surface for self-intersection. By default, **_params.***checkSelfInt = false*.

The mating types on the boundaries **_params.***type0*, **_params.***type1*, **_params.***type2*, **_params.***type3*, together with the corresponding mating surfaces **_params.***surface0*, **_params.***surface1*, **_params.***surface2*, **_params.***surface3*, define the behavior of the body at the edges for face parameter values *v = vMin*, *u = uMin*, *v = vMax*, *u = uMax*, respectively.

The mating parameters **_params.***type0*, **_params.***type1*, **_params.***type2*, **_params.***type3* can take values *trt_Position*, *trt_Tangent*, *trt_Normal*, *trt_SmoothG2* from the MbeMatingType enumeration (see *mb_enum.h*).

The value **trt_Position** is taken by default; it means the body surface passes through the curves. The surfaces **_params.***surface0*, **_params.***surface1*, **_params.***surface2*, **_params.***surface3* can be null in this case.

The value **trt_Tangent** means tangency of the corresponding body edge to the surface **_params.***surface0*, **_params.***surface1*, **_params.***surface2*, or **_params.***surface3*.

The value **trt_Normal** means orthogonality of the corresponding body edge to the surface **_params.***surface0*, **_params.***surface1*, **_params.***surface2*, or **_params.***surface3*.

The value **trt_SmoothG2** specifies tangent connection (along curvature), where both the surface constructed from the curve grid and the mating surface maintain equal curvatures at every boundary point in all directions.

The **tesselate** = *true* flag allows supplementing a user-defined curve network with additional curves, which in some cases may improve the geometry.

The **splitU** = *true*, **splitV** = *true* flags assign a property to the user curves that turns them into splitting edges of the resulting shell, even if they pass through smooth curves of the transverse direction.

The **usePropopt** = *true* flag reparameterizes the input curves proportionally to their length, which may lead to improved surface geometry.

The **autoDirection** flag changes the logic for selecting the default direction. If **autoDirection** = *true*, a geometric criterion is used (moving the loft boundary along with the lofted surface may result in a change of twisting); if **autoDirection** = *false*, the twisting is determined rigidly.

The **g2Cont** flag sets the degree of smoothness between surface patches along the curve network (for both directions at once). It can be controlled, but note that increasing the smoothness order may degrade the surface shape. In the absence of connections on transverse boundaries and with a single patch, linear interpolation is used in the considered direction. In the presence of any connection on transverse boundaries or multiple patches, cubic interpolation is used in the considered direction (providing G1 smoothness). In the presence of smooth connections on transverse boundaries or with the **g2Cont** option enabled, 5th-degree interpolation is used in the considered direction (providing G2 smoothness).

Fig. 2 shows a curve grid consisting of four cyclically closed curves from the set **_params.curvesU** and two open curves from the set **_params.curvesV**:

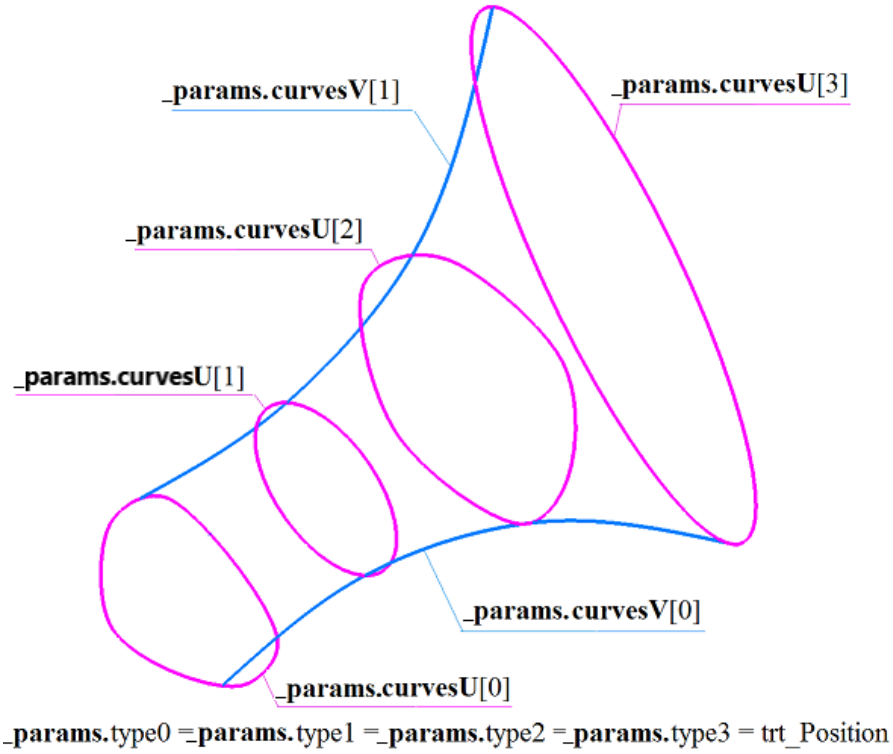


Fig. 2

Fig. 3 shows the body constructed from the curve grid presented in Fig. 2:

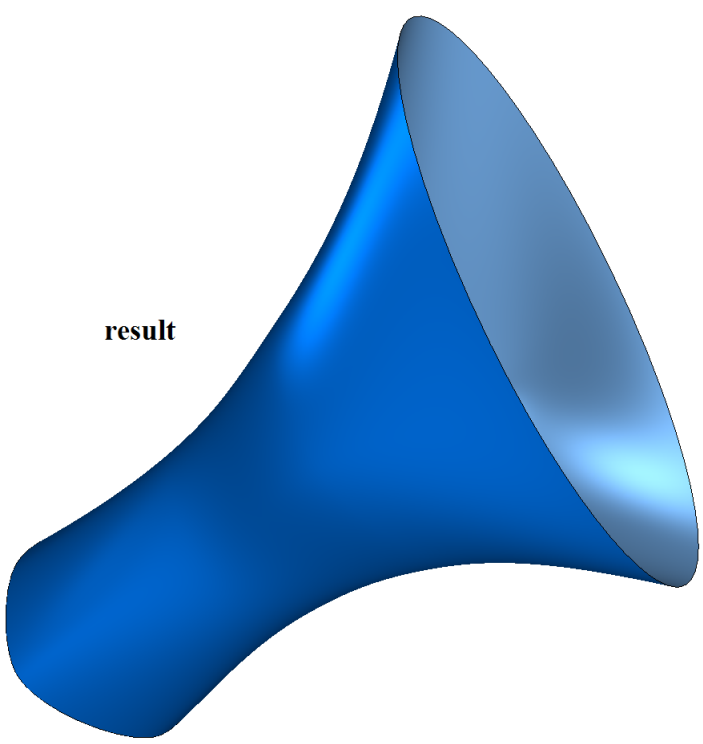


Fig. 3

Fig. 4 shows a curve grid consisting of two open curves from the set **_params.curvesU** and three open curves from the set **_params.curvesV**, two extreme ones of which are degenerated into points:

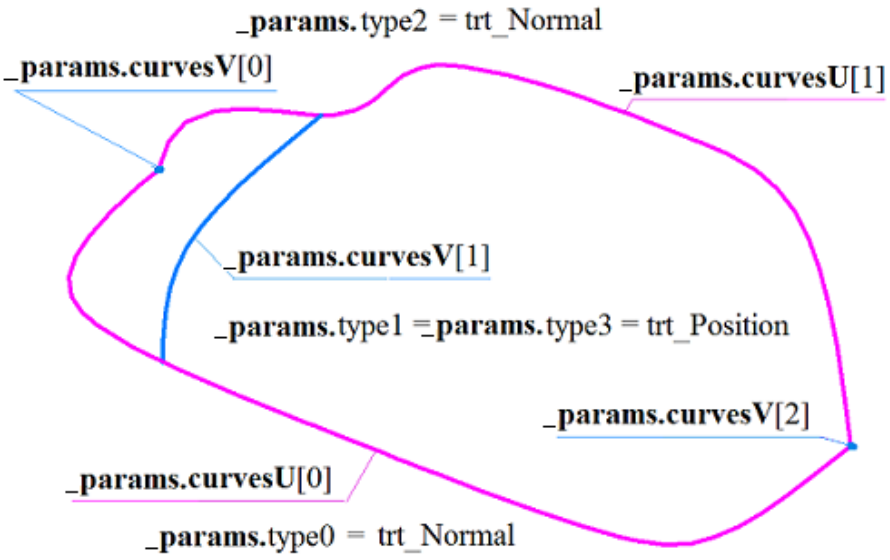


Fig. 4

Fig. 5 shows the body constructed from the curve grid presented in Fig. 2.9.4. The edges of the constructed body corresponding to the face parameters *v = vMin* and *v = vMax* are orthogonal to the planar surfaces **_params.***surface0* and **_params.***surface2*, since the parameters **_params.***type0* and **_params.***type2* have the values *trt_Normal*.

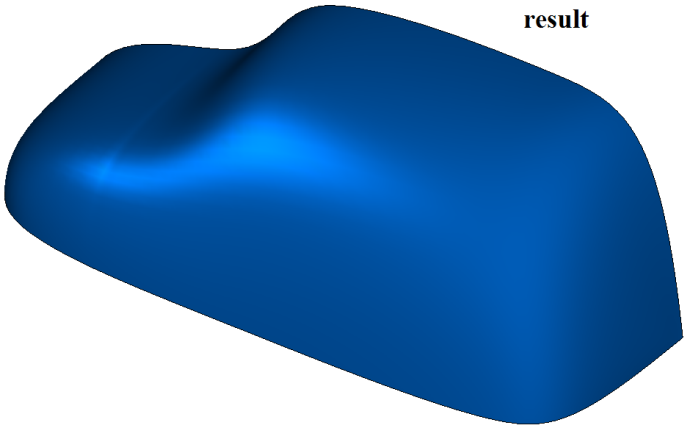


Fig. 5

The function adds the builder MbMeshShell to the log of the created body, which is declared in the file *cr_mesh_shell.h*.

In the test application *test.exe*, a body can be constructed from a curve grid using the menu command «Create→Solid→Based on curves».

3.1.9 Constructing a Patch

The function

```
MbResultType PatchShell( const MbPatchShellParams & params,
                               c3d::SolidSPtr &     result );
```

constructs a patch from a set of curves.

A **params** object of the MbPatchShellParams class is the input parameter of the function and contains the following parameters required for the operation (see *op_shell_parameter.h*):

- **initCurves** – is a set of curves;
- **patchValues** – is an object of the PatchValues structure, which contains construction parameters;
- **names** – is the operation's namer.

The MbPatchShellParams class contains three constructors for a set of curves, parameters, and the namer:

```
MbPatchShellParams( const std::vector<SPtr<MbPatchCurve>> & initCurves,
                    const PatchValues                     & patchValues,
                    const MbSNameMaker                    & names );
```

(continues on next page)

(continued from previous page)

```
MbPatchShellParams( const RPArray<MbPatchCurve> & initCurves,
                    const PatchValues           & patchValues,
                    const MbSNameMaker          & names );


MbPatchShellParams( const RPArray<MbCurve3D> & initCurves,
                    const PatchValues        & patchValues,
                    const MbSNameMaker       & names );
```

The function output parameter is the **result** parameter, which contains a constructed body.

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in *action_shell.h* file.

The function constructs a body formed by a face, the borders of which contain defined curves. The body will be constructed correctly if the curves of **initCurves** set connect end-to-end and form a closed contour.

If all **initCurves** in the set lie in a single plane, the patch is a portion of that plane bounded by the contour formed by the curves.

If all **initCurves** in the curve set lie on a single curved surface, the patch will be a portion of that surface bounded by the contour formed by the specified curves.

If **initCurves** set contains a single curve, that curve is split into two segments, and the resulting patch surface is a ruled surface with two poles.

A patch is constructed in a similar way when the **initCurves** set contains two curves: the patch surface will be a ruled surface with two poles.

If the number of **initCurves** is three, the patch surface will be an MbCornerSurface constructed from the three curves and having a pole.

If the number of **initCurves** is four, the patch surface will be an MbCoverSurface constructed by the four curves. In other cases, the construction may fail.

The **patchValues** object contains information about the patch type and the self-intersection check flag. To initialize the object the following constructor of the PatchValues structure is set by default:

```
PatchValues()
  : type               ( ts_none  )
  , checkSelfInt       ( false    )
  , mergeEdges         ( true     )
  , tolerantData       ( false    )
  , internalMatingType ( pmt_None )
{ }
```

where

- **type** – is the patch type;
- **checkSelfInt** – is the intersection check flag;
- **mergeEdges** – is the flag for merging similar edges (*true* – merge similar edges);
- **tolerantData** – is the flag for building an inaccurate patch based on inaccurate input data;

- **internalMatingType** – is the conjugation type between patch parts. Used for **type** = *ts_byCurves*.

The following patch types are defined for the **patchValues** object (see the SurfaceType enumeration):

- *ts_tang* – tangentially;
- *ts_norm* – according to a norm;
- *ts_none* – not defined;
- *ts_plane* – is a flat patch;
- *ts_byCurves* – the construction is set by conjugates for each curve.

The *ts_tang* and *ts_norm* types are set when constructing a patch tangentially or normally, respectively.

If the patch type *ts_plane* is set, a patch with flat edges is constructed.

The *ts_byCurves* type is set if you want to build a patch with conjugation along some boundaries. In this case, for a desired MbPatchCurve boundary, you need to set the conjugation by the SetMating() method.

Fig. 1 shows eight curves on the boundary of the inner face cut. Fig. 2 shows a patch constructed from the curves shown in Fig. 1:

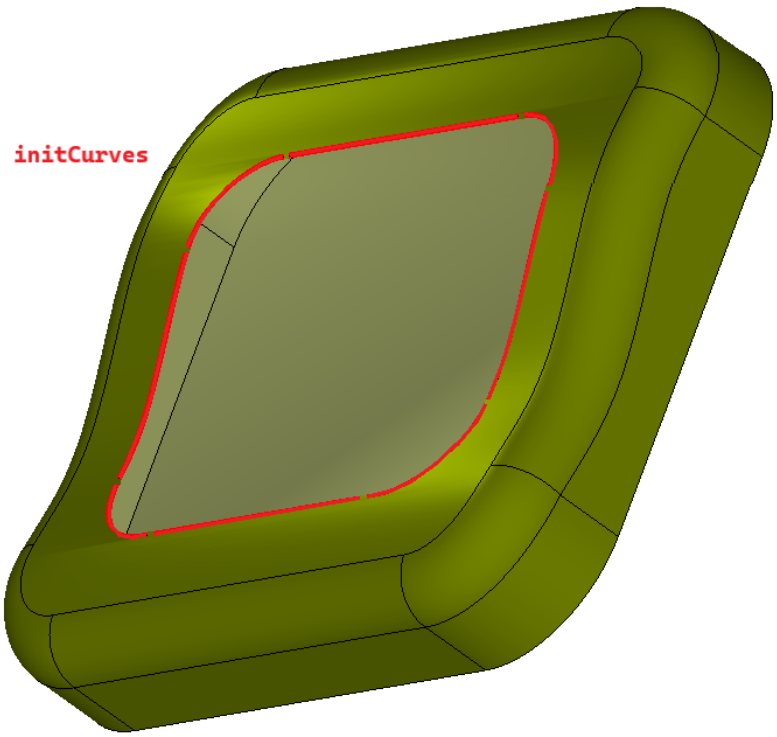


Fig. 1

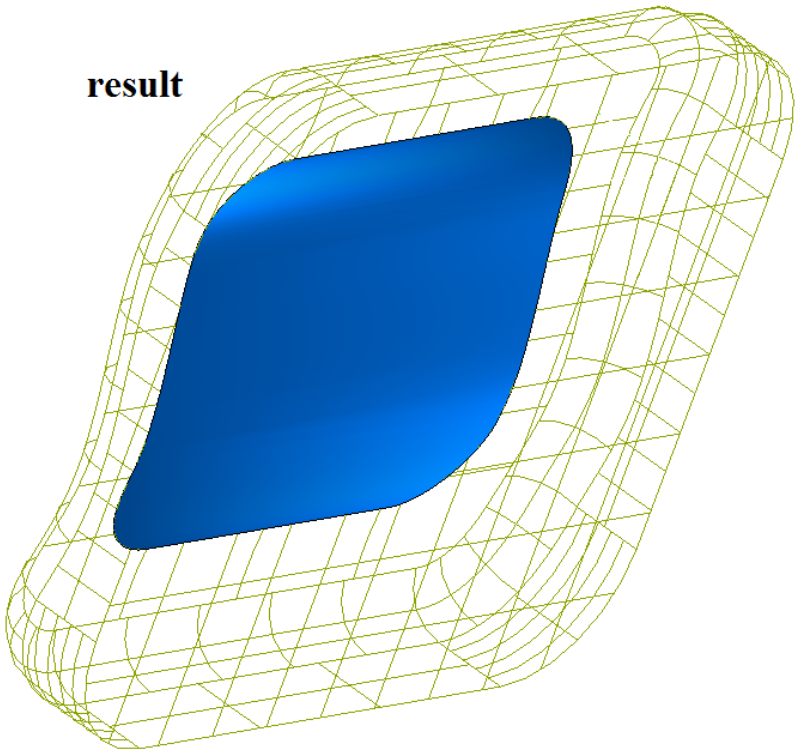


Fig. 2

Fig. 3 shows six curves lying on the borders of various faces produced by cutting faces with a common surface.

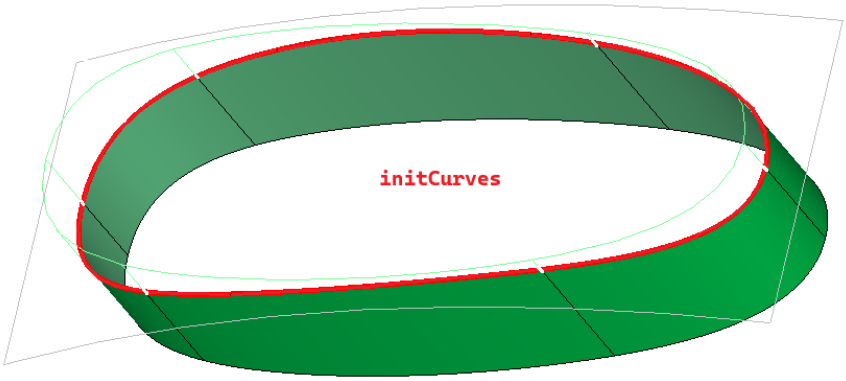


Fig. 3

If the boundary edges retain information about the cutting surface after the cut, that surface can be used to construct the patch. Fig. 4 shows an example of constructing a patch as a part of the cutting surface that formed border edges shown in Fig. 3:

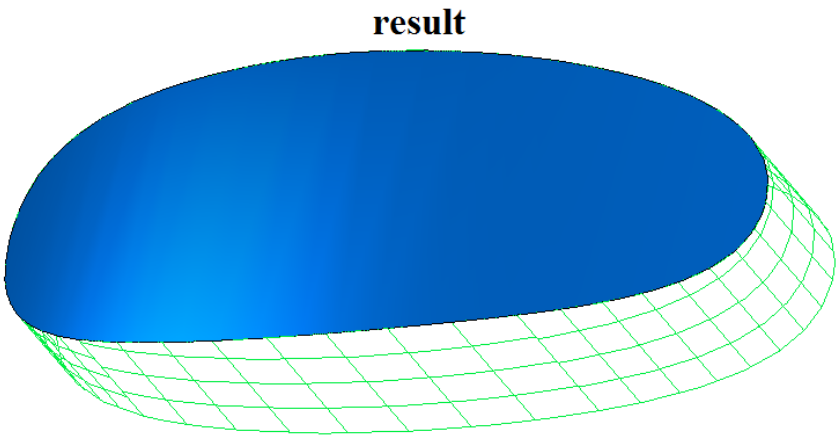


Fig. 4

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

PatchShell() function adds MbPatchCreator constructor into a log of the newly constructed body. MbPatchCreator constructor is declared in *cr_patch_creator.h* file.

3.1.10 Stiching Body Faces

```
MbeStitchResType StitchShells( const c3d::SolidsSPtrVector & initialSolids,
                               const MbShellStitchParams &   stitchParams,
                                     c3d::SolidSPtr &        resultSolid );
```

performs stitching of interlocking faces of several solids into one solid (the face orientation can be changed). The function is declared in the file *action_shell.h*.

The input parameters of the function are:

initialSolids - set of solids for stitching;

stitchParams - shell stitching parameters, which include the following:

- **_nameMaker** is the operation's namer;
- **_stitchAccuracy** is the stitching accuracy (accuracy of finding paired edges);
- **_formSolidBody** is the flag of forming a solid from a resulting shell;
- **_mergeEdges** is the merging of similar edges (true).

The output parameter of the method is the constructed solid or the shell **resultSolid** (depending on the **_formSolidBody** flag). If the parameter **_formSolidBody** = *true*, the constructed solid **resultSolid** is checked for absence of boundary edges, nested shells are searched and internal shells are reoriented if necessary. If the parameter **_formSolidBody** = *false*, then boundary edges are allowed in the constructed **resultSolid** solid, and the definition of shell nesting is not performed.

The function returns the operation result code from the *MbeStitchResType* enumeration (see *mb_operation_result.h*):

- **stch_Success** — the operation was completed successfully.
- **stch_PrepareError** — error when preparing a stitching operation.

- **stch_CoorientFaceError** — unable to set coordinated orientation of faces.
- **stch_SomeEdgesUnstitched** — some edges are still unstitched.
- **stch_OutwardOrientError** — failed to set face normals to the outside of the solid;
- **stch_NoEdgeWasStitched** — no edges were stitched.
- **stch_SeparatePartsResult** — there are still unconnected pieces after stitching;
- **stch_EdgeStitchError** — error of the edge stitching.
- **stch_InputTopologyError** — critical topology errors in input shells.

The function searches pairs of boundary edges that correspond to each other in length and location, and constructs the intersection curve of faces of the found edge in the given area. After successful construction of the intersection curve, a pair of boundary edges is replaced by a single edge connecting adjacent faces of the solids. If necessary, the function reorients the outer side of some faces. The set **initialSolids** can contain any finite number of solids.

Figure 1 shows two stitchable solids - **solids** [0] *и* **solids** [1]. Figure 2 shows the constructed solid **resultSolid**:

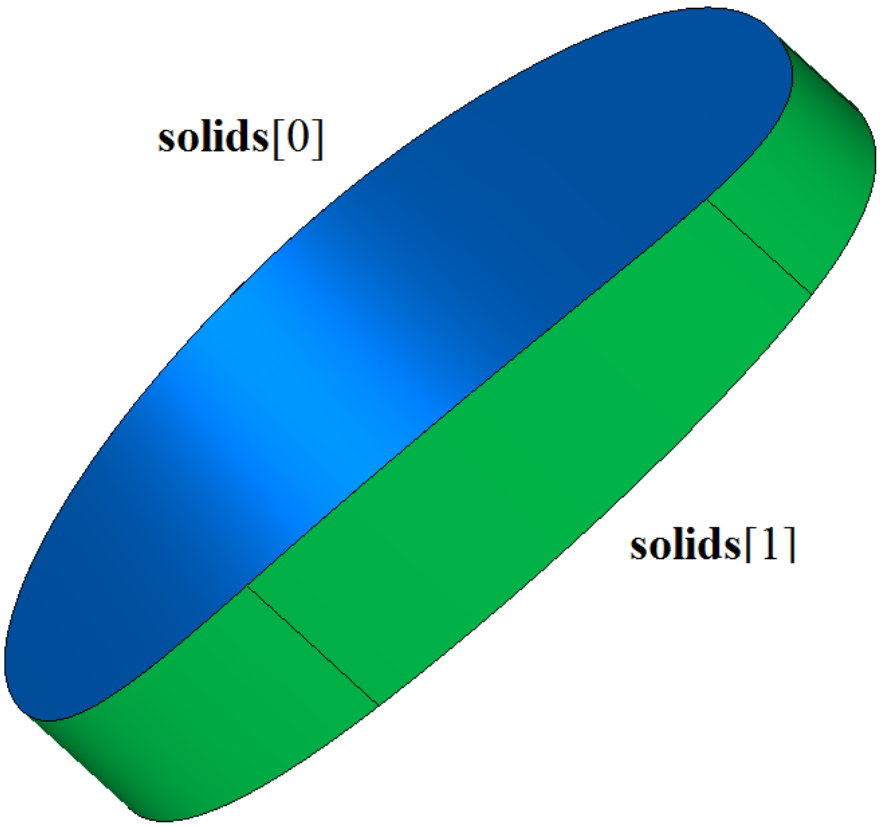


Fig. 1

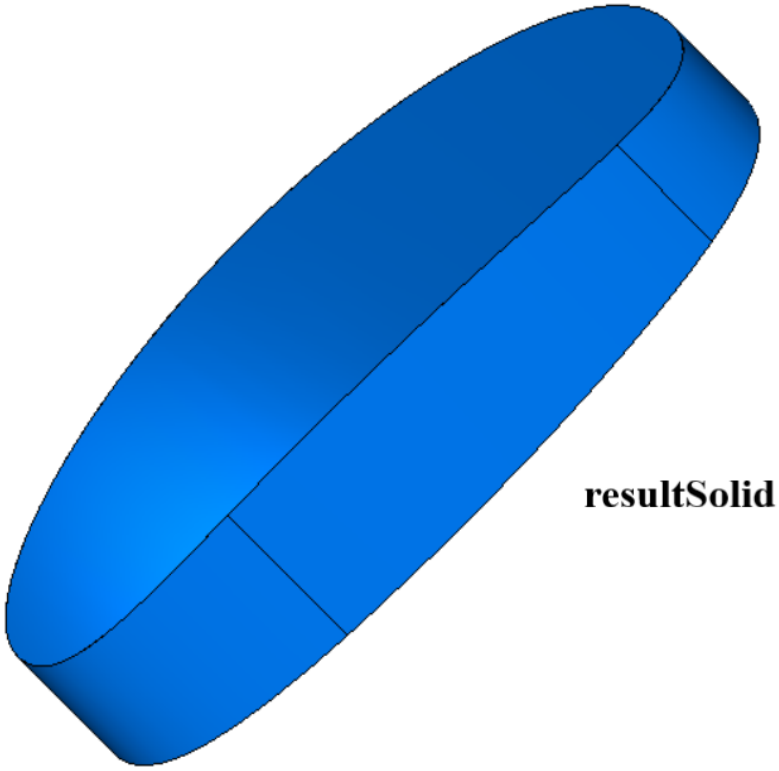


Fig. 2

The StitchShells() function adds the MbStitchedSolid builder to the built solid log, which contains all the necessary data for the solid building. The MbStitchedSolid builder is declared in the *cr_stitch_solid.h* file.

3.1.11 Construction of Body Based on Curves or Curve Points

Construction of Body Based on Curves

The function

```
MbResultType LoftedShell( const MbLoftedCurvesShellParams & params,
                                c3d::SolidSPtr &            result );
```

constructs a body with one face, the surface of which contains a set of curves.

The input parameter of the function is a **params** object of the MbLoftedCurvesShellParams class, which contains the parameters required for the operation (see *op_shell_parameter.h*):

- **curves** is a set of curves;
- **names** is the operation namer;
- **name** is the identifier.

The function output parameter is the **result** parameter, which contains the constructed body.

To initialize a **params** object in the MbLoftedCurvesShellParams class, the following constructor is defined:

```
template < class CurvesVector >
MbLoftedCurvesShellParams( const CurvesVector & curves,
                           const MbSNameMaker & names,
                           const SimpleName &   name );
```

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in *action_shell.h* file.

The 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. To prevent self-intersection and twisting of the body surfaces, 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 naming of faces, edges and vertices in the constructed body.

Fig. 1 shows **curves** set:

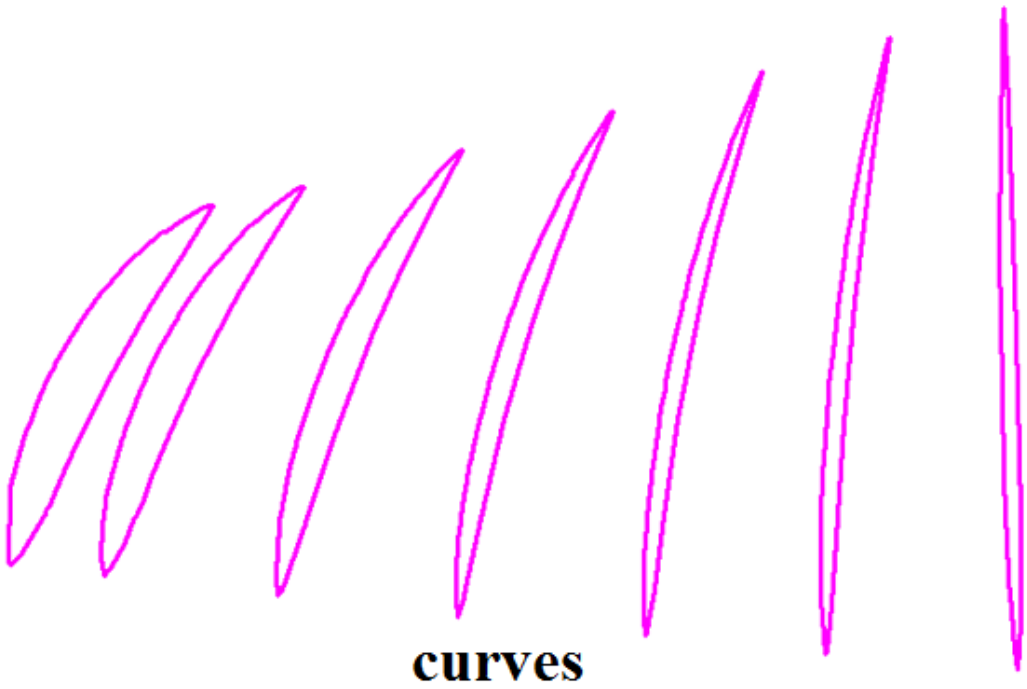


Fig. 1

Fig. 2 shows a body constructed from the curves shown in Fig. 1:

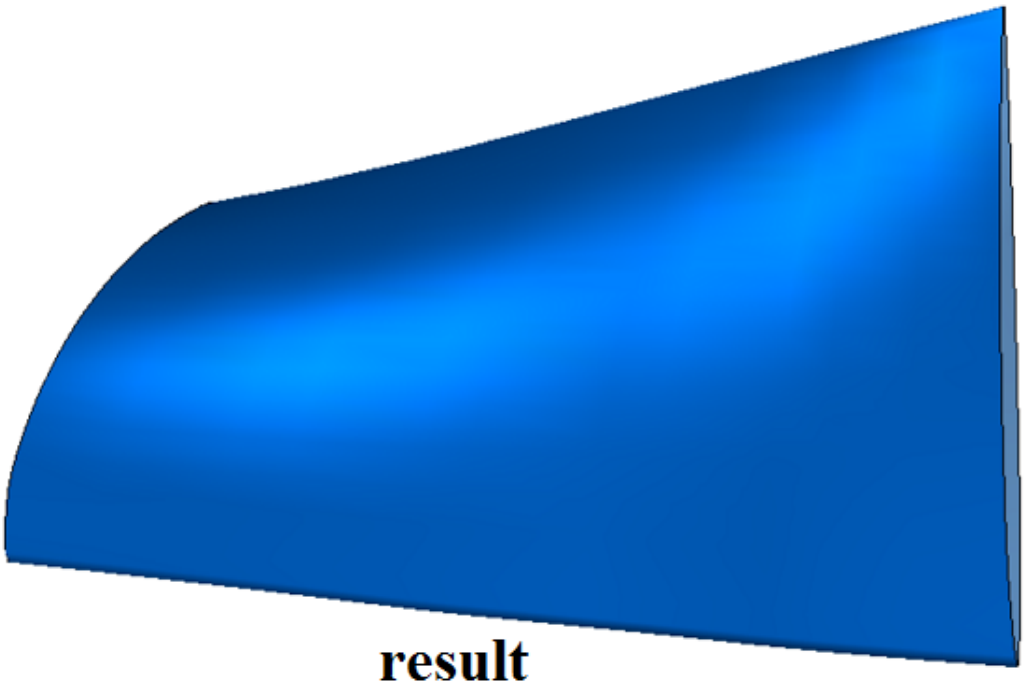


Fig. 2

Construction of Body Based on Curve Points

The function

```
MbResultType LoftedShell( const MbLoftedPointsShellParams & params,
                                c3d::SolidSPtr &            result );
```

constructs a non-closed body from one face, the surface of which contains control points.

A **params** object of the class MbLoftedPointsShellParams is the input parameter of the function and contains the parameters required for the operation (see *op_shell_parameter.h*):

- **points** is a set of control points;
- **names** is the operation namer;
- **name** is the identifier.

The function output parameter is the **result** parameter, which contains the constructed body.

To initialize a **params** object in the MbLoftedPointsShellParams class, the following constructors are defined:

```
MbLoftedPointsShellParams( const RPArray<SArray<MbCartPoint3D>> & points,
                           const MbSNameMaker &                   names,
                           const SimpleName &                     name );


MbLoftedPointsShellParams( const std::vector<c3d::SpacePointsVector> & points,
```

(continues on next page)

(continued from previous page)

```
                        const MbSNameMaker &                    names,
                        const SimpleName &                      name );
```

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in the *action_shell.h* file.

The control points are grouped into *SArray* containers or into a *c3d::SpacePointsVector* vector depending on the constructor type. Spline curves are constructed from point sets before the body is constructed. Then LoftedShell() function described in section «Construction of Body Based on Curves» constructs the body from the curve set.

The surface of the constructed body passes through all 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. To prevent self-intersection and twisting of the body surfaces, point sequences in adjacent groups should have the same direction, and starting points in adjacent groups should be close to each other. To construct a surface closed along one of the parametric directions, the first and the last points in every *SArray* container and in the *c3d::SpacePointsVector* vector must coincide.

The **names** and **name** parameters define naming of faces, edges and vertexes in the constructed body.

Fig. 3 shows a set of control points **points**, each group has a different color:

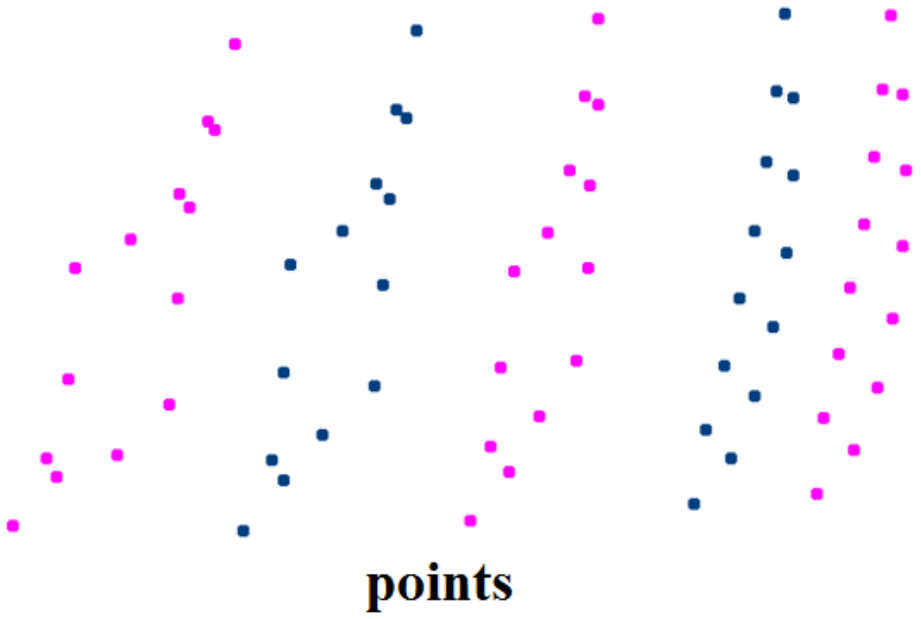


Fig. 3

Fig. 4 shows a body and control points used to construct it:

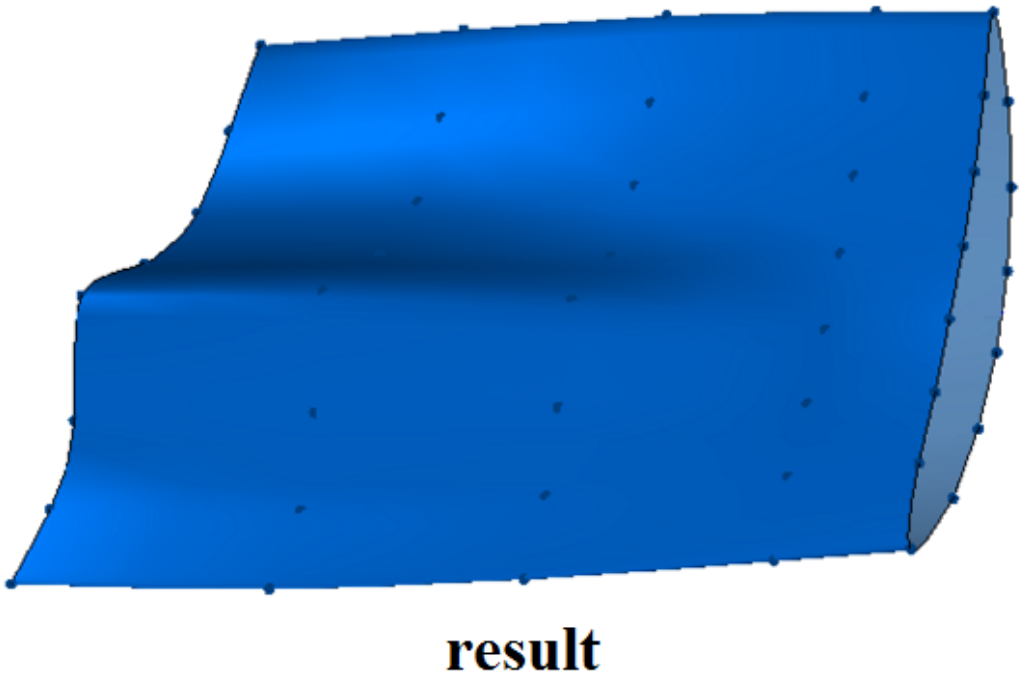


Fig. 4

The LoftedShell() function adds the MbThinShellCreator creator containing all construction data into the body log. The MbThinShellCreator creator is declared in the *cr_thin_sheet.h* file.

3.1.12 Construction of Equidistant Body

The function

```
MbResultType OffsetShell(       MbSolid &              solid,
                                MbeCopyMode            sameShell,
                                RPArray<MbFace> &      initFaces,
                                bool                   checkFacesConnection,
                                SweptValues &          p,
                          const MbSNameMaker &         operNames,
                                bool                   copyFaceAttrs,
                                MbSolid *&             result );
```

constructs an equidistant body for a set of faces.

Input parameters of the function are as follows:

- **solid** is an original body;
- **sameShell** is the copy mode of an original body;
- **initFaces** are faces of an original body;
- **checkFacesConnection** is the flag that checks the connection of selected faces:

  true – the operation checks that all faces in **initFaces** form a single connected region; if

the connectivity is violated, an error code is returned and the equidistant body is not constructed;

false – the check is not performed, which may lead to several disconnected parts of the result or to a failure of the operation;

- **p** is an object containing construction parameters;
- **operNames** is the operation namer;
- **copyFaceAttrs** is the flag for copying attributes from original faces to equidistant faces.

The function output parameter is the **result** parameter, which contains the constructed body.

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in *action_shell.h* file.

The function constructs a body, the faces of which are equidistant to **initFaces** of the **solid** original body.

The **sameShell** parameter controls transfer of faces, edges and vertices from the **solid** original body to the **result** constructed body. This parameter can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (an original shell and its copy have the same basic surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the original shell and its copy have the same basic surfaces);
- *cm_Copy* (the original shell and its copy do not have the same data).

The detailed description of the MbeCopyMode enumeration is located in section *Copying a Set of Faces*.

The **p** parameter contains information about the distance between constructed faces and the original body faces. The distance can be equal to **p**.*thickness1* in positive direction or to **p**.*thickness2* in negative direction of the normal of the original body faces.

If **p**.*shellClosed* = *false*, then a non-closed body will be constructed.

The **p**.*checkSelfInt* parameter indicates the need to check a construction result for self-intersection. By default, **p**.*checkSelfInt* = *false* – the check is not performed.

The **p**.*mergeFlags* flags control the merging of similar faces and edges (*true* – merge similar elements).

The **operNames** parameter is used to name faces, edges and vertices of the constructed body.

Fig. 1 shows a body, its selected faces will be used to construct an equidistant body:

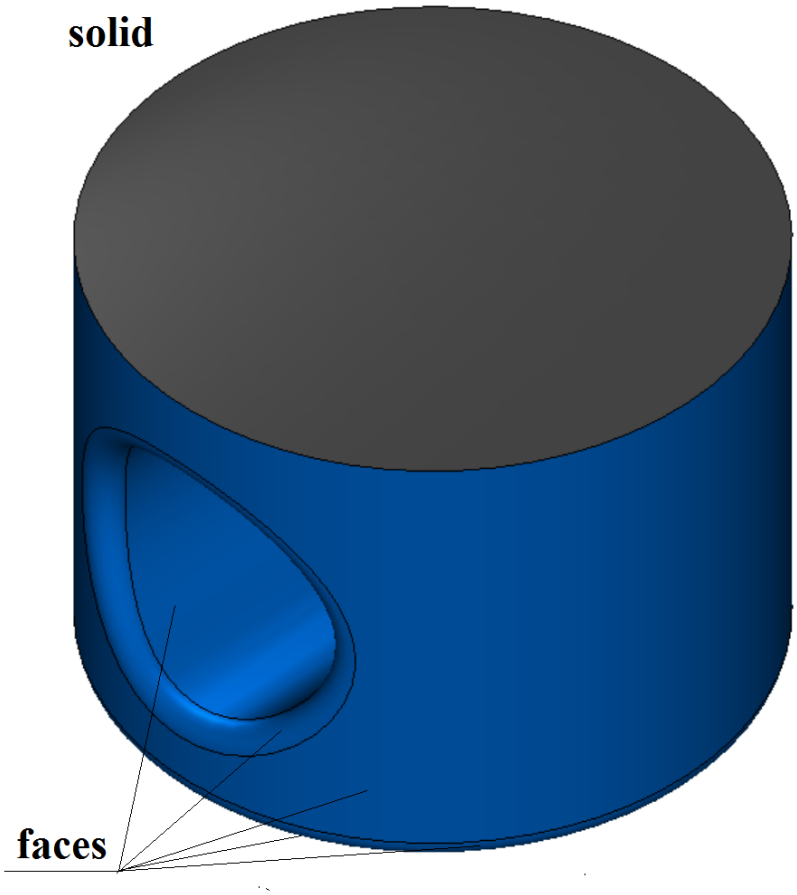


Fig. 1

Fig. 2 shows the constructed equidistant body:

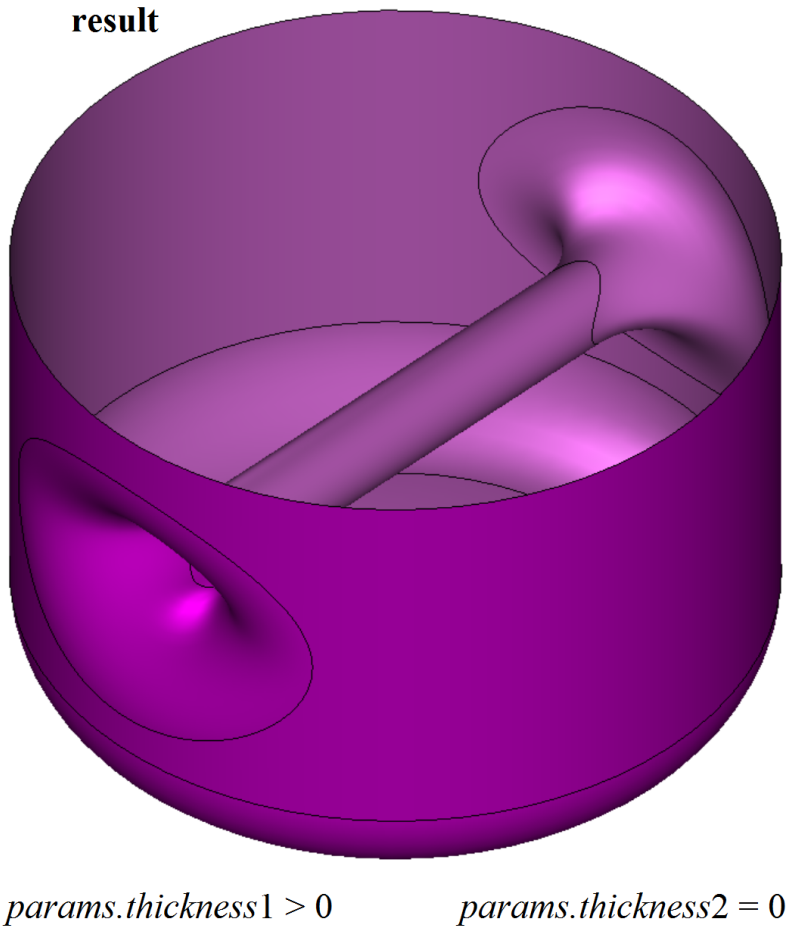


Fig. 2

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

The function OffsetShell() adds MbShellSolid builder containing all construction data required into the body log. MbShellSolid constructor is declared in *cr_thin_shell_solid.h* file.

Test.exe test application allow to construct an equidistant non-closed body using «Create→Solid→Based on a solid→Offset solid» menu.

3.1.13 Extension of Body Face

The function

```
MbResultType ExtensionShell(       MbSolid &                 solid,
                                   MbeCopyMode               sameShell,
                             const MbExtensionShellParams &  params,
                                   c3d::SolidSPtr &          result );
```

creates a non-closed body by extending the border edges of a selected body face.

The function's input parameters are:

- **solid** is an original body;
- **sameShell** is the copy mode for the original body;
- **params** is an object containing the construction parameters of the operation.

The MbExtensionShellParams class defines two constructors to initialize **params** objects with operation parameters:

The constructor for extending a non-closed body by extrusion for a given set of edges:

```
MbExtensionShellParams( const c3d::EdgesSPtrVector & edges,
                        const ExtensionValues &      params,
                        const MbSNameMaker &         operNames );
```

The constructor for extending a non-closed body by extrusion without specifying edges (the edges are specified by the appropriate MbExtensionShellParams::Add methods):

```
MbExtensionShellParams( const ExtensionValues & params,
                        const MbSNameMaker &    operNames );
```

where

- **edges** is a set of border edges through which the extension is performed;
- **params** is an object containing operation parameters;
- **operNames** is the namer of the operation.

The output parameter of the function is **result**, which contains the constructed body.

The function returns a result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in *action_shell.h* file.

The **sameShell** parameter controls transfer of faces, edges and vertices from the **solid** original body to the **result** constructed body. This parameter can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the original shell and its copy share base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the original shell and its copy share base surfaces);
- *cm_Copy* (the original shell and its copy have no shared data).

A detailed description of the MbeCopyMode enumeration is given in section *Copying a Set of Faces*.

The **params** object of the ExtensionValues class defines face extension methods and contains the data shown in Fig. 1:

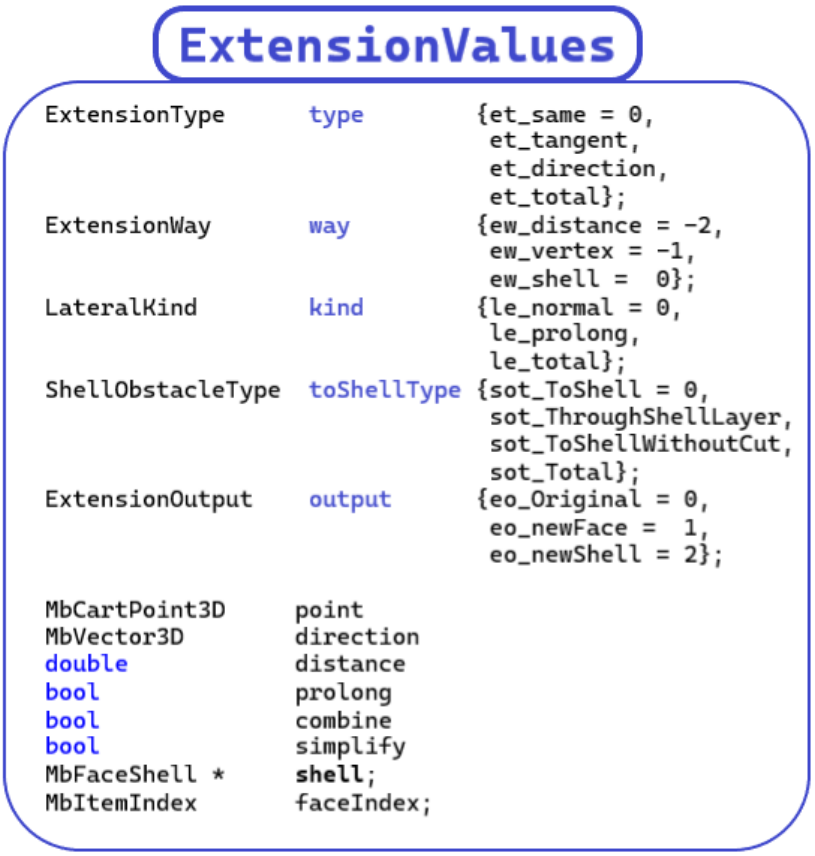


Fig. 1

There are three types of face extension. The **params**.*type* parameter defines the extension type:

- If **params**.*type* = *et_same*, the original **face** is extended by expanding its surface parameter domain and offsetting border **edges**;
- If **params**.*type* = *et_tangent*, new faces are smoothly joined to the face through the **edges**;
- If **params**.*type* = *et_direction*, new faces are joined to **face** of the original body using **edges**. New faces are received by extruding the curved **edges** in **params**.*direction*.

The **params**.*way* parameter defines the extension distance for the **face**.

- If **params**.*way* = *ew_distance*, then the face is extended by **params**.*distance*;
- If **params**.*way* = *ew_vertex*, then the face is extended to the vertex defined by **params**.*point*;
- If **params**.*way* = *ew_shell*, then the face is extended up to **params**.*shell*.

If **params**.*way* ≠ *ew_shell*, then **params**.*shell* can be null.

The **params**.*faceIndex* parameter sets an face index of **params**.*shell* used for extension to the shell.

The **params**.*kind* parameter defines how the borders of extension faces are trimmed 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 the border of the **face** extension is defined by the edges adjacent to the edges of the **edges** set.

The **params**.*output* parameter defines the representation options for the resulting body:

- If **params**.*output* = *eo_Original*, then the original faces of the body are extended;
- If **params**.*output* = *eo_newFace*, then the extension result is formed as new faces;
- If **params**.*output* = *eo_newShell*, the extension result is formed as a new shell.

The **params**.*toShellType* parameter defines the extension method to a shell:

- If **params**.*toShellType* = *sot_ToShell*, then the extension is performed to the shell;
- If **params**.*toShellType* = *sot_ThroughShellLayer*, then the extension is performed through the first layer of the shell (before the first exit from the shell);
- If **params**.*toShellType* = *sot_ToShellWithoutCut*, then the extension is performed to the shell without trimming.

The **operNames** parameter is used to name faces, edges and vertices of the constructed body.

Fig. 2 shows the body face, the two border edges of which will be extended:

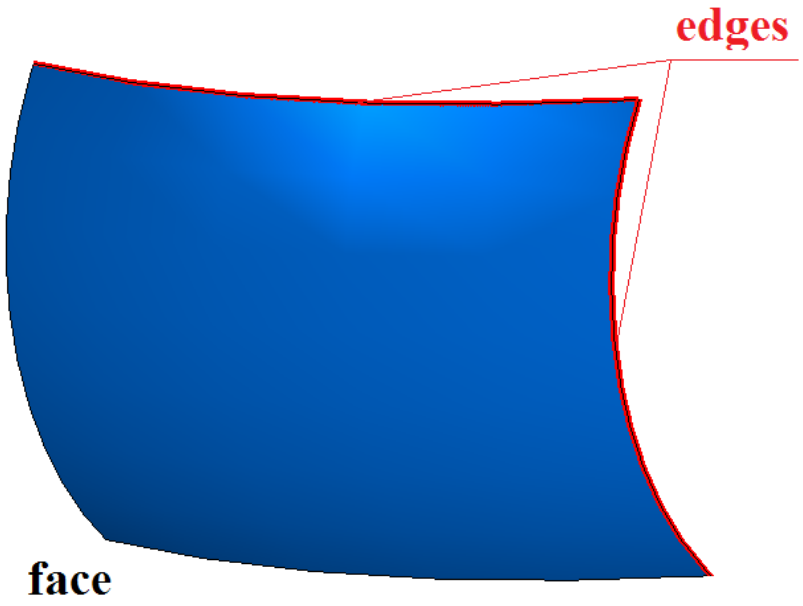


Fig. 2

Fig. 3 and 4 show the results of face extension by expanding the surface parameter domain using various values of **params**.*kind* parameter:

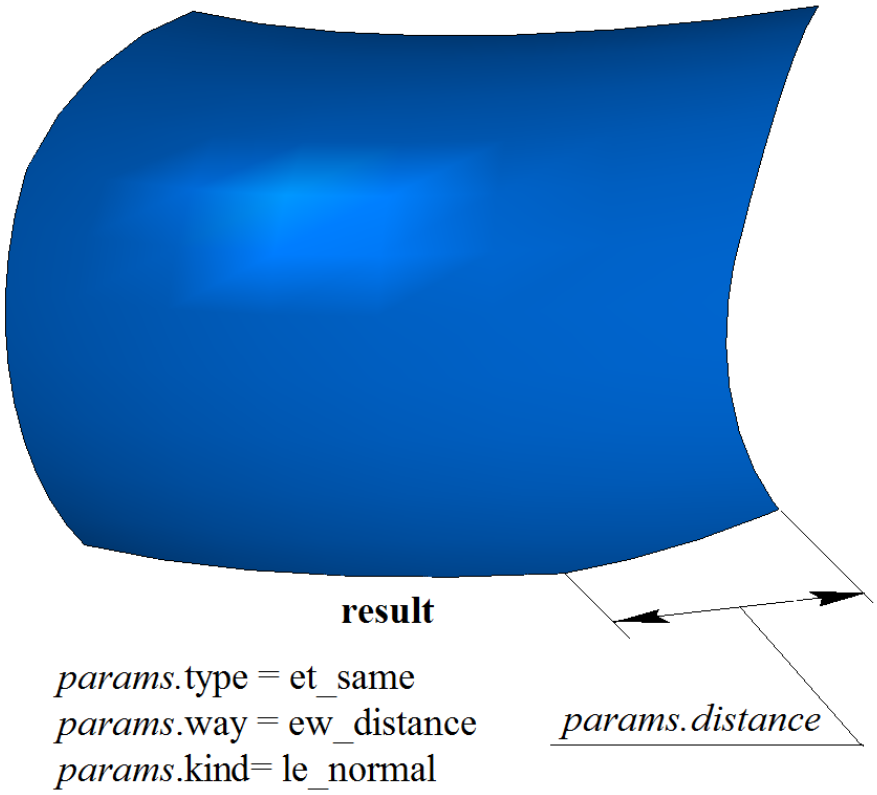

result
params.type = et_same
params.way = ew_distance
params.kind= le_normal
params.distance

Fig. 3

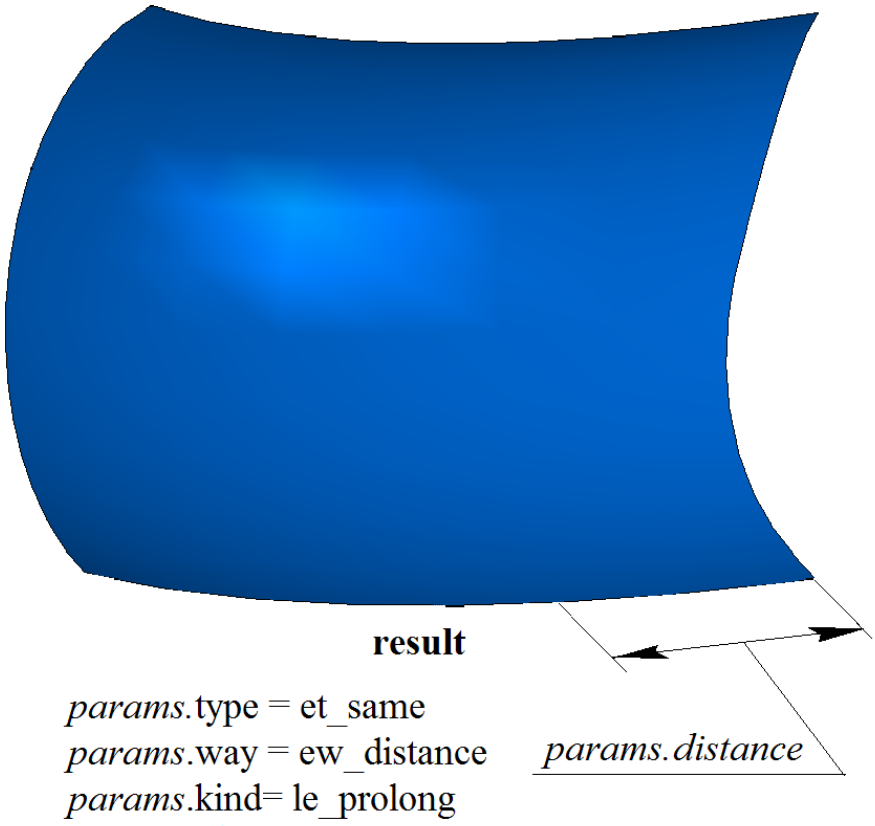


Fig. 4

Fig. 5 shows the result of **face** extension by adding other faces that smoothly join with **face**. Fig. 6 shows the result of extending **face** in the selected direction:

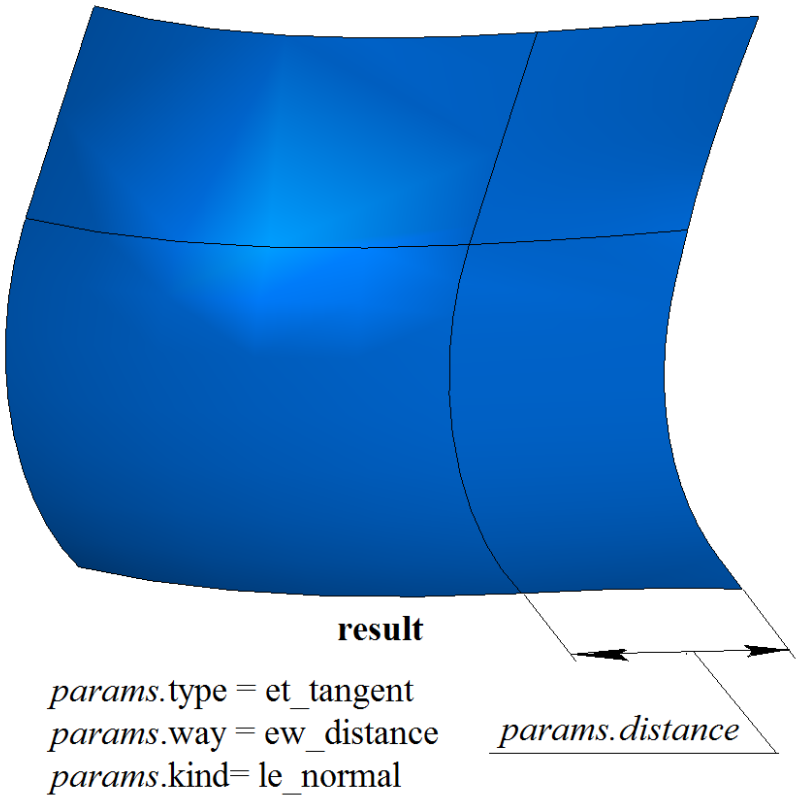

result
params.type = et_tangent
params.way = ew_distance
params.kind= le_normal
params.distance

Fig. 5

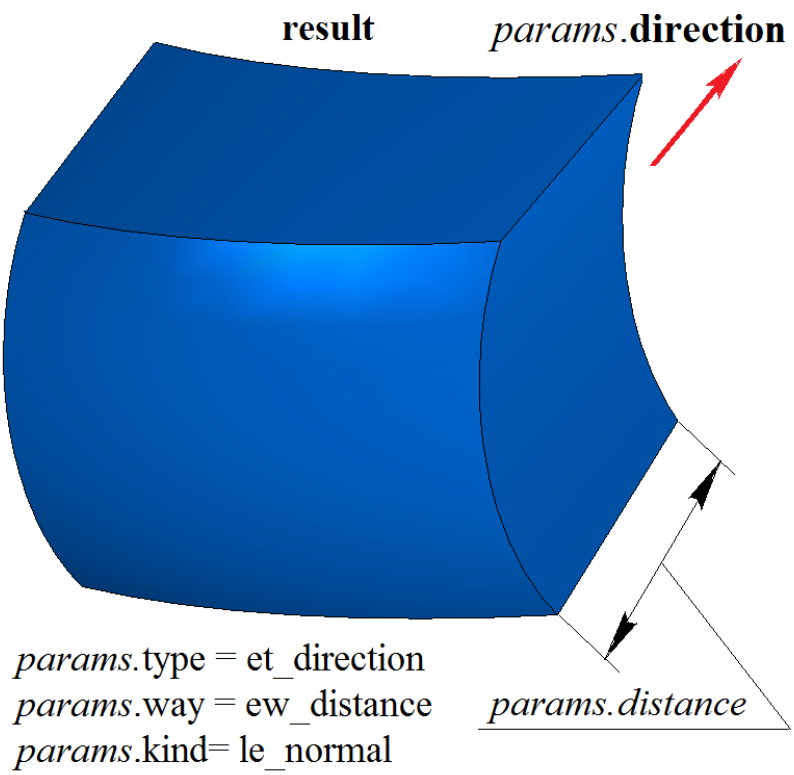


Fig. 6

The ExtensionShell() function adds the MbExtensionShell builder, which contains all construction data, into the body log. The MbExtensionShell constructor is declared in the *cr_extension_shell.h* file.

3.1.14 Constructing an Ingot Body from Given Objects

The function

```
MbResultType IngotSolid(      bool                    makeCopy,
                        const MbIngotSolidParams &    params,
                              c3d::SolidSPtr &        resIngot );
```

constructs an ingot solid from the given objects. The ingot solid is a simplified representation of the given set of solids, intended to speed up calculations, collision detection, etc., and is not intended for accurate detailing.

The given objects can be solids (MbSolid), solid instances (MbInstance), and solid assemblies (MbAssembly). A single solid is constructed from them, which coincides in appearance with the given solids and serves as their simplified substitute in terms of external parameters.

The input parameters of the function are:

- **makeCopy** – flag that specifies whether to copy the source solids;
- **params** – object that contains the operation parameters.

The output parameter of the function is the constructed solid **resIngot**.

The function returns the result code of the operation from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

To initialize the **params** object, the MbIngotSolidParams class defines constructors of the parameters for creating an ingot solid from a given set of solids (see *op_shell_parameter.h*):

```
MbIngotSolidParams( const c3d::SolidSPtr &     solid,
                    const MbSNameMaker &       operNames,
                          IProgressIndicator * prog = nullptr );
```

```
MbIngotSolidParams( const c3d::ItemSPtr &      item,
                    const MbSNameMaker &       operNames,
                          IProgressIndicator * prog = nullptr );
```

```
MbIngotSolidParams( const c3d::SolidsSPtrVector & solids,
                    const MbSNameMaker &          operNames,
                          IProgressIndicator *    prog = nullptr );
```

```
MbIngotSolidParams( const c3d::ItemsSPtrVector & items,
                    const MbSNameMaker &         operNames,
                          IProgressIndicator *   prog = nullptr );
```

```
MbIngotSolidParams( const MbSNameMaker &         operNames,
                          IProgressIndicator *   prog = nullptr );
```

- **solid** – given solid;
- **item** – given solid (can be an object of the MbAssembly, MbInstance, or MbSolid classes);
- **solids** – given container of solids;
- **items** – given container of solids (can contain MbAssembly, MbInstance, and MbSolid objects);
- **operNames** – operation name maker;
- **prog** – operation progress indicator.

The function performs uniting of multiple MbSolid solids, MbInstance solid instances, or MbAssembly solid assemblies into a single solid **resIngot**.

Fig. 1 shows the given container of **solids**:

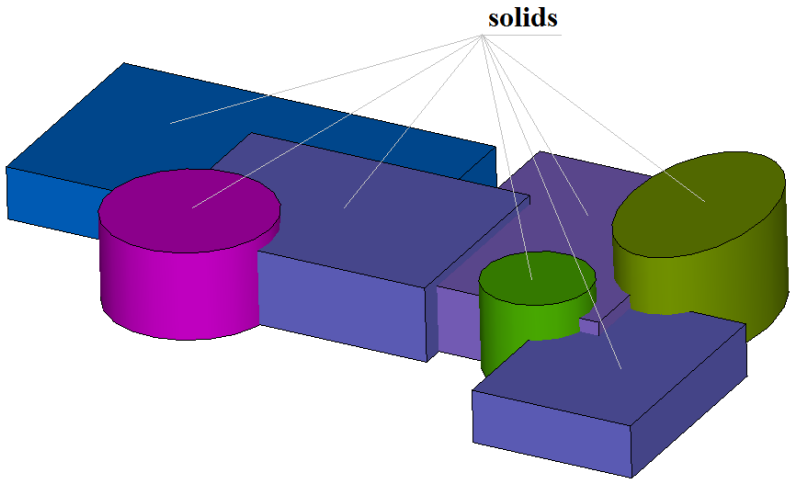


Fig. 1

Fig. 2 shows the **resIngot** result of uniting the **solids**:

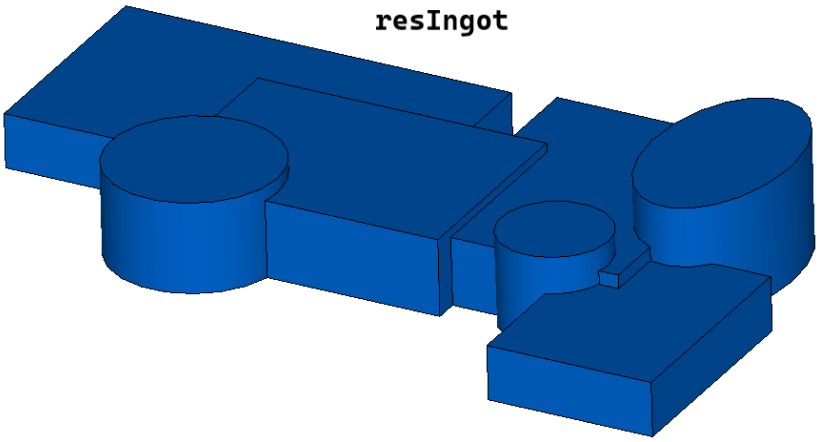


Fig. 2

In the test application *test.exe*, construction of an ingot solid is performed by the menu command “Create → Body → Based on body → Set of bodies”.

3.1.15 Constructing a Body Based on a Polygonal Model

The function

```
MbResultType MeshSolid( const MbMesh &                mesh,
                        const MbGridShellParams &     params,
                              c3d::SolidSPtr &        result );
```

constructs an MbSolid body based on an *MbMesh* polygonal model.

The input parameters of the function are:

- **mesh** – the polygonal model
- **params** – the set of operation parameters (see the MbGridShellParams class definition in *op_shell_parameter.h*)

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

The function returns the operation result code from the *MbResultType* enumeration.

The function is declared in the *action_solid.h* file.

The MbGridShellParams class defines the following two constructors to initialize **params** objects:

```
MbGridShellParams( const MbSNameMaker &          operNames,
                         IProgressIndicator * & prog = nullptr )
```

```
: _sewGrids       ( true                  )
, _mergeFaces     ( false                 )
, _useGridSurface ( false                 )
, _copyAttributes ( false                 )
, _operNames      ( &operNames.Duplicate() )
, _progress       ( prog                  ){ },
```

```
MbGridShellParams(       bool                    sg,
                         bool                    mf,
                         bool                    ugs,
                         bool                    ca,
                   const MbSNameMaker &          operNames,
                         IProgressIndicator * & prog = nullptr )
```

```
: _sewGrids       ( sg                    )
, _mergeFaces     ( mf                    )
, _useGridSurface ( ugs                   )
, _copyAttributes ( ca                    )
, _operNames      ( &operNames.Duplicate() )
, _progress       ( prog                  ){ }.
```

- **_sewGrids** – the flag for sewing face sets from different triangulation meshes into a single shell (*true* – sew face sets from different triangulation meshes; *false* – each mesh defines a separate shell part)
- The **_mergeFaces** flag controls the merging of *similar* adjacent faces to simplify the shell (*true* – merge similar faces)
- **_useGridSurface** – the flag for using a triangulation-based surface (*true* – use a triangulation-based surface; *false* – construct the shell without creating a mesh-based surface)
- The **_copyAttributes** flag controls the transfer of attributes (colors, names, custom data) from MbMesh faces to the faces of the resulting body (*true* – copy attributes)
- **_operNames** – the namer of the created faces, edges and vertices in the construction tree

Note

Similar faces are adjacent faces that can be merged without changing the shell geometry. For example, similar planar faces lie in the same plane and have consistently oriented normals.

Figs. 1 and 2 show the **mesh** polygonal model and the **result** body constructed on its basis:

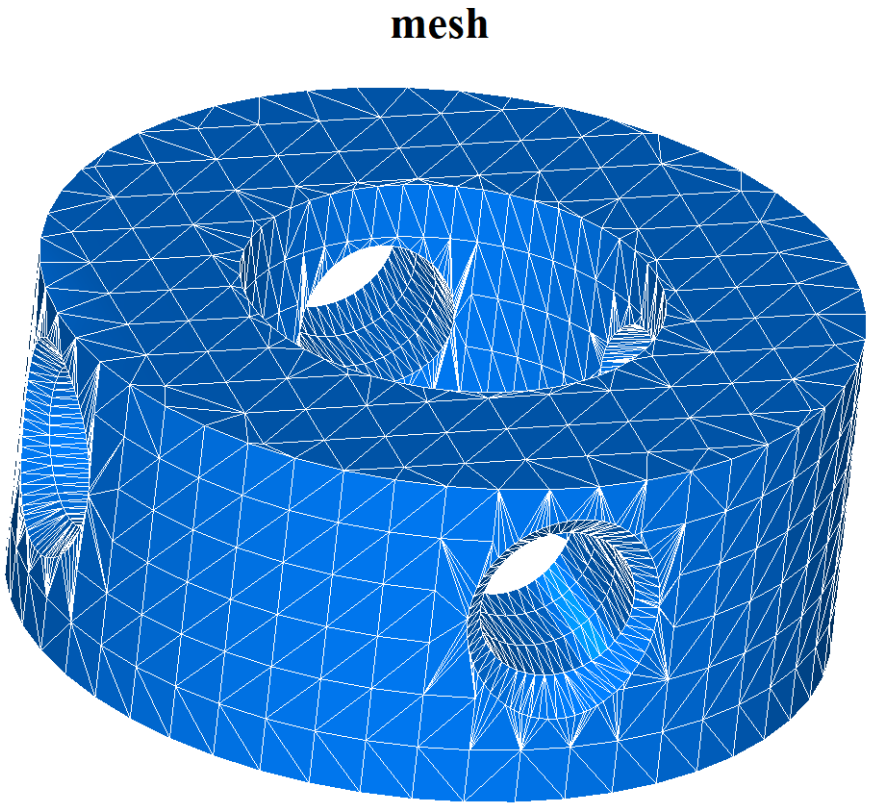


Fig. 1

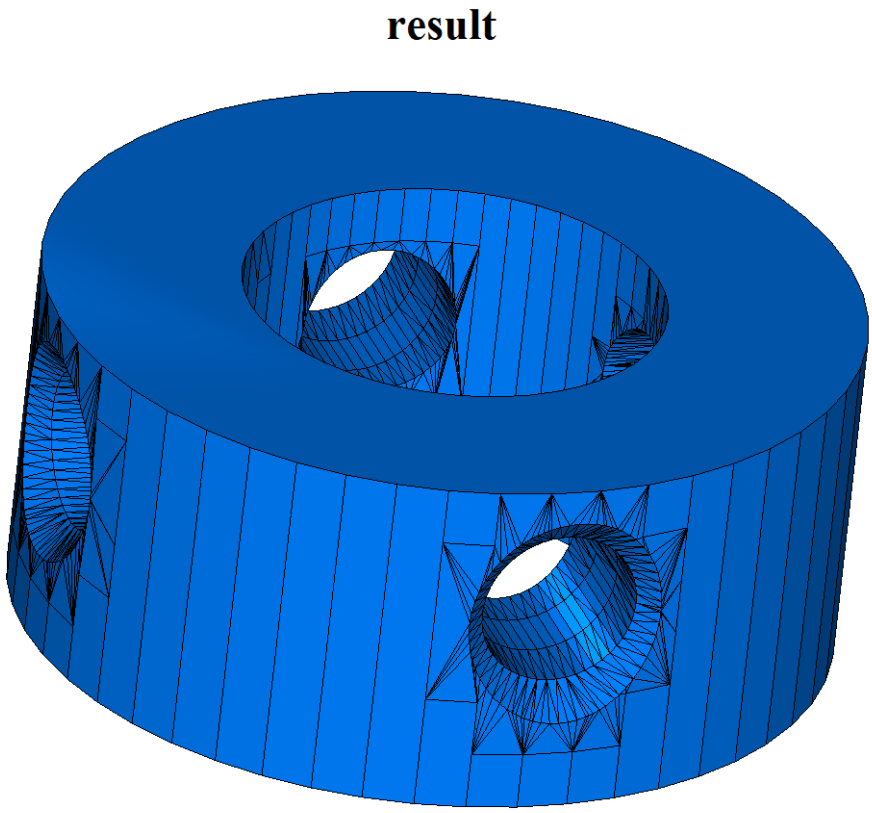


Fig. 2

The MeshSolid() function adds an MbSimpleMeshCreator builder, which contains all data required to perform the operation, to the log of the constructed body. The MbSimpleMeshCreator builder is declared in the *cr_mesh_creator.h* file.

In the *test.exe* application, a body based on a polygonal model is constructed by the «Create → Body → Based on a Polygonal Model» menu command.

3.2 OPERATIONS ON BODIES

One of approaches to construction of bodies in geometrical modeling 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.

3.2.1 Boolean Operation on Bodies

The function

```
MbResultType BooleanResult( const c3d::SolidSPtr &          solid1,
                            MbeCopyMode                     sameShell1,
```

(continues on next page)

(continued from previous page)

```
                              const c3d::SolidSPtr &           solid2,
                                    MbeCopyMode                sameShell2,
                              const MbBooleanOperationParams & params,
                                    c3d::SolidSPtr &           result );
```

constructs a new body by executing a Boolean operation on two solids with the ability to control face and edge merging.

The function input parameters are:

- **solid1** — the first solid for the Boolean operation;
- **sameShell1** — the copy mode for the first solid;
- **solid2** — the second solid for the Boolean operation;
- **sameShell2** — the copy mode for the second solid;
- **params** — the parameters of the Boolean operation.

The function output parameter is the **result** object containing the constructed solid.

The function returns a result code from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in the file *action_solid.h*.

The function performs union, intersection and difference operations on two solids **solid1** and **solid2**.

The parameters **sameShell1** and **sameShell2** control the transfer of faces, edges and vertices from the source solids **solid1** and **solid2** to the constructed solid **result**, and can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy do not share any common data).

A detailed description of the MbeCopyMode enumeration is provided in section *Copying a Set of Faces*.

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

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

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

where

- **oType** — the type of Boolean operation;
- **flags** — control flags for the Boolean operation;
- **operNames** — name maker;
- **closed** — indicator of closure of the operand shells;
- **progress** — progress indicator for operation execution.

The variant of MbBooleanOperationParams object with the **closed** parameter works only for non-closed solids and specifies the need to check the result for closure.

The parameter **oType** defines the type of Boolean operation and can take three values: *bo_Union*, *bo_Intersect*, *bo_Difference*:

- when **oType** = *bo_Union*, the function performs union of solids **solid1** and **solid2**;
- when **oType** = *bo_Intersect*, the function performs intersection of solids **solid1** and **solid2**;
- when **oType** = *bo_Difference*, the function performs subtraction of **solid2** from **solid1**.

The parameter **operNames** is used for naming resulting objects of the Boolean operation.

The parameter **flags** contains control flags for the Boolean operation.

The object has the following flags, which can be set in the constructor when creating the object, or using separate setter functions:

bool **mergeFaces** — merge similar faces;

bool **mergeEdges** — merge similar edges;

bool **closed** — closure of the operand shells;

bool **enclosureCheck** — check shells for nesting;

bool **allowNonIntersecting** — output the resulting shell if there are no intersections;

bool **cutting** — shell cutting flag when building cuts and sections;

bool **repairShellEdges** — shell edge repair flag for source shells.

Figure 1 shows the source solids **solid1** and **solid2**:

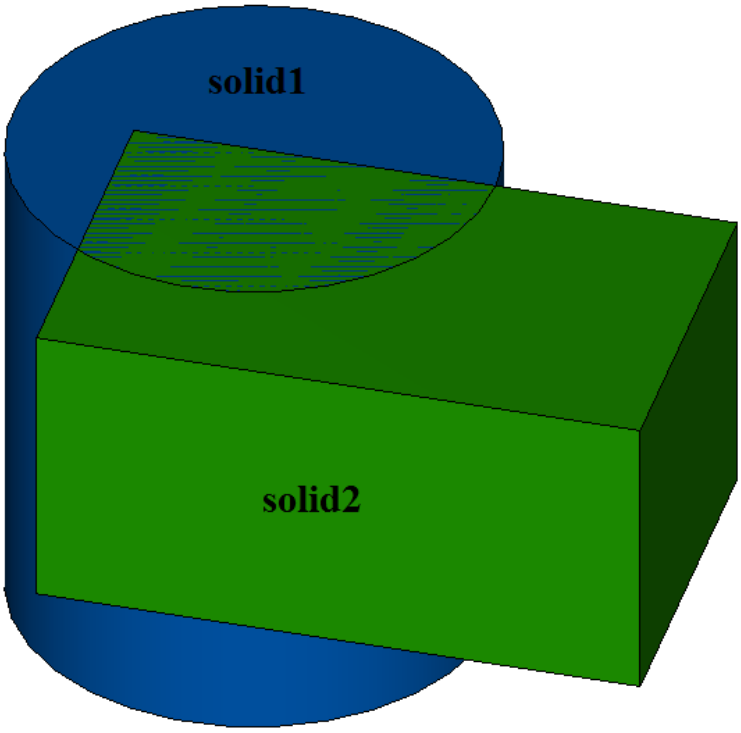
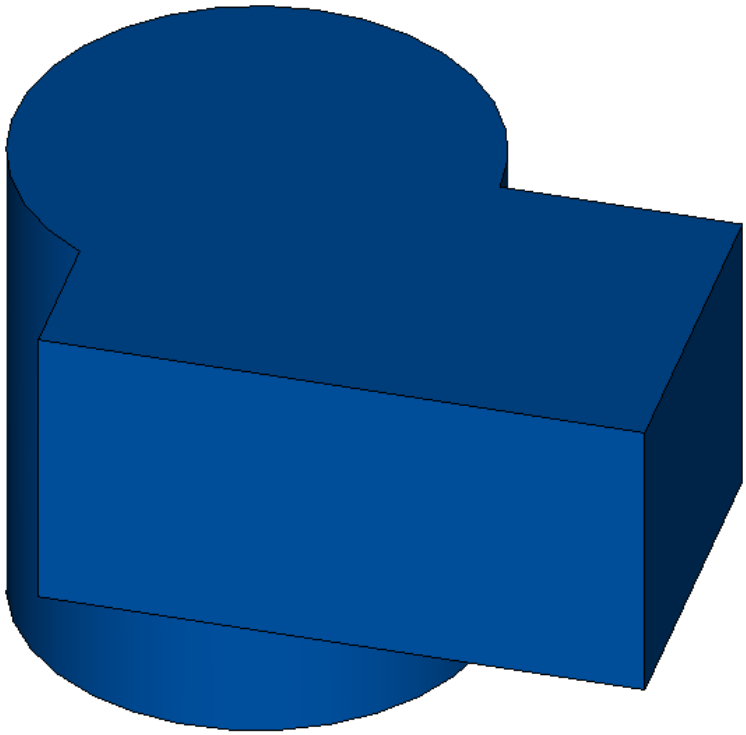


Рис. 1

Figure 2 shows the result of the Boolean union operation on solids **solid1** and **solid2** shown in Fig. 1:

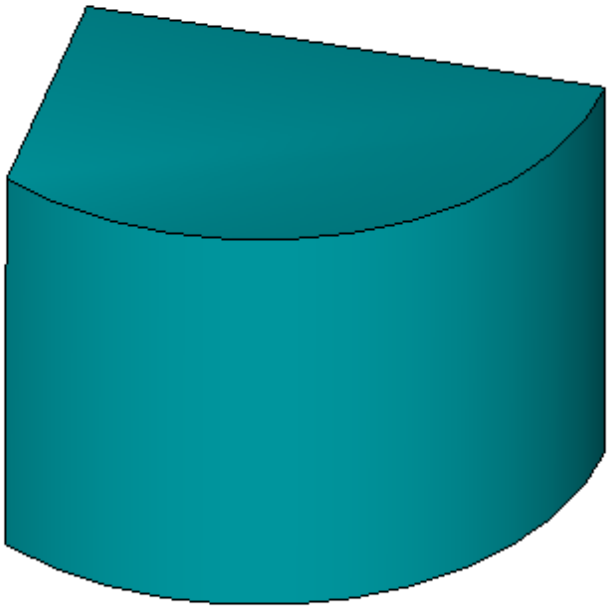


$oType = \text{bo\_Union}$

solid1 + solid2

Рис. 2

Figure 3 shows the result of the Boolean intersection operation on solids **solid1** and **solid2** shown in Fig. 1:

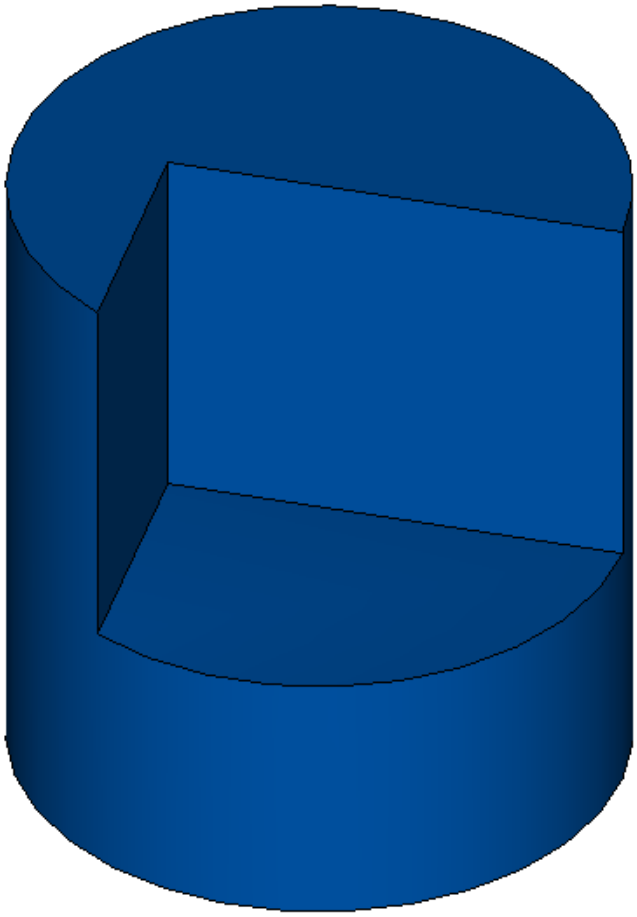


oType = bo_Intersect

solid1&solid2

Рис. 3

Figure 4 shows the result of the Boolean difference operation (subtracting **solid2** from **solid1**) on the solids shown in Fig. 1:

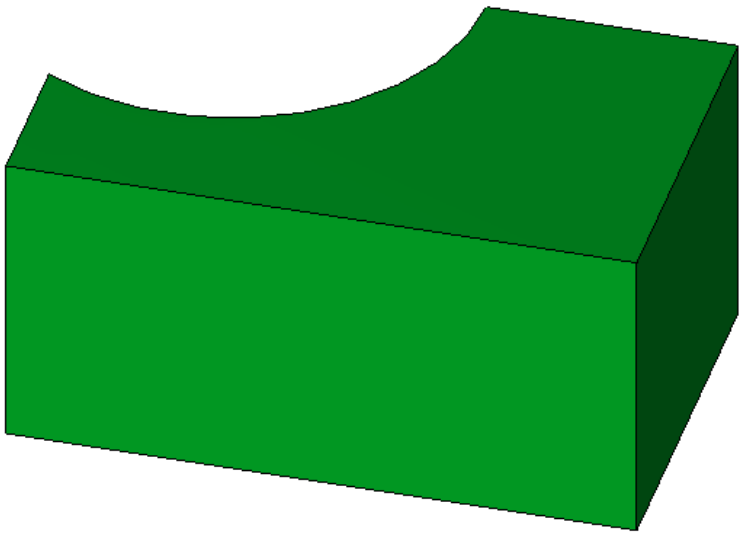


oType = bo_Difference

solid1 - solid2

Рис. 4

Figure 5 shows the result of the Boolean difference operation (subtracting solid **solid1** from solid **solid2**) on the solids shown in Fig. 1:



oType = bo_Difference

solid2 - solid1

Рис. 5

To demonstrate the function's operation with an MbBooleanFlags object with different values of the *mergeFaces* flag, consider Boolean operations on source solids **solid1** and **solid2** shown in Fig. 6:

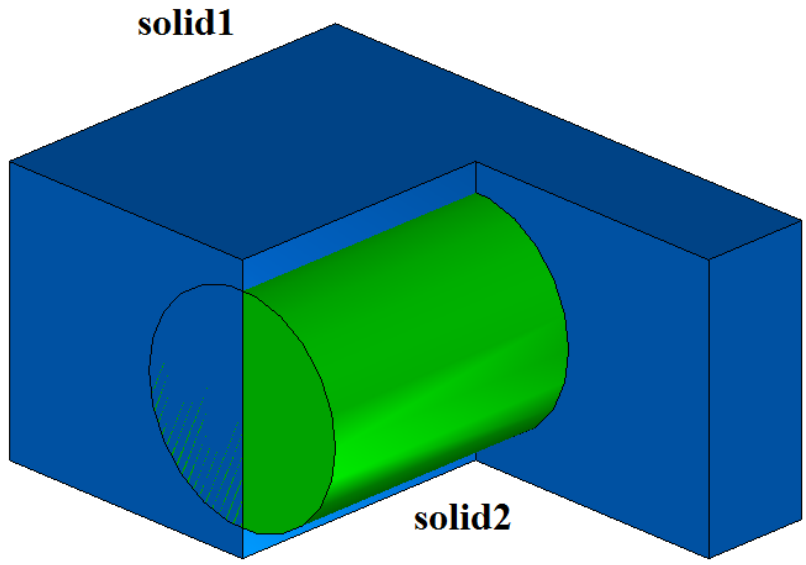


Рис. 6

Figure 7 shows the result of the union of solids **solid1** and **solid2** when the function operates with the flag *mergeFaces* = *true*:

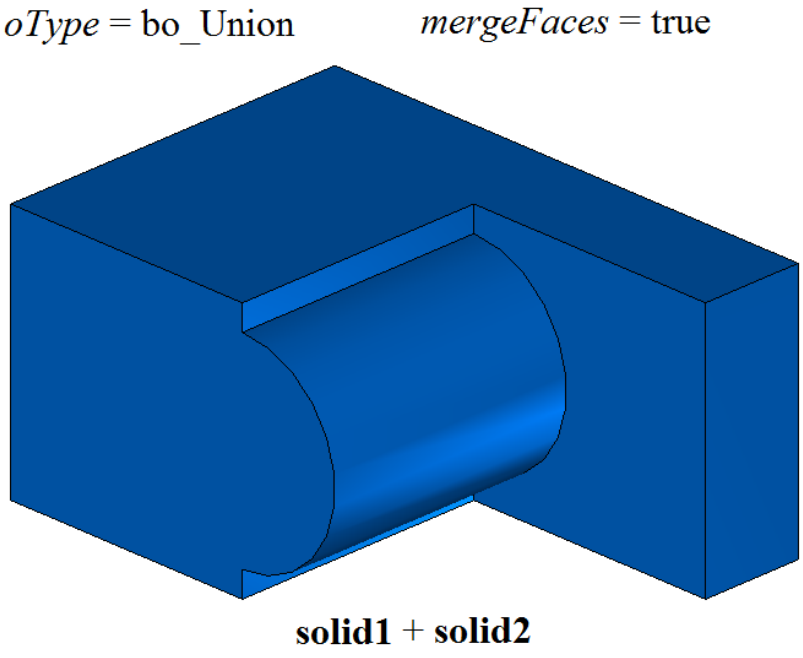


Рис. 7

Figure 8 shows the result of the union of solids **solid1** and **solid2** when the function operates

with the flag *mergeFaces* = *false*. Coinciding faces in Fig. 8 are not merged:

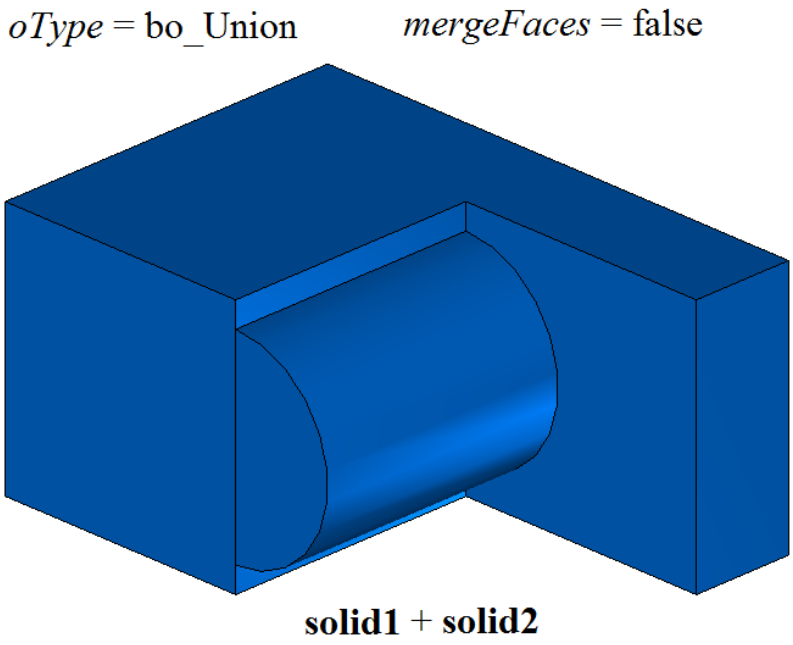


Рис. 8

Figure 9 shows the result of the difference operation (subtracting **solid2** from **solid1**) when the function operates with the parameter *mergeFaces* = *true* (merge similar faces):

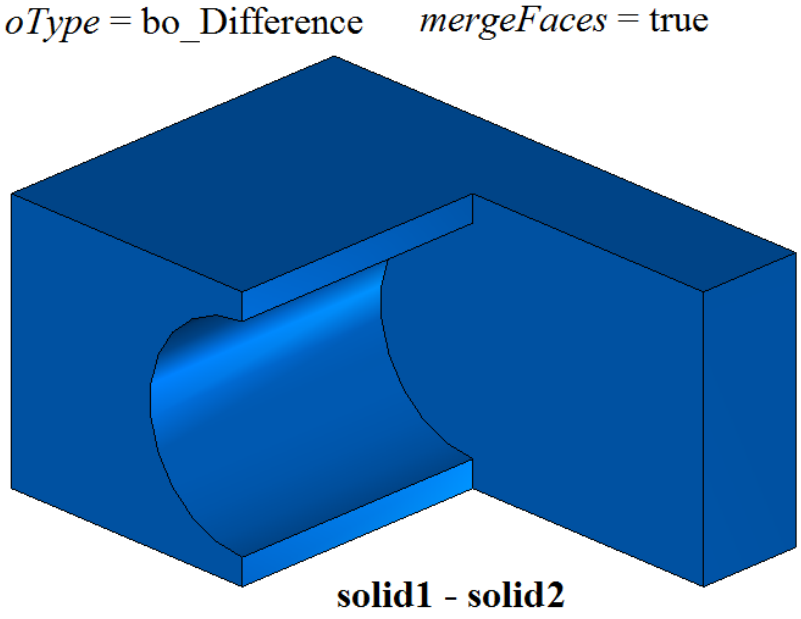


Рис. 9

Figure 10 shows the result of the difference operation (subtracting **solid2** from **solid1**) when the function operates with the parameter *mergeFaces* = *false*. Coinciding faces in Fig. 10 are not merged:

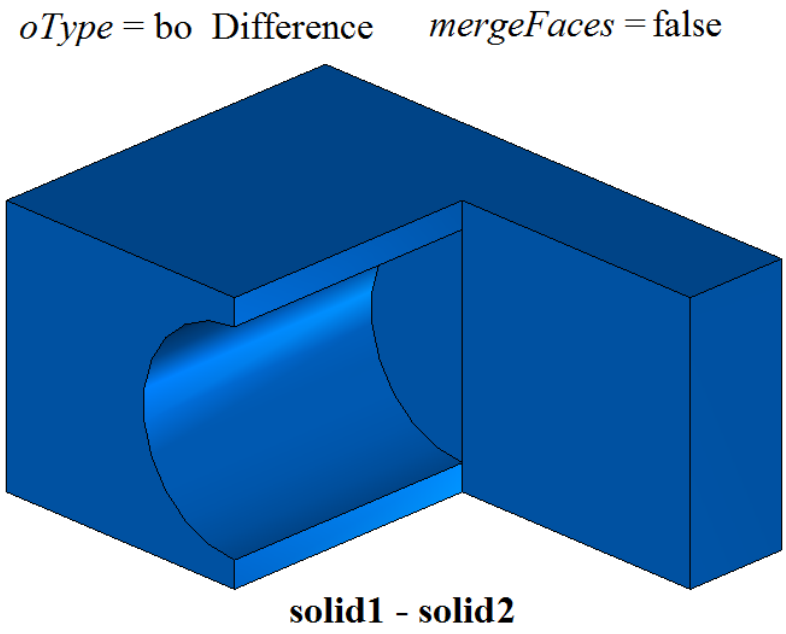


Рис. 10

The function BooleanResult() adds to the constructed solid's history log a builder MbBoolean-Solid, which contains all the necessary data to execute the operation. The builder MbBoolean-Solid is declared in the file *cr_boolean_solid.h*.

Test.exe application executes Boolean operations on bodies using Create→Solid→As union with a solid→Solid, Create→Solid→As subtraction with a solid→Solid, Create→Solid→As intersection with a solid→Solid menu commands.

3.2.2 Boolean Operation on Extrusion Body

The function

```
MbResultType ExtrusionResult(       MbSolid &                solid,
                                    MbeCopyMode              sameShell,
                              const MbSweptData &            sweptData,
                              const MbVector3D &             direction,
                              const ExtrusionValues &        params,
                                    OperationType            oType,
                              const MbSNameMaker &           operNames,
                              const RPArray<MbSNameMaker> &  contoursNames,
                                    MbSolid *&               result );
```

constructs an extrusion body and executes a Boolean operation between a specified body and the constructed extrusion body.

The input parameters of the function are:

- **solid** – a given body for the Boolean operation;
- **sameShell** – the copy mode for the given body;
- **sweptData** – data about the generatrix curves for constructing the extrusion body;
- **direction** – an extrusion direction;
- **params** – an object containing construction parameters that provide information for constructing array operation elements up to the surface;
- **oType** – a type of Boolean operation:

  bo_Union – union of bodies,

  bo_Intersect – intersection of bodies,

  bo_Difference – difference of bodies;
- **operNames** – an operation name maker;
- **contoursNames** – name makers for the generatrix curves.

The output parameter of the function is the object **result** containing the constructed body.

The function returns a result code from the enumeration MbResultType (see *mb_operation_result.h*).

The function is declared in the file *action_solid.h*.

The function is implemented as a sequential combination of two functions: the function ExtrusionSolid(), which constructs a body by extruding the curves **sweptData** according to the specified parameters **params** in the direction **direction**, and the function BooleanResult(), which performs the Boolean operation **oType** of body **solid** with the body constructed in the previous step.

The function ExtrusionSolid() is described in Section *Constructing an Extrusion Body*, the function BooleanResult() is described in Section *Boolean Operation on Bodies*.

The parameter **sameShell** controls the transfer of faces, edges, and vertices from the original body **solid** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the original shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the original shell and its copy share common base surfaces);
- *cm_Copy* (the original shell and its copy have no shared data).

A detailed description of the enumeration MbeCopyMode is provided in Section *Copying a Set of Faces MbFaceShell*.

The parameter **oType** defines the type of Boolean operation and can take three values: *bo_Union*, *bo_Intersect*, *bo_Difference*.

When **oType** = *bo_Union*, the function performs a union of bodies **solid** and the extrusion body.

When **oType** = *bo_Intersect*, the function performs an intersection of bodies **solid** and the extrusion body.

When **oType** = *bo_Difference*, the function performs a difference operation, subtracting the extrusion body from body **solid**.

The parameters **operNames** and **contoursNames** provide naming for faces, edges, and vertices of the constructed body.

The function ExtrusionResult() when constructing a body by extruding curves has the same capabilities as the function ExtrusionSolid(): the extruded curves can be located on a plane, a curved surface, or in space. Extrusion can be performed in forward, reverse, or both directions, with or without face tapering. The body can completely fill closed curves or have thin walls.

The following sections focus on selected capabilities of this function related to Boolean operations.

Figure 1 shows body **solid**, the generatrix curve contained in data **sweptData**, and the extrusion direction **direction**:

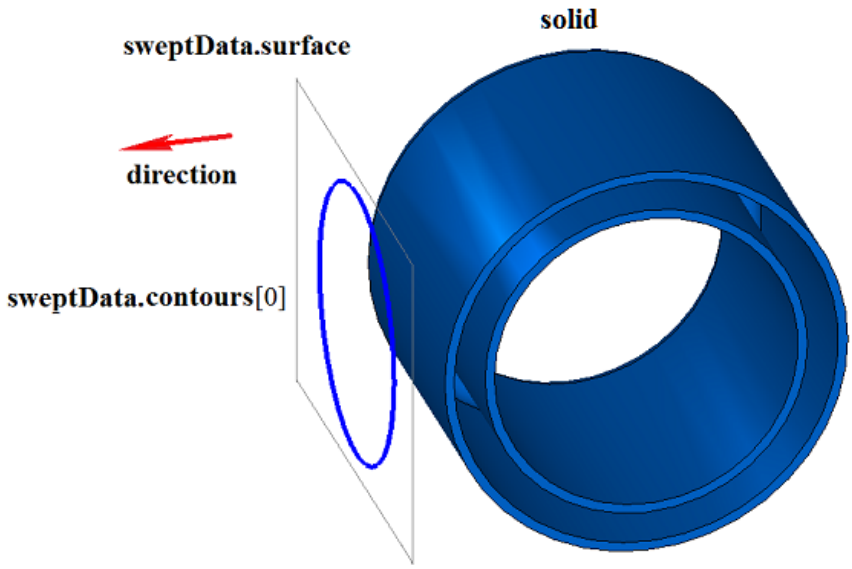


Fig. 1

Figure 2 shows the result of the union Boolean operation of body **solid** and a thin-walled body obtained by extruding the generatrix curve **sweptData** in the direction **direction** shown in Figure 1 by a specified distance:

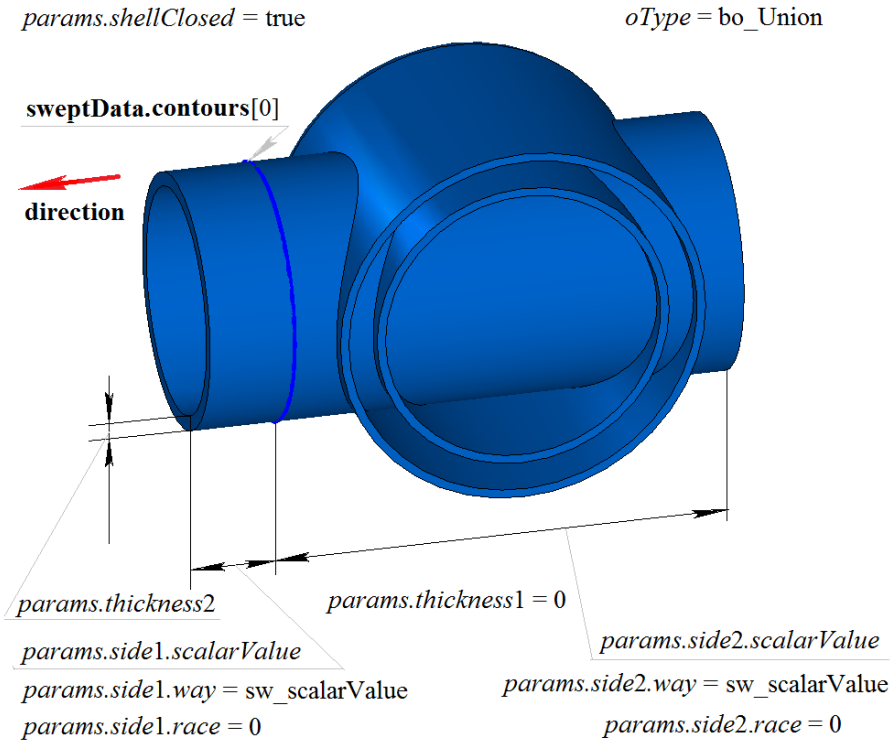


Fig. 2

Figure 3 shows the result of the Boolean union of body **solid** and the body obtained by extruding the generatrix curve **sweptData** in the direction **direction** shown in Figure 1. The extrusion of the generatrix curve was performed in the reverse direction without tapering with the “Up to Nearest Objects” option (*params.side2.way* = sw_shell):

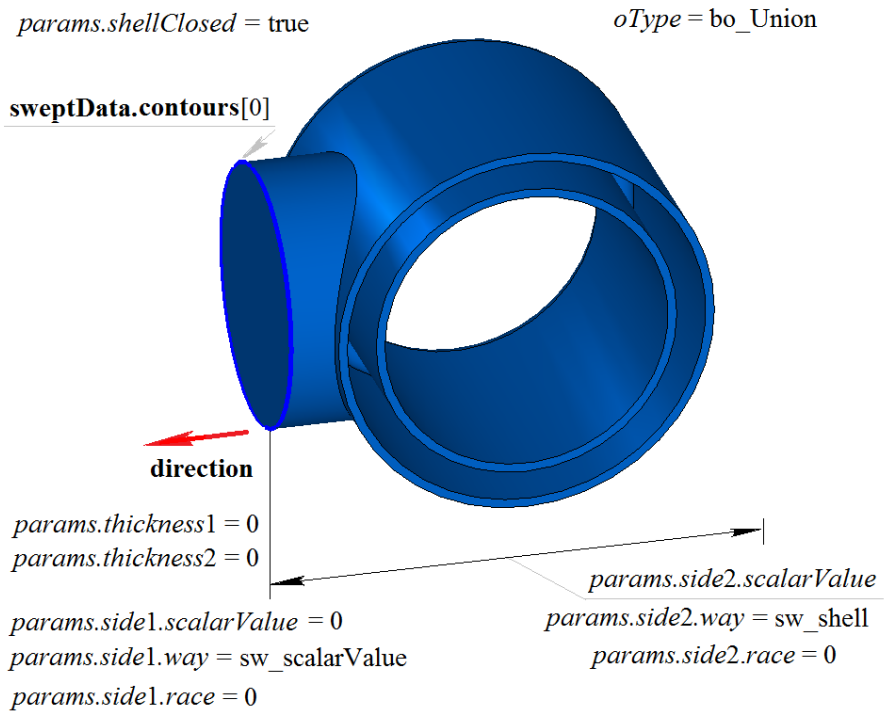


Fig. 3

Figure 4 shows the result of the Boolean difference of subtracting from body **solid** a body obtained by extruding the generatrix curve **sweptData** in the direction **direction** shown in Figure 1:

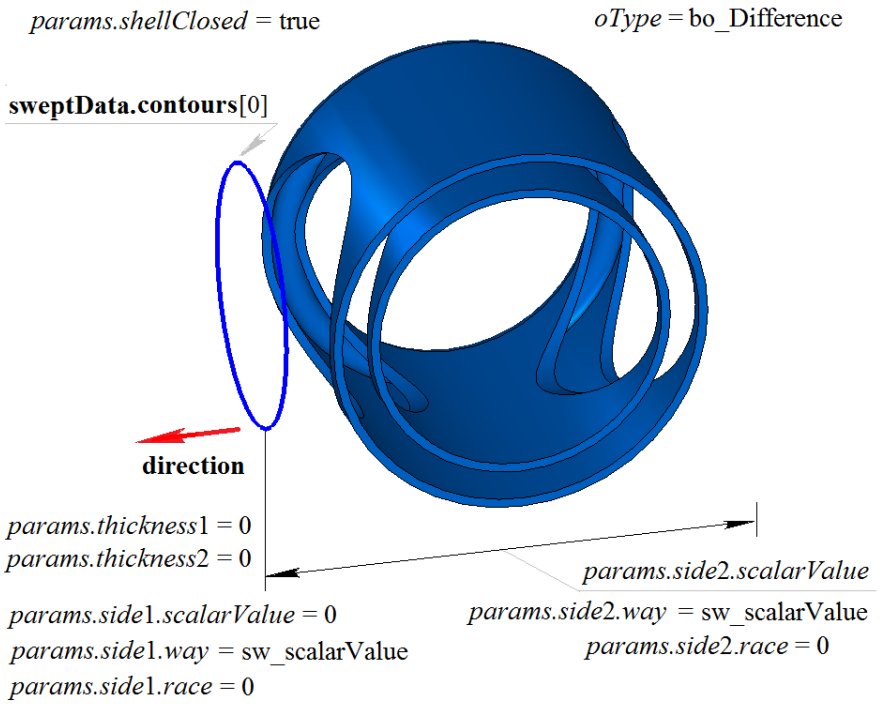


Fig. 4

Figure 5 shows the result of the Boolean difference of subtracting from body **solid** a body obtained by extruding the generatrix curve **sweptData** in the direction **direction** shown in Figure 1. The extrusion of the generatrix curve was performed in the reverse direction without tapering with the “Up to Nearest Objects” option (*params.side2.way* = sw_shell):

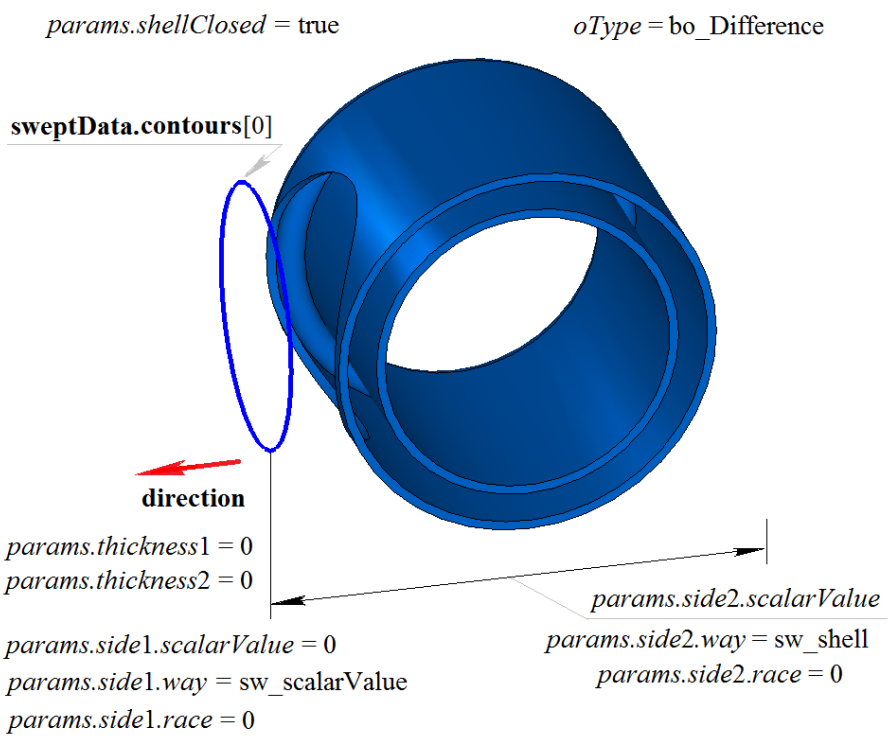


Fig. 5

Figure 6 shows the result of the Boolean intersection of body **solid** and a body obtained by extruding the generatrix curve **sweptData** in the direction **direction** shown in Figure 1:

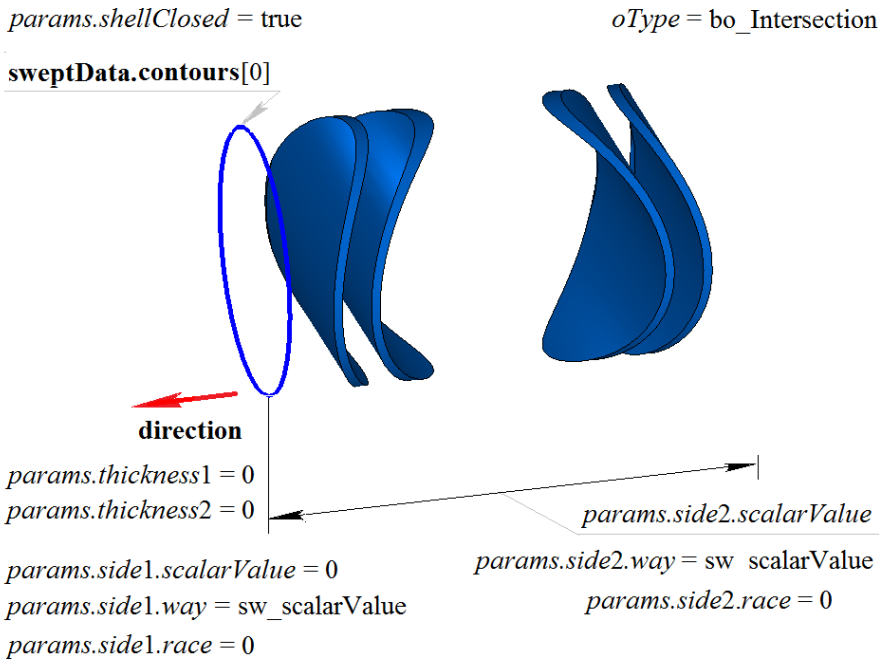


Fig. 6

Figure 7 shows the result of the intersection Boolean operation of body **solid** and a body obtained by extruding the generatrix curve **sweptData** in the direction **direction** shown in Figure 1. The extrusion of the generatrix curve was performed in the reverse direction without tapering with the “Up to Nearest Objects” option:

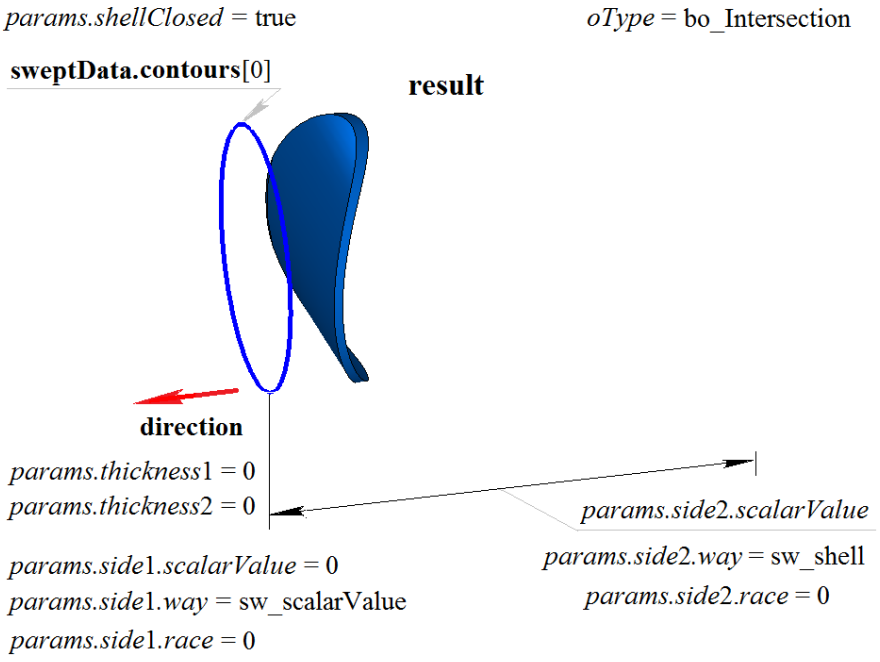


Fig. 7

The function ExtrusionResult() adds the builder MbCurveExtrusionSolid to the construction log of the created body, which contains all necessary data for executing the operation. The builder MbCurveExtrusionSolid is declared in the file *cr_extrusion_solid.h*.

In the test application *test.exe*, Boolean operations with the constructed extrusion body are executed through the following menu commands:

- Create→Solid→As union with a solid→Curve extrusion;
- Create→Solid→As subtraction with a solid→Curve extrusion;
- Create→Solid→As intersection with a solid→Curve extrusion.

3.2.3 Boolean Operation on Revolution Body

The function

```
MbResultType RevolutionResult(       MbSolid &                solid,
                                     MbeCopyMode              sameShell,
                               const MbSweptData &            sweptData,
                               const MbAxis3D &               axis,
                               const RevolutionValues &       params,
                                     OperationType            oType,
                               const MbSNameMaker &           operNames,
                               const RPArray<MbSNameMaker> &  contoursNames,
                                     MbSolid *&               result );
```

performs the construction of a revolution body using **sweptData** and a Boolean operation of the given body **solid** with the constructed body.

The input parameters of the function are:

- **solid** – the given body for the Boolean operation;
- **sameShell** – a copy mode for the given body;
- **sweptData** – parameters defining the generatrix curves for construction of the revolution body;
- **axis** – an axis of revolution;
- **params** – parameters of the revolution construction;
- **oType** – a type of Boolean operation:

  bo_Union – union of bodies,

  bo_Intersect – intersection of bodies,

  bo_Difference – difference of bodies;
- **operNames** – an operation name maker;
- **contoursNames** – name makers for segments of generatrix curves (contours) used in construction of the revolution body.

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

The function returns the operation result code from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in the file *action_solid.h*.

The function represents a sequential combination of two functions: the RevolutionSolid() function, which performs body construction by revolving the curves **sweptData** according to the given parameters **params** around the axis **axis**, and the BooleanResult() function, which performs the Boolean operation **oType** on the body **solid** and the body constructed at the previous step. The RevolutionSolid() function is described in the section *Constructing a Revolution Body*, and the BooleanResult() function is described in the section *Boolean Operation on Bodies*.

The parameter **sameShell** controls the transfer of faces, edges, and vertices from the original body **solid** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the original shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the original shell and its copy share common base surfaces);
- *cm_Copy* (the original shell and its copy have no common data).

The enumeration MbeCopyMode is described in detail in the section *Copying a Set of Faces MbFaceShell*.

The rotation parameters and the structure of the result are defined by the fields of **params**: the directions and revolution angles are specified by **params**.*side1* and **params**.*side2*, the wall thickness is specified by **params**.*thickness1* and **params**.*thickness2*, the shell closedness is specified by **params**.*shellClosed*, the merging of similar faces and edges is controlled by the **params**.MergingFlags() flags, and the topology (spherical or toroidal) is specified by **params**.*shape*.

The parameter **oType** defines the type of Boolean operation and can take three values: *bo_Union*, *bo_Intersect*, *bo_Difference*.

If **oType** is *bo_Union*, the union of the body **solid** and the revolution body is performed.

If **oType** is *bo_Intersect*, the intersection of the body **solid** and the revolution body is performed.

If **oType** is *bo_Difference*, the revolution body is subtracted from the body **solid**.

The parameters **operNames** and **contoursNames** provide naming of faces, edges, and vertices of the constructed body.

The RevolutionResult() function has the same capabilities as RevolutionSolid() when constructing a body by revolving curves: the revolved curves can lie on a plane, on a curved surface, or in space. The revolution can be performed in the forward, reverse, or both directions. The body can completely fill closed curves or have a thin wall.

The following examples demonstrate the function's capabilities related to Boolean operations.

Figure 1 shows the body **solid**, the generatrix curve included in the **sweptData**, and the axis of revolution **axis**:

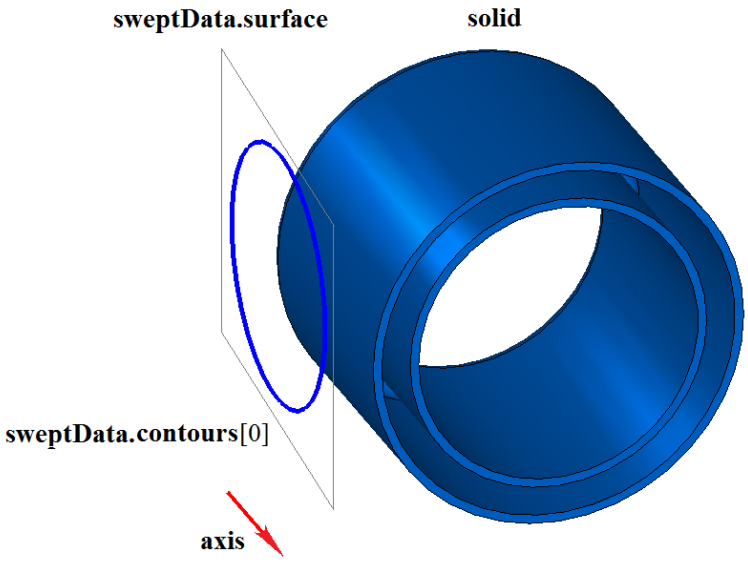


Fig. 1

Figure 2 shows the result of the Boolean union operation of the body **solid** and a thin-walled revolution body obtained by revolving the generatrix curve **sweptData** around the axis **axis**, shown in Figure 1:

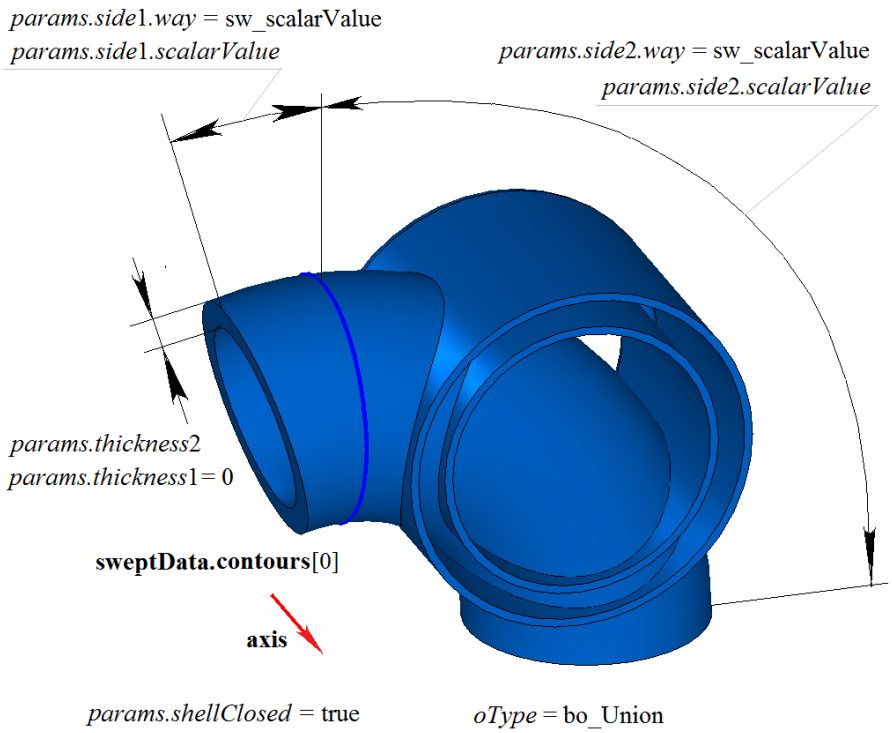


Fig. 2

Figure 3 shows the result of the Boolean union operation of the body **solid** and a thin-walled revolution body obtained by revolving the generatrix curve **sweptData** around the **axis**, shown in Figure 1. The revolution of the generatrix curve is performed in the reverse direction with the option “Up to Surface” (**params**.*side2.way* = *sw_surface*).

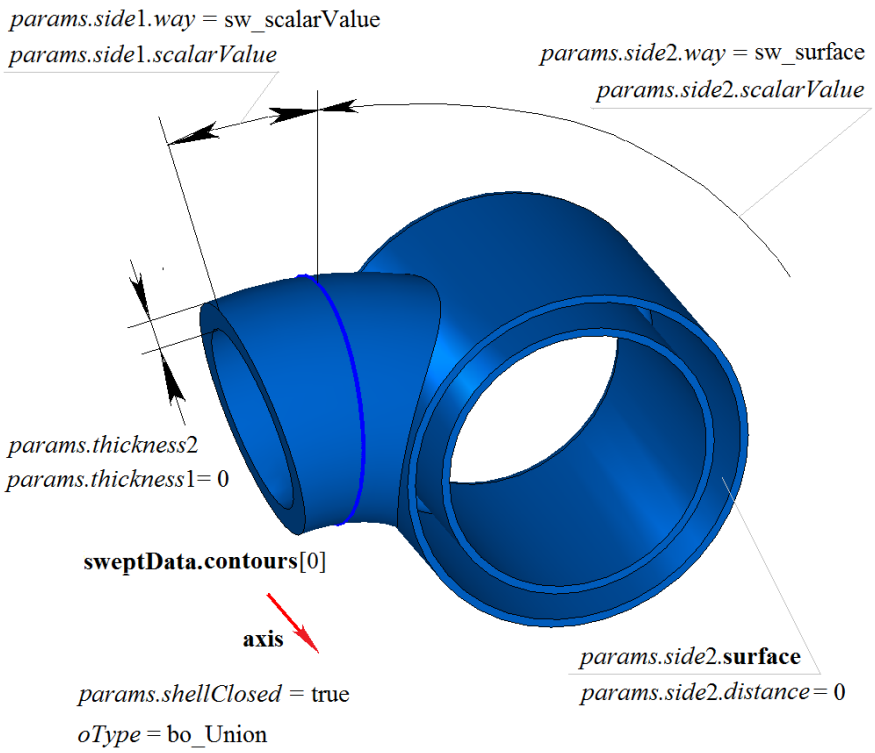


Fig. 3

Figure 4 shows the result of the Boolean difference operation of the body **solid** and the revolution body obtained by revolving the generatrix curve **sweptData** around the axis **axis**, shown in Figure 1:

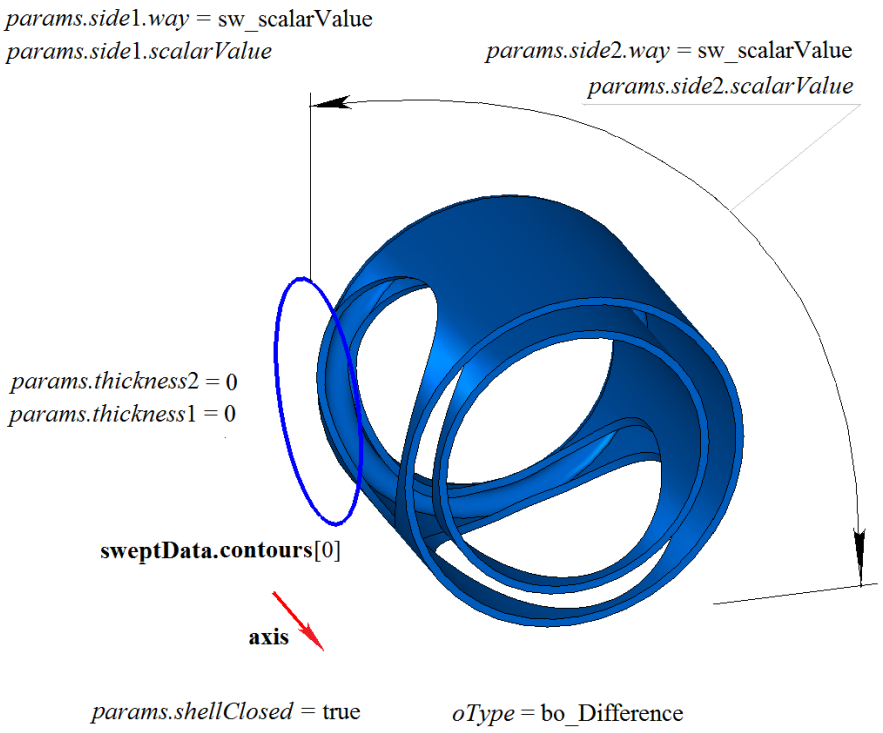


Fig. 4

Figure 5 shows the result of the Boolean difference operation of the body **solid** and the revolution body obtained by revolving the generatrix curve **sweptData** around the axis **axis**, shown in Figure 1. The revolution of the generatrix curve is performed in the reverse direction with the option “Up to Surface” (*params.side2.way = sw_surface*).

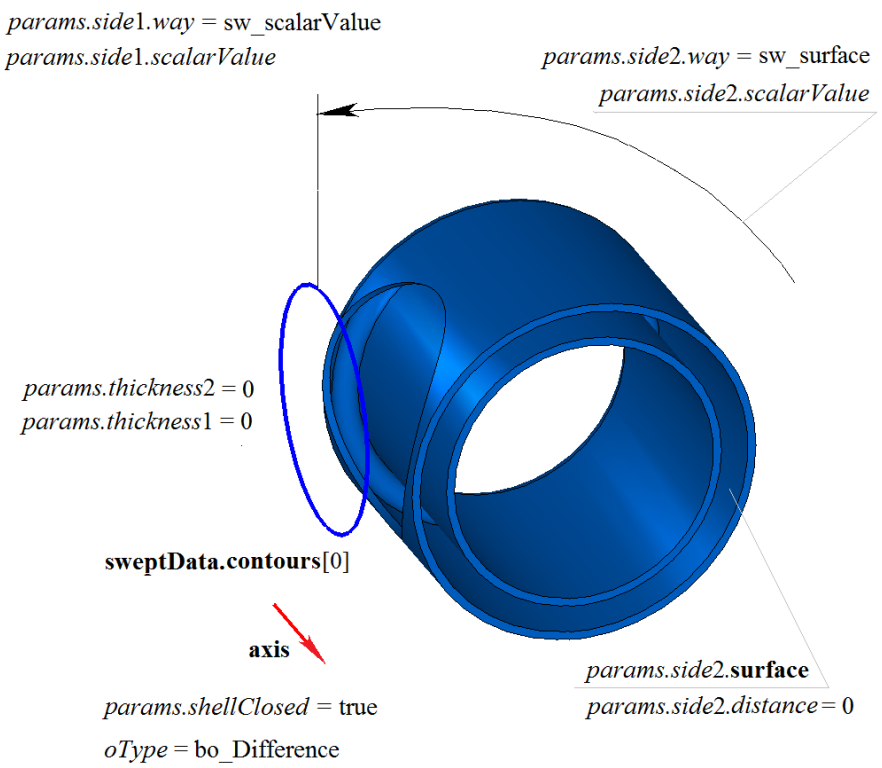


Fig. 5

Figure 6 shows the result of the Boolean intersection operation of the body **solid** and the revolution body obtained by revolving the generatrix curve **sweptData** around the **axis**, shown in Figure 1:

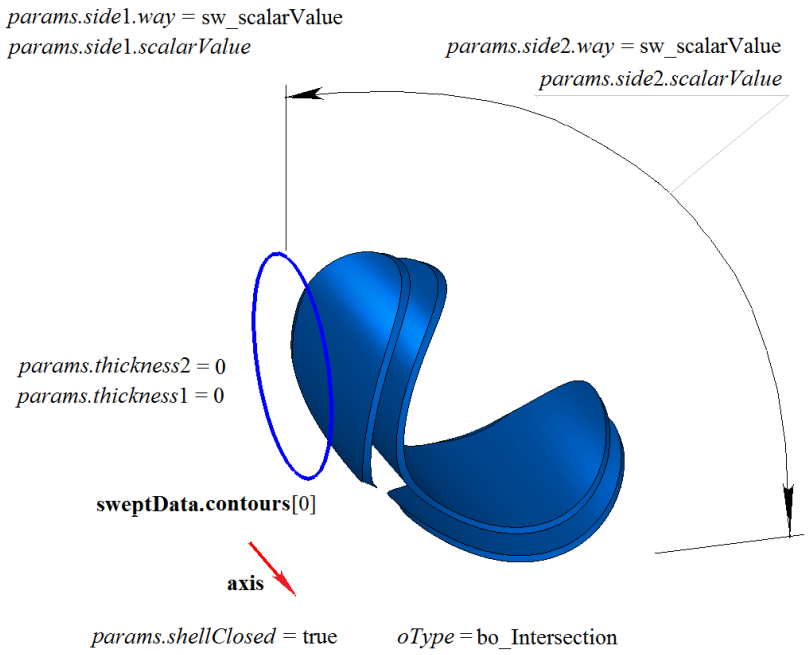


Fig. 6

Figure 7 shows the result of the Boolean intersection operation of the body **solid** and the revolution body obtained by revolving the generatrix curve **sweptData** around the **axis**, shown in Figure 1. The revolution of the generatrix curve is performed in the reverse direction with the option “Up to Surface” (*params.side2.way* = *sw_surface*).

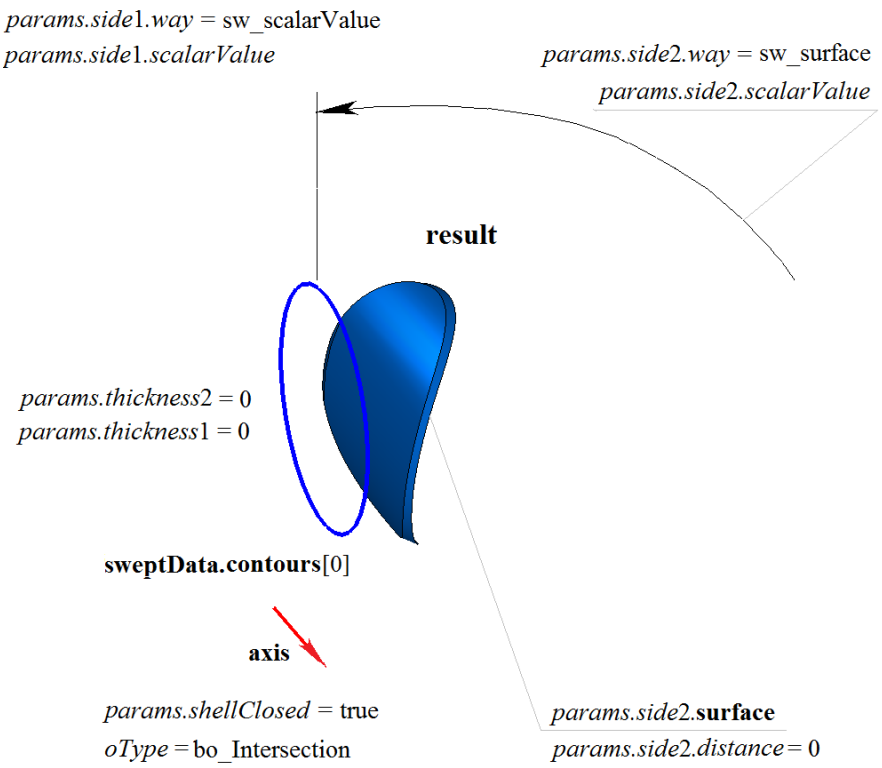


Fig. 7

The function RevolutionResult() adds the MbCurveRevolutionSolid builder to the construction log, which contains all necessary data for performing the operation. The builder MbCurveRevolutionSolid is declared in the file *cr_revolution_solid.h*.

In the test application *test.exe*, the Boolean operation with the constructed revolution body is performed by menu commands:

- Create→Solid→As union with a solid→Revolving a Curve revolution;
- Create→Solid→As subtraction with a solid→Revolving a Curve revolution;
- Create→Solid→As intersection with a solid→Revolving a Curve revolution.

3.2.4 Boolean Operation with Kinematic Body

The function

```
MbResultType EvolutionResult( const c3d::SolidSPtr &          srcSolid,
                              const MbeCopyMode               copyMode,
                                    OperationType             oType,
                              const MbEvolutionShellParams & evShellParams,
                                    c3d::SolidSPtr &          result );
```

performs the construction of a kinematic body and then a Boolean operation between the specified body and the constructed body.

The input parameters of the function are:

- **srcSolid** – a given body for the Boolean operation;

- **copyMode** – a copy mode of the specified body;
- **oType** – a type of Boolean operation:

  bo_Union – union of bodies,

  bo_Intersect – intersection of bodies,

  bo_Difference – subtraction of bodies;
- **evShellParams** – an object, that contains source data for constructing the kinematic body.

The output parameter of the function is the **result** object containing the constructed body.

The function returns an operation result code from the enumeration MbResultType (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

When creating the **evShellParams** object with source data for the construction, one of the following constructors is used (see *op_shell_parameter.h*):

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbCurve3D &              spineCurve,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                              IProgressIndicator *     prog = nullptr );
```

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbWireFrame &            spineFrame,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                              IProgressIndicator *     prog = nullptr );
```

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbSpine &                spine,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                              IProgressIndicator *     prog = nullptr );
```

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbSpine &                spine,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                        const VariableSectionValues &  varSectVals,
                              IProgressIndicator *     prog = nullptr );
```

where

- **sweptData** – data about the generatrix curve;

- **spineCurve** – a guide curve;
- **spineFrame** – a guide curve as a wire frame;
- **spine** – a guide curve with additional information;
- **evolParams** – an object containing operation parameters;
- **operName** – a name maker object for the operation;
- **contoursNames** – a name maker object for generatrix contours;
- **spineName** – a name maker object for the guide curve;
- **varSectVals** – data required for constructing a body along a path with changing section;
- **prog** – progress indicator for operation execution.

The function represents a sequential combination of two functions: the function EvolutionShell(), which performs body construction by moving generatrix curves **sweptData** along the guide curve **spine** (**spineCurve**, **spineFrame**), and the function BooleanResult(), which performs the Boolean operation **oType** of the body **srcSolid** with the kinematic body constructed in the previous step. The function EvolutionShell() is described in the section Constructing a Kinematic Body, the function BooleanResult() is described in the section *Boolean Operations on Bodies*.

The parameter **copyMode** controls the transfer of faces, edges and vertices from the source body **srcSolid** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy have no shared data).

A detailed description of the enumeration MbeCopyMode is provided in the section *Copying a Set of Faces MbFaceShell*.

The parameter **oType** determines the type of Boolean operation and can take three values: *bo_Union*, *bo_Intersect*, *bo_Difference*.

When **oType** = *bo_Union*, the function performs a union of the body **srcSolid** and the kinematic body.

When **oType** = *bo_Intersect*, the function performs an intersection of the body **srcSolid** and the kinematic body.

When **oType** = *bo_Difference*, the function subtracts the kinematic body from the body **srcSolid**.

The parameters **operName** and **contoursNames** provide naming for the faces, edges and vertices of the constructed body.

When constructing a body by moving curves, the EvolutionResult() function has the same capabilities as the EvolutionShell() function: guide curves can lie on a plane, on a curved surface or in space; the body can completely fill closed curves or have a thin wall. The following are some capabilities of the function related to Boolean operations.

Fig. 1 shows the body **srcSolid**, the generatrix curve **sweptData** and the guide curve **spine**:

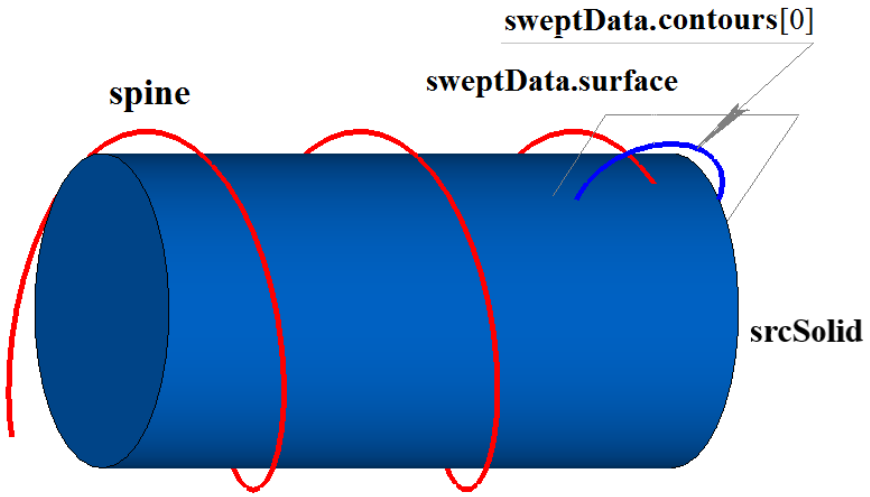


Fig. 1

Fig. 2 shows the result of the union Boolean operation of the body **srcSolid** and the body obtained by moving the generatrix curve **sweptData** along the guide curve **spine** shown in Fig. 1:

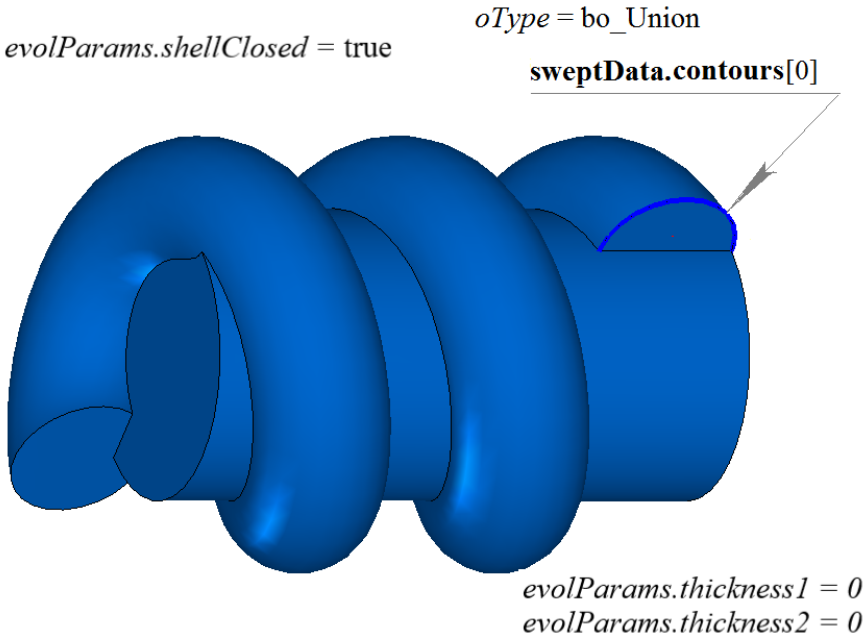


Fig. 2

Fig. 3 shows the result of the difference Boolean operation (subtracting the body obtained by moving the generatrix curve **sweptData** along the guide curve **spine** from the body **srcSolid**

shown in Fig. 1):

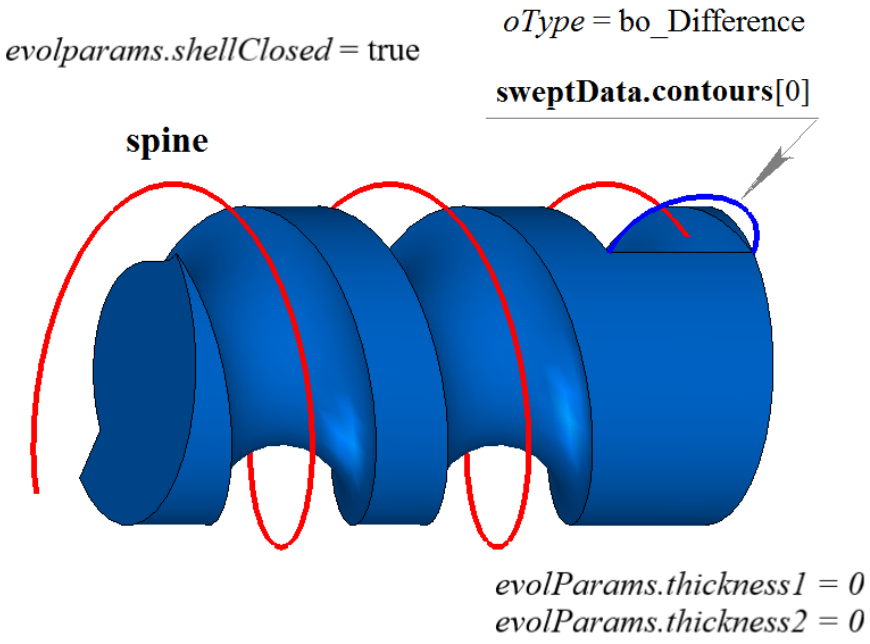


Fig. 3

Fig. 4 shows the result of the intersection Boolean operation of the body **srcSolid** and the body obtained by moving the generatrix curve **sweptData** along the guide curve **spine** shown in Fig. 1:

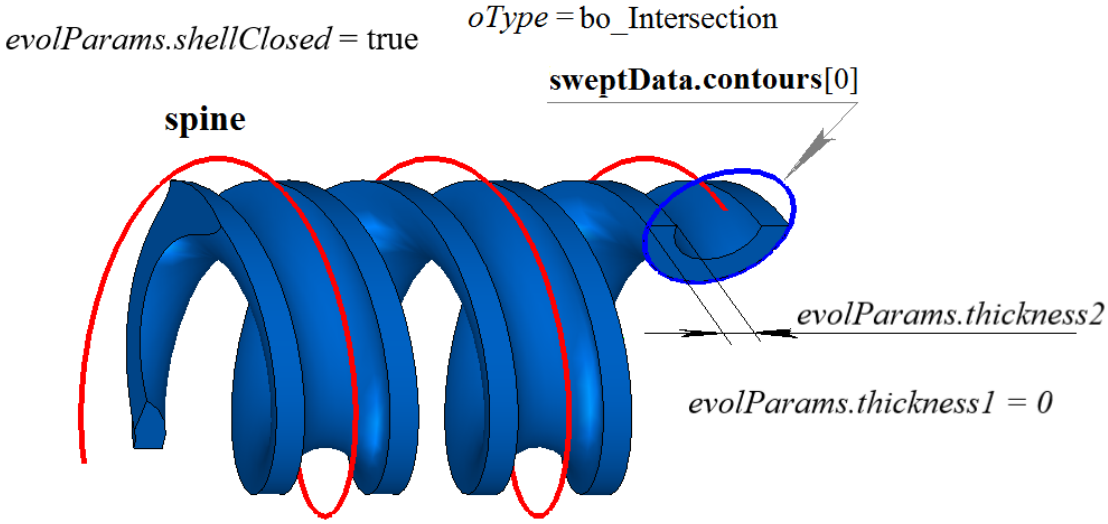


Fig. 4

The EvolutionResult() function adds the creator MbCurveEvolutionSolid to the construction

log of the body, which contains all necessary data for performing the operation. The creator MbCurveEvolutionSolid is declared in the file *cr_evolution_solid.h*.

In the test application *test.exe* the Boolean operation with the constructed kinematic body is performed by menu commands:

“Create→Solid→As union with a solid→Curve sweeping”,

“Create→Solid→As subtraction with a solid→Curve sweeping”,

“Create→Solid→As intersection with a solid→Curve sweeping”.

3.2.5 Boolean Operation with a Body Constructed on the Basis of Planar Sections

The function

```
MbResultType LoftedResult( const c3d::SolidSPtr &         srcSolid,
                           const MbeCopyMode              copyMode,
                           const MbLoftedSolidParams &    loftSolParams,
                                 c3d::SolidSPtr &         result );
```

constructs a solid from planar sections and performs a Boolean operation between the source and constructed solid.

The input parameters of the function are:

- **srcSolid** – a source body for the Boolean operation;
- **copyMode** – a copy mode of the specified body;
- **loftSolParams** – input data for the operation.

The function output parameter is the constructed body **result**.

The function returns an operation result code from the enumeration MbResultType (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

When creating the object **loftSolParams** with source data for construction, one of the following four constructors is used (see *op_shell_parameter.h*). The first two constructors do not require specifying the Boolean operation type, while the third and fourth constructors allow you to specify the operation type **oType**:

```
MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );
```

```
MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
```

(continues on next page)

(continued from previous page)

```
                      const c3d::SpaceCurvesSPtrVector *    guideCurves,
                      const c3d::SpacePointsVector *        guidePoints,
                      const MbSNameMaker &                  operName,
                      const c3d::SNamesMakerSPtrVector &    contourNames,
                            IProgressIndicator *            prog = nullptr );
```

```
MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                      const c3d::PlaneContoursSPtrVector &  planeConours,
                      const c3d::ConstSpaceCurveSPtr &      spine,
                      const LoftedValues &                  params,
                      const OperationType                   oType,
                      const c3d::SpaceCurvesSPtrVector *    guideCurves,
                      const c3d::SpacePointsVector *        guidePoints,
                      const MbSNameMaker &                  operName,
                      const c3d::SNamesMakerSPtrVector &    contourNames,
                            IProgressIndicator *            prog = nullptr );
```

```
MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                      const c3d::PlaneContoursSPtrVector &  planeConours,
                      const c3d::ConstSpaceCurveSPtr &      spine,
                      const LoftedValues &                  params,
                      const OperationType                   oType,
                      const c3d::SpaceCurvesSPtrVector *    guideCurves,
                      const c3d::SpacePointsVector *        guidePoints,
                      const MbSNameMaker &                  operName,
                      const c3d::SNamesMakerSPtrVector &    contourNames,
                            IProgressIndicator *            prog = nullptr );
```

where

- **places** – a set of coordinate systems of the generatrix contours;
- **surfaces** – a set of surfaces of the generatrix contours;
- **planeContours** – a set of generatrix contours;
- **spine** – an axis curve (can be nullptr);
- **params** – an object containing operation parameters;
- **oType** – a type of Boolean operation;
- **guideCurves** – a set of guide curves that defines trajectories for corresponding contour points (optional, can be nullptr);
- **guidePoints** – a set of control points on the generatrix contours that defines corresponding points or chain of points (optional, can be nullptr);
- **operName** – an operation name maker;
- **contourNames** – name makers for the generatrix contours.

The function represents a sequential combination of two functions: the function LoftedSolid(), which performs body construction by planar sections **planeContours** on the planes **places** according to the given parameters **params**, and the function BooleanResult(), which performs the Boolean operation **oType** of the body **srcSolid** with the body constructed in the previous step. The function LoftedSolid() is described in the section *Constructing a Body on the Basis*

of Sections, the function BooleanResult() is described in the section *Boolean Operations on Bodies*.

The parameter **copyMode** controls the transfer of faces, edges and vertices from the source body **srcSolid** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy have no shared data).

A detailed description of the enumeration MbeCopyMode is provided in the section *Copying a Set of Faces MbFaceShell*.

The parameter **oType** determines the type of Boolean operation and can take three values: *bo_Union*, *bo_Intersect*, *bo_Difference*.

When **oType** = *bo_Union*, the function performs a union of the body **srcSolid** and the body constructed by sections.

When **oType** = *bo_Intersect*, the function performs an intersection of the body **srcSolid** and the body constructed by sections.

When **oType** = *bo_Difference*, the function subtracts the body constructed by sections from the body **srcSolid**.

The parameters **operName** and **contourNames** provide naming for the faces, edges, and vertices of the constructed body.

When constructing a body by planar sections, the LoftedResult() function has the same capabilities as the LoftedSolid() function. These include:

- the body can be constructed as a non-closed or closed solid;
- the body can have various shapes at its ends;
- the body can completely fill closed curves or have a thin wall;
- the shape of the body can be controlled by a guide curve.

The following are some capabilities of the function related to Boolean operations.

Fig. 1 shows the body **srcSolid** and plane closed curves:

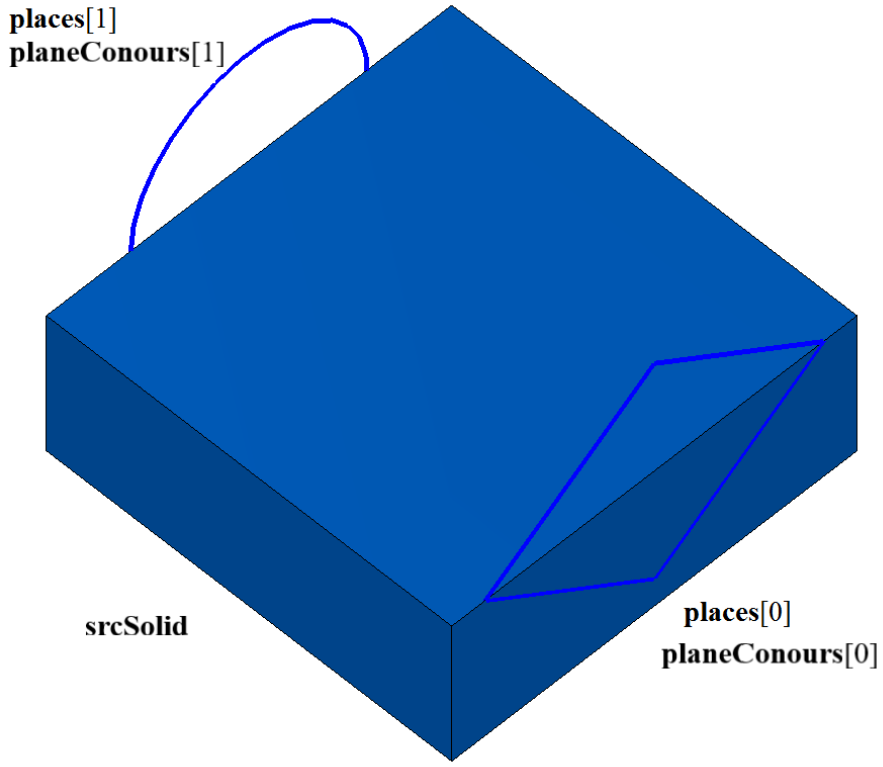


Fig. 1

Fig. 2 shows the result of the Boolean union operation of the body **srcSolid** and the body constructed by planar sections **planeConours** shown in Fig. 1:

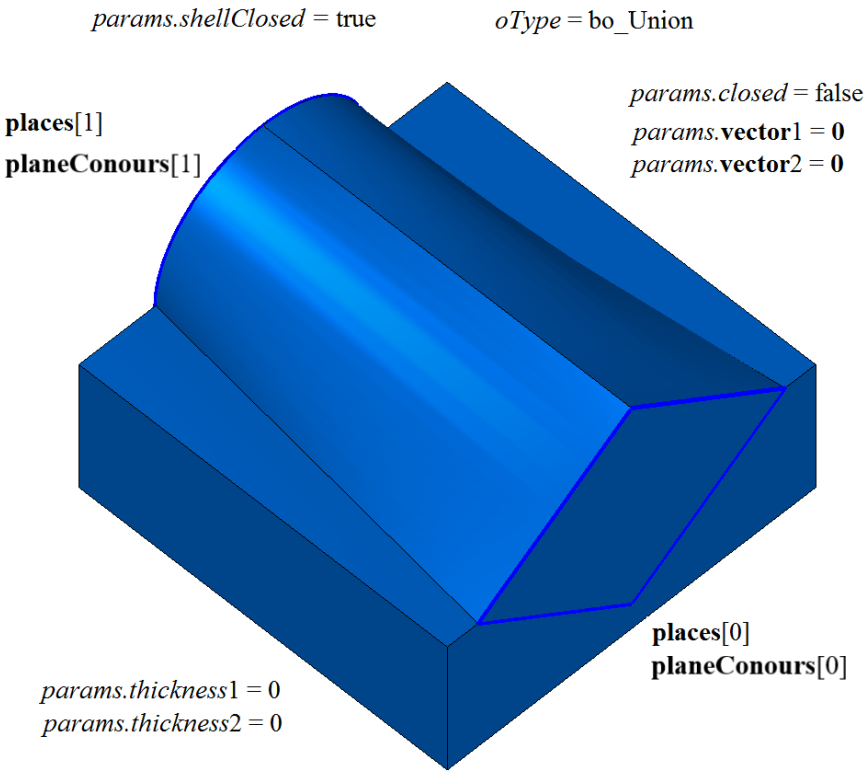


Fig. 2

Fig. 3 shows the result of the Boolean subtraction operation (subtracting from the body **srcSolid** the body constructed by planar sections **planeContours** shown in Fig. 1):

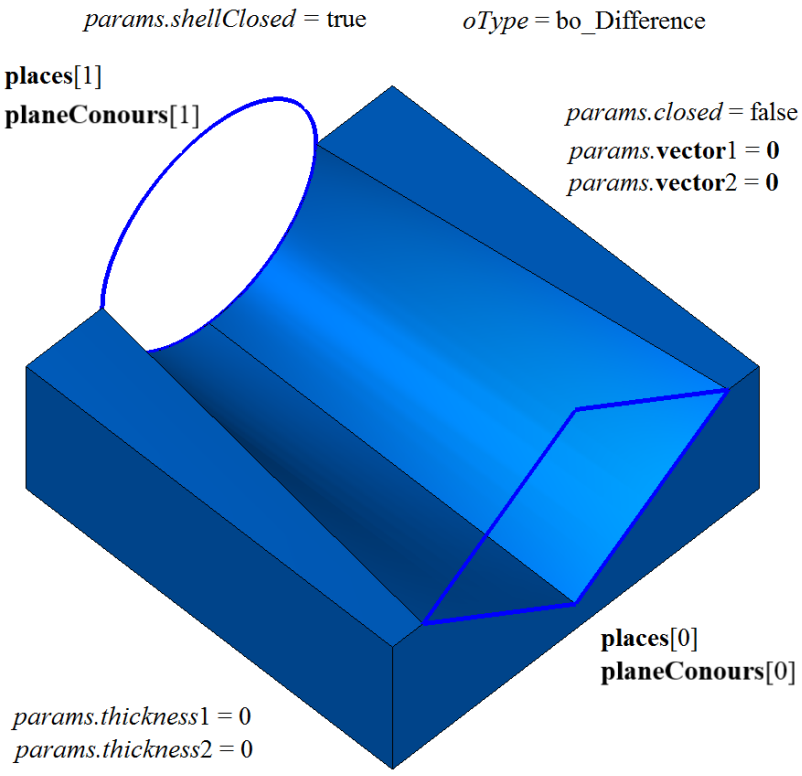


Fig. 3

Fig. 4 shows the result of the Boolean intersection operation of the body **srcSolid** and the body constructed by planar sections **planeContours** shown in Fig. 1:

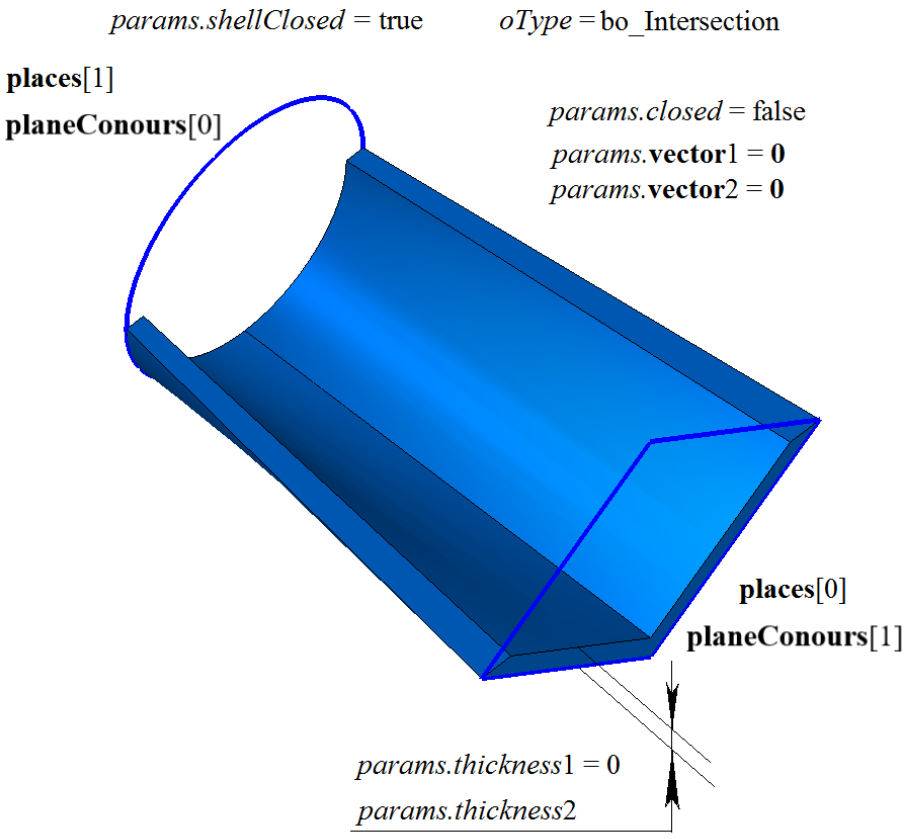


Fig. 4

The LoftedResult() function adds to the history tree of the body the creator MbCurveLoftedSolid, which contains all necessary data for performing the operation. The builder MbCurveLofted-Solid is declared in the file *cr_lofted_solid.h*.

In the test application *test.exe* the Boolean operation with a body constructed on the basis of planar sections is performed by the following menu commands:

“Create→Solid→As union with a solid→Lofting”,

“Create→Solid→As union with a solid→Lofting with a guide curve”,

“Create→Solid→As subtraction with a solid→Lofting”,

“Create→Solid→As subtraction with a solid→Lofting with a guide curve”,

“Create→Solid→As intersection with a solid→Lofting”,

“Create→Solid→As intersection with a solid→Lofting with a guide curve”.

3.2.6 Cutting a Body into Parts

The function

```
MbResultType SolidCutting( const c3d::SolidSPtr &        solid,
                                 MbeCopyMode             sameShell,
                           const MbShellCuttingParams &  cuttingParams,
                                 c3d::SolidsSPtrVector & results );
```

cuts off part of a body by an intersecting surface or shell.

The input parameters of the function are:

- **srcSolid** – an original body;
- **sameShell** – a copy mode of the original body;
- **cuttingParams** – parameters of the operation.

The output parameter of the function is the constructed bodies **results**.

The function returns an operation result code from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in the *action_solid.h* file.

To initialize objects **cuttingParams** with operation parameters, the following default constructor is defined in the MbShellCuttingParams class:

```
MbShellCuttingParams(       int                  part,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr )

: cutterData  (                     )
, booleanFlags(                     )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral          )
, prolongState(                     )
, progress    ( prog                )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

- **part** – a retained part of the original shell (+1, 0, -1), 0 – keep both parts;
- **mergingFlags** – flags for merging shell elements;
- **cutAsClosed** – a flag for building a closed shell;
- **snMaker** – an operation namer.

Cutting a Body by a Flat Contour

The function constructs a non-closed shell by extruding in direction **dir** a two-dimensional contour **contour**, located in the XY plane of the local coordinate system **place**, and performs a Boolean intersection operation of the original body **solid** with the non-closed shell. If the vector **dir** equals zero, the contour extrusion is performed along the vector **place.axisZ**.

For the operation to execute, the cutting contour must fully intersect the projection of the original body onto the XY plane of the local coordinate system **place** in the direction of the extrusion

vector. The extrusion length of the contour is calculated so that the non-closed shell completely intersects the original body.

To initialize objects **cuttingParams** of the function with operation parameters, the following contour constructors are defined in the MbShellCuttingParams class:

The constructor with the ability to save a part:

```
MbShellCuttingParams( const MbPlacement3D &      place,
                      const MbContour &          contour,
                            bool                 sameContour,
                      const MbVector3D &         dir,
                            int                  part,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr )

: cutterData  ( place, dir, contour, sameContour )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

The constructor without selecting a saved part (all parts are saved by default):

```
MbShellCuttingParams( const MbPlacement3D &      place,
                      const MbContour &          contour,
                            bool                 sameContour,
                      const MbVector3D &         dir,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr )

: cutterData  ( place, dir, contour, sameContour )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
}
```

where

- **place** – a local coordinate system in the XY plane of which the two-dimensional contour is located;

- **contour** – a two-dimensional contour for extrusion is located in the XY plane of the local coordinate system **place**;
- **sameContour** – the flag to use the original contour (true) or its copy (false);
- **dir** – a direction of contour extrusion;
- **part** – a retained part of the original body (+1, 0, -1), 0 – keep both parts;
- **mergingFlags** – flags for merging shell elements;
- **cutAsClosed** – the flag for building a closed shell;
- **snMaker** – the operation namer.

The parameter **part** determines the retained part of the original body **solid**:

- if *part* = +1, the part of the body located to the right of the contour is retained when viewing along the contour towards the **place.axisZ** axis;
- if *part* = -1, the part of the body located to the left of the contour is retained when viewing along the contour towards the **place.axisZ** axis;
- if *part* = 0, both parts of the body are retained.

The parameter **mergingFlags** controls the merging of similar faces and edges of the resulting shell.

The parameter **cutAsClosed** indicates how the original body **solid** should be treated: as closed or non-closed. When **cutAsClosed** = *true*, the operation is performed on the set of points located inside and on the surface of the body. When **cutAsClosed** = *false*, the operation is performed on the set of points located on the body surface.

The parameter **snMaker** provides naming of faces, edges and vertices of the constructed body.

Fig. 1 shows the original body **solid**, the cutting contour **contour** and the XY plane of the local coordinate system **place**.

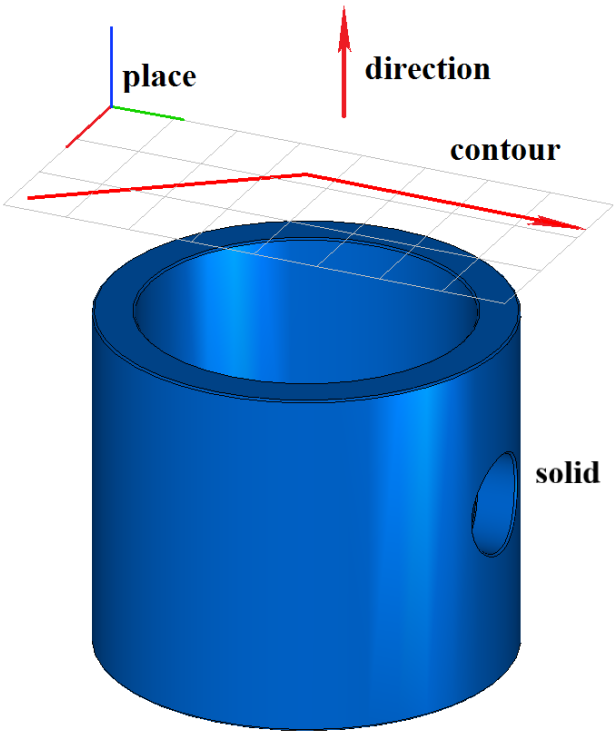


Fig. 1

Fig. 2 shows the constructed body **result** when *part* = +1 and *cutAsClosed* = *true*. Fig. 3 shows the constructed body **result** when *part* = -1 and *cutAsClosed* = *true*:

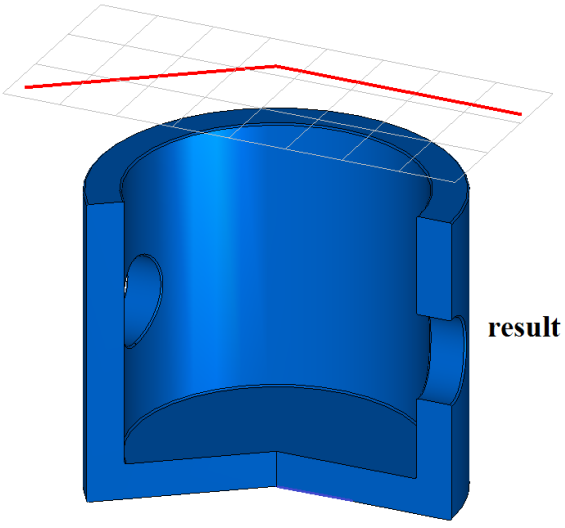


Fig. 2

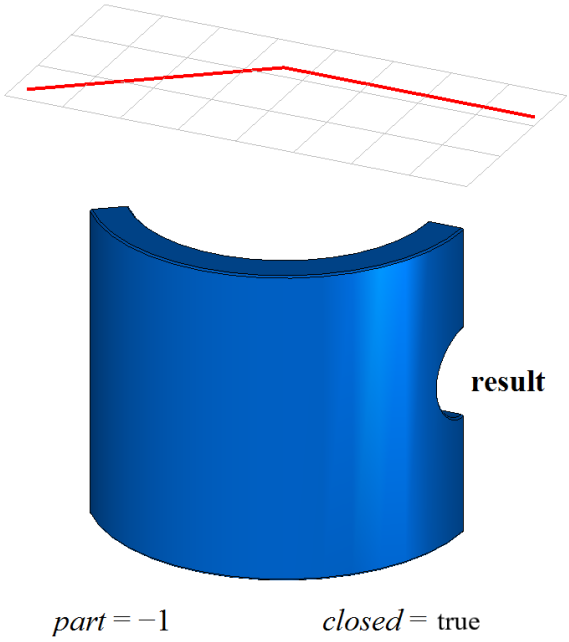


Fig. 3

Fig. 4 shows the constructed body **result** when *part* = +1 and *cutAsClosed* = *false*:

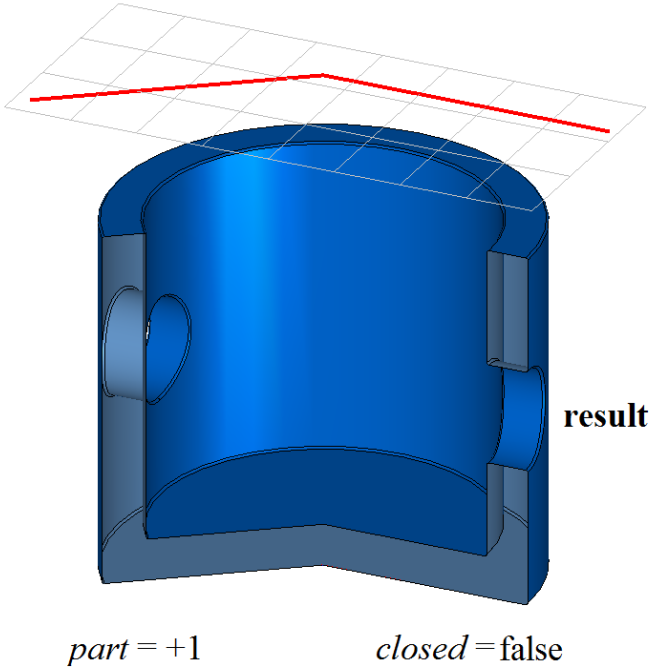


Fig. 4

When *part* = 0, the function constructs the body **result**, into which all parts of the original body

solid obtained after cutting are included.

To divide the body **result** into parts, use the functions DetachParts() or CreateParts(), described in section Divide a Body into Disconnected Parts.

When using the constructor by contour:

```
MbShellCuttingParams( const MbPlacement3D &       place,
                      const MbContour &           contour,
                            bool                  sameContour,
                      const MbVector3D &          dir,
                      const MbMergingFlags &      mergingFlags,
                            bool                  cutAsClosed,
                      const MbSNameMaker &        snMaker,
                            IProgressIndicator * prog = nullptr );
```

the function SolidCutting() cuts the body by a surface obtained by extruding a flat contour and constructs separate bodies from all parts. Fig. 5 shows the body **result** constructed in this way when *cutAsClosed* = *true*:

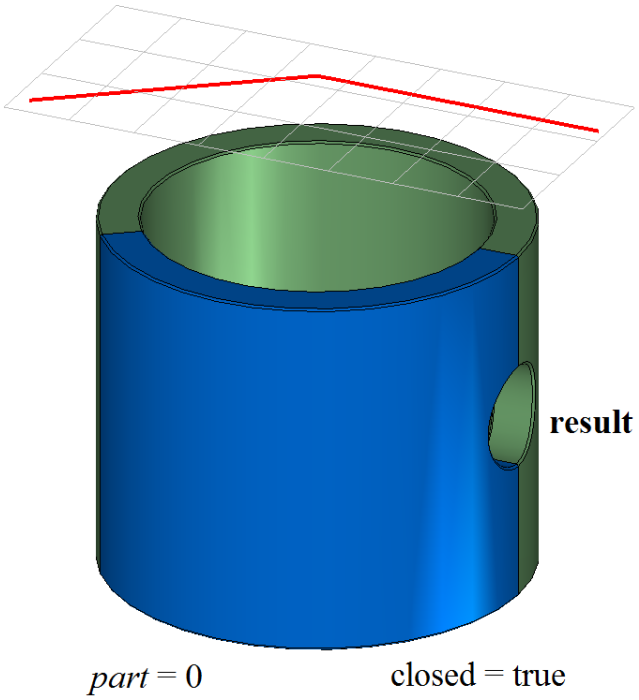


Fig. 5

The function SolidCutting() adds to the history tree of the constructed body the creator MbCuttingSolid, which contains all necessary data for performing the operation. The creator MbCuttingSolid is declared in the file *cr_cutting_solid.h*.

In the test application *test.exe*, cutting a body by a flat contour is performed using the menu commands “Create→Solid→Based on a solid→Cut by a 2D-Curve” and “Create→Shell→Based on a shell→Cut by a curve”.

Cutting a Body by a Surface

The function constructs a non-closed shell with one face based on the cutting surface **surface** and performs a Boolean intersection operation of the original body **solid** with the non-closed shell. For the operation to execute, the cutting surface must fully intersect the original body.

To initialize objects **cuttingParams** of the function with operation parameters, the following surface constructors are defined in the MbShellCuttingParams class:

```
MbShellCuttingParams( const MbSurface &          surface,
                      bool                       sameSurface,
                      int                        part,
                    const MbMergingFlags &       mergingFlags,
                      bool                       cutAsClosed,
                    const MbSNameMaker &         snMaker,
                      IProgressIndicator * prog = nullptr )

: cutterData  ( surface, sameSurface )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

```
MbShellCuttingParams( const MbSurface &          surface,
                      bool                       sameSurface,
                    const MbMergingFlags &       mergingFlags,
                      bool                       cutAsClosed,
                    const MbSNameMaker &         snMaker,
                      IProgressIndicator * prog = nullptr )

: cutterData  ( surface, sameSurface )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
}
```

- **surface** – a cutting surface;
- **sameSurface** – the flag to use the original surface (true) or its copy (false);
- **part** – a retained part of the original body (+1, 0, -1), 0 – keep both parts;
- **mergingFlags** – flags for merging shell elements;
- **cutAsClosed** – the flag for building a closed shell;

- **snMaker** – the operation namer.

The parameter **part** determines the retained part of the original body **solid**:

- if **part** = +1, the part of the body located above the surface is retained (on the side towards which the surface normal is directed);
- if **part** = -1, the part of the body located below the surface is retained;
- if **part** = 0, both parts of the body are retained.

The parameter **cutAsClosed** indicates how the original body **solid** should be treated: as closed or non-closed:

- if **cutAsClosed** = *true*, the operation is performed on the set of points located inside and on the surface of the body;
- if **cutAsClosed** = *false*, the operation is performed on the set of points located on the body surface.

The parameter **snMaker** provides naming of faces, edges and vertices of the constructed body.

Fig. 6 shows the original body **solid** and the cutting surface **surface**.

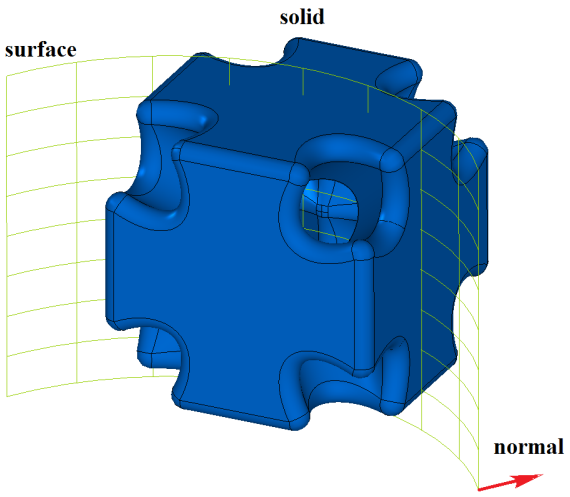


Fig. 6

Fig. 7 shows the constructed body **result** when *part* = +1 and *cutAsClosed* = true. Fig. 8 shows the constructed body **result** when *part* = -1 and *cutAsClosed* = true.

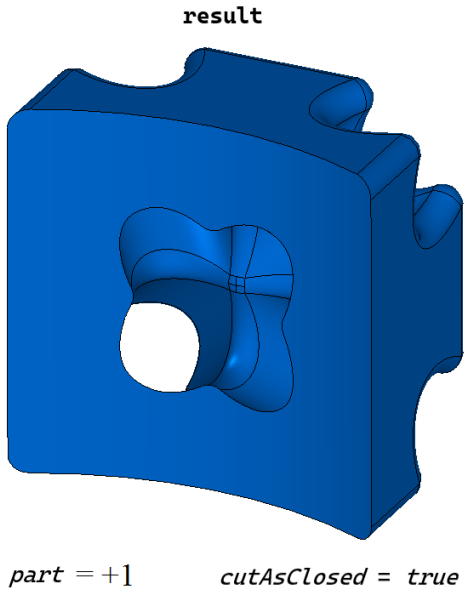


Fig. 7

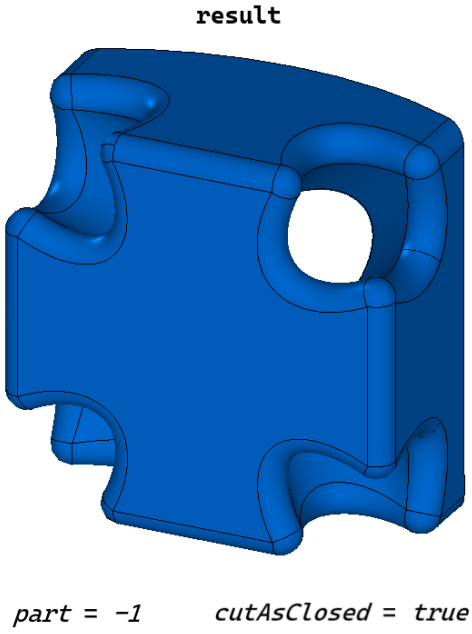


Fig. 8

Fig. 9 shows the constructed body **result** when *part* = +1 and *cutAsClosed* = false:

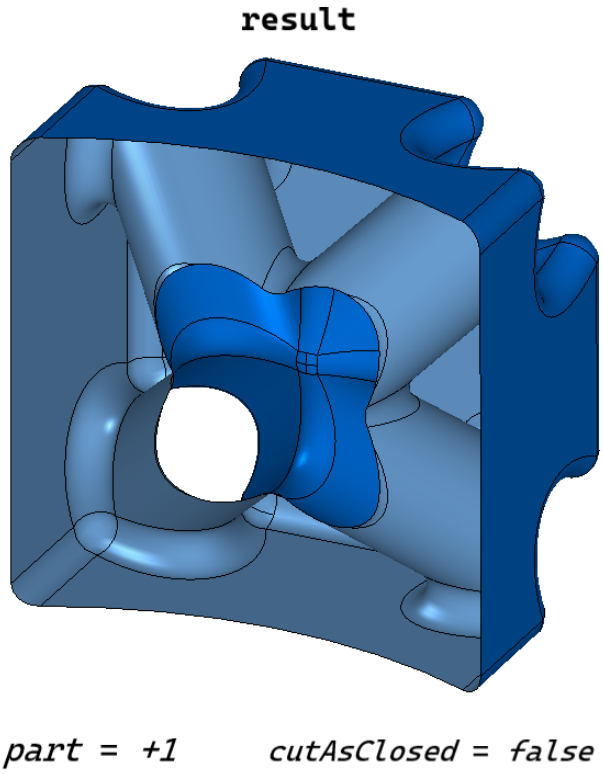


Fig. 9

When **part** = 0, the function constructs the body **result**, into which all parts of the original body **solid** obtained after cutting are included.

To divide the body **result** into parts, use the functions DetachParts() or CreateParts(), described in section Divide a Body into Disconnected Parts.

When using the surface constructor

```
MbShellCuttingParams( const MbSurface &          surface,
                      bool                       sameSurface,
                      const MbMergingFlags &     mergingFlags,
                      bool                       cutAsClosed,
                      const MbSNameMaker &       snMaker,
                      IProgressIndicator * prog = nullptr )
```

the function cuts the body by an intersecting surface and constructs separate bodies from all parts. Fig. 10 shows the body **result** constructed in this way when *cutAsClosed* = *true*:

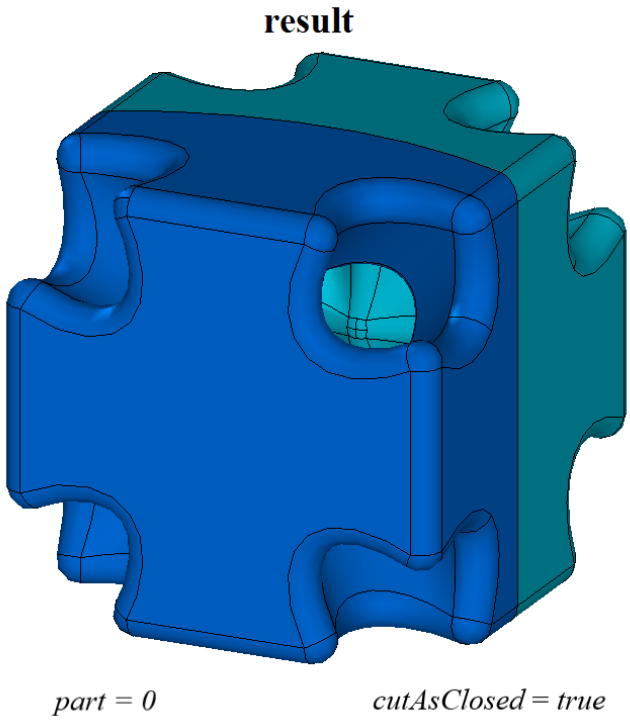


Fig. 10

In the test application *test.exe*, cutting a body by a surface is performed using the menu commands “Create→Solid→Based on a body→Cut by a surface” and “Create→Shell→Based on a shell→Cut by a surface”.

Cutting a Body by a Shell

The function performs cutting of the original body **solid** by an intersecting shell **cutSolid**. For the operation to execute, the cutting shell **cutSolid** must fully intersect the original body **solid**.

To initialize objects **cuttingParams** of the function with operation parameters, the following shell constructor is defined in the MbShellCuttingParams class:

```
MbShellCuttingParams( const MbSolid &            solid,
                      bool                       sameSolid,
                      int                        part,
                    const MbMergingFlags &       mergingFlags,
                      bool                       cutAsClosed,
                    const MbSNameMaker &         snMaker,
                      IProgressIndicator * prog = nullptr )

: cutterData  ( solid, sameSolid, true )
, booleanFlags(                         )
, nameMaker   ( &snMaker.Duplicate()    )
, retainedPart( ts_neutral              )
, prolongState(                         )
, progress    ( prog                    )
{
```

(continues on next page)

(continued from previous page)

```
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

- **solid** – a cutting shell;
- **sameSolid** – the flag to use the original body (true) or its copy (false);
- **part** – a retained part of the original body (+1, 0, -1), 0 – keep both parts;
- **mergingFlags** – flags for merging shell elements;
- **cutAsClosed** – the flag for building a closed shell;
- **snMaker** – the namer of the operation.

The parameter **part** determines the retained part of the original body **solid**:

- if **part** = +1, the part of the body located outside the shell is retained;
- if **part** = -1, the part of the body located inside the shell is retained;
- if **part** = 0, both parts of the body are retained.

The parameter **cutAsClosed** indicates how the original body **solid** should be treated: as closed or non-closed:

- if **cutAsClosed** = *true*, the operation is performed on the set of points located inside and on the surface of the body;
- if **cutAsClosed** = *false*, the operation is performed on the set of points located on the body surface.

The parameter **snMaker** provides naming of faces, edges and vertices of the constructed body.

Fig. 11 shows the original body **solid** and the cutting shell **cutSolid**.

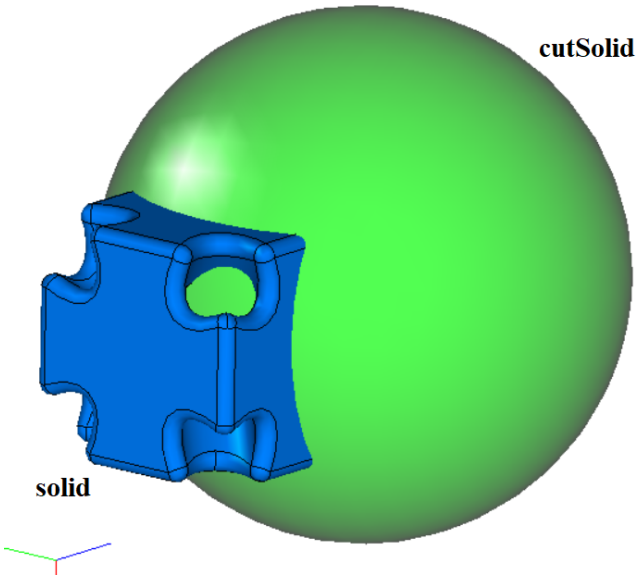


Fig. 11

Fig. 12 shows the constructed body **result** when *part* = +1 and *cutAsClosed* = *true*. Fig. 13 shows the constructed body **result** when *part* = -1 and *cutAsClosed* = *true*:

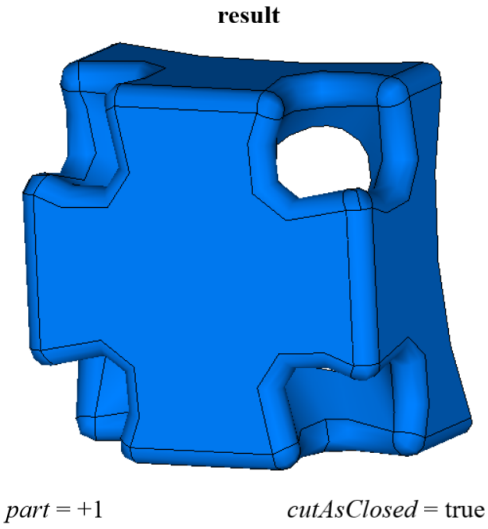


Fig. 12

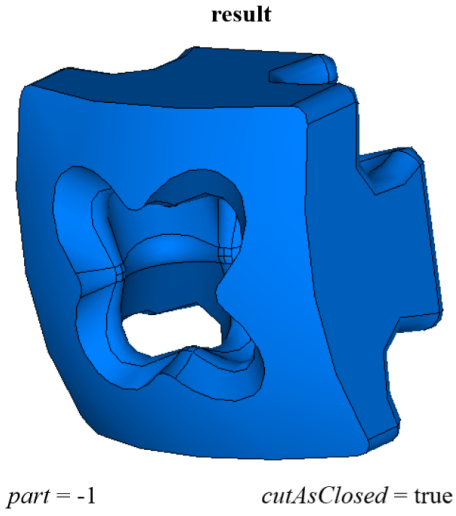


Fig. 13

When **part** = 0, the function constructs the body **result**, into which all parts of the original body **solid** obtained after cutting are included (Fig. 14):

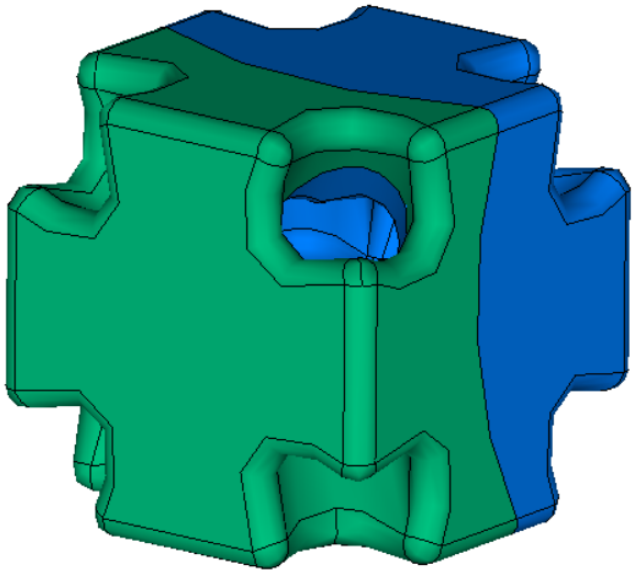


Fig. 14

In the test application *test.exe*, cutting a body by a shell is performed using the menu commands “Create→Solid→Based on a solid→Cut by a shell” and “Create→Shell→Based on a shell→Cut by a shell”.

3.2.7 Constructing a Symmetrical Body

The function

```
MbResultType SymmetrySolid(       MbSolid &               solid,
                                  MbeCopyMode             sameShell,
                            const MbSymmetrySolidParams & symParams,
                                  c3d::SolidSPtr &        result );
```

constructs a symmetrical body with a given symmetry plane.

The input parameters of the function are:

- **solid** – the original body;
- **sameShell** – the copying mode of the original body;
- **symParams** – the parameters of the operation.

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

The function returns an operation result code from the MbResultType enumeration.

The function is declared in the *action_solid.h* file.

The parameter **sameShell** can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the original shell and its copy have common base surfaces, vertices and faces unchanged by the operation);

- *cm_KeepSurface* (the original shell and its copy have common base surfaces);
- *cm_Copy* (the original shell and its copy have no common data).

The enumeration MbeCopyMode is described in section *Copying a Set of Faces MbFaceShell*.

In the MbSymmetrySolidParams class, the following constructor is defined to initialize the object **symParams** (see *op_shell_parameter.h*):

```
MbSymmetrySolidParams( const MbPlacement3D & place,
                       const MbSNameMaker &  operNames,
                             ThreeStates     side );
```

where

- **place** – the local coordinate system;
- **side** – the retained part of the original body relative to the symmetry plane (+1 for the part above the plane, -1 for the part below the plane);
- **operNames** – the namer for faces, edges and vertices of the constructed body.

If the parameter **symParams**.*side* = 1, the function creates a symmetrical body with respect to the specified symmetry plane as follows: the source body is cut by the XY-plane of the local coordinate system, the part of the source body located in the positive direction of the Z-axis from this plane is taken, a mirror copy of the selected part of the source body is constructed and then united with the selected part.

If **symParams**.*side* = -1, the function unites the part of the source body located in the negative direction of the Z-axis from the XY-plane of the local coordinate system with the mirror copy of this part with respect to the symmetry plane.

The parameter **operNames** provides naming for faces, edges and vertices of the constructed body.

Fig. 1 shows the original body **solid** and the symmetry plane **place**:

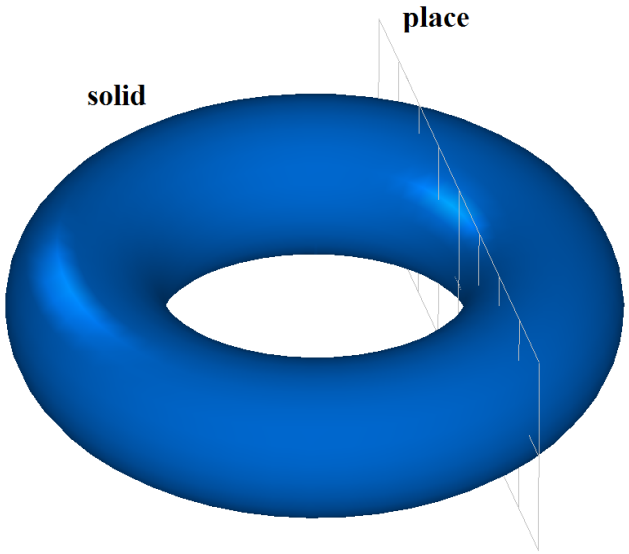


Fig. 1

Fig. 2 shows the constructed body **result**:

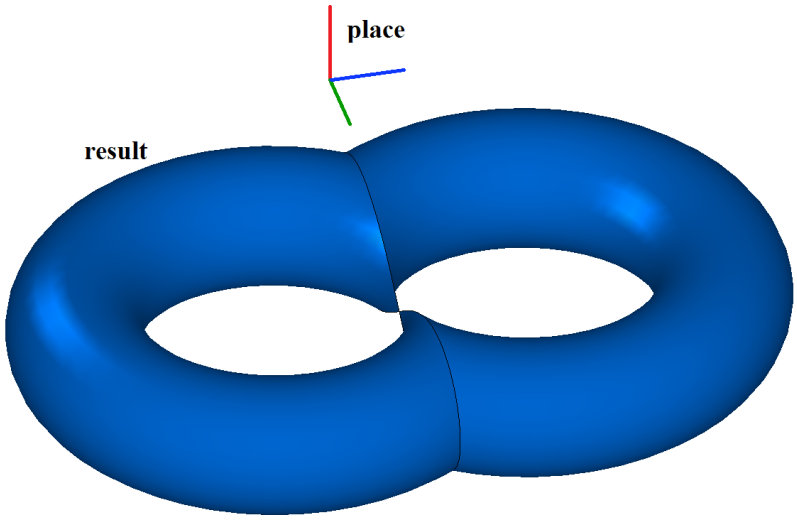


Fig. 2

Fig. 3 shows the constructed body **result** for a symmetry plane with opposite normal:

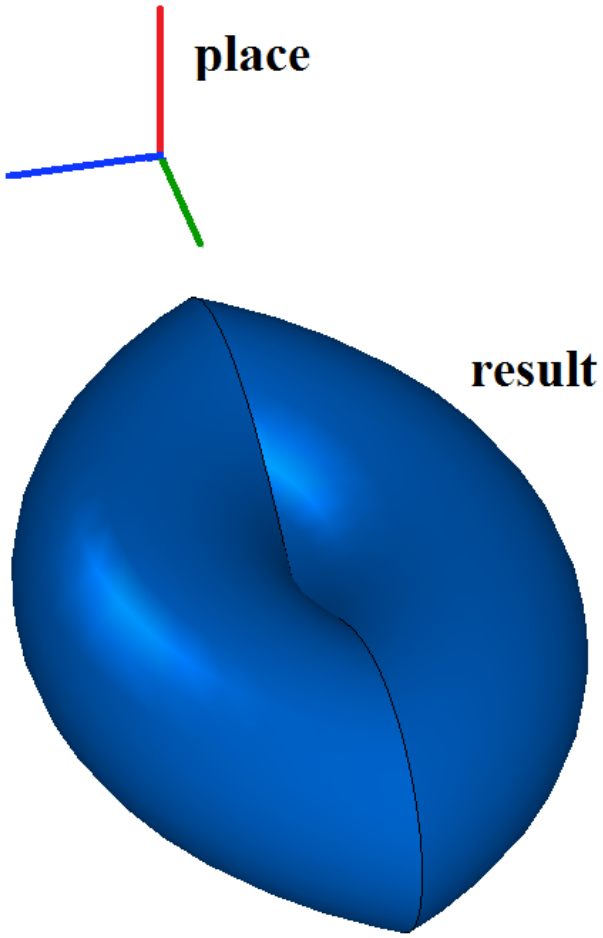


Fig. 3

Fig. 4 shows the original body **solid** and the symmetry plane **place**, which do not intersect:

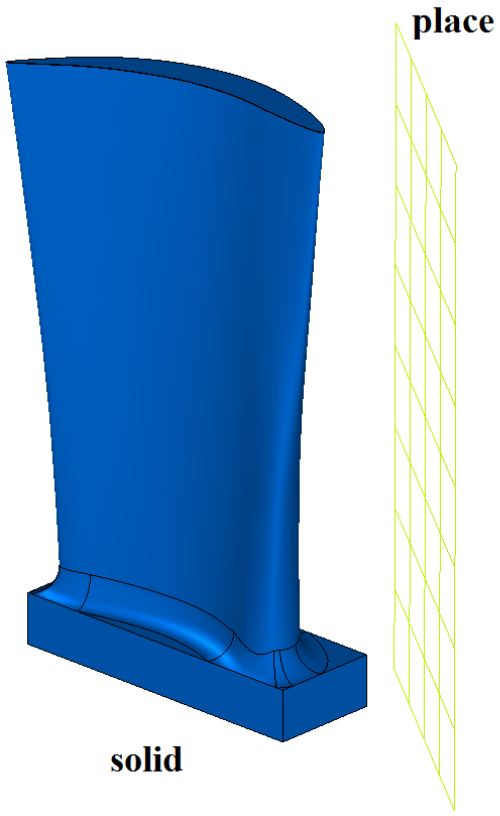


Fig. 4

Fig. 5 shows the original body **solid** and the constructed mirror body **result**:

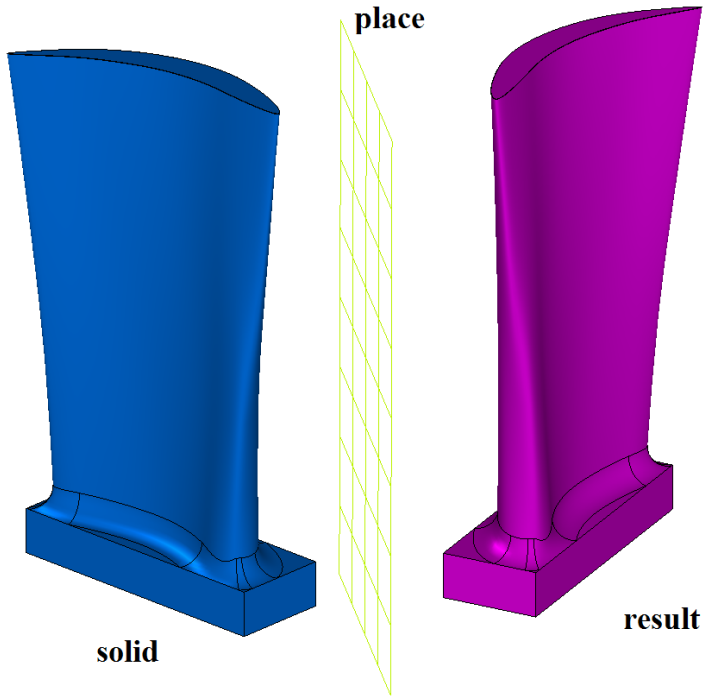


Fig. 5

The function SymmetrySolid() adds to the operation log of the constructed body the builder MbSymmetrySolid, which contains all necessary data for performing the operation. The builder MbSymmetrySolid is declared in the file *cr_symmetry_solid.h*.

In the test application *test.exe*, constructing a symmetrical body is performed using the menu command “Create→Solid→Based on a solid→Symmetric”.

3.2.8 Rounding Off Body Vertices and Edges

The function

```
MbResultType FilletSolid(       MbSolid &               solid,
                                MbeCopyMode             sameShell,
                          const MbShellFilletValues &   filletParams,
                                c3d::SolidSPtr &        result,
                                IProgressIndicator *    progInd = nullptr );
```

replaces the specified edges of a source body with faces that smoothly connect the adjacent faces of the specified edges. In a cross-section, the connecting faces can have the shape of a circle arc, an ellipse, a hyperbola, or a parabola.

Methods of rounding the body edges:

- Rounding the specified body edges with a constant radius;
- Rounding the vertices and the edges adjacent to them with a constant radius (three edges must meet at a vertex);
- Rounding the specified body edges with a variable radius defined by MbEdgeFunction.function.

The parameters of a cross-section can vary according to a given law. The method of constructing a fillet is defined by the set of parameters in the **filletParams** object.

The input parameters of the function are:

- **solid** – a source body whose edges are to be processed;
- **sameShell** – the mode of copying the source body;
- **filletParams** – the construction parameters.

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

The function returns a result code of the operation from the MbResultType enumeration.

The function is declared in the *action_solid.h* file.

Rounding Edges with a Constant Radius

When rounding edges with a constant radius, the following constructor is defined in the MbShellFilletValues class to initialize **filletParams** objects with the operation parameters:

```
MbShellFilletValues(       c3d::EdgesSPtrVector      initCurves,
                           c3d::FacesSPtrVector &    initBounds,
                    const  SmoothValues &            params,
                    const  MbSNameMaker &            names )
: MbShellChamferParams( initCurves, params, names )
, _initBounds( initBounds ) {}
```

where

- **initCurves** – a set of body edges to be rounded;
- **initBounds** – a set of faces for trimming the ends;
- **params** – an object that contains parameters of edge rounding;
- **names** – a name maker.

The fillet construction parameter *params* contains information about the shape and the method of conjugation of the adjacent faces of the processed edges, see Fig. 1. The SmoothValues class is described in the *op_shell_parameter.h* file.

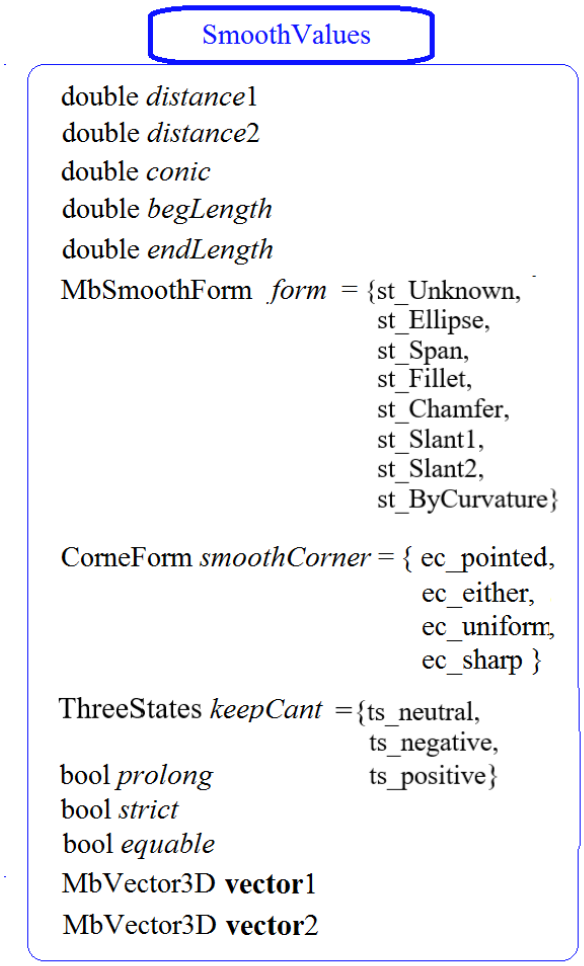


Fig. 1

- **distance1** – the curvature radius on the first surface;
- **distance2** – the curvature radius on the second surface;
- **conic** – the shape coefficient of the conjugation surface, takes the values 0, 0.05...0.95 (0 means an arc of a circle or an ellipse);
- **begLength** – the distance from the start of the fillet to the stop point (UNDEFINED_DBL means there is no stop);
- **endLength** – the distance from the end of the fillet to the stop point (UNDEFINED_DBL means there is no stop);
- **form** – the type of conjugation (fillet) from the MbeSmoothForm enumeration;
- **smoothCorner** – the method of processing corners where three edges meet;
- **prolong** – the flag of extending the fillet along tangent edges;
- **keepCant** – the parameter for automatic detection of preserving an edge (*ts_neutral*), preserving a face (*ts_negative*), or preserving an edge (*ts_positive*);
- **strict** – the construction strictness flag: if *false*, round at least what is possible;

- **equable** – the flag of inserting a toroidal surface in the corners of the junction of the conjugation surface (for sheet-metal stamping);
- **vector1** – the normal vector to the plane by which the fillet is cut off at the start of the chain;
- **vector2** – the normal vector to the plane by which the fillet is cut off at the end of the chain.

The fillet type is controlled by the **form** parameter. For edge rounding, the **form** parameter values *st_Ellipse*, *st_Fillet* and *st_Span* are used.

If **form** = *st_Fillet*, the function constructs a fillet surface with the given radii defined by the **distance1** and **distance2** parameters. Fig. 2 shows a fillet with given radii of an edge connecting two cylindrical surfaces:

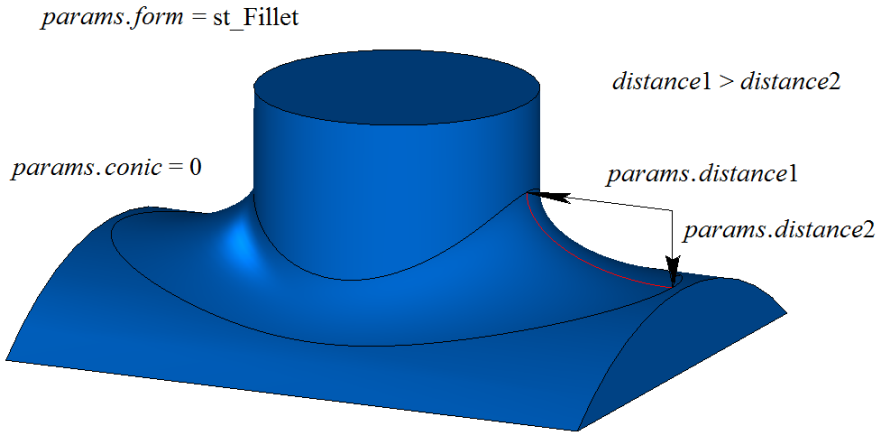


Fig. 2

If **distance1** = **distance2** and **conic** = 0, the fillet surface is constructed by moving a sphere that is tangent to the two faces adjacent to the edge being rounded. The support edges of the conjugation face pass through the points of tangency of the sphere and the corresponding adjacent face. A cross-section of the conjugation face is an arc of a circle. Fig. 3 shows a fillet with equal specified radii of an edge connecting two cylindrical surfaces:

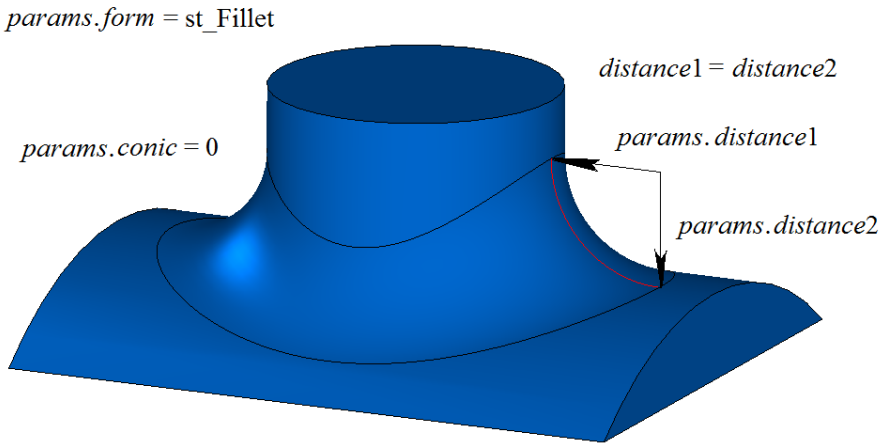


Fig. 3

The **conic** coefficient controls the shape of the fillet surface. If **conic** = 0 (the _ARC_ macro), a cross-section of the conjugation surface has the shape of an arc of a circle or an ellipse with the specified radii. In addition to zero, the shape coefficient can take values from 0.05 to 0.95.

If **conic** = 0.5, a cross-section of the fillet face is an arc of a parabola.

If **conic** < 0.5, a cross-section of the fillet face is an arc of an ellipse.

If **conic** > 0.5, a cross-section of the fillet face is an arc of a hyperbola.

Figs. 4, 5 show fillets with equal radii of an edge connecting two cylindrical surfaces and with different shape coefficients:

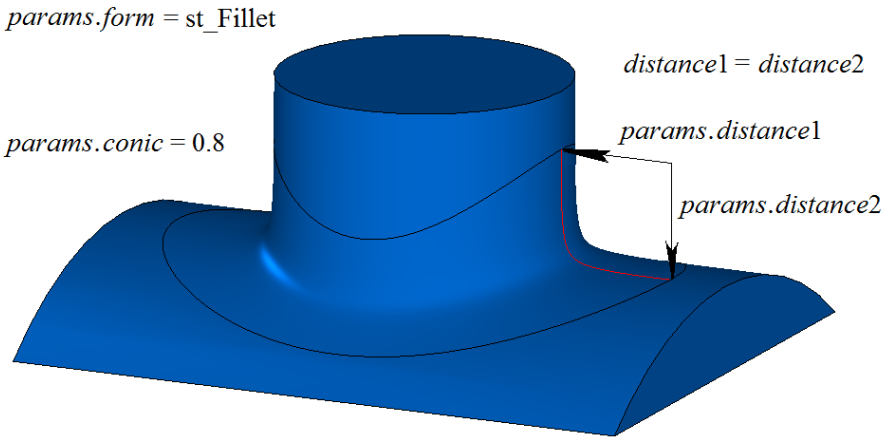


Fig. 4

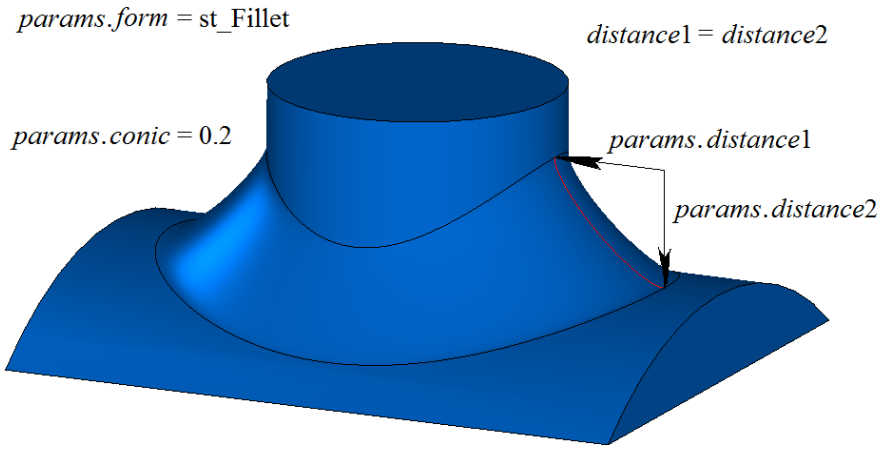


Fig. 5

If **form** = *st_Span*, the function constructs a fillet surface with a given chord. The **distance1** and **distance2** parameters are equal and define the distance between the support edges of the conjugation face. A cross-section of the fillet face is an arc of a circle. In the general case, in each cross-section of the fillet face the radius of the arc is different, and the **distance1** and **distance2** parameters are equal to the chord of the circular arc. Fig. 6 shows a fillet with a given chord of an edge connecting two cylindrical surfaces:

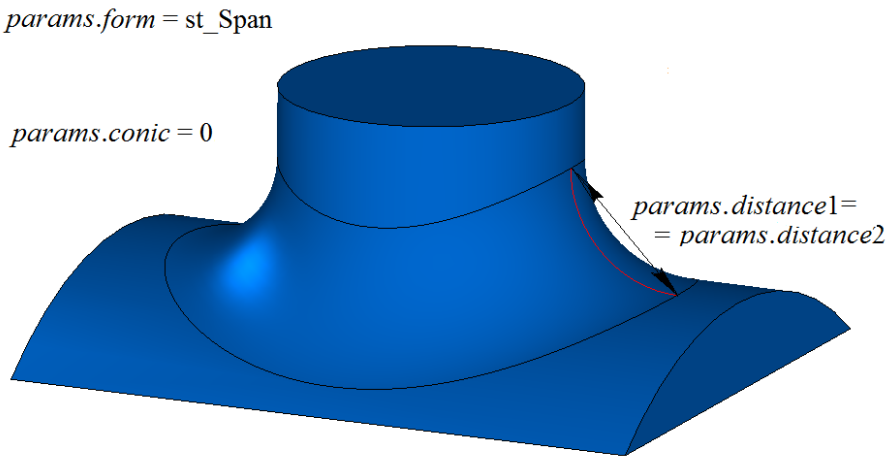


Fig. 6

Fig. 7 shows a fillet with a given chord of an edge connecting two cylindrical surfaces and a nonzero shape coefficient:

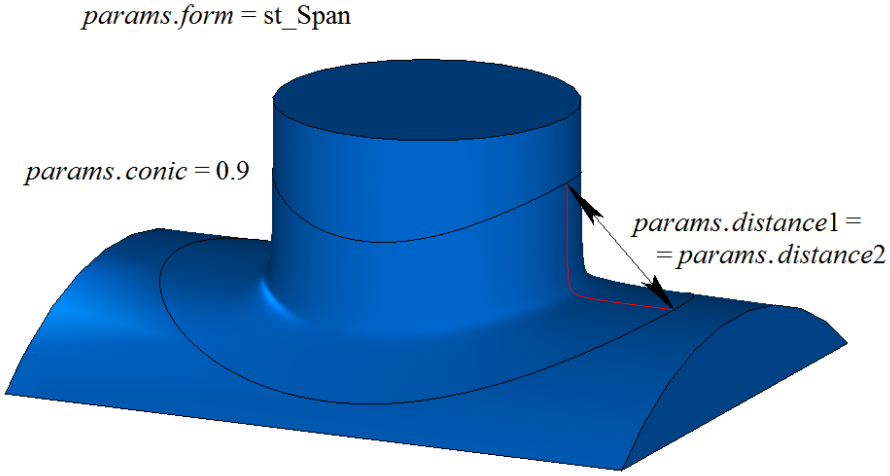


Fig. 7

Fig. 8 shows an example of stopping a fillet at the **begLength** distance from the initial vertex and at the **endLength** distance from the terminal vertex of the processed edge. If conjugation is not to be stopped, the **begLength** and **endLength** distances must be set to *UNDEFINED_DBL*. By default, the fillet is stopped by faces that are perpendicular to the edge being rounded. You can override the fillet stopping behavior by using the **vector1** vector as the normal of the stop face at the start of the conjugation and the **vector2** vector as the normal of the stop face at the end of the conjugation.

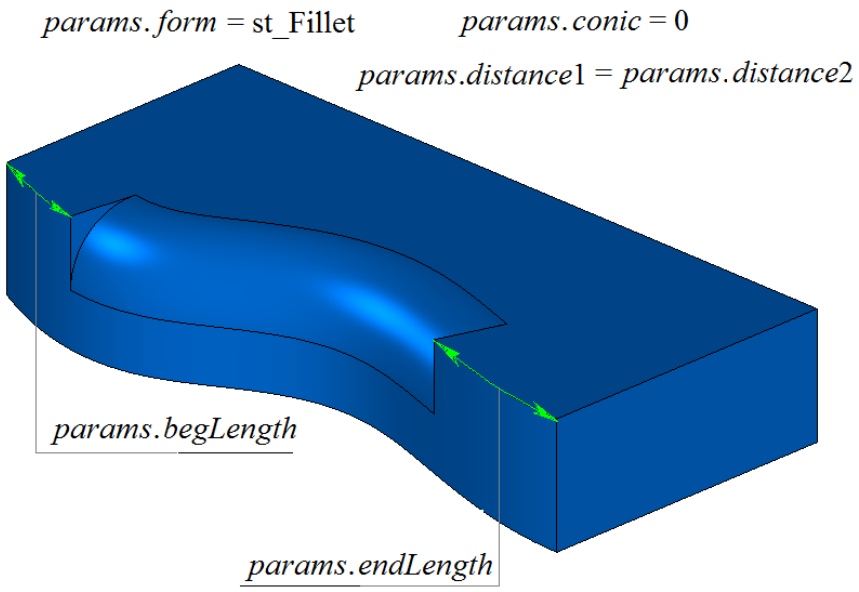


Fig. 8

Using the body in Fig. 9 as an example, we will demonstrate how the **prolong** and **keepCant** flags work when rounding the edge highlighted in the figure:

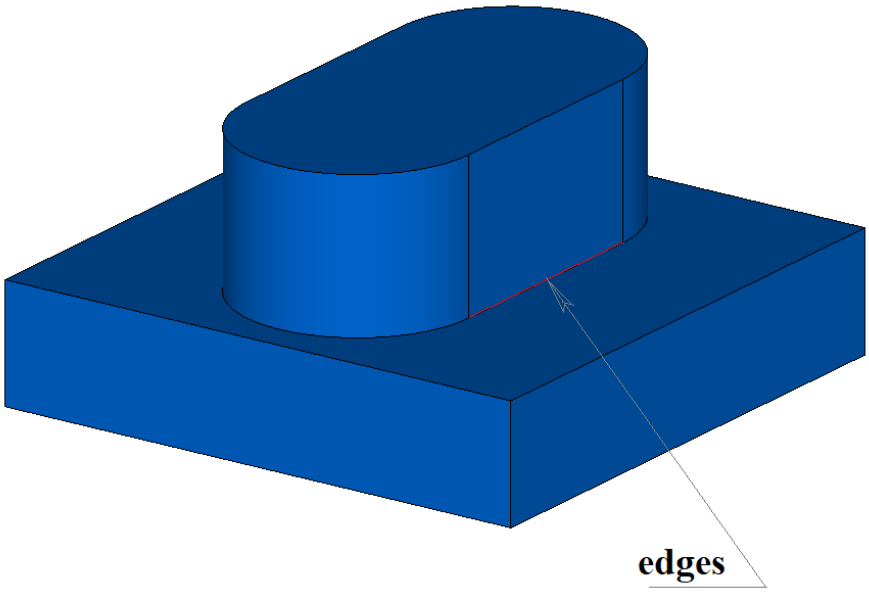


Fig. 9

The **prolong** flag specifies which body edges must be processed. If **prolong** = *false*, only the edges specified in the **initCurves** container are processed, see Fig. 10:

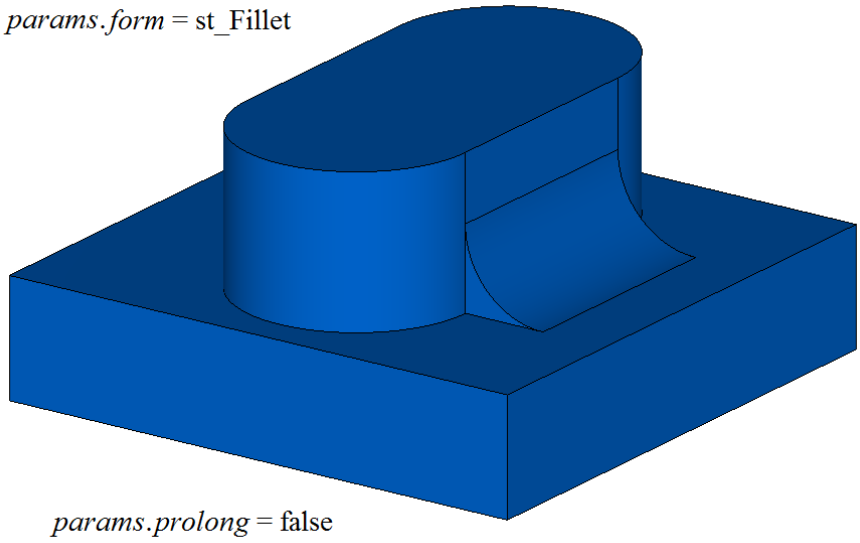


Fig. 10

If **prolong** = *true*, the edges specified in the **initCurves** container as well as the edges that are smoothly connected to them are processed, see Fig. 11:

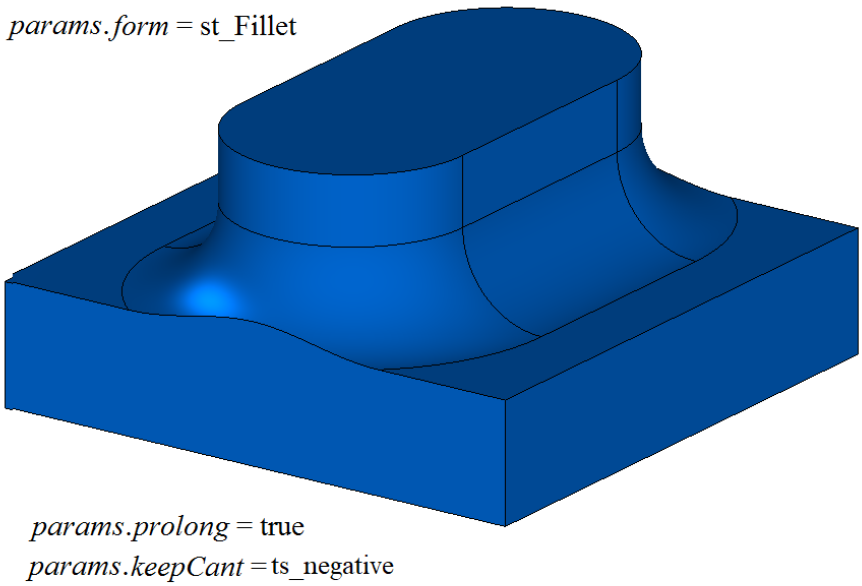


Fig. 11

The **keepCant** parameter controls the processing of cases where the support edges of the conjugation face go beyond the boundary of an adjacent face.

If **keepCant** = *ts_negative*, when the support edge of the conjugation face goes beyond the boundary of an adjacent face across a sharp edge, the conjugation face does not change its

shape but is “cut off” by the neighboring face, as shown in Fig. 11.

If **keepCant** = *ts_positive*, when the support edge of the conjugation face goes beyond the boundary of an adjacent face across a sharp edge, the conjugation face changes its shape and follows the boundary with its support edge, keeping the boundary unchanged, as shown in Fig. 12:

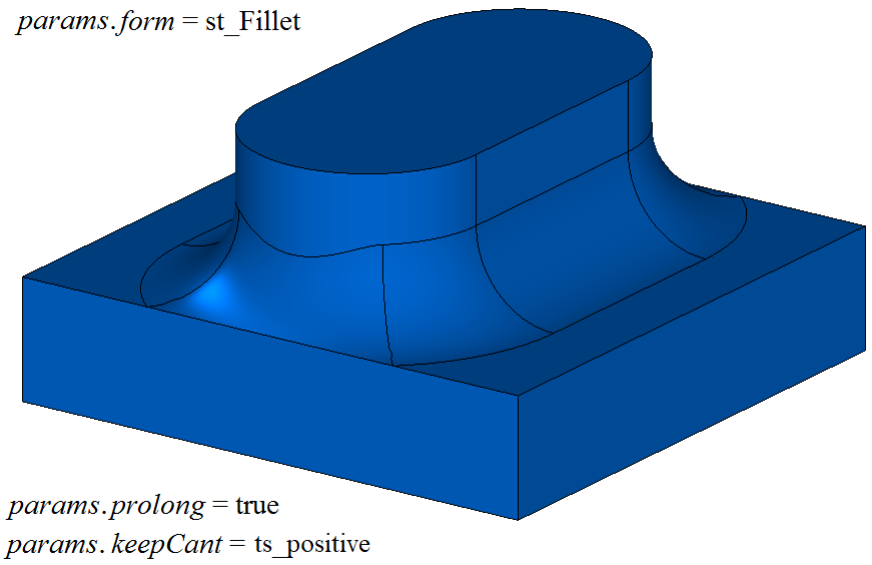


Fig. 12

If **keepCant** = *ts_neutral*, when the support edge of the conjugation face goes beyond the boundary of an adjacent face across a smooth edge, the conjugation face replaces the adjacent face with a neighboring one, changing its shape in this region, see Fig. 13:

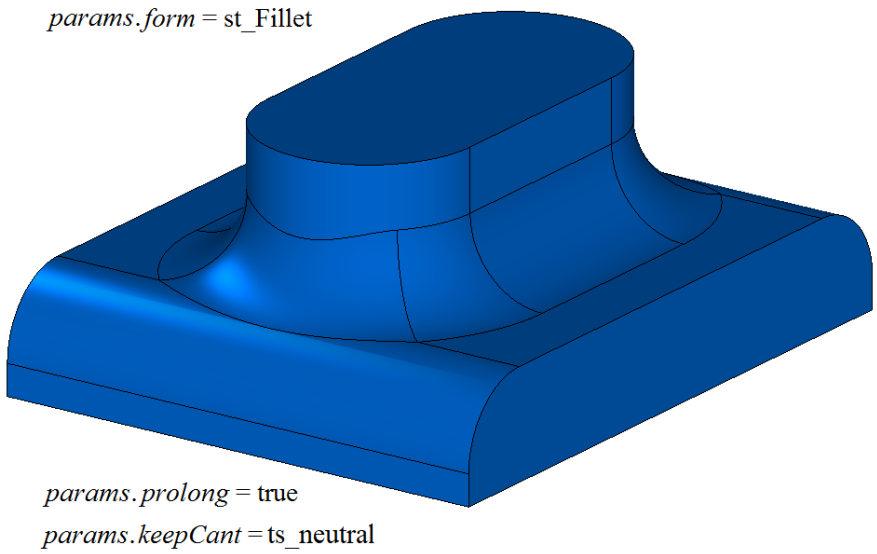


Fig. 13

Fig. 14 shows the filleting of four edges performed by a single function call with the **equable** flag set to *false* (no toroidal surface in the corners of the junction):

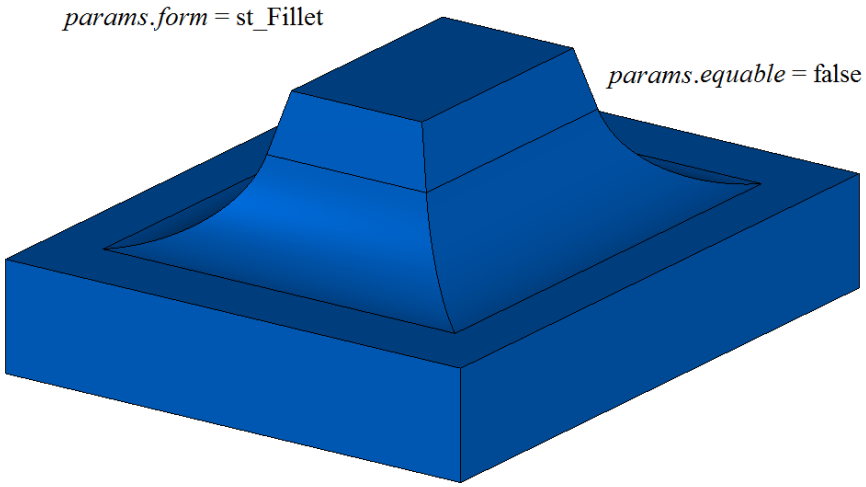


Fig. 14

Fig. 15 shows the filleting of four edges performed by a single function call with the **equable** flag set to *true*, which indicates that a toroidal surface must be inserted in the corners of the conjugation surfaces:

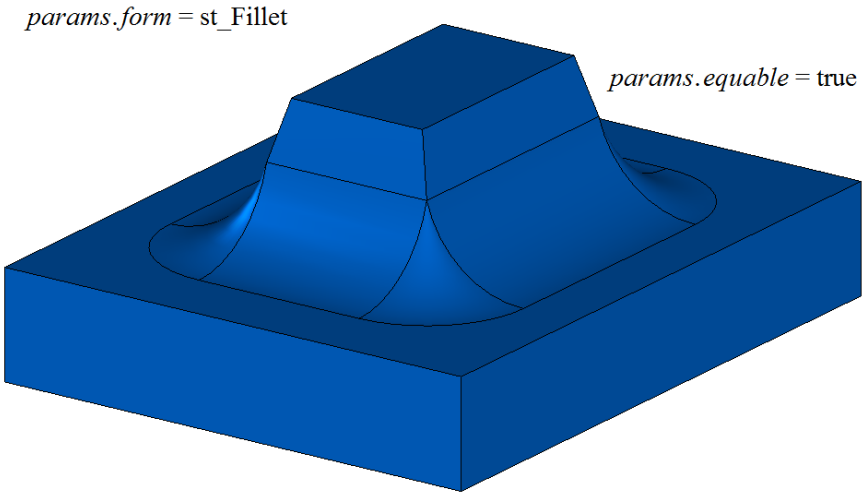


Fig. 15

When constructing a fillet of three edges that meet at one vertex, the **smoothCorner** parameter defines how suitcase-like corners are processed. If **smoothCorner** = *ec_pointed*, corners where three edges of the same convexity meet are not processed (see Fig. 16):

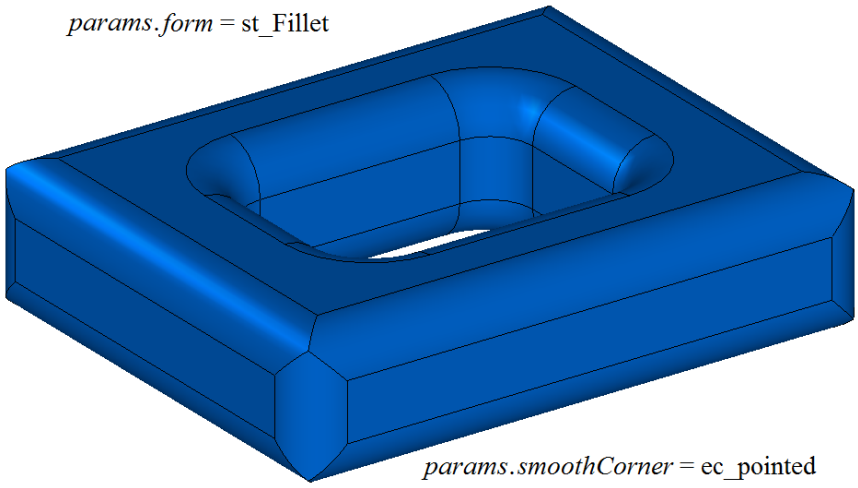


Fig. 16

If **smoothCorner** = *ec_uniform*, corners where three edges of different convexity meet are processed in the same way: if at a point two convex (concave) edges and one concave (convex) edge meet, the two convex (concave) edges are processed first (see Fig. 17):

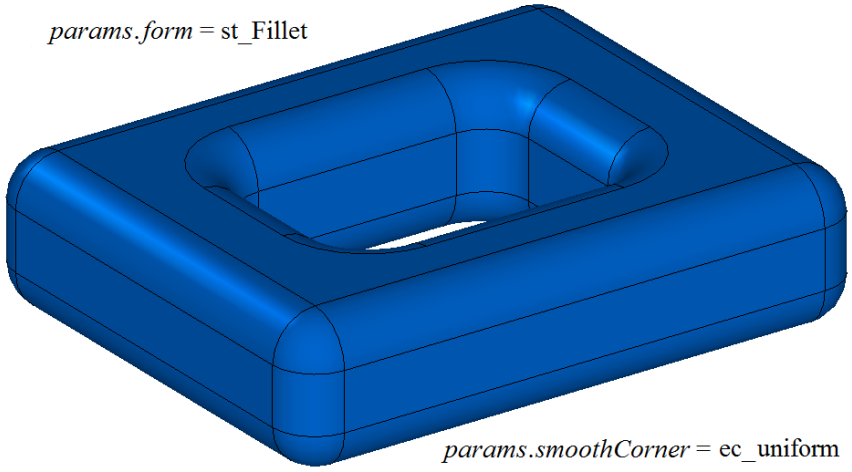


Fig. 17

If **smoothCorner** = *ec_sharp*, corners where three edges of different convexity meet are processed in the same way, as shown in Fig. 18. If at a point two convex (concave) edges and one concave (convex) edge meet, the convex (concave) edges are processed first.

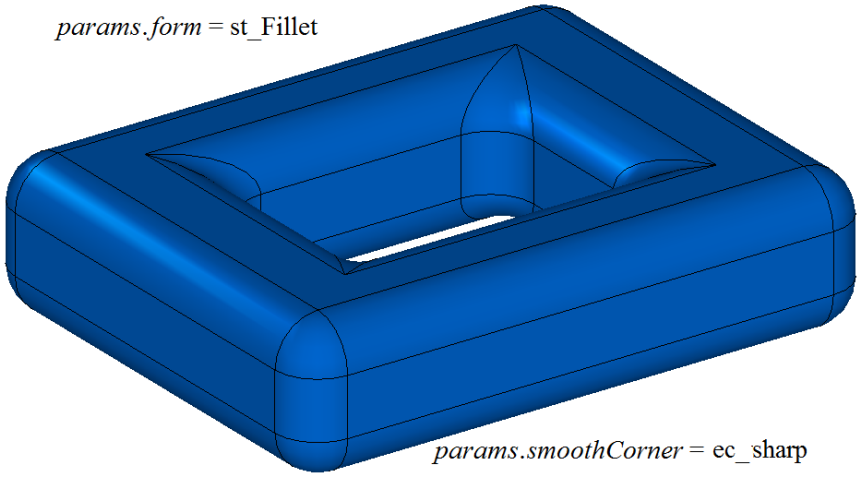


Fig. 18

If **smoothCorner** = *ec_either*, corners where three edges of different convexity meet can be processed in different ways: three edges meeting at one point are processed in the order of their internal numbering, regardless of whether they are convex or concave.

In an ambiguous situation at the ends of a conjugation face, the **initBounds** parameter with the **solid** body faces that must be used to trim the conjugation faces can be used. An example of using the **initBounds** parameter is shown in Figs. 19 and 20:

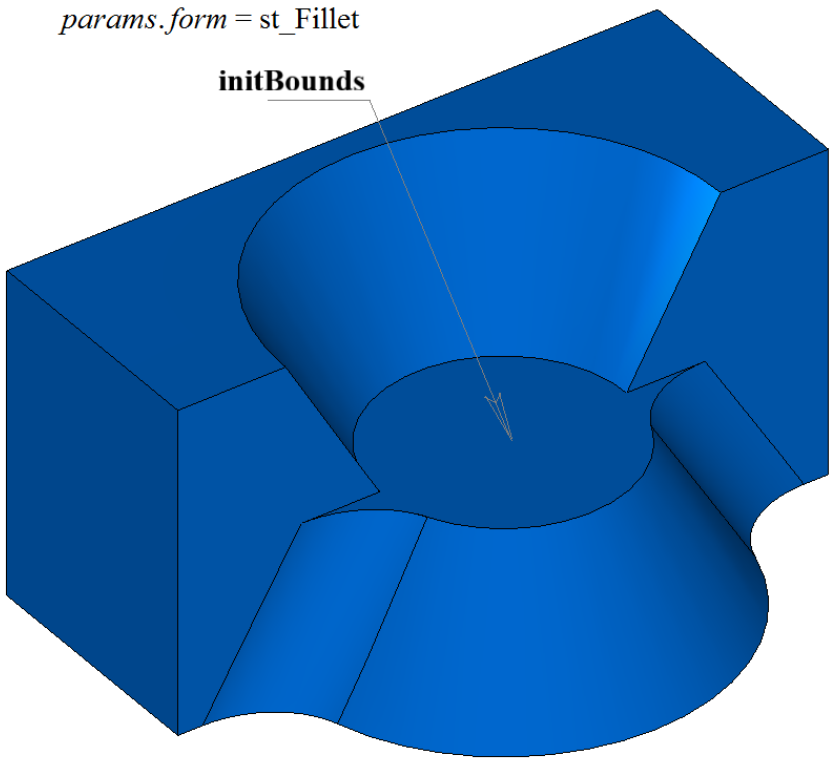


Fig. 19

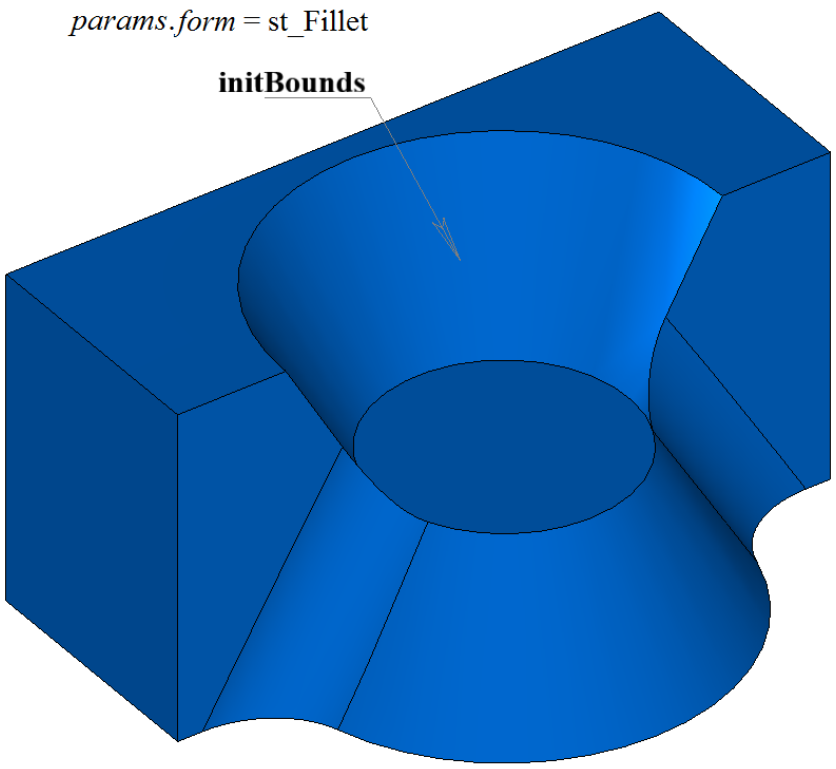


Fig. 20

The **initBounds** parameter can be used to stop conjugation faces at the start and at the end. In this case, the faces defined by the **initBounds** parameter must belong to the source body **solid**.

Fig. 21 shows a model for which it is required to construct fillets of the specified edges that completely embrace a hole and a protrusion:

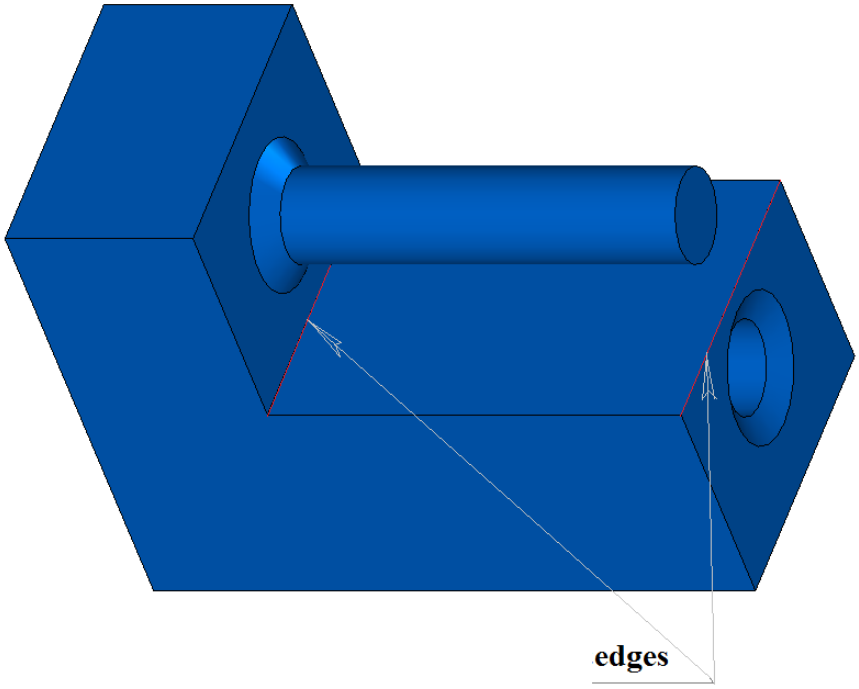


Fig. 21

The result of constructing a fillet that bypasses obstacles is shown in Fig. 22:

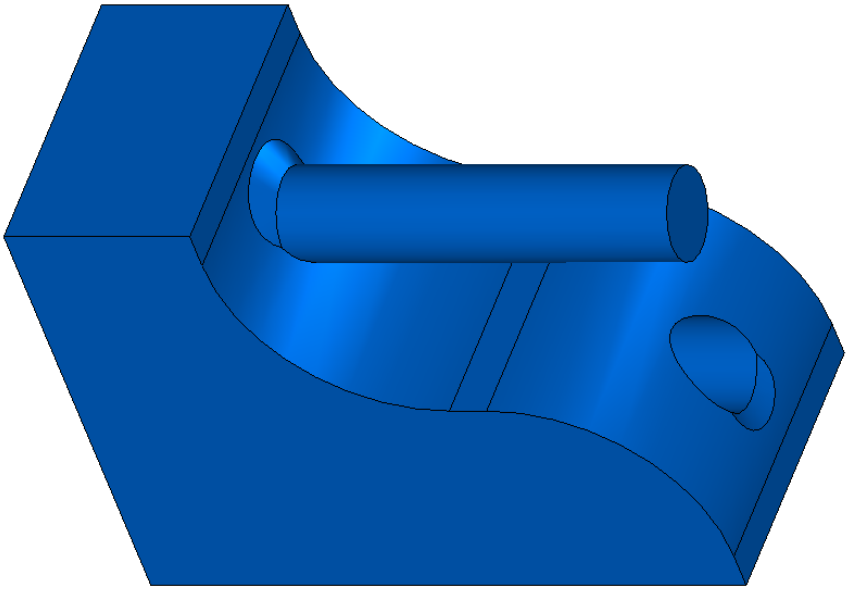


Fig. 22

Fig. 23 shows an example of simultaneously rounding a group of six edges that meet at a common vertex:

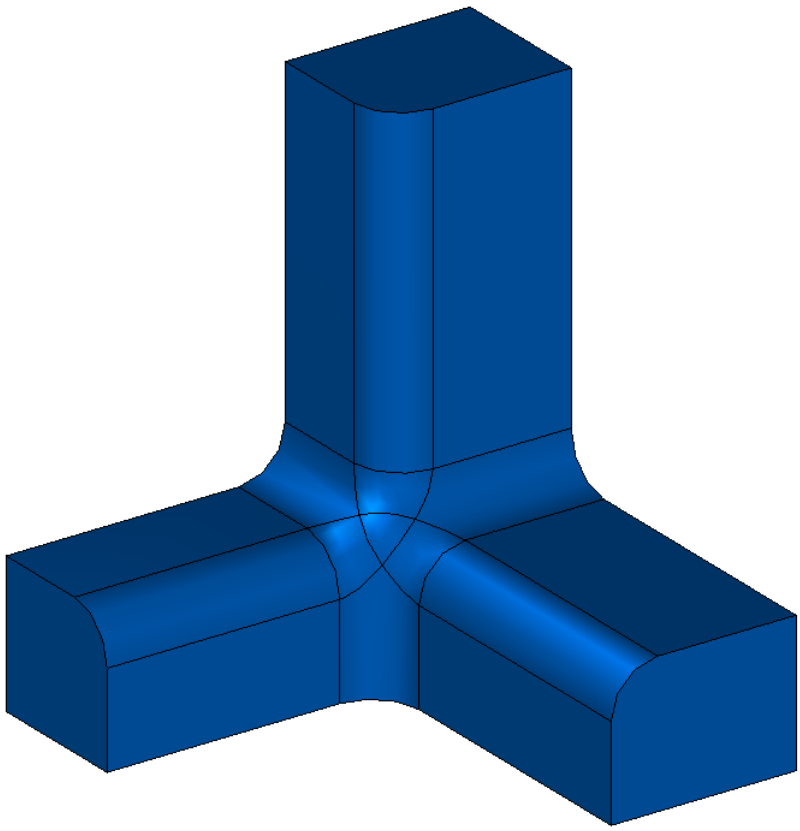


Fig. 23

Fig. 24 shows an example of rounding several groups of four edges that meet at common vertices. A specific feature of the fillet is that the groups are interconnected and can only be processed simultaneously. The source body for the body shown in the figure is obtained by subtracting four cylinders whose axes coincide with the cube diagonals from a cube.

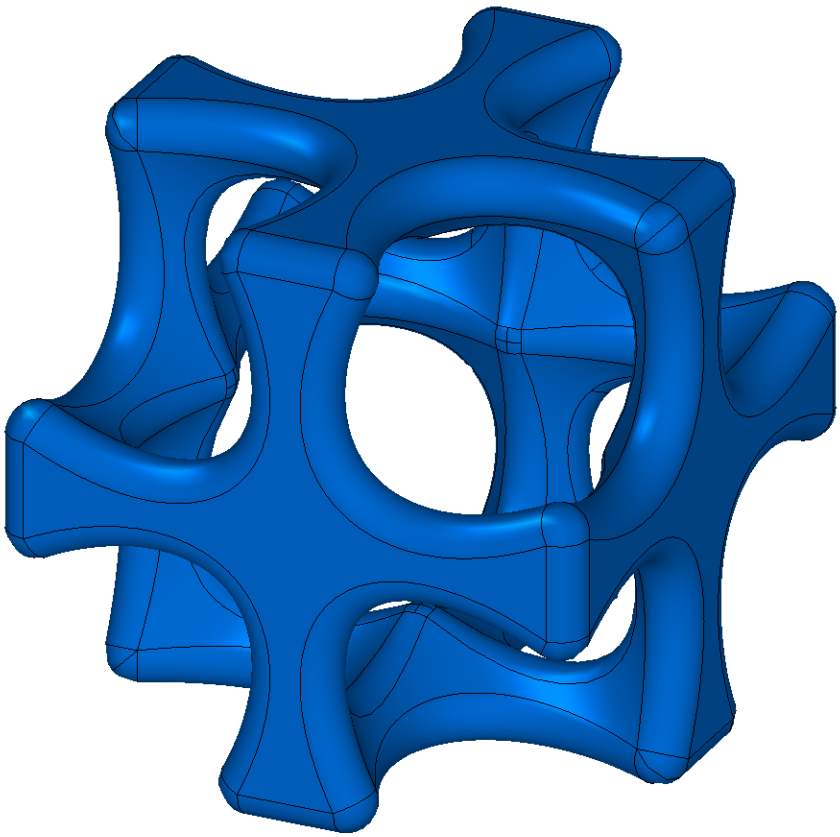


Fig. 24

When in the fillet construction methods the **distance1** and **distance2** parameters are not equal and several edges are specified for processing, it is hard to understand which parameter corresponds to which object. In such a situation, auxiliary methods are suggested: SmoothPhantom(), SmoothSequence(), and SmoothPositionData(), which are declared in the *action_phantom.h* file.

The SmoothPhantom() function constructs simplified surfaces to simulate future fillets.

The SmoothSequence() function constructs edge sequences, to which smoothly connected edges to be processed together can optionally be added.

The SmoothPositionData() function calculates a triple of fillet support points that make it possible to evaluate the specified fillet parameters.

Usually, using these functions, a certain phantom is drawn, which makes it possible to understand what the result of the operation will be and to change the parameters if necessary.

Before constructing fillets, the processed edges are sorted, and if necessary or on demand, smoothly connected edges are added to them, and sequences of smoothly connected edges are created from the edges.

The **distance1** and **distance2** parameters are associated with the first and the second surfaces and, accordingly, with the intersection curve of the first edge in the sequence of smoothly connected edges.

The surfaces can be obtained by the intersection curve functions GetCurveOneSurface() and

GetCurveTwoSurface().

The **distance1** and **distance2** of the first edge define the parameters for the other edges of each sequence. Thus, if for the first edge the pointer edge→GetIntersectionCurve().GetSurfaceOne() is equal to the pointer edge→GetFacePlus()→GetSurface().GetSurface(), then, when constructing a fillet, **distance1** will correspond to the radius for the **facePlus** face and **distance2** will correspond to the radius for the **faceMinus** face of the first edge in the sequence.

When constructing edge fillets, the function adds the MbFilletSolid builder to the body log; this builder is declared in the *cr_fillet_solid.h* file.

In the test application *test.exe*, edge processing of a body is performed by the menu commands "Create → Solid → By edge processing → Fillet with radius" and "Create → Solid → By edge processing → Fillet by a chord".

Rounding Edges with a Variable Radius

The function is similar to the one described in the previous subsection and differs from it by the **initFunctions** parameter. When rounding edges, the conjugation faces in a cross-section can have the shape of a circular arc with a variable radius.

When rounding edges with a variable radius, the following constructor is defined in the MbShellFilletValues class to initialize **filletParams** objects with the operation parameters:

```
MbShellFilletValues( const std::vector<MbEdgeFunction> & initFunctions,
                           c3d::FacesSPtrVector &        initBounds,
                     const MbSNameMaker &                names,
                     const SmoothValues &                params )
: MbShellChamferParams( c3d::EdgesSPtrVector(), params, names )
, _initFunctions( initFunctions )
, _initBounds( initBounds ) {}
```

where

- **initFunctions** – a set of body edges to be rounded with the functions of radius variation;
- **initBounds** – a set of faces for trimming the ends;
- **params** – an object that contains the parameters of edge rounding;
- **names** – a name maker.

The **initFunctions** parameter contains the processed edges of the **solid** body and the radius variation functions along an edge. Each element of the **initFunctions** set consists of a pointer to an edge and a pointer to a scalar function whose values are used to scale the fillet radii **distance1** and **distance2**, see Fig. 25:

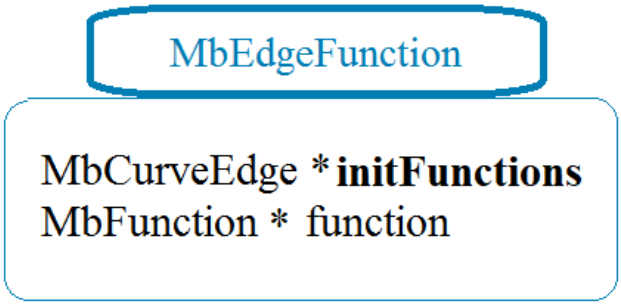


Fig. 25

For rounding edges with a variable radius, the **form** parameter value *st_Fillet* is used. For each point of a processed edge, **initFunctions**[i].initFunctions→Point(*t*, **point**), the radii of the fillet surface are equal to the **distance1** and **distance2** parameters multiplied by the value of the edge function **initFunctions**[i].function→Value(*t*).

Fig. 26 shows a fillet with variable radii of an edge of a rectangular prism. If **distance1** = **distance2** and **conic** = 0, the fillet surface is constructed by moving a sphere of variable radius that is tangent to the two faces adjacent to the edge being rounded. The support edges of the conjugation face pass through the points of tangency of the sphere and the corresponding adjacent face. A cross-section of the conjugation face is an arc of a circle.

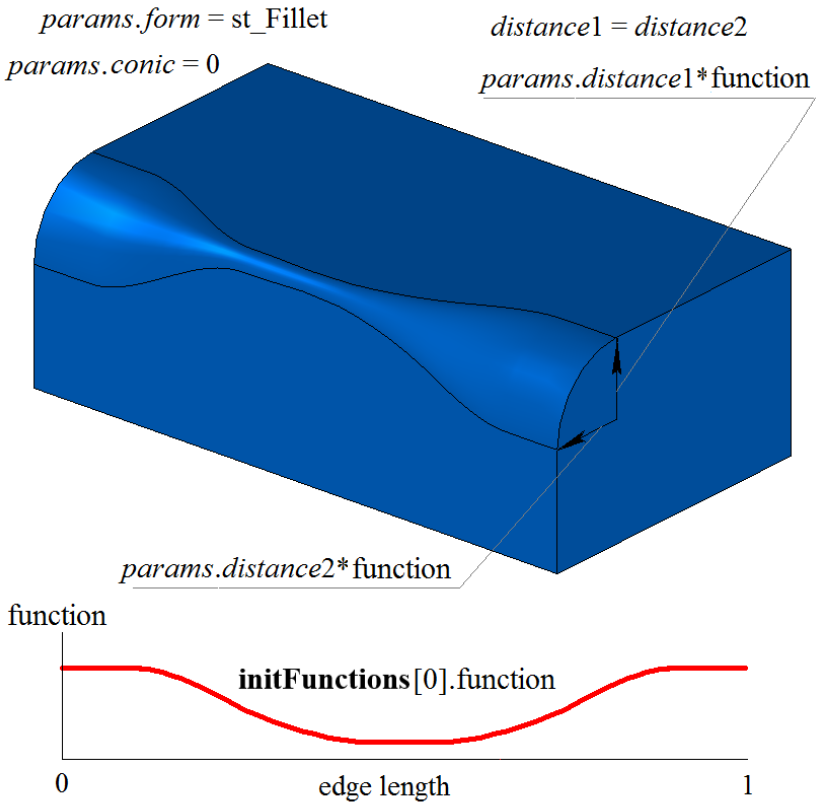


Fig. 26

Fig. 27 shows an elliptic fillet with variable radii of an edge of a rectangular prism:

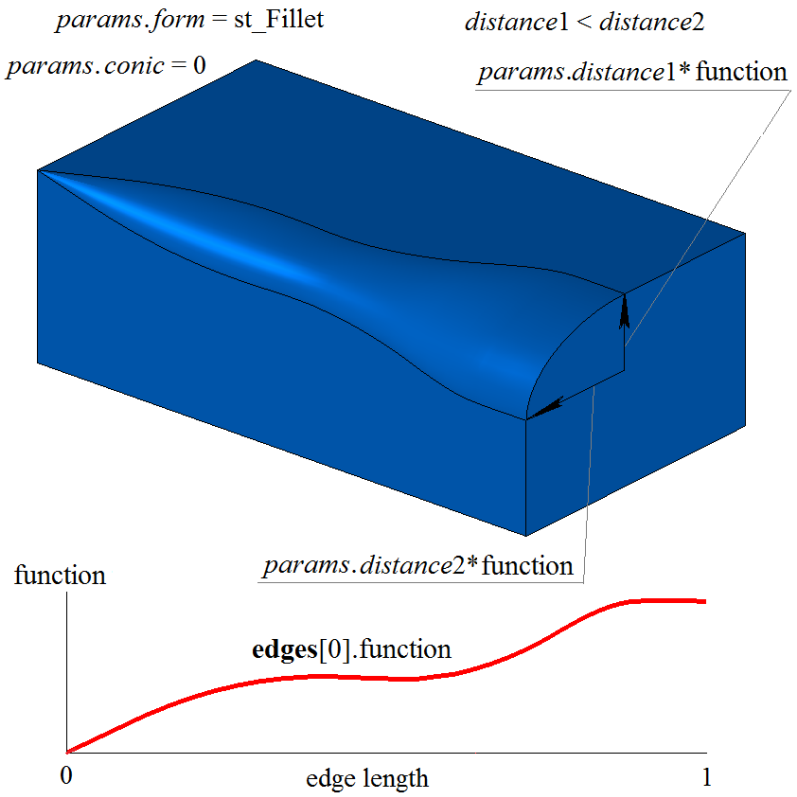


Fig. 27

Figs. 28, 29 show fillets with variable radii of an edge of a rectangular prism and different shape coefficients **conic**:

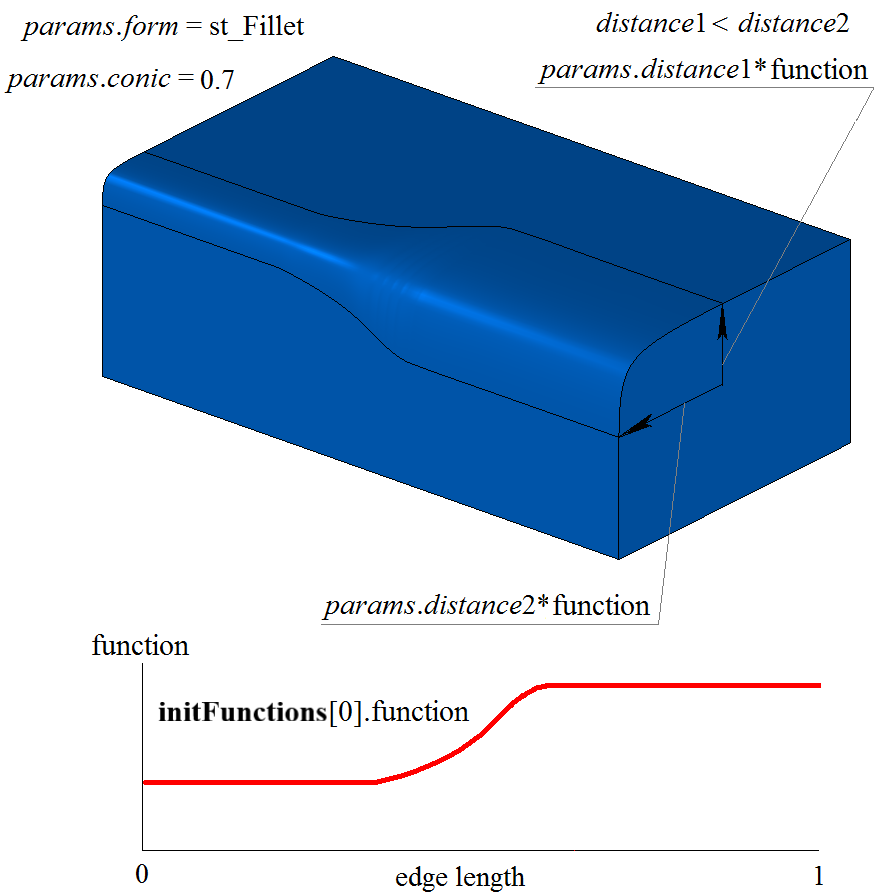

params.form = st_Fillet
params.conic = 0.7
distance1 < distance2
params.distance1*function
params.distance2*function
function
initFunctions[0].function
0
edge length
1

Fig. 28

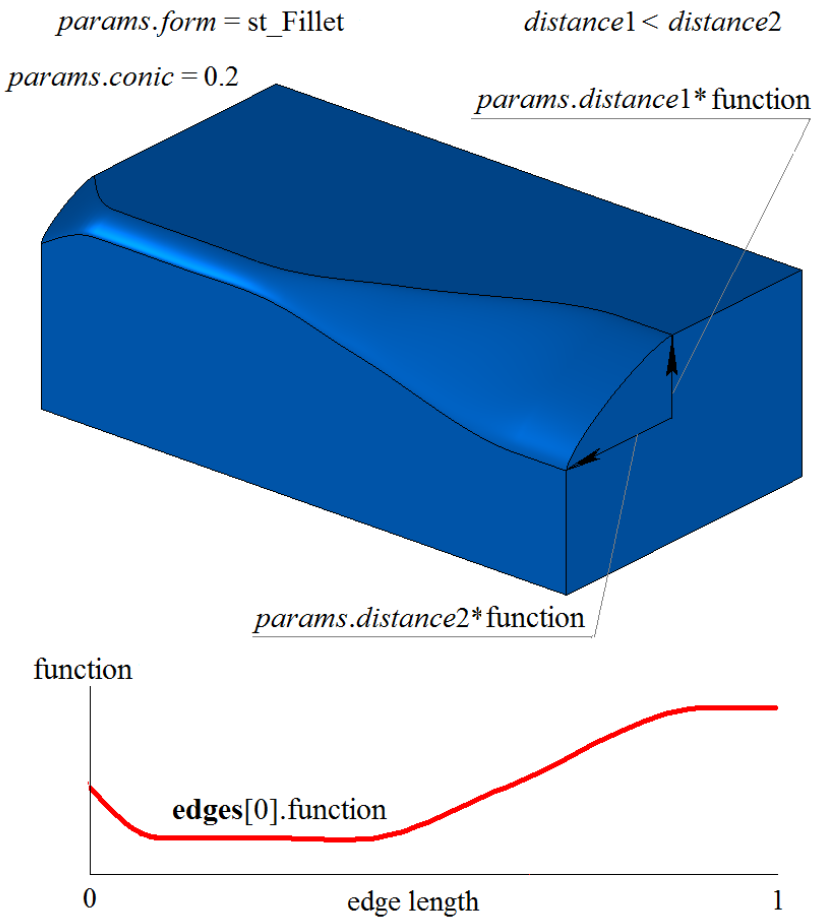


Fig. 29

Fig. 30 shows an example of stopping a fillet at the **begLength** distance from the initial vertex and at the **endLength** distance from the terminal vertex of the processed edge. When specifying the parameters of stopping a fillet, the fillet radius variation method is defined for the entire edge. If conjugation is not to be stopped, the **begLength** and **endLength** distances must be set to *UNDEFINED_DBL*.

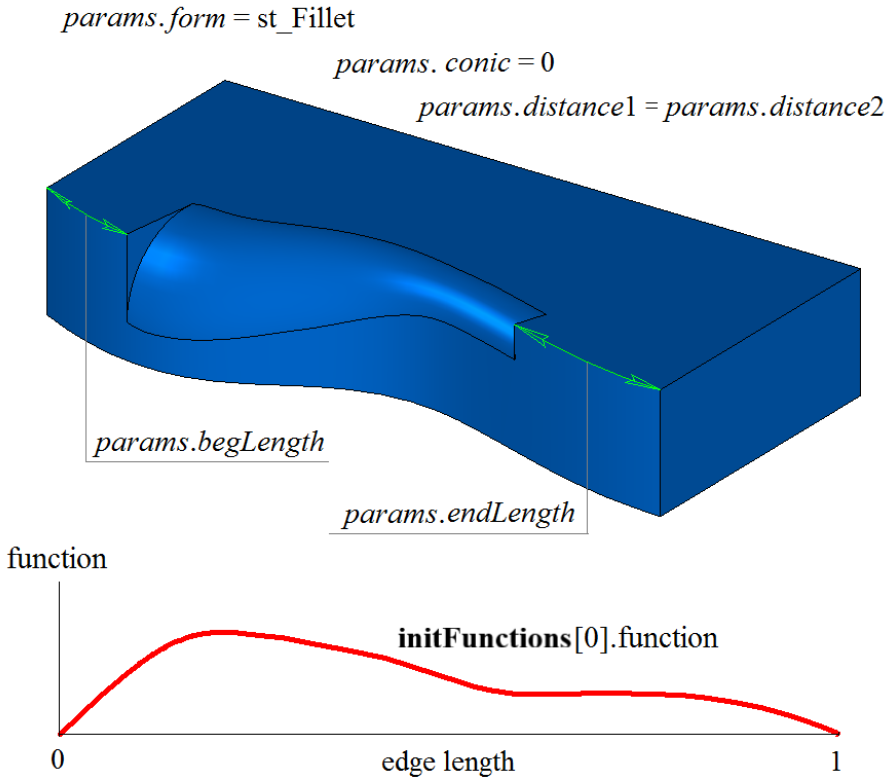


Fig. 30

For continuation edges of a fillet, the radius variation function takes a constant value equal to the value of the function at the end of the previous smoothly connected edge. Fig. 31 shows the source body, the edges to be rounded, and the radius variation function for the specified edges:

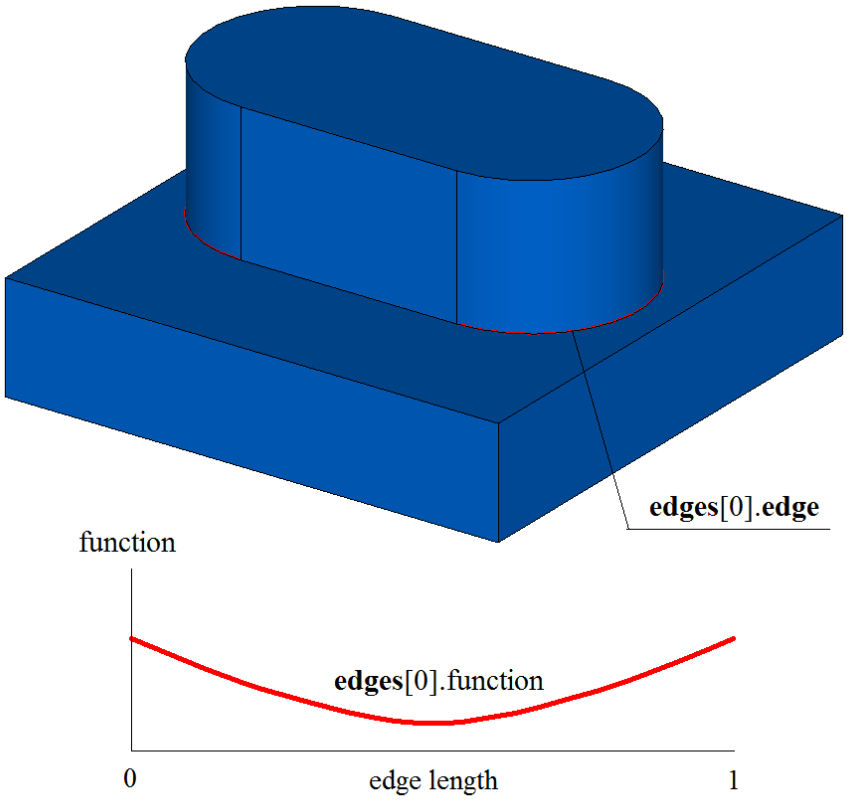


Fig. 31

Fig. 32 shows the result of the function for the source body. The figure shows that at the ends the radius variation function is chosen so as to provide smooth conjugation with a neighboring fillet face:

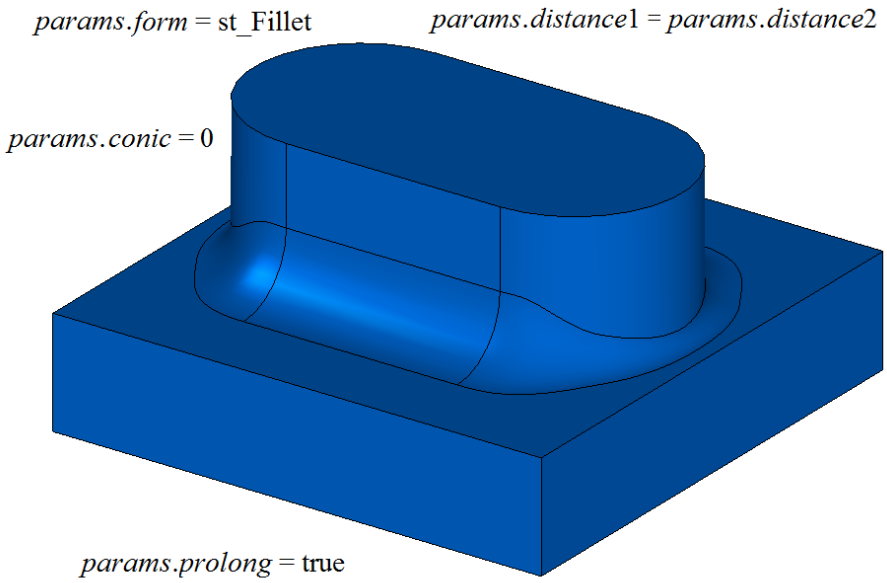


Fig. 32

In the test application *test.exe*, edge processing of a body with a variable radius is performed by the menu command “Create→Solid→By edges processing→Variable fillet”.

Rounding Vertices and Adjacent Edges with a Constant Radius

When rounding vertices and the edges adjacent to them with a constant radius, the following constructor is defined in the MbShellFilletValues class to initialize **filletParams** objects with the operation parameters:

```
MbShellFilletValues(       c3d::EdgesSPtrVector &    initCurves,
                           c3d::FacesSPtrVector &    initBounds,
                           c3d::VerticesSPtrVector & initVertices,
                    const CornerValues &             cornerData,
                    const SmoothValues &             params,
                    const MbSNameMaker &             names )
: MbShellChamferParams( initCurves, params, names )
, _initBounds  ( initBounds )
, _initVertices( initVertices )
, _cornerData  ( cornerData ) {}
```

where

- **initCurves** – a set of body edges to be rounded;
- **initBounds** – a set of faces for trimming the ends;
- **initVertices** – a set of vertices to be rounded;
- **cornerData** – an object that contains vertex rounding parameters;
- **params** – an object that contains parameters of edge rounding;
- **names** – a name maker.

The fillet construction parameter **params** contains information about the shape and the method of conjugation of the adjacent faces of the processed edges (see *Rounding Edges with a Constant Radius*).

The vertex rounding parameters **cornerData** for a vertex where three edges meet contain the information required to perform the operation, see Fig. 33. The CornerValues class is described in the *op_shell_parameter.h* file.

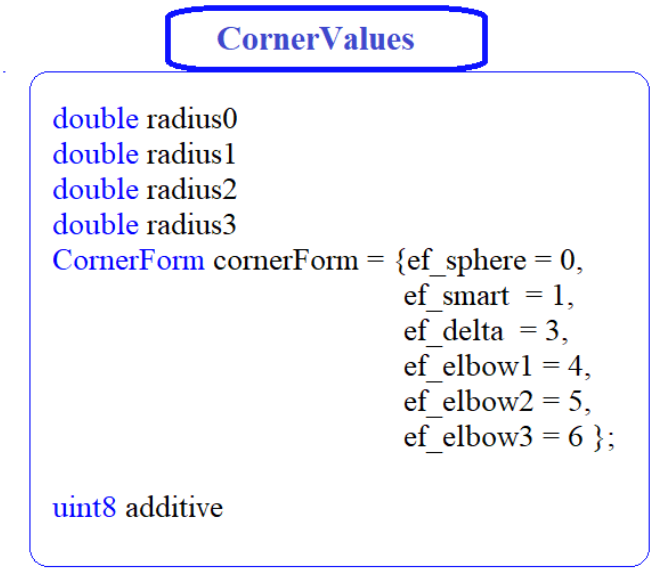


Fig. 33

- **radius0** – the radius of the sphere at the vertex;
- **radius1** – the radius of the first edge of the vertex;
- **radius2** – the radius of the second edge of the vertex;
- **radius3** – the radius of the third edge of the vertex;
- **cornerForm** – the method of rounding a vertex where three edges meet;
- **additive** – an offset in the numbering of the vertex edges (an increment to an edge number).

The vertex rounding type is controlled by the **cornerForm** parameter. For vertex rounding, the following **cornerForm** parameter values are used:

ef_sphere – rounding a vertex with a spherical surface;

ef_smart – rounding a vertex with a smooth surface;

ef_delta – rounding a vertex with a triangular surface;

ef_elbow1 – rounding a vertex with a quadrilateral surface (the fourth side is opposite the first edge of the vertex);

ef_elbow2 – rounding a vertex with a quadrilateral surface (the fourth side is opposite the second edge of the vertex);

ef_elbow3 – rounding a vertex with a quadrilateral surface (the fourth side is opposite the third edge of the vertex).

If **cornerForm** = *ef_sphere*, vertices where three edges of different convexity meet are rounded with a spherical surface, as shown in Fig. 34:

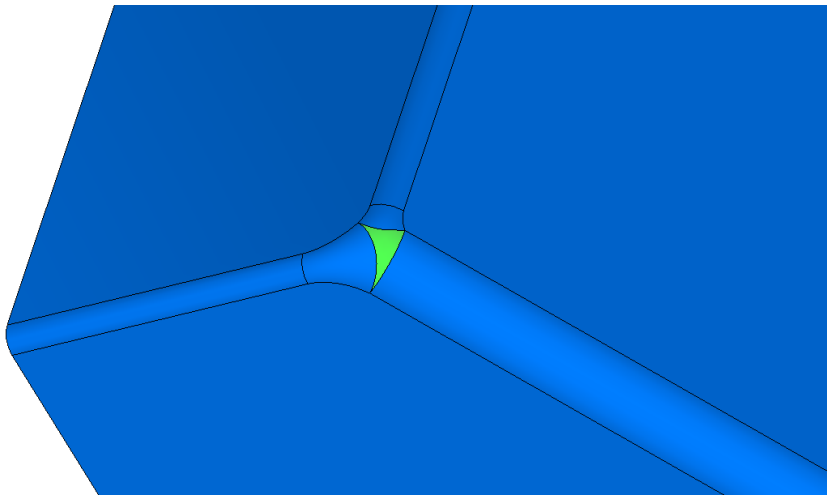


Fig. 34

If **cornerForm** = *ef_smart*, vertices where three edges of different convexity meet are rounded with a smooth surface, as shown in Fig. 35:

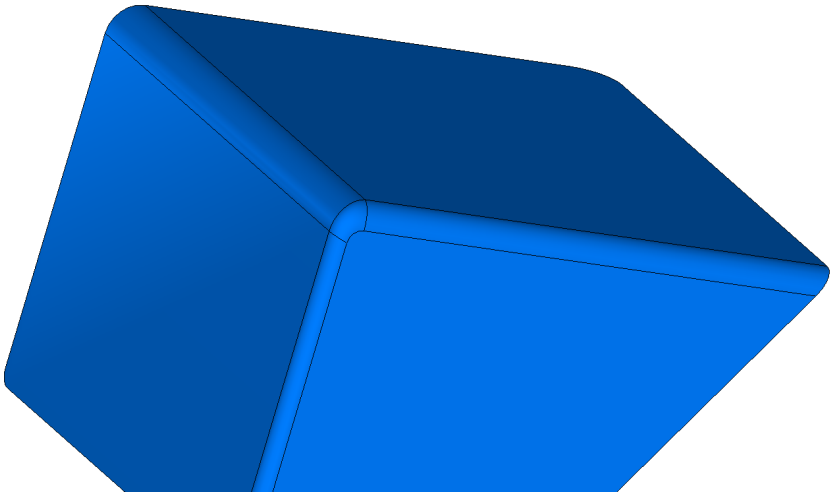


Fig. 35

If **cornerForm** = *ef_delta*, vertices where three edges of different convexity meet are rounded with a triangular surface, as shown in Fig. 36:

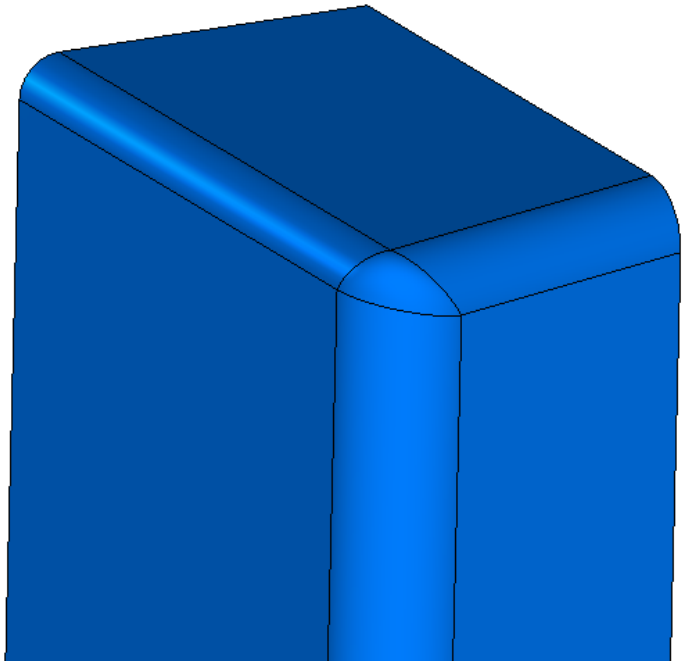


Fig. 36

If **cornerForm** = *ef_elbow2*, vertices where three edges of different convexity meet are rounded with a quadrilateral surface whose fourth side is opposite the second edge of the vertex, as shown in Fig. 37:

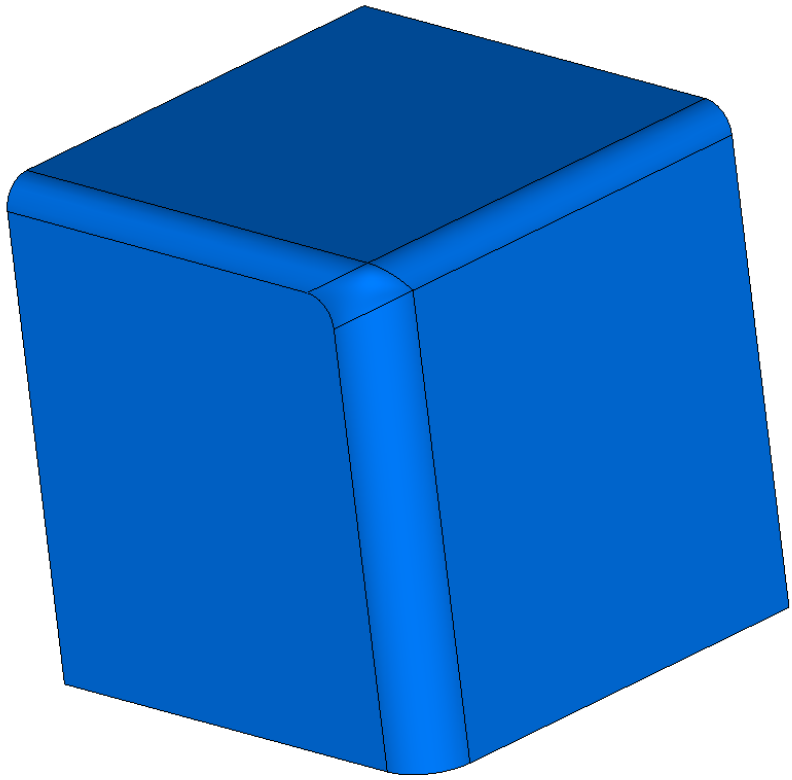


Fig. 37

In the test application *test.exe*, rounding vertices and the edges adjacent to them with a constant radius is performed by the menu command "Create→Solid→By vertices processing→Vertices fillet".

3.2.9 Constructing a Body with Edge Chamfers

The function

```
MbResultType ChamferSolid(       MbSolid &                  solid,
                                 MbeCopyMode                sameShell,
                           const MbShellChamferParams &     chamferParams,
                                 c3d::SolidSPtr &           result );
```

constructs chamfers on the specified edges of the source body.

The input parameters of the function are:

- **solid** is the source body;
- **sameShell** is the mode of copying the source body;
- **chamferParams** are the construction parameters.

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

The function returns the operation result code from the *MbResultType* enumeration.

The function is declared in the *action_solid.h* file.

When constructing bodies with edge chamfers, to initialize the **chamferParams** objects with the operation parameters, the following constructor is defined in the MbShellChamferParams class:

```
MbShellChamferParams( const c3d::EdgesSPtrVector &  initCurves,
                      const SmoothValues &          params,
                      const MbSNameMaker &          names )
: _initCurves( initCurves )
, _nameMaker( &names.Duplicate() )
, _params( params ){}
```

where

- **initCurves** is an array of body edges to be processed;
- **params** is an object with the chamfer construction parameters;
- **names** is a name maker.

The **solid** parameter contains the source body whose edges are to be processed.

The **sameShell** parameter controls the transfer of faces, edges and vertices from the source body **solid** to the resulting body **result** and can take one of four values:

- *cm_Same = 0* (the shell is not copied);
- *cm_KeepHistory* (the source shell and its copy have common base surfaces, vertices and faces not modified by the operation);
- *cm_KeepSurface* (the source shell and its copy have common base surfaces);
- *cm_Copy* (the source shell and its copy have no common data).

The MbeCopyMode enumeration is described in detail in the section *Copying a Set of Faces MbFaceShell*.

To construct edge chamfers, the same SmoothValues & **params** parameters are used as for edge filleting. The SmoothValues class is described in the *op_shell_parameter.h* file. The **params** object contains information about the shape and the method of mating adjacent faces of the processed edges. For chamfer construction, the following data of the **params** parameter are used*:

- **distance1** is the first leg of the chamfer;
- **distance2** is the second leg of the chamfer;
- **begLength** is the distance from the start vertex to the mating stop point (a negative value means no stop; the value must be equal to *UNDEFINED_DBL* if no stop is required);
- **endLength** is the distance from the end vertex to the mating stop point (a negative value means no stop; the value must be equal to *UNDEFINED_DBL* if no stop is required);
- **form** is the mating type from the MbeSmoothForm enumeration;
- **smoothCorner** is the method of rounding suitcase corners;
- **prolong** is a flag of extension of the chamfer along tangent edges;
- **vector1** is the normal vector of the mating stop plane at the start;
- **vector2** is the normal vector of the mating stop plane at the end.

* All public fields will be moved to the private section. To specify chamfer parameters, specialized initialization methods should be used:

```
// initialization of a chamfer with equal offsets
bool InitChamfer( double dist );

// initialization of a chamfer with different offsets
bool InitChamfer( double dist1, double dist2 );

// initialization of a chamfer by one leg and an angle
bool InitOneSideChamfer( bool firstSide, double dist, double ang );
```

Additional properties of the operation are specified by the SetProperties() method, whose arguments are:

- **c** is a shape coefficient, varying from 0.05 to 0.95 (at 0 — a circular arc);
- **pro** is a flag of extension along the tangent;
- **cor** is the method of rounding suitcase corners;
- **autoS** is automatic detection of edge/surface preservation;
- **keep** specifies whether to preserve the edge (true) or to preserve the fillet/chamfer surface (false);
- **str** is strict filleting. If false, fillet at least what is possible;
- **equ** specifies inserting a toroidal surface at mating corners.

To specify stops, the SetBegLength ()/SetEndLength() methods are defined (by default, no stop is specified).

The chamfer mating method is controlled by the **form** parameter. For edge chamfer construction, the values of the **form** parameter equal to *st_Chamfer*, *st_Slant1* and *st_Slant2* are used.

If **form** = *st_Chamfer*, the function constructs a chamfer surface with the specified legs defined by the **distance1** and **distance2** parameters, see Fig. 1:

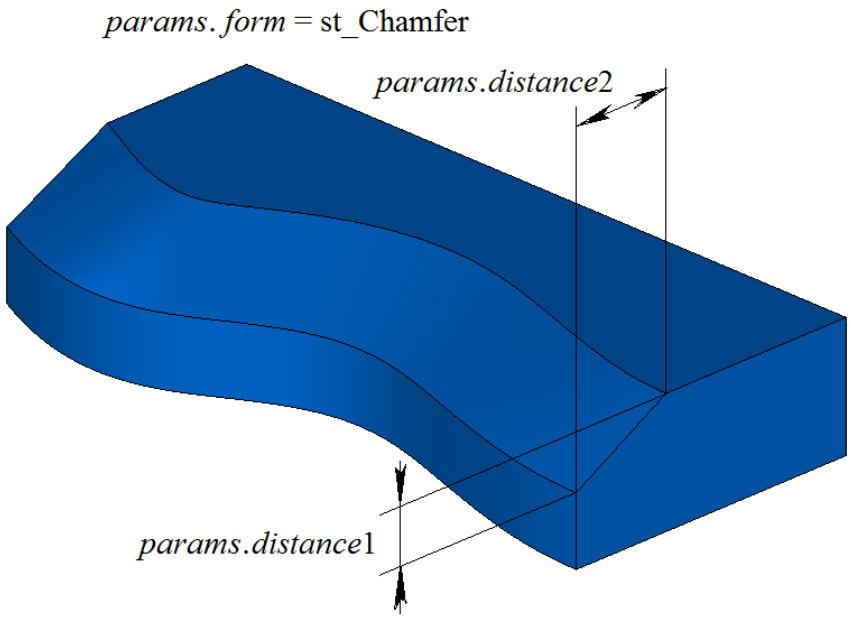


Fig. 1

If **form** = *st_Slant1*, the function constructs a chamfer surface with the specified leg **distance1** and the adjacent angle determined by the **distance2** leg, see Fig. 2:

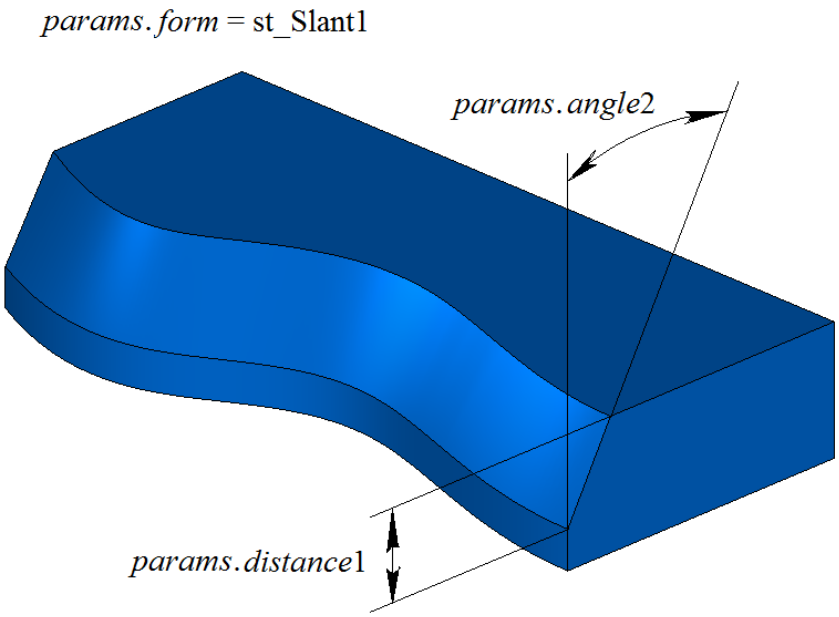


Fig. 2

If **form** = *st_Slant2*, the function constructs a chamfer surface with the specified angle and the adjacent leg. The **distance1** parameter corresponds to the leg that ensures the specified angle, and **distance2** defines the adjacent leg, see Fig. 3:

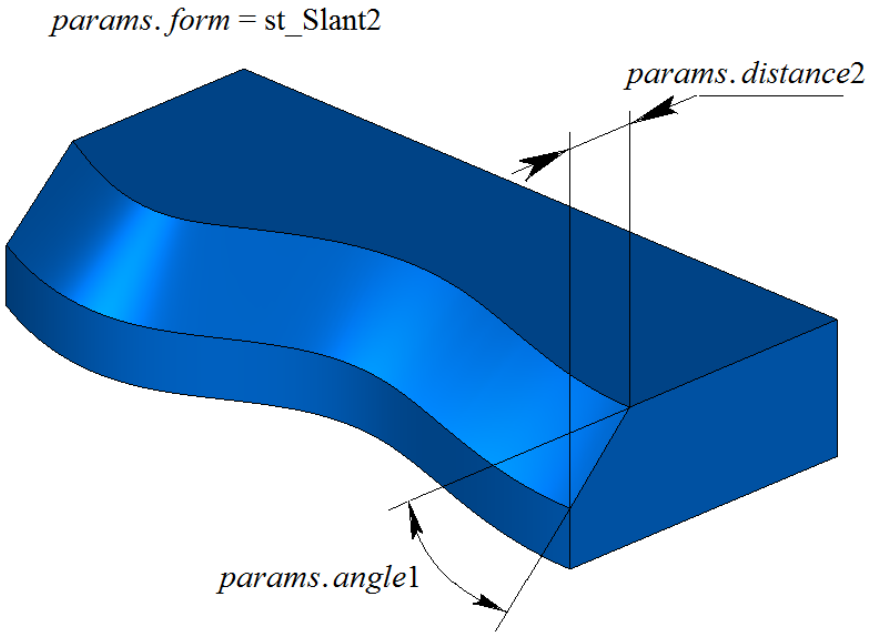


Fig. 3

Fig. 4 shows an example of stopping the chamfer at the distance **begLength** from the start vertex and at the distance **endLength** from the end vertex of the processed edge. If mating stopping is not required, then the **begLength** and **endLength** distances must take negative values.

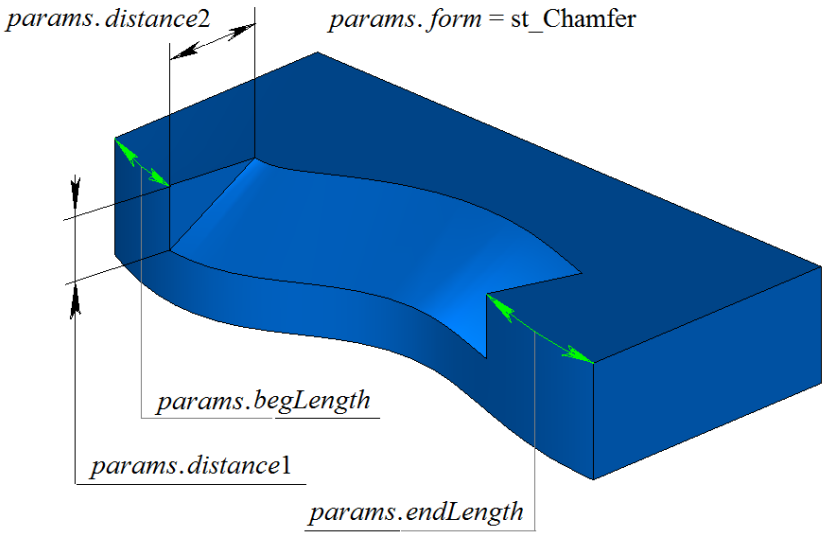


Fig. 4

Using the body example shown in Fig. 5, we demonstrate how the **prolong** flag works when constructing a chamfer on the selected edge:

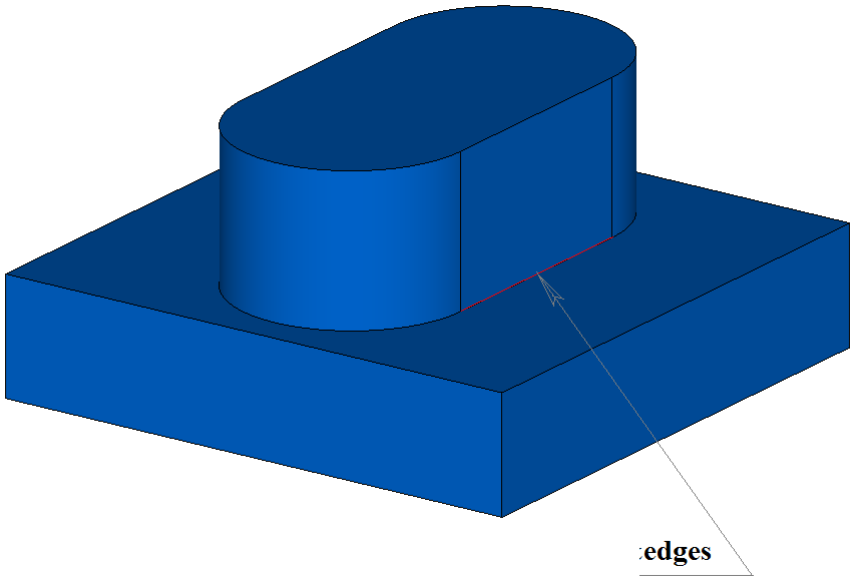


Fig. 5

The **prolong** flag specifies which edges of the body must be processed. If **prolong** = *false*, only those edges that are specified in the **initCurves** container are processed, see Fig. 6:

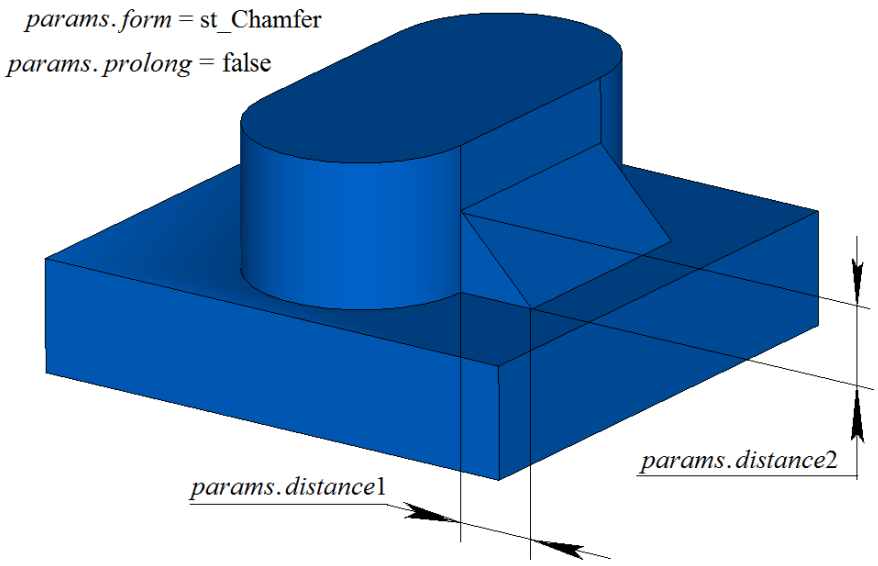


Fig. 6

If **prolong** = *true*, the edges specified in the **initCurves** container as well as edges smoothly joined with them are processed, see Fig. 7:

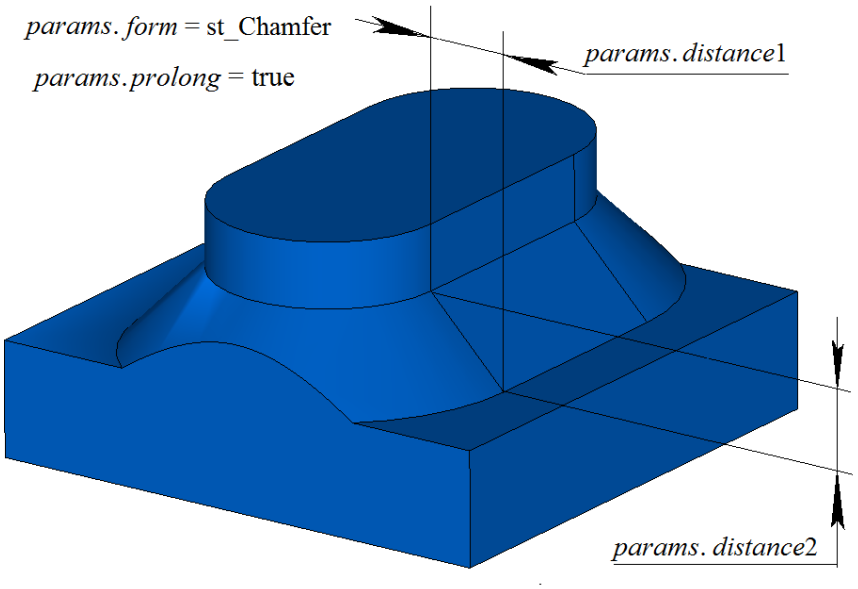


Fig. 7

When a chamfer is constructed on three edges meeting at one vertex, the **smoothCorner** parameter defines the method of processing suitcase corners. Using the body example shown in Fig. 8, we demonstrate the effect of the **smoothCorner** parameter when constructing chamfers on all edges of the body.

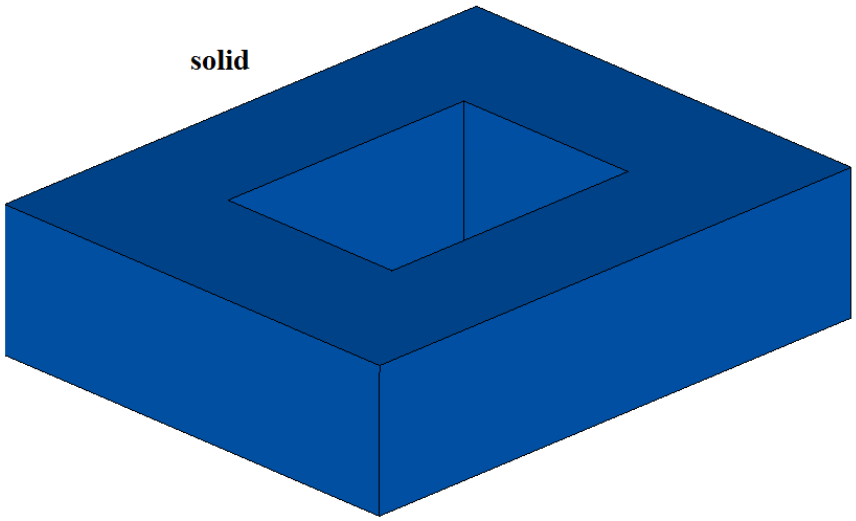


Fig. 8

If **smoothCorner** = *ec_pointed*, then corners where three edges of the same convexity meet are not processed, and the resulting body contains a point where three chamfer faces meet, see Fig. 9:

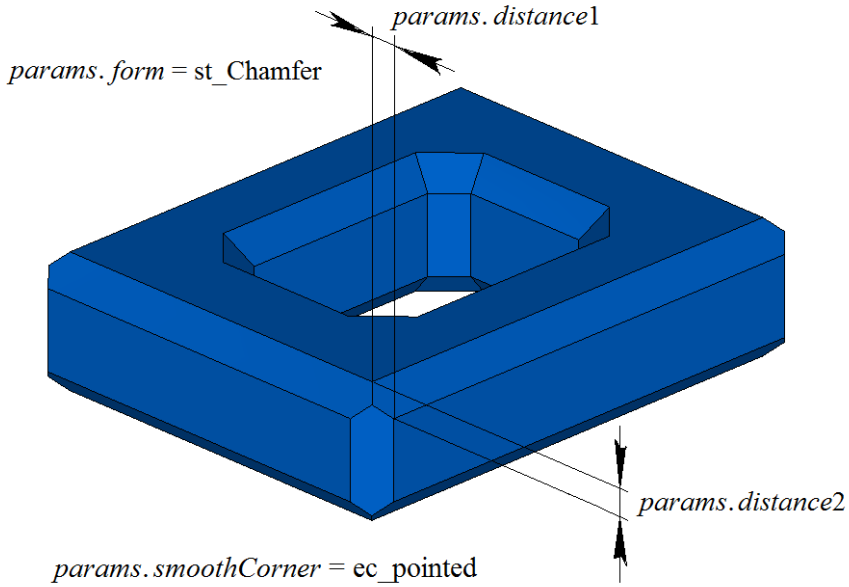


Fig. 9

For all other values of the **smoothCorner** parameter, corners where three edges of the same convexity meet are processed by constructing an additional face, as shown in Fig. 10:

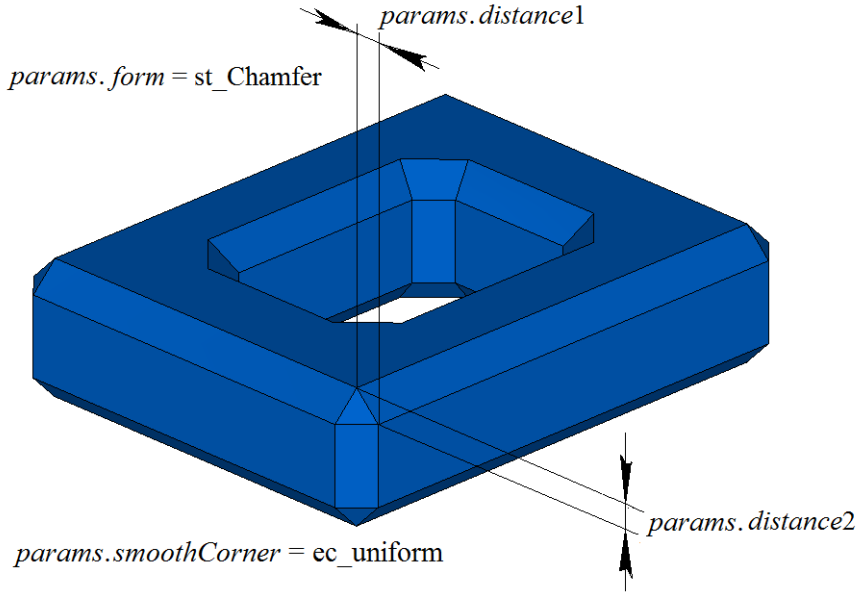


Fig. 10

Using the pyramid-shaped body example shown in Fig. 11, we demonstrate the behavior of the considered function when constructing a chamfer in special cases. The result of construction for a symmetric edge configuration and a symmetric chamfer is shown in Fig. 12:

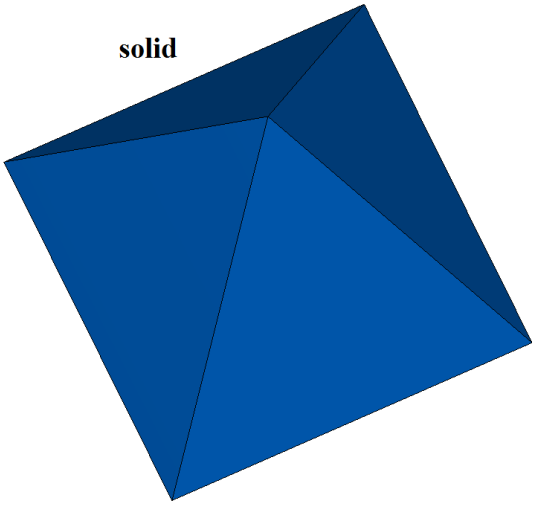


Fig. 11

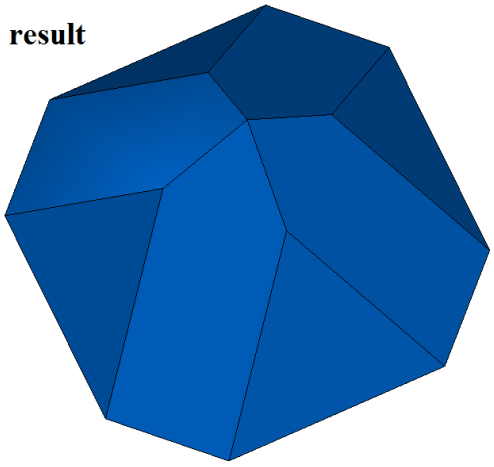


Fig. 12

It should be noted that to construct a chamfer on four or more edges meeting at one vertex, the intersection of all chamfer surfaces at one point is required. An example of a symmetric chamfer on seven edges meeting at a common vertex is shown in Fig. 13:

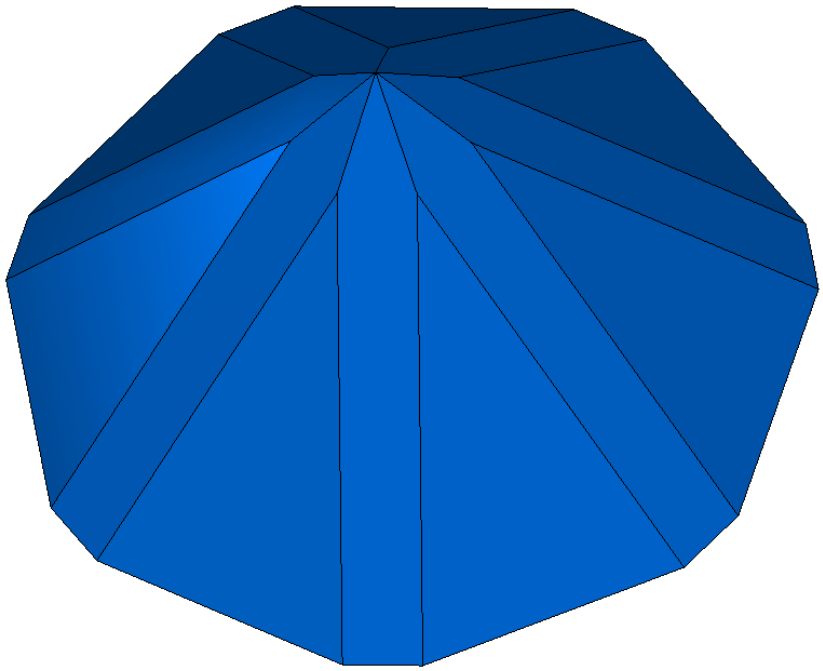


Fig. 13

When the **distance1** and **distance2** parameters in the chamfer construction function are not equal and several edges are specified for processing, it is difficult to understand which parameter corresponds to what. In this situation it is recommended to use the auxiliary functions SmoothPhantom(), SmoothSequence() and SmoothPositionData(), which are declared in the *action_phantom.h* file.

The SmoothPhantom() function constructs simplified surfaces to simulate the future chamfers.

The SmoothSequence() function constructs sequences of edges to which, if desired, smoothly joined edges to be processed together can be added.

The SmoothPositionData() function computes three points for the processed edges, which are used for phantom dimensions of legs and angles. Usually these functions are used to draw a phantom that makes it possible to understand what will be obtained as a result of the operation and to change the parameters if necessary.

Before constructing chamfers, the processed edges are sorted; if necessary or on demand, smoothly joined edges are added to them, and sequences of smoothly joined edges are created from the edges. The **distance1** and **distance2** parameters are associated with the first and second surfaces respectively of the intersection curve of the first edge in the sequence of smoothly joined edges.

The surfaces can be obtained by the intersection-curve methods GetCurveOneSurface() and GetCurveTwoSurface().

By the **distance1** and **distance2** values of the first edge, the parameters for the remaining edges of each sequence are determined.

When constructing edge chamfers, the considered function adds the MbChamferSolid builder to the history of the constructed body; this builder is declared in the *cr_chamfer_solid.h* file.

In the test application *test.exe*, body construction with chamfers is performed by the menu commands:

- "Create → Body → By edges processing → Cathetus-cathetus chamfer";
- "Create → Body → By edges processing → Cathetus-angle chamfer";
- "Create → Body → By edges processing → Angle-cathetus chamfer".

3.2.10 Constructing a Thin-Wall Body

The function

```
MbResultType ThinSolid(       MbSolid &           solid,
                              MbeCopyMode         sameShell,
                              RPArray<MbFace> &   outFaces,
                              SweptValues &       params,
                        const MbSNameMaker &      names,
                              bool                copyFaceAttrs,
                              MbSolid *&          result );
```

constructs a thin-wall body by excluding the specified faces from the source body and optionally assigning a uniform thickness to the remaining faces.

The input parameters of the function are:

- **solid** is the source body;
- **sameShell** is the copying mode of the source body;
- **outFaces** are the faces to be removed from the body;
- **params** are the operation parameters;
- **names** is the name maker;
- **copyFaceAttrs** is the flag for copying attributes from source faces to offset faces.

The output parameter of the function is the **result** object that contains the constructed body.

The function returns an operation result code from the *MbResultType* enumeration.

The function is declared in the *action_solid.h* file.

The function removes the specified faces **outFaces** from the source body **solid** and "assigns the specified thickness" to the remaining faces.

The parameter **sameShell** controls the transfer of faces, edges, and vertices from the source body **solid** to the constructed body **result** and can take one of four values:

- *cm_Same = 0* (shell is not copied);
- *cm_KeepHistory* (source shell and its copy share base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (source shell and its copy share base surfaces);
- *cm_Copy* (source shell and its copy have no shared data).

The MbeCopyMode enumeration is described in detail in the section *Copying a Set of MbFaceShell Faces*.

The parameter **params** contains information about the wall thickness of preserved faces and the closure of the constructed body **result** (see *op_swept_parameter.h*).

The wall thickness of preserved faces can be equal to **params**.*thickness1* in the positive normal direction of the face or **params**.*thickness2* in the negative normal direction.

If **params**.*shellClosed* = *false*, a non-closed body will be constructed.

The parameter **params**.*checkSelfInt* specifies whether the construction result should be checked for self-intersection. By default, **params**.*checkSelfInt* = *false* (no check is performed).

The **params**.*mergeFlags* field controls merging of geometrically similar faces and edges (*mergeFaces* = *true* — merge similar faces, *mergeEdges* = *true* — merge similar edges).

The parameter **names** provides naming for faces, edges, and vertices of the constructed body.

For the operation to succeed, the faces to be removed **outFaces** must not have smooth mating edges along their common boundary with the preserved faces of the source body.

The parameter **copyFaceAttrs** controls copying attributes from source faces to offset faces.

Figure 1 shows the source body **solid** and faces to be removed **outFaces**:

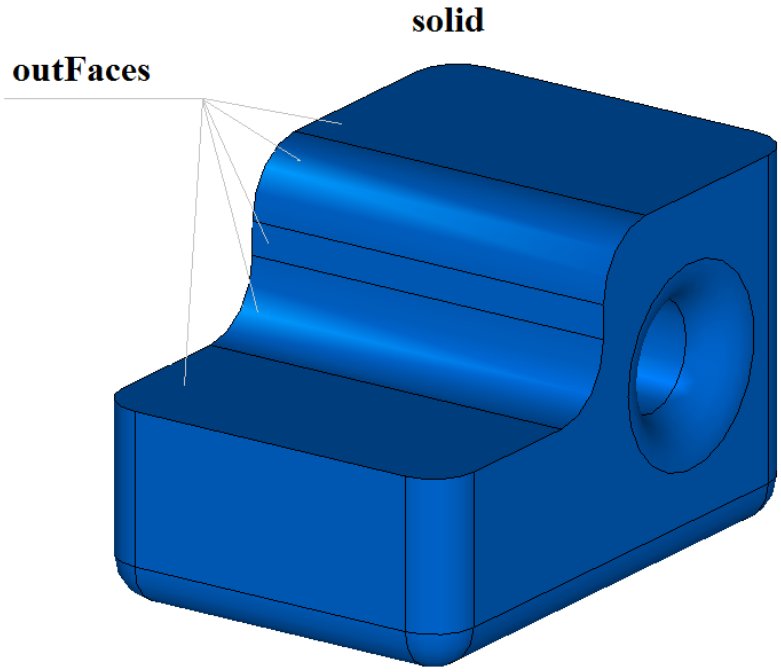


Fig. 1

Figure 2 shows the constructed thin-wall body **result** with preserved faces thickened inward from the source body:

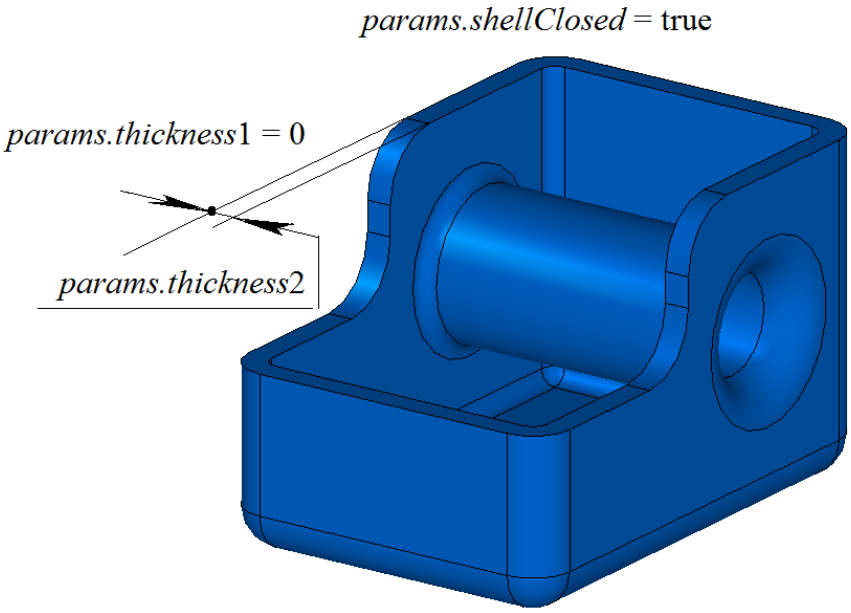


Fig. 2

Figure 3 shows the constructed thin-wall body **result** with preserved faces thickened outward from the source body:

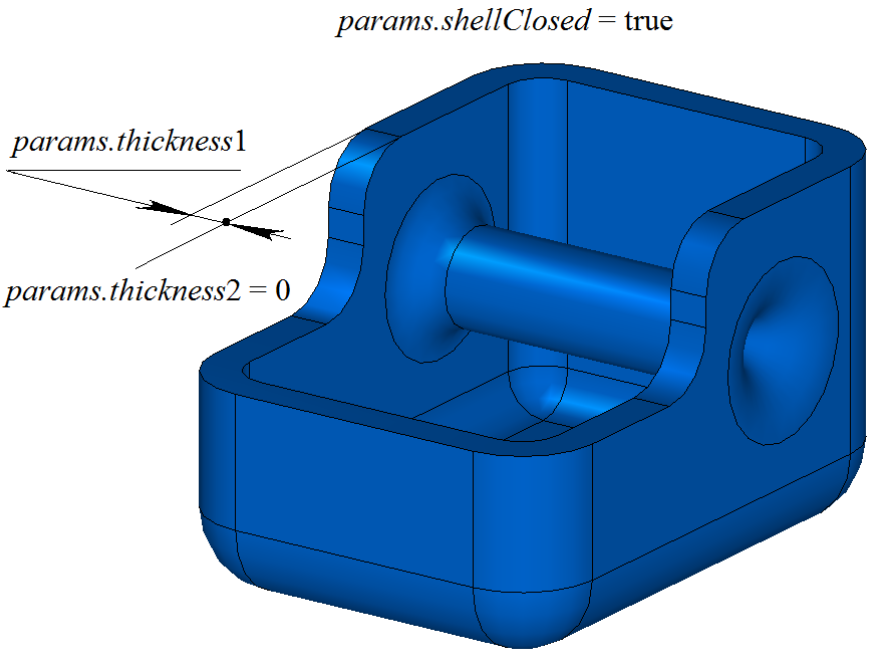


Fig. 3

Figure 4 shows the constructed non-closed body **result**:

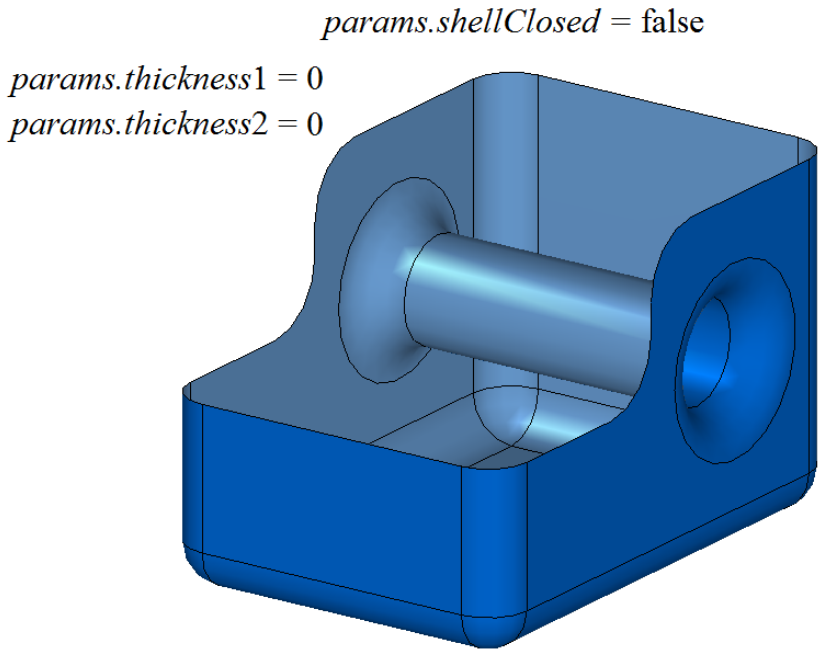


Fig. 4

Figure 5 shows the thin-wall body constructed with an empty set of faces to be removed:

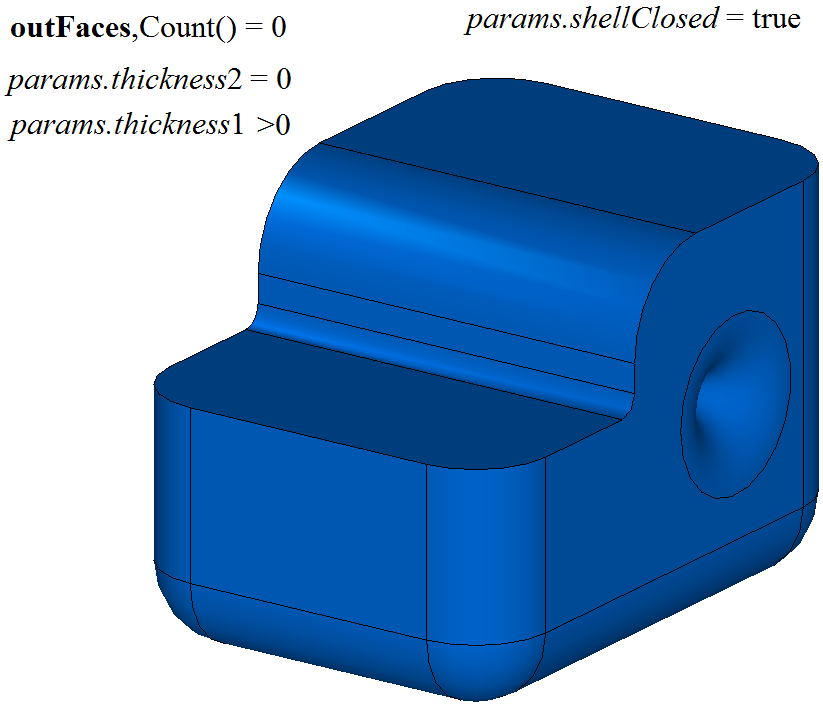


Fig. 5

The ThinSolid() function adds the MbShellSolid builder to the log of the constructed body, which contains all necessary data for performing the operation. The MbShellSolid builder is declared in the *cr_thin_shell_solid.h* file.

In the test application *test.exe*, thin-wall body construction is performed using the menu command "Create → Solid → By faces processing → Thin-walled with a constant thickness".

3.2.11 Constructing a Thin-Wall Body with Various Wall Thickness

The function

```
MbResultType ThinSolid(       MbSolid &            solid,
                              MbeCopyMode          sameShell,
                              RPArray<MbFace> &    outFaces,
                              RPArray<MbFace> &    offFaces,
                              SArray<double> &     offDists,
                              SweptValues &        params,
                        const MbSNameMaker &       names,
                              bool                 copyFaceAttrs,
                              MbSolid *&           result );
```

constructs a thin-walled body by excluding the specified faces from the original body and assigning different wall thicknesses to the remaining faces.

The input parameters of the function are:

- **solid** — the source body;
- **sameShell** — the copying mode of the source body;
- **outFaces** — faces to be excluded from the body;
- **offFaces** — a set of faces for which individual thickness values are specified;
- **offDists** — a set of individual thickness values (must be synchronized with the **offFaces** array);
- **params** — an object that contains information about the closure of the constructed body and the wall thickness of preserved faces;
- **names** — the name maker of the operation;
- **copyFaceAttrs** — a flag for copying attributes from source faces to equidistant faces.

The output parameter of the function is the **result** object that contains the constructed body.

The function returns an operation result code from the MbResultType enumeration.

The function is declared in the file *action_solid.h*.

The function removes the specified faces **outFaces** from the source body **solid** and "assigns the specified thickness" to the remaining faces, where different faces can have different thicknesses.

The parameter **sameShell** controls the transfer of faces, edges and vertices from the source body **solid** to the constructed body **result** and can take one of four values:

- cm_Same = 0 (the shell is not copied);
- cm_KeepHistory (the source shell and its copy share base surfaces, vertices, and faces unchanged by the operation);
- cm_KeepSurface (the source shell and its copy share base surfaces);

- cm_Copy (the source shell and its copy have no shared data).

The MbeCopyMode enumeration is described in paragraph *Copying a Set of Faces MbFaceShell*.

The parameter **offFaces** contains faces with individual thickness values **offDists**.

Face **offFaces**[*i*] will be assigned thickness **offDists**[*i*]. The remaining faces will be assigned thickness determined by the **params** parameter. The parameter **params** contains information about the closure of the constructed body **result** and the wall thickness of the preserved faces not included in the **offFaces** set.

The wall thickness of preserved faces can be either **params**.*thickness1* in the positive normal direction or **params**.*thickness2* in the negative normal direction.

The parameter **names** provides naming for faces, edges and vertices of the constructed body.

To perform the operation, the faces **outFaces** to be removed from the body must not have smooth connection edges along their common perimeter with the preserved faces of the source body.

Figure 1 shows the source body **solid**, faces to be removed **outFaces**, and faces **offFaces** with individual thickness values **offDists**:

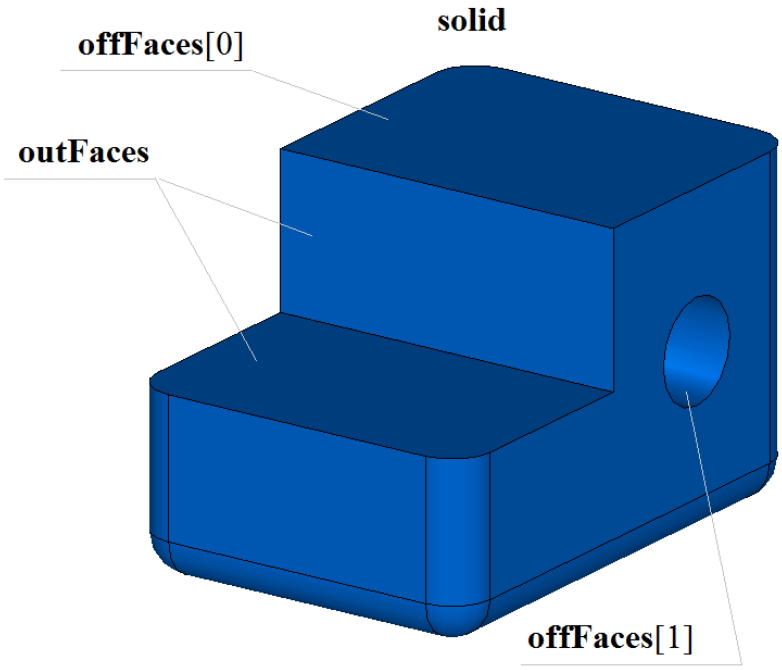


Fig. 1

Figure 2 shows the constructed body **result** with preserved and newly constructed faces:

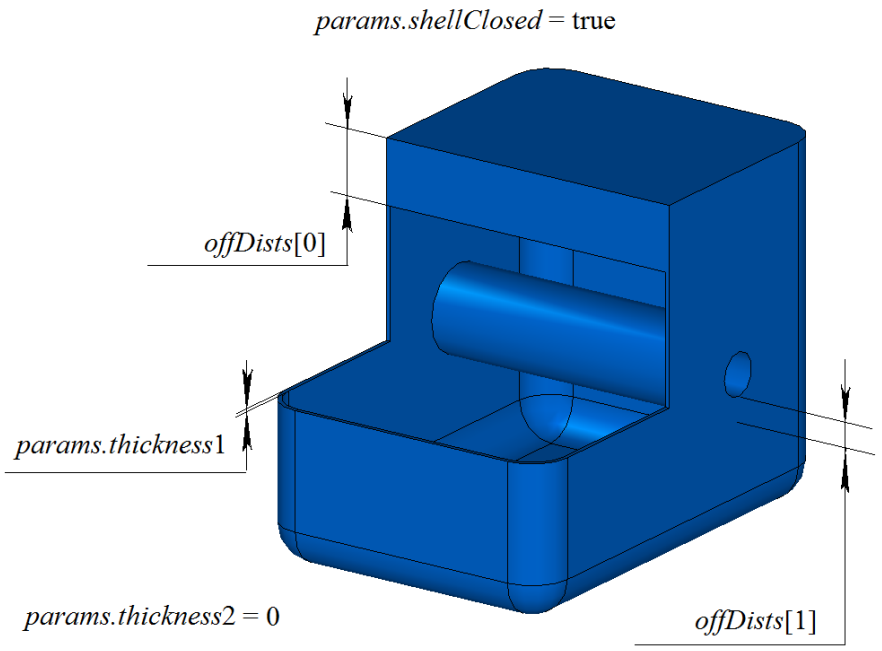


Fig. 2

To perform the operation, each face from **offFaces** must not have smooth connection edges along its common perimeter with faces of the source body having different thicknesses.

The ThinSolid() function adds to the construction log of the constructed body a builder MbShellSolid containing all necessary data for performing the operation. The builder MbShellSolid is declared in the file *cr_thin_shell_solid.h*.

In the test application *test.exe*, constructing a thin-walled body with different wall thicknesses is performed using the menu command “Create → Solid → By faces processing → Thin-walled with multiple thicknesses”.

3.2.12 Constructing a Body by Thickening the Surface

The function

```
MbResultType ThinSolid( const MbSurface &       surface,
                              bool              faceSense,
                              SweptValues &     params,
                        const MbSNameMaker &    names,
                              SimpleName        name,
                              MbSolid *&        result )
```

constructs a body by thickening the given surface.

The input parameters of the function are:

- **solid** — the given surface;
- **faceSense** — orientation of the normal of the face constructed on the surface;
- **params** — an object containing construction parameters;

- **names** — the operation namer;
- **name** — the operation name.

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

The function returns an operation result code from the MbResultType enumeration.

The function is declared in the file *action_solid.h*.

The function constructs a face based on the **surface** surface and then builds a body by assigning the specified thickness to this face.

The **faceSense** parameter indicates whether the direction of the **surface** normal matches the direction of the face normal (*true* if they match, *false* otherwise). The face will be assigned the thickness specified in the **params** parameter.

The **params** parameter contains information about the wall thickness of the constructed body **result**. The wall thickness can be either **params**.*thickness1* in the positive direction of the face normal or **params**.*thickness2* in the negative direction of the face normal.

The **names** and **name** parameters provide naming for the faces, edges, and vertices of the constructed body.

Figure 1 shows the source surface **surface**:

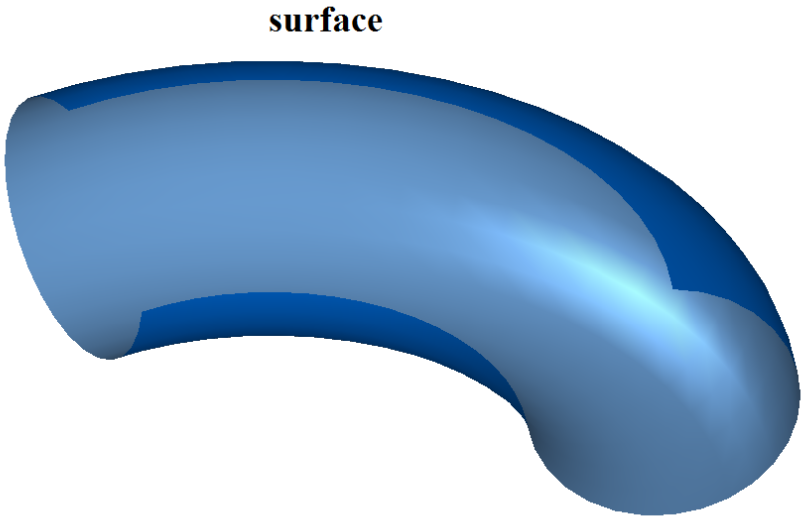


Fig. 1

Figure 2 shows the constructed body **result**:

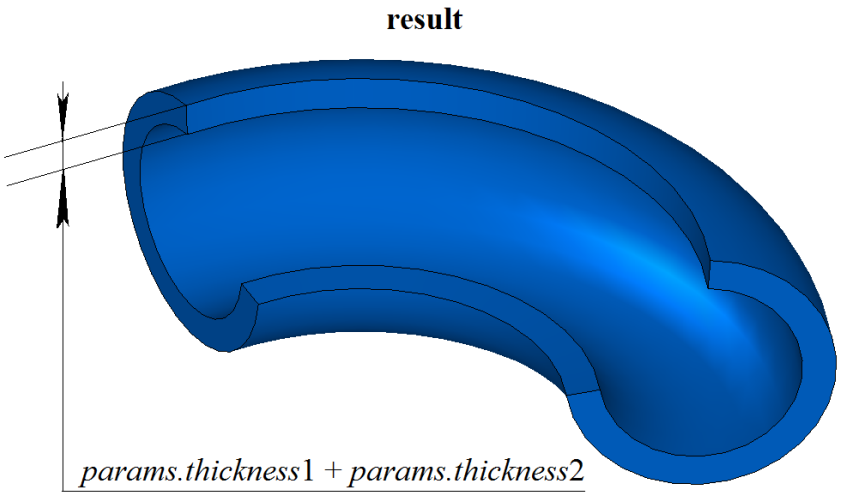


Fig. 2

The ThinSolid() function adds to the construction log of the constructed body a builder MbShellSolid that contains all the necessary data for performing the operation. The MbShellSolid builder is declared in the file *cr_thin_shell_solid.h*.

In the test application *test.exe*, construction of a thin-walled body is performed by the menu command “Create → Solid → Based on a surface → By thickening”.

3.2.13 Boolean Operation on a Body and a Set of Bodies

The function

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

performs a union of the bodies from the given set and the specified Boolean operation with the source body, if it is provided.

The input parameters of the function are:

- **solid** — the source body (can be null);
- **sameShell** — the copy mode for the source body;
- **solids** — the set of bodies;
- **sameShells** — the copying mode for bodies in the set;
- **oType** — the type of Boolean operation between the body and the array of bodies:

— *bo_Union* — union of bodies,

— *bo_Intersect* — intersection of bodies,

— *bo_Difference* — subtraction of the set of bodies **solids** from body **solid**;

- **checkIntersect** — the flag for checking intersection of bodies in the set (*false* — do not check);
- **mergeFlags** — the control flags for merging shell elements;
- **names** — the operation namer;
- **isArray** — the flag for regularity of the set of bodies.

The output parameters of the function are: the constructed body **result** and the set of bodies **notGluedSolids** that were not used in the operation (can be null).

The function returns an operation result code from the MbResultType enumeration.

The function is declared in the file *action_solid.h*.

The function is a variant of the Boolean operation BooleanSolid() and accelerates execution when it is necessary to perform the same Boolean operation on body **solid** with a large number of other bodies. First, the function merges the set of bodies **solids** into a common intermediate body, and then performs the specified Boolean operation **oType** of body **solid** with the intermediate body. The bodies **solids** may not intersect with each other.

The **sameShell** parameter controls whether and how faces, edges and vertices from the source body **solid** are transferred to the constructed body **result** and can take one of four values:

- cm_Same = 0 (the shell is not copied);
- cm_KeepHistory (the source shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- cm_KeepSurface (the source shell and its copy share common base surfaces);
- cm_Copy (the source shell and its copy share no common data).

The MbeCopyMode enumeration is described in section *Copying a Set of Faces MbFaceShell*.

The **oType** parameter defines the type of Boolean operation and takes three values: *bo_Union*, *bo_Intersect*, *bo_Difference*:

- when **oType** = *bo_Union*, the union of bodies **solid** and the set of bodies **solids** is performed;
- when **oType** = *bo_Intersect*, the intersection of bodies **solid** and the set of bodies **solids** is performed;
- when **oType** = *bo_Difference*, subtraction of the set of bodies **solids** from body **solid** is performed.

The **checkIntersect** and **isArray** parameters control the construction of a common intermediate body for the set of bodies **solids**.

The **checkIntersect** parameter instructs the function to check for intersections between bodies in the **solids** set. If **checkIntersect** == *true*, a Boolean union operation is performed on all intersecting bodies in the **solids** set to form a single body during construction of the common intermediate body. Otherwise, merging of bodies in the given set is performed by simply combining faces of all bodies into one new body. Regardless of the value of the **checkIntersect** parameter, all non-intersecting bodies in the **solids** set contribute their faces to the common intermediate body.

The **isArray** parameter is applicable only when **checkIntersect** == *true* and indicates the regularity of the set of bodies **solids**. If **isArray** == *true*, the bodies in the set are positioned at the nodes of a rectangular or circular grid with their positions encoded in the face names.

The **mergeFlags** parameter controls whether similar faces and edges in the constructed body **result** are merged or left separate. When **mergeFlags**.*mergeFaces* == *false*, similar faces are not merged, and when **mergeFlags**.*mergeEdges* == *false*, similar edges are not merged.

The **names** parameter provides naming for faces, edges and vertices of the constructed body.

The **notGluedSolids** parameter contains bodies that were not used in the operation due to inability to merge them with the common intermediate body.

Figure 1 shows the source body **solid** and the set of bodies **solids**:

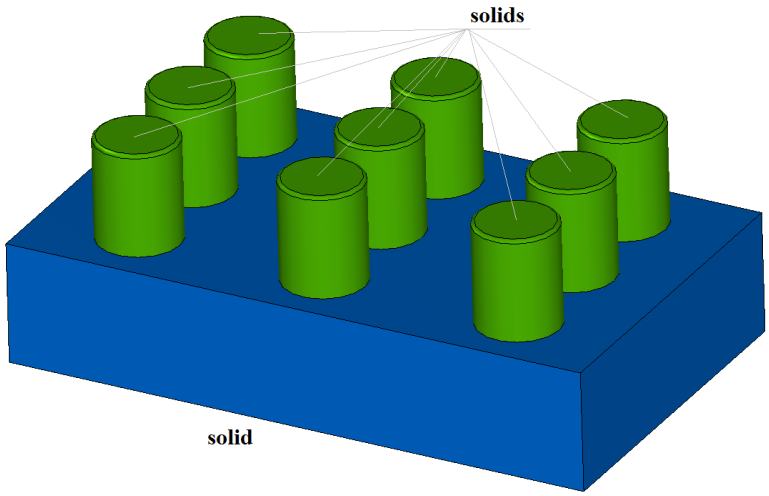


Fig. 1

Figure 2 shows the result **result** of gluing the bodies **solids** to the body **solid**. In this case, the *checkIntersect* parameter can be set to *false*, since the bodies **solids** do not intersect with each other:

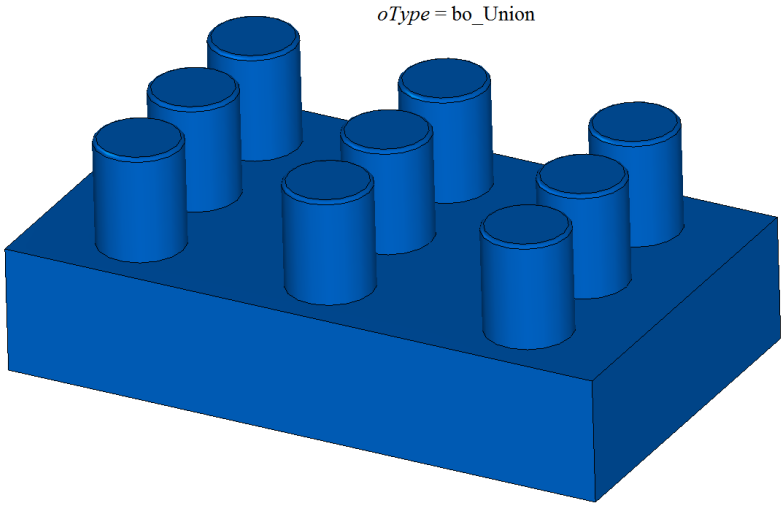


Fig. 2

Figure 3 shows the source body **solid** and the set of bodies **solids**:

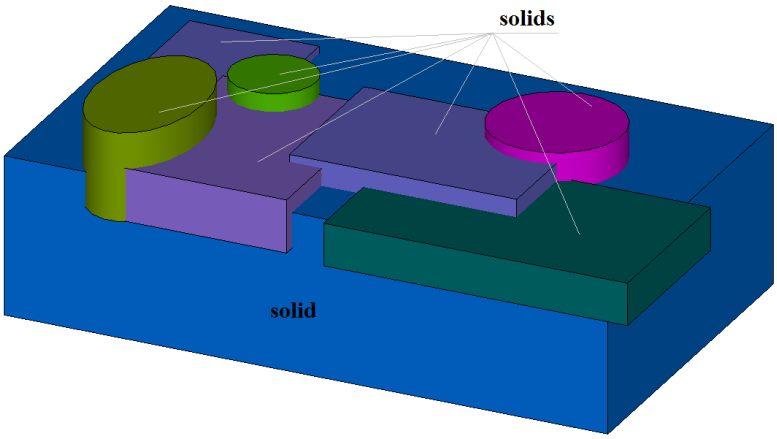


Fig. 3

Figure 4 shows the result **result** of subtracting the bodies **solids** from the body **solid** when using this function with the parameter **mergeFlags**.*mergeFaces* == *true*. In this case, the **checkIntersect** parameter must be set to *true*, since the bodies **solids** intersect with each other:

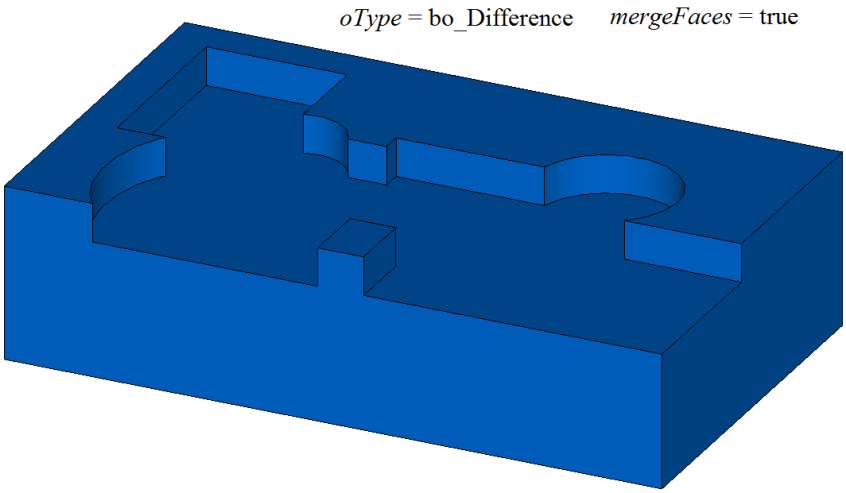


Fig. 4

Figure 5 shows the result **result** of subtracting the bodies **solids** from the body **solid** with the parameter **mergeFlags**.*mergeFaces* == *false*:

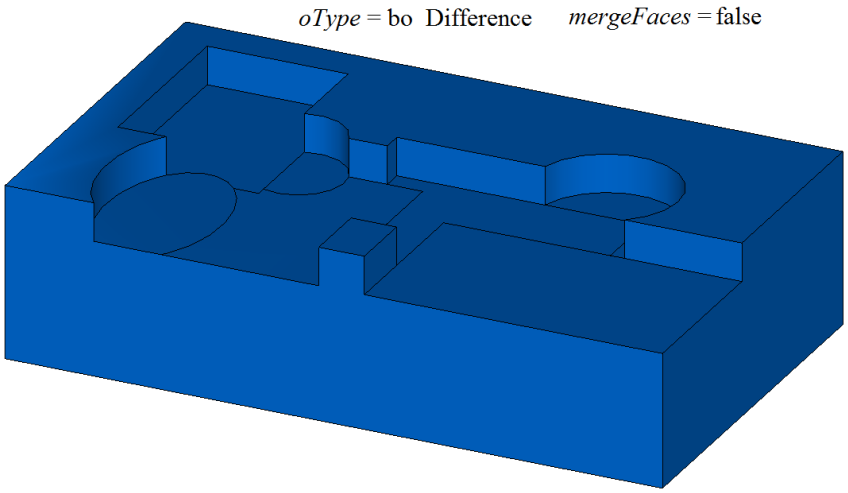


Fig. 5

Figure 6 shows, for clarity, the faces of the subtraction result of bodies **solids** from body **solid**, presented in Figure 3, colored according to the original bodies. The sequence of inclusion of bodies **solids** into the intermediate body can be determined from the face shapes: a body leaves a more complete imprint if it is included in the intermediate body earlier than the others:

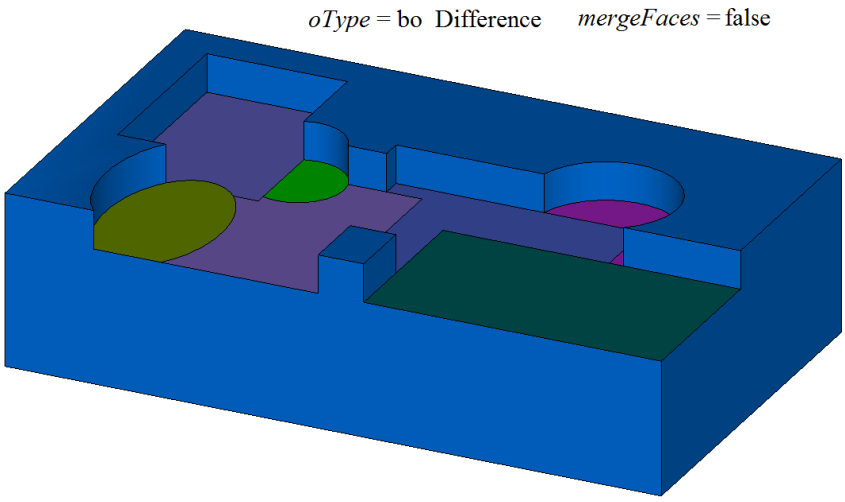


Fig. 6

The UnionResult() function adds to the construction log of the built body a builder MbUnionSolid that contains all the necessary data for performing the operation. The MbUnionSolid builder is declared in the file *cr_union_solid.h*.

In the test application *test.exe*, Boolean operations on a body with a set of bodies are performed by the menu commands:

- "Create → Solid → As union with a solid → Set of solids";
- "Create → Solid → As subtraction with a solid → Set of solids";
- "Create → Solid → As intersection with a solid → Set of solids".

3.2.14 Merging a Set of Bodies

The function

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

unites bodies from a given set into a single body.

The input parameters of the function are:

- **solids** — a set of bodies;
- **sameShells** — the copy mode for bodies in the set;
- **checkIntersect** — the flag that enables intersection checking for bodies in the set (*false* means no check);
- **names** — the name maker for the operation;
- **isArray** — the flag indicating whether the set of bodies is regular.

The output parameters of the function are the constructed body **result** and the set of bodies **notGluedSolids**, which were not used in the operation (may be null).

The function returns a result code from the enumeration MbResultType.

The function is declared in the file *action_solid.h*.

This function operates as the function UnionResult() with parameter values **solid** = 0, **sameShell** = *cm_Same*, **oType** = *bo_Union*, **mergeFlags**.*mergeFaces* = *true*. The function performs union of the set of bodies **solids** into a single body **result** even if the bodies do not intersect one another.

The parameter **sameShells** controls the transfer of faces, edges and vertices from bodies **solids** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy have no shared data).

The enumeration MbeCopyMode is described in the section *Copying a Set of Faces MbFaceShell*.

The parameter **checkIntersect** issues a command to check for intersection of bodies in the set **solids** with one another. If the parameter **checkIntersect** == *true*, then during result construction, a Boolean union operation is performed on all intersecting bodies in the set **solids**. Otherwise, the union of bodies from the given set is performed by simply transferring faces from all bodies into the constructed body. Regardless of the value of the parameter **checkIntersect**, all non-intersecting bodies in the set **solids** transfer their faces to a common intermediate body.

The parameter **isArray** works only when **checkIntersect** == *true* and indicates the regularity of the set of bodies **solids**. If **isArray** == *true*, then the bodies in the set are positioned at nodes of a rectangular or circular grid, and the body positions are specified in the face names.

The parameter **names** provides naming for the faces, edges and vertices of the constructed body.

The parameter **notGluedSolids** contains bodies that were not used in the operation due to the impossibility of merging them.

Figure 1 shows the source bodies **solids**:

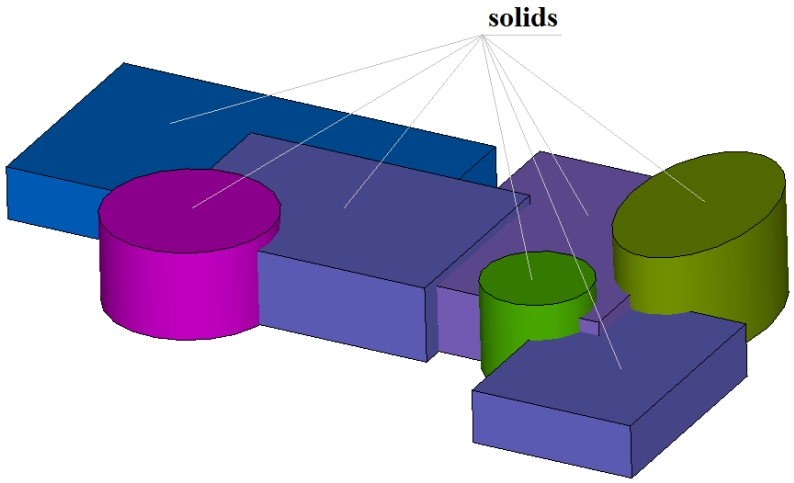


Fig. 1

Figure 2 shows the result **result** of the union of bodies **solids**. In this case, the parameter **checkIntersect** must be set equal to *true*, since the bodies in **solids** intersect with one another.

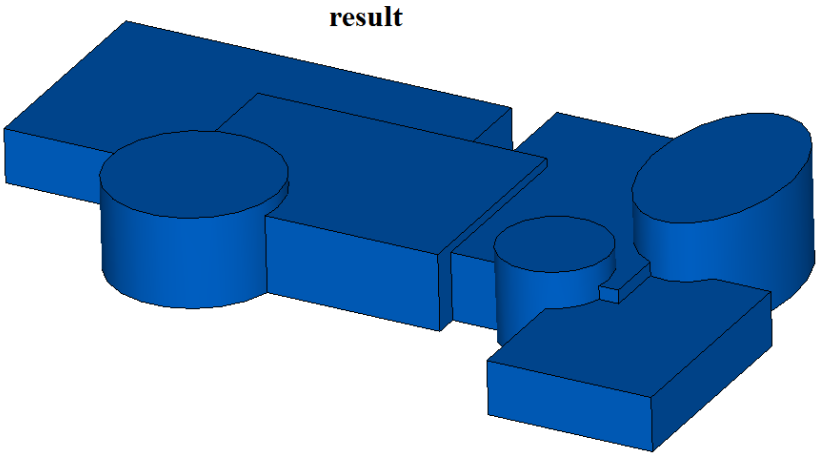


Fig. 2

The functions

```
MbResultType UnionSolid( const RPArray<MbSolid> & solids,
                         const MbSNameMaker &     names,
                               MbSolid *&         result ),
```

```
MbResultType UnionSolid( const c3d::SolidsVector & solids,
                         const MbSNameMaker &      names,
                               MbSolid *&          result ),
```

```
MbResultType UnionSolid( const c3d::SolidsSPtrVector & solids,
                         const MbSNameMaker &          names,
                               MbSolid *&              result )
```

are simplified variants of the previously discussed function of the same name, which unite bodies from a given set by simply transferring faces from all bodies into one new body. Thus, the constructed body and the source bodies share the same faces.

The functions UnionSolid() add a builder MbUnionSolid to the history log of the constructed body, which contains all necessary data to perform the operation. The builder MbUnionSolid is declared in the file *cr_union_solid.h*.

In the test application *test.exe*, the Boolean operation of a body with a set of bodies is performed by the menu command "Create → Solid → Based on a solid → Set of solids".

3.2.15 Splitting Body Faces

The function

```
MbResultType SplitSolid(       MbSolid &                    solid,
                               MbeCopyMode                  sameShell,
                         const RPArray<MbSpaceItem> &       spItems,
                               bool                         spSame,
                               RPArray<MbFace> &            selFaces,
                         const MbMergingFlags &             flags,
                         const MbSNameMaker &               names,
                               MbSolid *&                   result );
```

splits the specified body faces with spatial curves, surfaces and shells. The spatial splitting elements must not have full or partial overlaps and self-intersections.

The input parameters of the function are:

- **solid** - the source body;
- **sameShell** - a copy mode for the source body;
- **spItems** - spatial splitting elements of faces;
- **spSame** - the flag controlling use of spatial splitting elements;
- **selFaces** - the set of faces to be split;
- **flags** - flags controlling merging of shell elements;
- **names** - a name maker.

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

The function returns a result code from the *MbResultType* enumeration.

The function is declared in the file *action_solid.h*.

The function splits the specified faces **selFaces** of the source body **solid** with spatial objects **spItems**, if the specified faces intersect these objects. The objects in **spItems** may be curves, surfaces, or a solid.

For successful operation, cutting objects must fully intersect the specified faces of the source body.

Parameter **sameShell** controls the transfer of faces, edges and vertices from the source body **solid** to the constructed body **result** and may take one of four values:

- *cm_Same = 0* (shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy have no shared data).

Enumeration MbeCopyMode is described in section *Copying a Set of Faces MbFaceShell*.

The parameter **spSame** controls copying of spatial splitting elements (*true* – use originals, *false* – use copies).

Parameter **flags** controls merging of shell elements after splitting:

- **flags**.*mergeFaces* = *true* – merge geometrically similar faces arising as a result of splitting;
- **flags**.*mergeEdges* = *true* – merge geometrically similar edges.

The parameter **names** provides naming of the faces, edges, and vertices of the constructed body.

Fig. 1 shows the source body **solid**, the faces for splitting **selFaces**, and the cutting surface **spItems**[0].

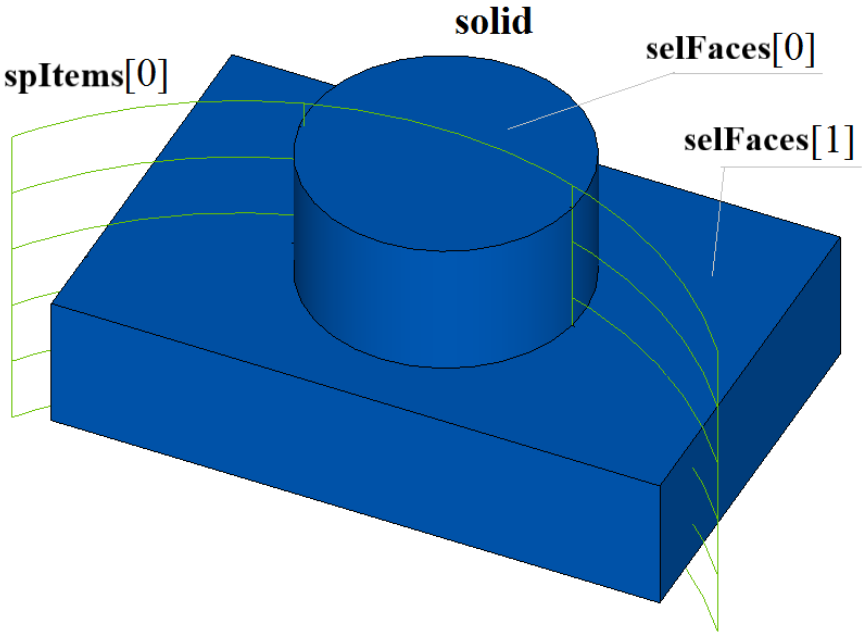


Fig. 1

Fig. 2 shows the constructed body **result** with faces split into parts. New edges return *true* when queried with MbEdge::IsSplit().

In Fig. 3, the split faces of the constructed body are painted in different colors:

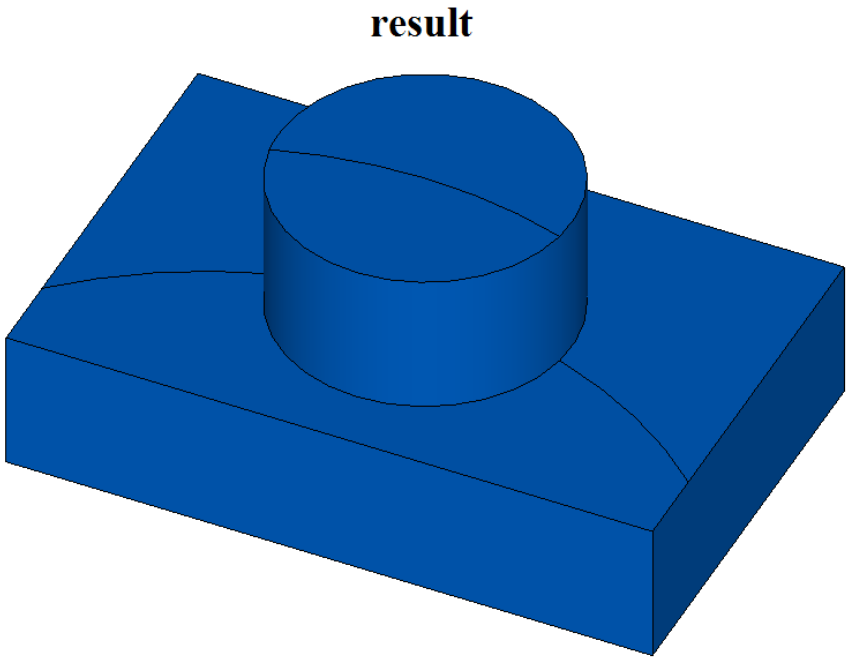


Fig. 2

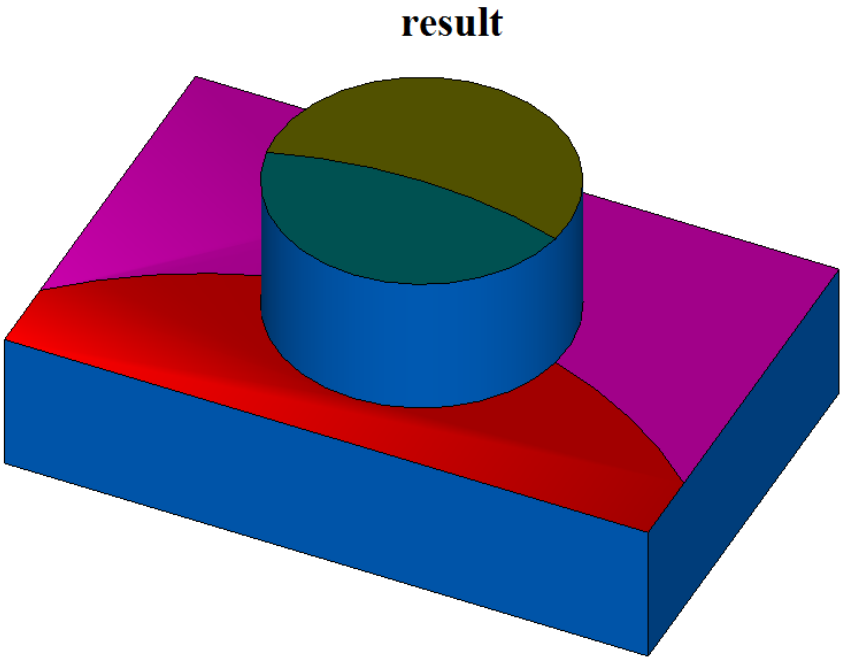


Fig. 3

The function

```
MbResultType SplitSolid(       MbSolid &                    solid,
                               MbeCopyMode                  sameShell,
                         const MbPlacement3D &              spPlace,
                               MbeSenseValue                spType,
                         const RPArray<MbContour> &         spContours,
                               bool                         spSame,
                               RPArray<MbFace> &            selFaces,
                         const MbMergingFlags &             flags,
                         const MbSNameMaker &               names,
                               MbSolid *&                   result );
```

performs the same operations as the function considered above with the difference that instead of splitting objects **spItems**, the splitting of the body **solid** faces is performed by surfaces constructed by extrusion of two-dimensional contours **spContours** located in the XY plane of the local coordinate system **spPlace**. Extrusion of contours is performed in the direction of the **spPlace**.*axisZ* axis of the local coordinate system for a length sufficient to fully intersect the source body.

Functions SplitSolid() add to the history log of the constructed body the builder MbSplitShell, which contains all necessary data for performing the operation. The builder MbSplitShell is declared in the file *cr_split_shell.h*.

In the test application *test.exe*, splitting of the specified body faces is performed by the menu command «Create → Solid → By faces processing → Face splitting».

3.2.16 Constructing a Hole, Pocket or Slot in a Body

The function

```
MbResultType HoleSolid( const c3d::SolidSPtr &          solid,
                              MbeCopyMode               sameShell,
                        const MbHoleSolidParams &       params,
                              c3d::SolidSPtr &          result );
```

constructs a hole, pocket or profiled slot in a body.

The input parameters of the function are:

- **solid** - the source body (may be null);
- **sameShell** - the body copying mode;
- **params** - construction parameters.

To initialize the **params** object, the following constructor is used (see *op_shell_parameter.h*):

```
MbHoleSolidParams( const MbPlacement3D &   place,
                   const HoleValues &      params,
                   const MbSNameMaker &    operNames )
: _place     ( place                   )
, _parameters( params                  )
, _operNames ( &operNames.Duplicate() ){}
```

where

- **place** - the local coordinate system;
- **params** - construction parameters;

- **operNames** - the name maker.

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

The function returns an operation result code from the *MbResultType* enumeration.

The function is declared in the file *action_solid.h*.

The function constructs an auxiliary body in the form of a volume to be removed for a hole, pocket, or slot.

If the source body **solid** is specified, the function returns the difference between the source body and the auxiliary body.

If the source body is not specified (**solid** = 0), the function returns the auxiliary body (drill, boss).

For successful operation, the auxiliary body must intersect the source body.

The parameter **sameShell** controls the transfer of faces, edges and vertices from the source body **solid** to the constructed body **result** and can take one of four values:

- *cm_Same = 0* (the shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy have no common data).

The MbeCopyMode enumeration is described in the section *Copying a Set of Faces MbFaceShell*.

The parameter **params** defines the shape of the hole, pocket, or slot.

Construction is performed in the local coordinate system **place** with consideration of the rotation angles **params**.*placeAngle* and **params**.*azimuthAngle*.

The parameter **params**.*placeAngle* defines the rotation angle of the local coordinate system **place** relative to the axis **place**.**axisY**.

The parameter **params**.*azimuthAngle* defines the rotation angle of the local coordinate system **place** relative to the axis **place**.**axisZ**.

The surface **params**.*surface* may not be specified. If **params**.*surface* is not null, it is used for proper processing of the entrance to the hole, pocket, or slot.

Fig. 1 shows the data used in construction and the inheritance scheme of construction parameters from the abstract class HoleValues.

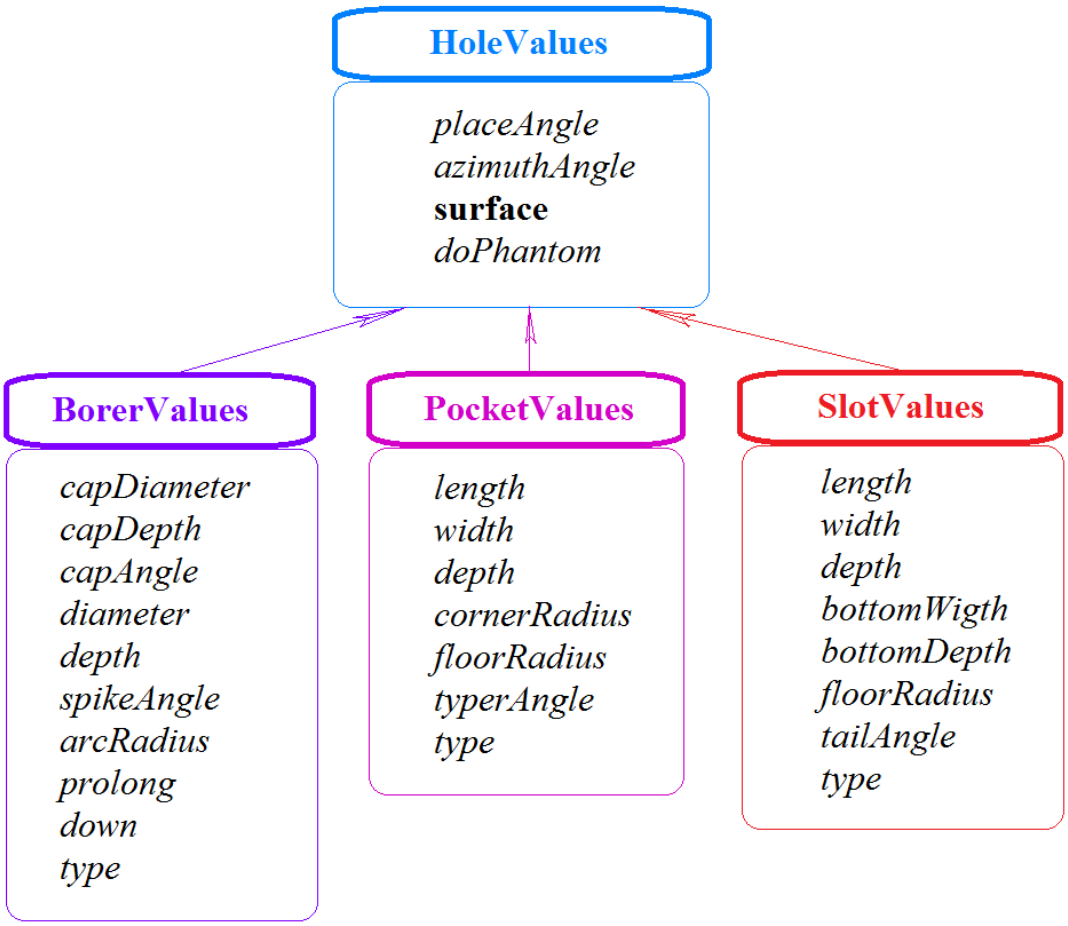
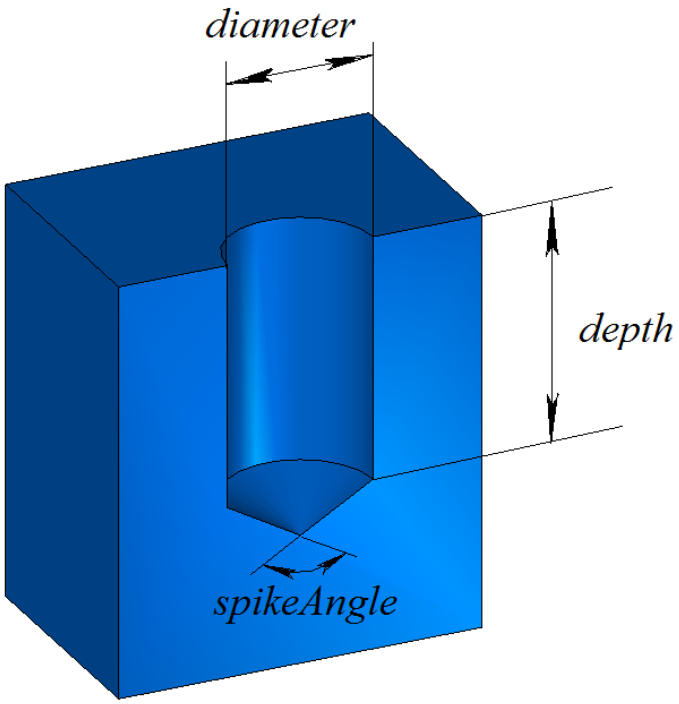


Fig. 1

To construct a hole, the parameters BorerValues should be used. The following hole types are supported, which are determined by the parameter BorerValues::*type*, taking the values:

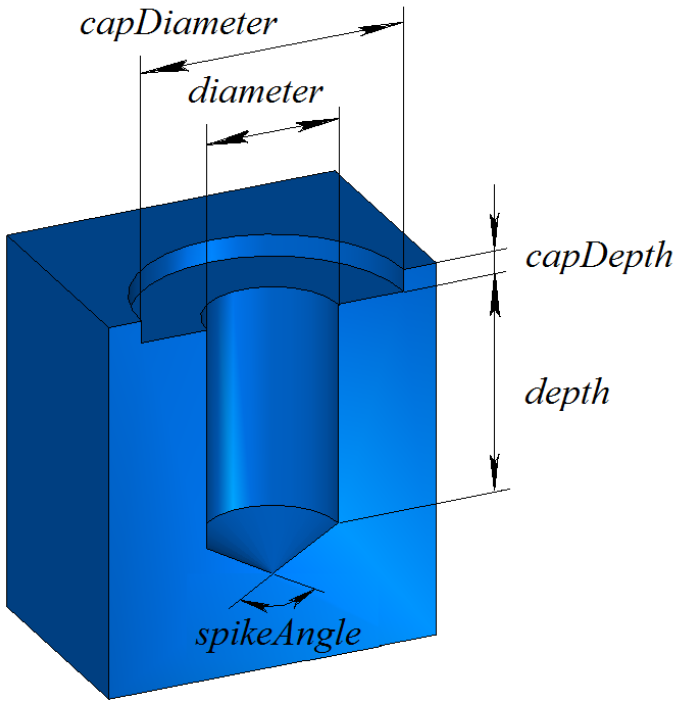
- bt_SimpleCylinder;
- bt_TwofoldCylinder;
- bt_ChamferCylinder;
- bt_ComplexCylinder;
- bt_SimpleCone;
- bt_ArcCylinder;
- bt_DoubleCylinder.

Figs. 2...8 show bodies with holes of various shapes:



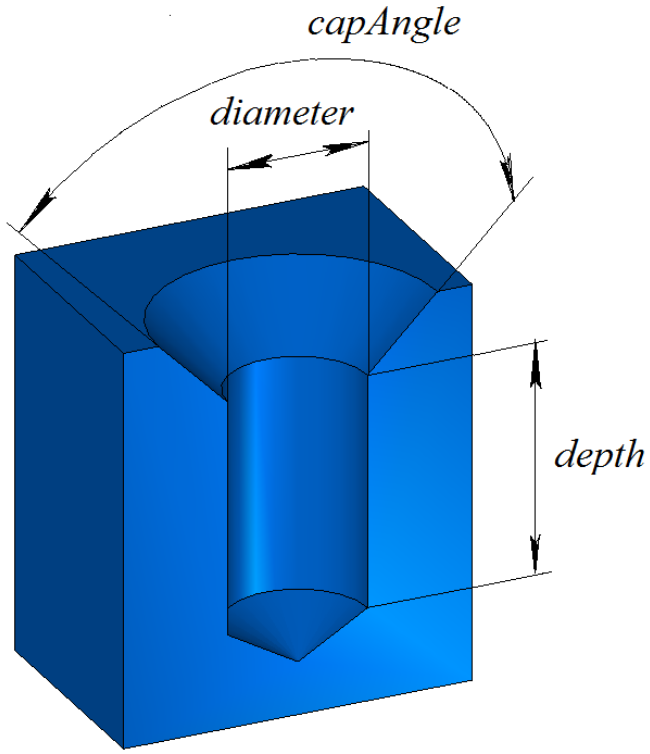
BorerValues::*type* = bt_SimpleCylinder

Fig. 2



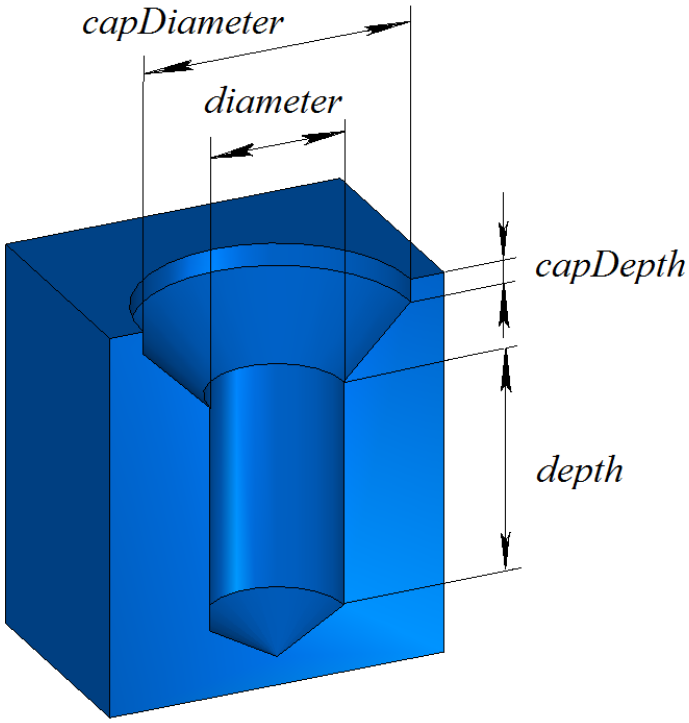
BorerValues::*type* = bt_TwofoldCylinder

Fig. 3



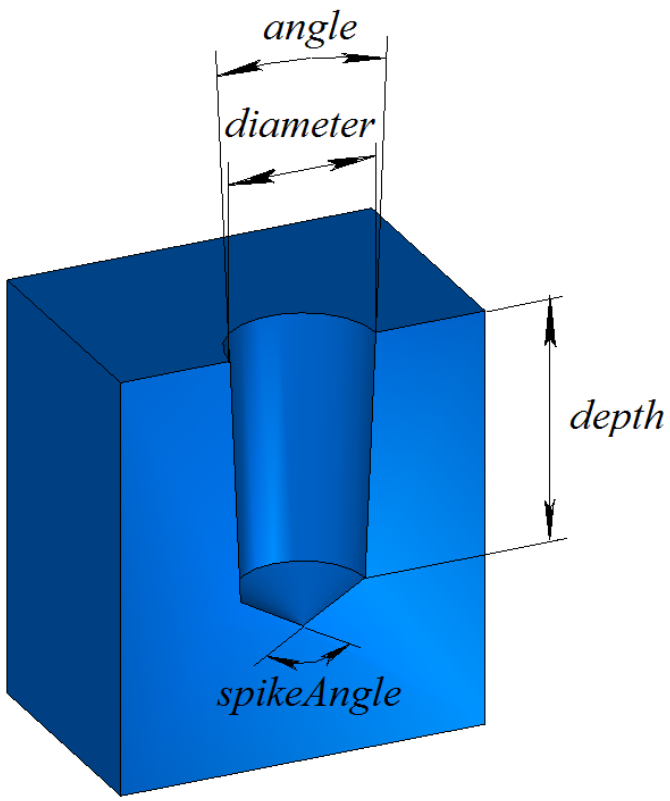
BorerValues::*type* = bt_ChamferCylinder

Fig. 4



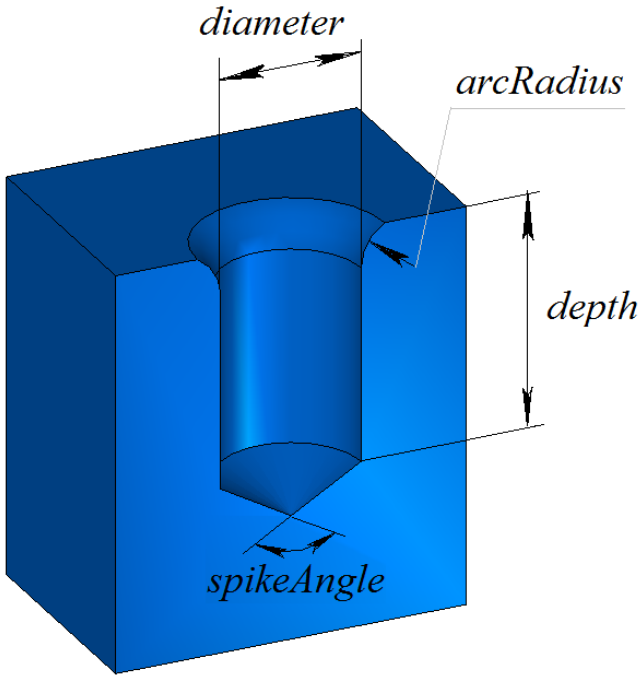
BorerValues::*type* = bt_ComplexCylinder

Fig. 5



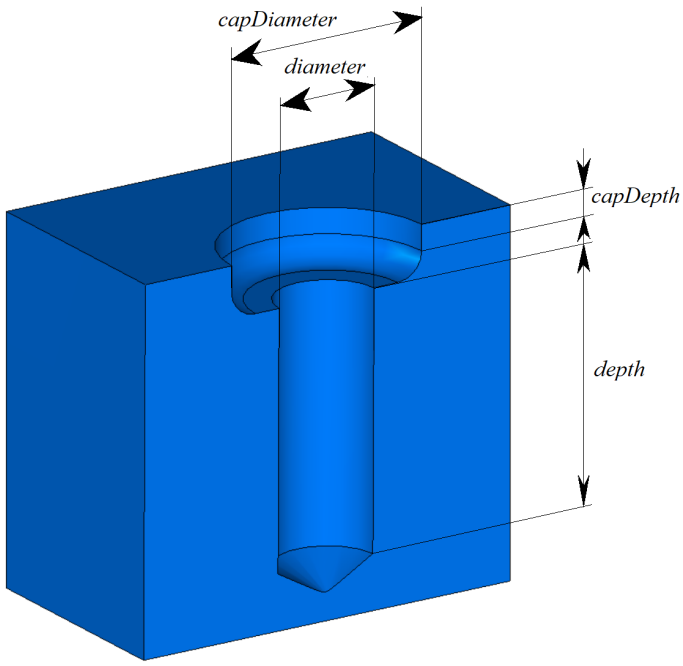
BorerValues::*type* = bt_SimpleCone

Fig. 6



BorerValues::*type* = bt_ArcCylinder

Fig. 7



BorerValues::*type* = bt_DoubleCylinder

Fig. 8

To construct a pocket or boss, the parameters PocketValues should be used.

When PocketValues::*type* = *false*, a pocket is constructed according to the specified parameters **params**; when PocketValues::*type* = *true*, a boss is constructed according to the specified parameters **params**.

Fig. 9 shows a body with a rectangular pocket without a draft angle on the side faces.

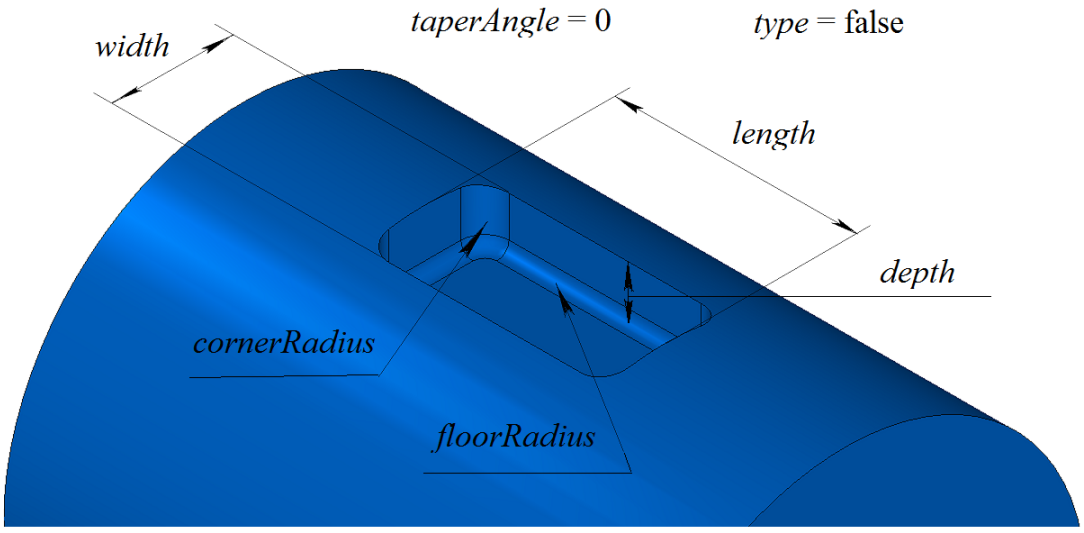


Fig. 9

To construct a slot, the parameters SlotValues should be used. Four types of slots are supported, which are determined by the parameter SlotValues::*type*:

- st_BallEnd - cylindrical at the bottom;
- st_Rectangular - rectangular;
- st_TShaped - T-shaped;
- st_DoveTail - dovetail.

The function HoleSolid() adds a builder MbHoleSolid to the log of the constructed body, which contains all necessary data for performing the operation. The builder MbHoleSolid is declared in the file *cr_hole_solid.h*.

In the test application *test.exe*, construction of holes, pockets, or slots in a body is performed by the menu commands:

- "Create → Solid → Based on a solid → With a hole";
- "Create → Solid → Based on a solid → With a pocket or a boss";
- "Create → Solid → Based on a solid → With a slot".

3.2.17 Constructing a Body with an Enforcement Rib

The function

```
MbResultType RibSolid(       MbSolid &              solid,
                             MbeCopyMode            sameShell,
                       const MbRibSolidParameters & pars,
                             c3d::SolidSPtr &       result );
```

constructs an enforcement rib along a given contour and merges it with the source body. The contour segment with the specified index determines the slope vector for the rib side faces.

The input parameters of the function are:

- **solid** – the source body;
- **sameShell** – the source body copying mode;
- **pars** – enforcement rib parameters.

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

The function returns an operation result code from the MbResultType enumeration.

The function is declared in the file *action_solid.h*.

The parameter **sameShell** controls the transfer of faces, edges, and vertices from the source body **solid** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the source shell is not copied);
- *cm_KeepHistory* (the source shell and its copy share common base surfaces, vertices, and faces unchanged by the operation);
- *cm_KeepSurface* (the source shell and its copy share common base surfaces);
- *cm_Copy* (the source shell and its copy have no common data).

The MbeCopyMode enumeration is described in the section *Copying a Set of Faces MbFaceShell*.

To initialize the **pars** object, the following constructor is used (see *op_shell_parameter.h*):

```
MbRibSolidParameters( const MbPlacement3D & place,
                      const MbContour &     contour,
                            size_t          index,
                      const RibValues &     pars,
                      const MbSNameMaker &  names );
```

where

- **place** – the local coordinate system whose XY plane contains the generatrix contour. The Z-axis of this system defines the primary extrusion direction of the rib (before applying the *side* parameter and slope angles);
- **contour** – the shape-forming contour in the XY plane of the coordinate system **place**;
- **index** – the contour segment number (starting from 0) that defines the slope vector of the enforcement rib lateral faces. The direction of this segment's normal sets the slope direction. If an invalid segment number is specified (beyond the number of contour segments), the function returns an error code;
- **pars** – the enforcement rib parameters;
- **names** – the name maker of the operation.

The object **pars** of class RibValues contains the data shown in Figure 1.

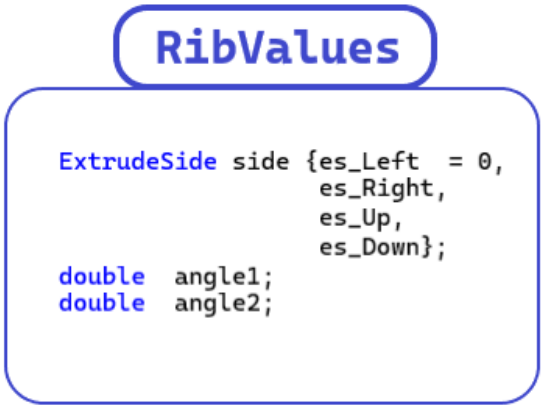


Figure 1

The RibValues structure is defined in the file *swept_parameter.h*.

The parameter **pars**.*side* determines the side toward which the rib is extruded:

1. When **pars**.*side* = *es_Left*, the enforcement rib is extruded to the left of the curve along the XY plane;
2. When **pars**.*side* = *es_Right*, the enforcement rib is extruded to the right of the curve along the XY plane;
3. When **pars**.*side* = *es_Up*, the enforcement rib is extruded in the direction of the XY plane normal;
4. When **pars**.*side* = *es_Down*, the enforcement rib is extruded in the direction opposite to the XY plane normal.

The parameters **pars**.*angle1* and **pars**.*angle2* determine the draft angles of the rib plane in the forward and reverse directions, respectively.

Figure 2 shows the source body **solid**, the local coordinate system **place**, and the contour **contour** located in the XY plane of **place**:

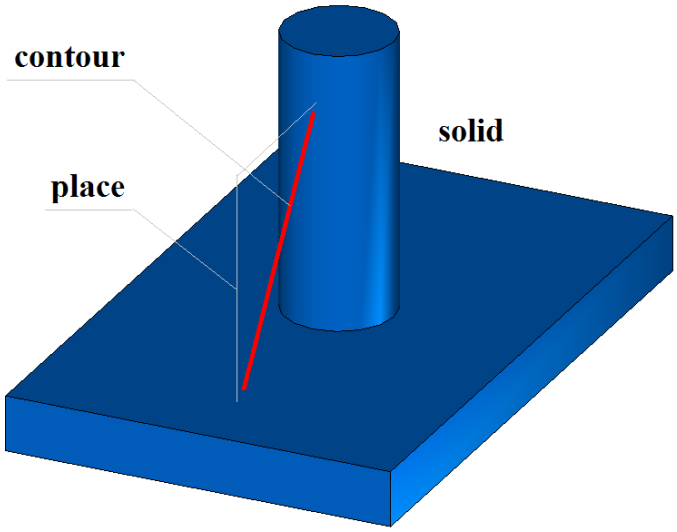


Figure 2

Figure 3 shows the result of constructing an enforcement rib without a draft angle on the rib side faces:

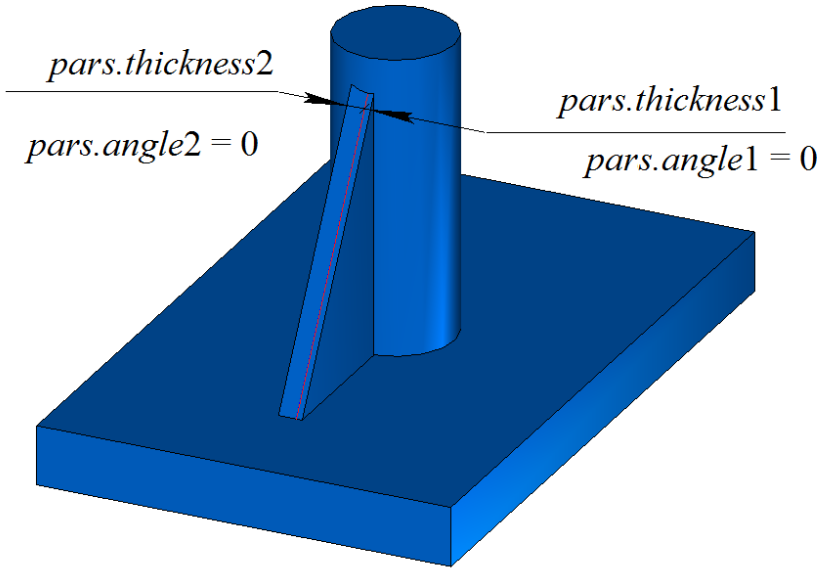


Figure 3

Figure 4 shows the result of constructing an enforcement rib with a draft angle on the rib side faces:

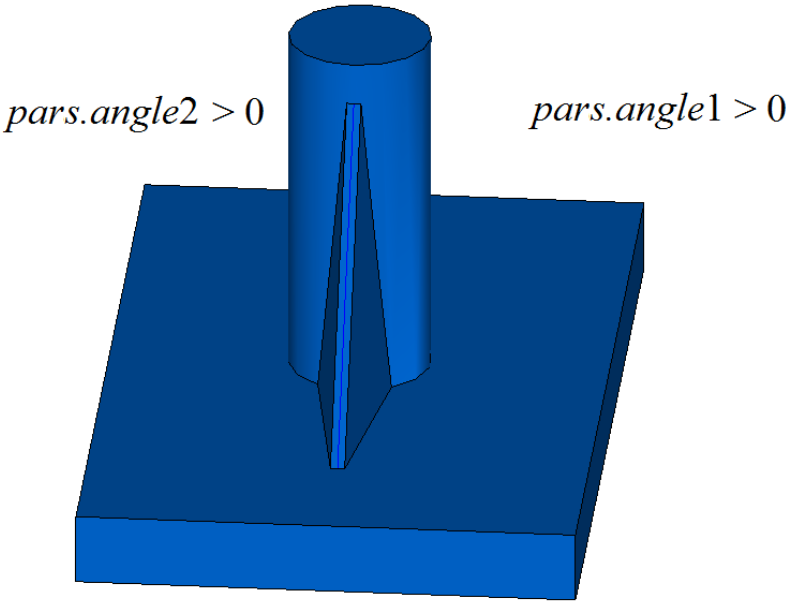


Figure 4

The function RibSolid() adds the builder MbRibSolid to the log of the constructed body, which contains all necessary data for performing the operation. The MbRibSolid builder is declared in the file *cr_rib_solid.h*.

For creating a separate rib without union with the original body, use the RibElement() function. It has parameters analogous to those of RibSolid(), but returns only the enforcement rib body without the Boolean union operation. This is useful when you need to process the rib separately before integrating it into the model.

In the test application *test.exe*, the construction of enforcement ribs in a body is performed by the menu command “Create → Solid → Based on a solid → With a rib”.

3.2.18 Face Body Sloping

```
MbResultType DraftSolid(       MbSolid &              solid,
                               MbeCopyMode            sameShell,
                         const MbDraftSolidParams &   draftParams,
                               MbDraftSolidResults &  result );
```

The function DraftSolid() constructs a body with sloping of specified body faces from a neutral plane or neutral edges at a given angle.

The input parameters of the function are:

- **solid** - the source body;
- **sameShell** - a copy mode for the source body;
- **draftParams** - draft parameters.

The output parameter of the function is the **result** object of the MbDraftSolidResults class, which contains the constructed draft body and a hot point attribute of the draft.

The function returns an operation result code from the *MbResultType* enumeration.

The function is declared in the file *action_solid.h*.

The function constructs a body whose faces are sloped relative to their position in the source body **solid**.

The parameter **sameShell** controls the transfer of faces, edges and vertices from the source body **solid** to the constructed body **result** and can take one of four values:

- *cm_Same* = 0 (the source shell is not copied).
- *cm_KeepHistory* (the source shell and its copy have common base surfaces, vertices, and unchanged operation faces).
- *cm_KeepSurface* (the source shell and its copy have common base surfaces).
- *cm_Copy* (the source shell and its copy have no common data).

The enumeration MbeCopyMode is described in detail in section *Copying a Set of Faces MbFaceShell*.

The parameter **draftParams** specifies the construction parameters.

In the class MbDraftSolidParams, to initialize the object **draftParams** with sloping parameters, the following constructors are used (see *op_shell_parameter.h*):

Constructors by a neutral plane and an operation namer:

```
MbDraftSolidParams( const c3d::FacesSPtrVector &  faces,
                    const MbPlacement3D &         neutralPlane,
                    const MbSNameMaker &          names,
                          double                  angle,
                          MbeFacePropagation      faceProp,
                          bool                    reverse );
```

```
MbDraftSolidParams( const c3d::FacesSPtrVector &  faces,
                    const MbPlacement3D &         neutralPlane,
                    const MbSNameMaker &          names,
                          double                  angle,
                          MbeFacePropagation      faceProp,
                          bool                    reverse,
                          bool                    rebuildFillets );
```

```
MbDraftSolidParams( const RPArray<MbFace> &       faces,
                    const MbPlacement3D &         neutralPlane,
                    const MbSNameMaker &          names,
                          double                  angle,
                          MbeFacePropagation      faceProp,
                          bool                    reverse );
```

where

- **faces** - sloped faces;
- **neutralPlane** - a neutral plane;
- **names** - a name maker;

- **angle** - a sloping angle;
- **faceProp** - a flag for capturing faces that smoothly join with sloped faces;
- **reverse** - a flag for reverse sloping direction.

Constructors for edge set and an operation namer:

```
MbDraftSolidParams( const c3d::FacesSPtrVector &  faces,
                    const c3d::EdgesSPtrVector &  edges,
                    const MbVector3D &            dir,
                    const MbSNameMaker &          names,
                          double                  angle,
                          MbeFacePropagation      faceProp,
                          bool                    reverse );
```

```
MbDraftSolidParams( const c3d::FacesSPtrVector &  faces,
                    const c3d::EdgesSPtrVector &  edges,
                    const MbVector3D &            dir,
                    const MbSNameMaker &          names,
                          double                  angle,
                          MbeFacePropagation      faceProp,
                          bool                    reverse,
                          bool                    rebuildFillets );
```

where

- **faces** - sloped faces;
- **edges** - neutral edges;
- **dir** - a base direction;
- **names** - a namer;
- **angle** - a sloping angle;
- **faceProp** - a flag for capturing faces that smoothly join with sloped faces;
- **reverse** - a flag for reverse sloping direction;
- **rebuildFillets** - a flag for rebuilding adjacent fillets.

The parameter **angle** specifies the draft angle of the specified body faces from the set **faces** relative to the neutral plane **neutralPlane** or relative to the neutral edges **edges** and the direction **dir** (depending on the constructor used).

The parameter **faceProp** controls the addition to the set of sloped faces **faces** of other body faces that smoothly join with the sloped faces.

The parameter **reverse** controls the sloping direction.

The parameter **rebuildFillets** controls the rebuilding of adjacent fillets.

The parameter **names** provides naming of faces, edges and vertices of the constructed body.

Figure 1 shows the source body **solid**, the neutral plane **neutralPlane** relative to whose XY plane the face sloping is performed, and the sloped faces **faces**:

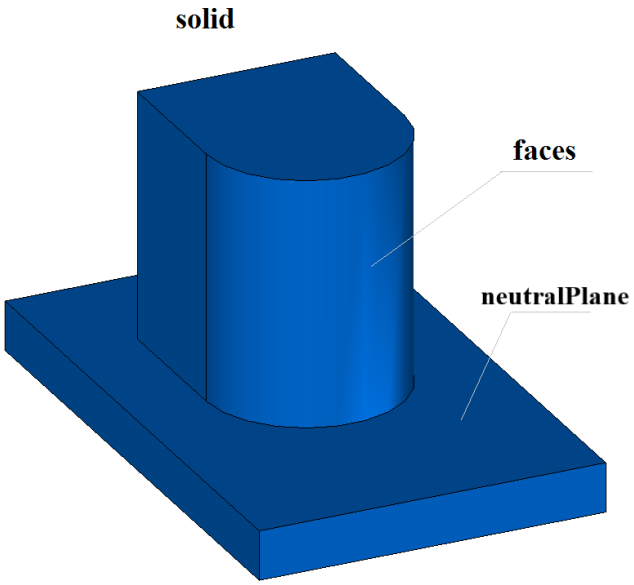


Fig. 1

Figure 2 shows the result of constructing a body with a slope of the specified faces:

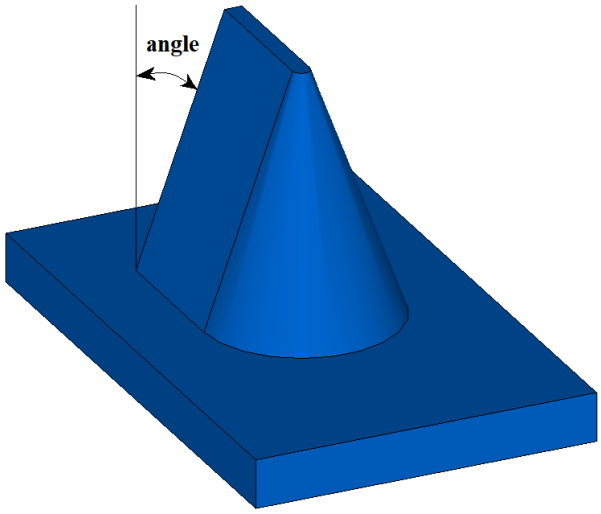


Fig. 2

The function adds the builder MbDraftSolid to the construction history of the resulting body. The builder MbDraftSolid contains all the data required to perform the operation. The builder MbDraftSolid is declared in the file *cr_draft_solid.h*.

In the test application *test.exe* the sloping of specified body faces is performed by the menu command “Create→Solid→By faces processing→Faces drafting”.

3.2.19 Multiplication of Body

The function

```
MbResultType DuplicationSolid( const MbSolid &                    solid,
                               const MbDuplicationSolidParams &   params,
                                     c3d::SolidSPtr &             result );
```

creates copies of an original body, transforms them according to the specified law, and combines them into a single body.

The function input parameters are:

- **solid** - the original body;
- **params** - construction parameters.

In the MbDuplicationSolidParams class, the following constructors are used to initialize the **params** object with construction parameters (see *op_shell_parameter.h*):

```
MbDuplicationSolidParams( const DPtr<DuplicationValues> &  values,
                                IProgressIndicator *       prog = nullptr )

_duplicationValues ( values )
, _selIndices        (        )
, _adaptMode         ( false  )
, _names             (        )
, _progress          ( prog   )
{
   if ( _duplicationValues == nullptr )
   {
      C3D_ASSERT( _duplicationValues != nullptr );
      _duplicationValues = new DuplicationMatrixValues();
   }
}
```

```
MbDuplicationSolidParams( const DPtr<DuplicationValues> &  values,
                          const MbSNameMaker &             names,
                                IProgressIndicator *       prog = nullptr )

_duplicationValues ( values )
, _selIndices        (        )
, _adaptMode         ( false  )
, _names             (        )
, _progress          ( prog   )
{
   if ( _duplicationValues == nullptr )
   {
      C3D_ASSERT( _duplicationValues != nullptr );
      _duplicationValues = new DuplicationMatrixValues();
   }
}
```

Constructor based on the DuplicationMeshValues class for copying along a grid:

```
MbDuplicationSolidParams(       bool                  isPolar,
                          const MbVector3D &          dir1,
                          const double                step1,
                          const uint                  num1,
                          const MbVector3D &          dir2,
                          const double                step2,
                          const uint                  num2,
                          const MbCartPoint3D *       center,
                                bool                  isAlongAxis,
                                IProgressIndicator *  prog = nullptr )

_duplicationValues( new DuplicationMeshValues( isPolar, dir1, step1, num1, dir2,␣
↪step2, num2, center, isAlongAxis ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

Constructors based on the DuplicationMatrixValues class for copying along matrices:

```
MbDuplicationSolidParams( const MbMatrix3D &          matr,
                                IProgressIndicator *  prog = nullptr )

_duplicationValues( new DuplicationMatrixValues( matr ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

```
MbDuplicationSolidParams( const SArray<MbMatrix3D> &  matr,
                                IProgressIndicator *  prog = nullptr )

_duplicationValues( new DuplicationMatrixValues( matr ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

```
MbDuplicationSolidParams( const std::vector<MbMatrix3D> &  matr,
                                IProgressIndicator *       prog = nullptr )

: _duplicationValues( new DuplicationMatrixValues( matr ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

where

- **_duplicationValues** - duplication parameter object;

- **_selIndices** - indices of selected shell faces;
- **_adaptMode** - adaptation mode for open shells;
- **_names** - name maker for bodies and faces created by the operation;
- **IProgressIndicator * _progress** - operation progress indicator.

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

The function returns an operation result code from the *MbResultType* enumeration.

The function is declared in the file *action_solid.h*.

The **_names** parameter provides naming for the faces, edges and vertices of the constructed body.

The **values** parameter specifies the construction parameters. Figure 1 shows the data used in the construction and the inheritance scheme of construction parameters from the abstract class DuplicationValues:

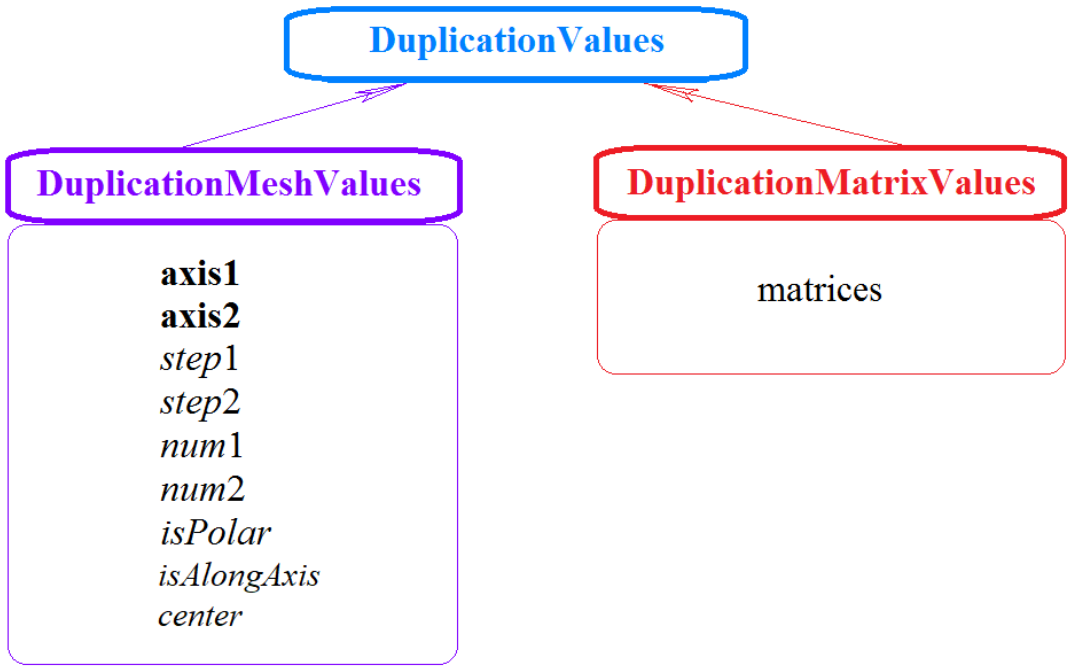


Fig. 1

To duplicate a body whose copies are arranged along a two-dimensional grid, use the DuplicationMeshValues parameters.

Two types of duplication are supported: along two directions and along a polar grid.

The **values**.*isPolar* parameter determines the grid type.

axis1 and **axis2** are vectors in the global coordinate system that define the directions of the grid (either linear or polar).

If **values**.*isPolar* = *false*, the original body and constructed copies are arranged at the nodes of a two-dimensional grid with the axes **values**.*axis1* and **values**.*axis2*. The original body serves as the reference point. Along the **values**.*axis1* axis, **values**.*num1* copies of the body are arranged with step **values**.*step1*; along the **values**.*axis2* axis, **values**.*num2* copies of the body are arranged with step **values**.*step2*, including the original body.

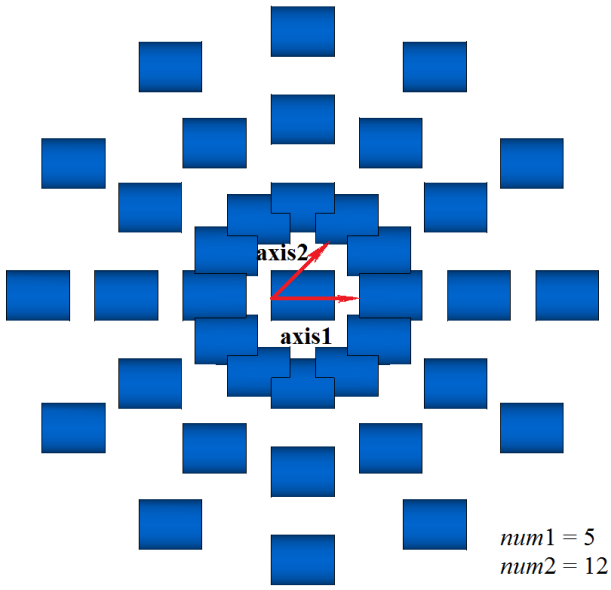
If **values**.*isPolar* = *true*, the constructed copies are arranged at the nodes of a polar grid. The original body serves as the origin. The radial direction of the grid is determined by the **values**.*axis1* vector, and the rotation axis is determined by the cross product of **values**.*axis1* and **values**.*axis2*. Along the radial directions, **values**.*num1* copies of the body are arranged with step **values**.*step1*; on each circle, **values**.*num2* copies of the body are arranged with angular step **values**.*step2*.

If **values**.*isAlongAxis* = *true*, the bodies are arranged along the radial axis. If **values**.*isAlongAxis* = *false*, the bodies are arranged parallel to the original body.

The **values**.*center* parameter determines the center of the local coordinate system (as well as the application point of the rotation axis in the case of a polar system).

To duplicate a body whose copies are transformed according to a set of matrices, use the DuplicationMatrixValues parameters. The set of transformation matrices is defined by the **values**.*matrices* parameter.

If, after construction, the copies and the original body intersect with each other, a Boolean union operation is performed on the intersecting bodies. Figure 2 shows the result of constructing a body duplicated along a polar grid:



DuplicationMeshValues::*isPolar* = true

Fig. 2

The DuplicationSolid() function adds the builder MbDuplicationSolid to the construction log of

the body, which contains all the necessary data for executing the operation. The MbDuplicationSolid builder is declared in the file *cr_duplication_solid.h*.

In the test application *test.exe*, duplication of the body is performed using the menu commands “Create→Solid→Based on a solid→Duplication by grid” and “Create→Solid→Based on a solid→Duplication by matrices”.

3.2.20 Dividing a shell into parts using a given set of edges

The function

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

divides the shell contained in **solid** using a specified set of edges defined in the parameters. The division of the solid/shell is performed by removing topological relationships between adjacent faces at the specified edges. After the operation completes, the entire result is placed in the resulting shell.

Input parameters:

- **solid** - the source solid (shell) to be divided;
- **sameShell** - the copy mode for the source solid (see description of MbeCopyMode);
- **params** - operation parameters for shell division (set of edges and additional settings).

The output parameter is the returned solid **resSolid**, which contains the resulting shell.

The function returns an operation result code from the *MbResultType* enumeration.

The function is declared in the file *action_shell.h*.

The MbDivideShellParams class contains the input data for executing the shell division operation. When creating an instance of the MbDivideShellParams class, it can be immediately initialized with a vector of edges along which the division will be performed.

Constructors of the MbDivideShellParams class:

```
MbDivideShellParams( const RPArray<MbCurveEdge> &   edges,
                     const MbSNameMaker &           operNames );
```

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

```
MbDivideShellParams( const MbSNameMaker &           operNames );
```

where

- **edges** - a vector of edges along which division will be performed;
- **operNames** - a name maker.

As a result of successful execution, the source solid **solid** will be divided and all resulting open solids will be placed in the shell of the returned solid **resSolid**.

The test application *test.exe* performs division of specified body faces using the menu commands “Create→Shell→Based on a shell→Dividing by a group of edges”.

3.3 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 section *TWO-DIMENSIONAL CURVES* . A curve can be constructed by direct call of corresponding constructor or using the functions described in this section.

3.3.1 Constructing a Two-Dimensional Straight Line/Segment

The function

```
MbResultType Line( const MbCartPoint &   point1,
                   const MbCartPoint &   point2,
                         MbCurve *&      result );
```

constructs a two-dimensional straight line based on two non-matching points.

Input parameters of the function are as follows:

- **point1** is the first point that lies on the straight line;
- **point2** is the second point that lies on the straight line.

The output parameter of the function is the constructed straight line **result**.

If successful, the function returns rt_Success, otherwise it returns an error code from *MbResultType*.

The function is declared in *action_curve.h* file.

The **point1** parameter defines the start point of the straight line that corresponds to zero parameter value. The tangent vector (normalized derivative) of the straight line has unit length and is directed from **point1** towards **point2**, Fig. 1:

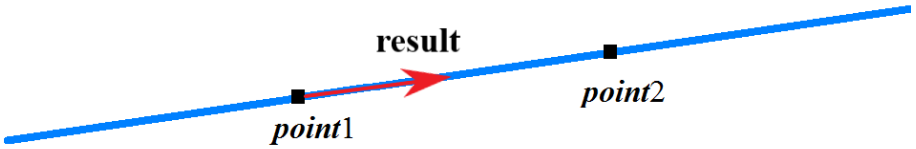


Fig. 1

Straight line is described in subsection *MbLine_Two-Dimensional_Straight_Line*.

The function

```
MbResultType Segment( const MbCartPoint &   point1,
                      const MbCartPoint &   point2,
                            MbCurve *&      result );
```

constructs a two-dimensional straight line segment based on two non-matching points.

Input parameters of the function are as follows:

- **point1** is the segment starting point;

- **point2** is the segment end point.

The output parameter of the function is the constructed line segment **result**.

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

The function is declared in *action_curve.h* file.

point1 parameter defines the starting point of the line segment, it corresponds to zero value of curve parameter. **point2** parameter defines the end point of the line segment, it corresponds to curve parameter equal to one, see Fig. 2:

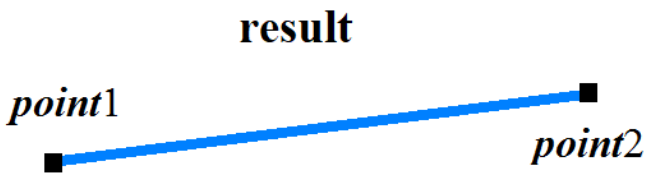


Fig. 2

Line segment is described in subsection *MbLineSegment Two-Dimensional Straight Line Segment*.

3.3.2 Constructing a Two-Dimensional Circle, Ellipse and their Arcs

The function

```
MbResultType Arc(       MbeArcCreateWay           createWay,
                 const  MbCartPoint &             center,
                 const  c3d::ParamPointsVector &  points,
                        double &                  a,
                        double &                  b,
                        double &                  c,
                        bool                      option,
                        MbCurve *&                result );
```

constructs a two-dimensional elliptic arc. In a special case, it constructs a two-dimensional circular arc.

The function input parameters are:

- **createWay** - the method for creating an ellipse (circle) (see the enum MbeArcCreateWay):
  - *acw_CircleByCenterAndRadius* - a circle by center and radius,
  - *acw_ArcByCenterAnd2Points* - a circular arc by center and two points,
  - *acw_ArcByCenterAnd2Angles* - a circular arc by center and two angles,
  - *acw_ArcBy3Points* - a circular arc by three points,
  - *acw_EllipseByCenterAndSemiaxis* - an ellipse with specified semi-axes and inclination angle,
  - *acw_EllipseByCenterAnd3Points* - an ellipse by center and three points,

  - *acw_EArcByCenterAnd2Points* - an elliptical arc bounded by two rays from the center to the specified points;

- **center** - the center of the ellipse (circle);
- **points** - the end points or the points through which the curve passes;
- **a** - interpretation depends on the arc creation method (see the enum MbeArcCreateWay);
- **b** - interpretation depends on the arc creation method (see the enum MbeArcCreateWay);
- **c** - interpretation depends on the arc creation method (see the enum MbeArcCreateWay);
- **option** - the curve direction (*true* - clockwise, *false* - counterclockwise).

The output parameter of the function is an **result** object that contains the constructed ellipse (circle) or its arc.

If the function completes successfully, it returns rt_Success; otherwise, the function returns an error code from the *MbResultType* enumeration.

The function is declared in the *action_curve.h* file.

The curve can be constructed from either specified points or scalar parameters (values of semi-axes, radius and inclination angles).

The parameter **center** defines the center point of the ellipse.

The **points** array can be empty; in this case, the lengths of the semi-axes **a** and **b** must be non-zero.

If the **points** array contains two elements, then the points **center**, **points**[0] and **points**[1] define the plane containing the axes **axisX** and **axisY** of the local coordinate system of the ellipse: the **axisX** axis is directed from the center to point **points**[0], the **axisY** axis is orthogonal to the **axisX** axis and is directed from the center toward point **points**[1]. The distance between **center** and **points**[0] defines the length of the first semi-axis of the ellipse **a**, while the distance between **center** and the projection of **points**[1] onto the **axisY** axis defines the length of the second semi-axis of the ellipse **b**, see Fig. 1:

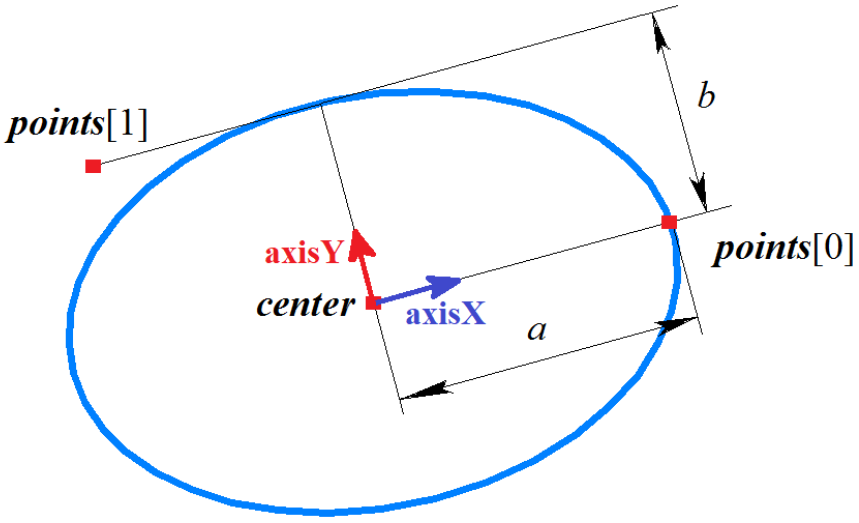


Fig. 1

If the **points** array is empty and **center** and radius **c** are specified, then a circle will be constructed; at the same time the axes **axisX** and **axisY** of the local coordinate system will coincide with the global axes (see the enum MbeArcCreateWay).

If the **points** array contains three elements, then an ellipse will be constructed with center at point **center**, passing through points **points**[0], **points**[1] and **points**[2]. The position of the axes of the ellipse's local coordinate system and the lengths of the semi-axes will be computed from points **center**, **points**[0], **points**[1], and **points**[2], see Fig. 2:

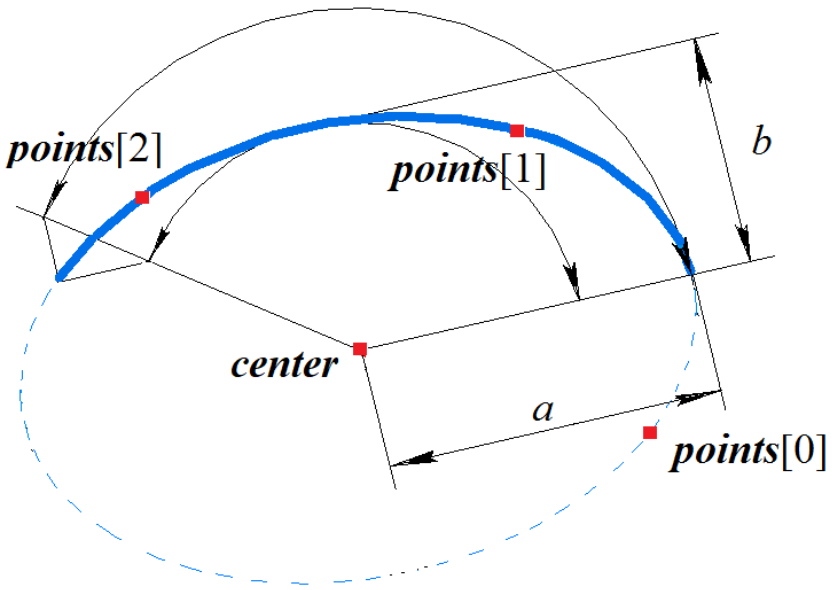


Fig. 2

The elliptical arc is described in subsection Two-Dimensional Elliptical Arc MbArc.

If the **points** array is empty, then the lengths of the semi-axes **a** and **b** and the arc angle **c** are input parameters, while the axes **axisX** and **axisY** of the local coordinate system of the ellipse coincide with the global axes.

In the special case where the lengths of the semi-axes are equal, the function constructs a circle or circular arc.

3.3.3 Constructing Two-Dimensional NURBS Curve

The function

```
MbResultType NurbsCurve( const SArray<MbCartPoint> & pointList,
                         const SArray<double> &      weightList,
                               size_t                degree,
                         const SArray<double> &      knotList,
                               bool                  curveClosed,
                               MbCurve *&            result );
```

constructs a two-dimensional NURBS curve based on a specified set of control points.

The input parameters of the function are:

- **pointList** - the set of control points;
- **weightList** - the set of control point weights;
- **degree** - the order of the B-spline;
- **knotList** - the set of parametric knots (knot vector);
- **curveClosed** - the flag indicating periodic closure of the curve.

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

Upon successful completion, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration.

The function is declared in the file *action_curve.h*.

The function constructs a NURBS curve (a curve based on *B*-splines), which is described in subsection Two-Dimensional NURBS Curve MbNurbs.

The set of weights **weightList** must correspond in size to the set of control points **pointList**.

The curve order **degree** must not exceed the number of control points.

The parameter **curveClosed** determines the periodic closure of the constructed curve.

The knot vector **knotList** represents a non-decreasing sequence of real numbers that defines the parameter domain and shape of the curve.

For **curveClosed** = *false*, the knot vector must contain a number of elements equal to the number of control points plus the curve order.

For **curveClosed** = *true*, the knot vector must contain a number of elements equal to the number of control points plus twice the curve order minus one.

To make an open NURBS curve pass through end control points, the first **degree** elements of the knot vector **knotList** must be equal to each other, and the last **degree** elements of the knot vector **knotList** must also be equal to each other.

Figure 1 shows a closed NURBS curve of the fourth order:

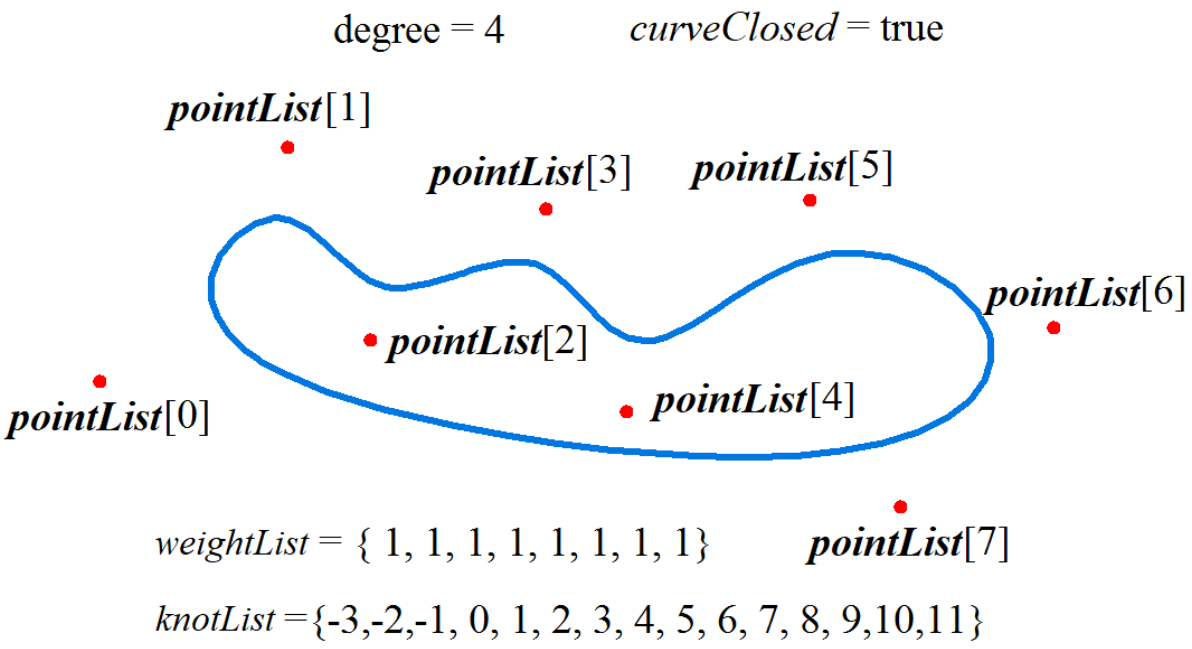


Fig. 1

Figure 2 shows an open NURBS curve of the fourth order passing through the end control points:

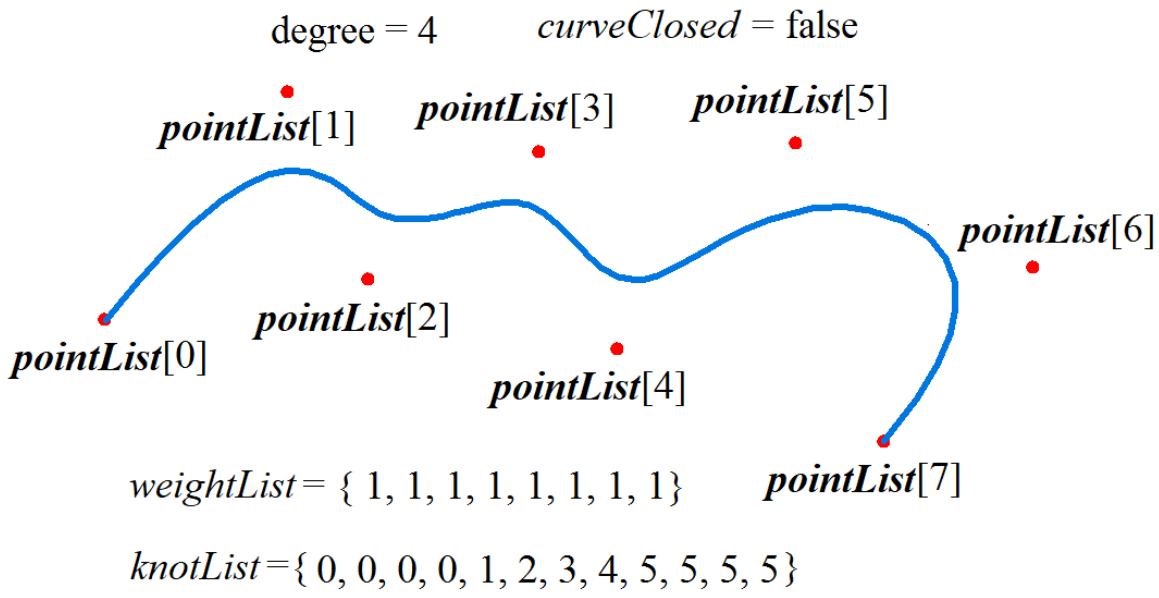


Fig. 2

Figure 3 shows three closed NURBS curves of different orders constructed from the same control points:

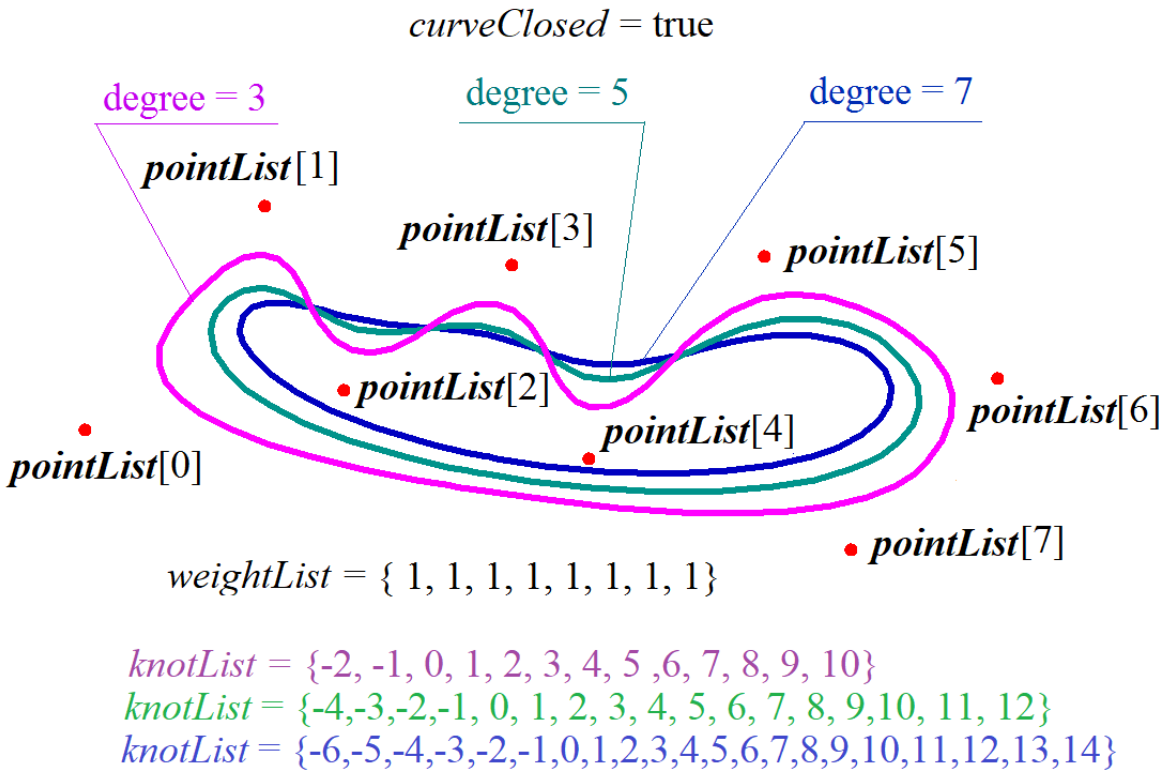


Fig. 3

Figure 4 shows three open NURBS curves of different orders constructed from the same control points:

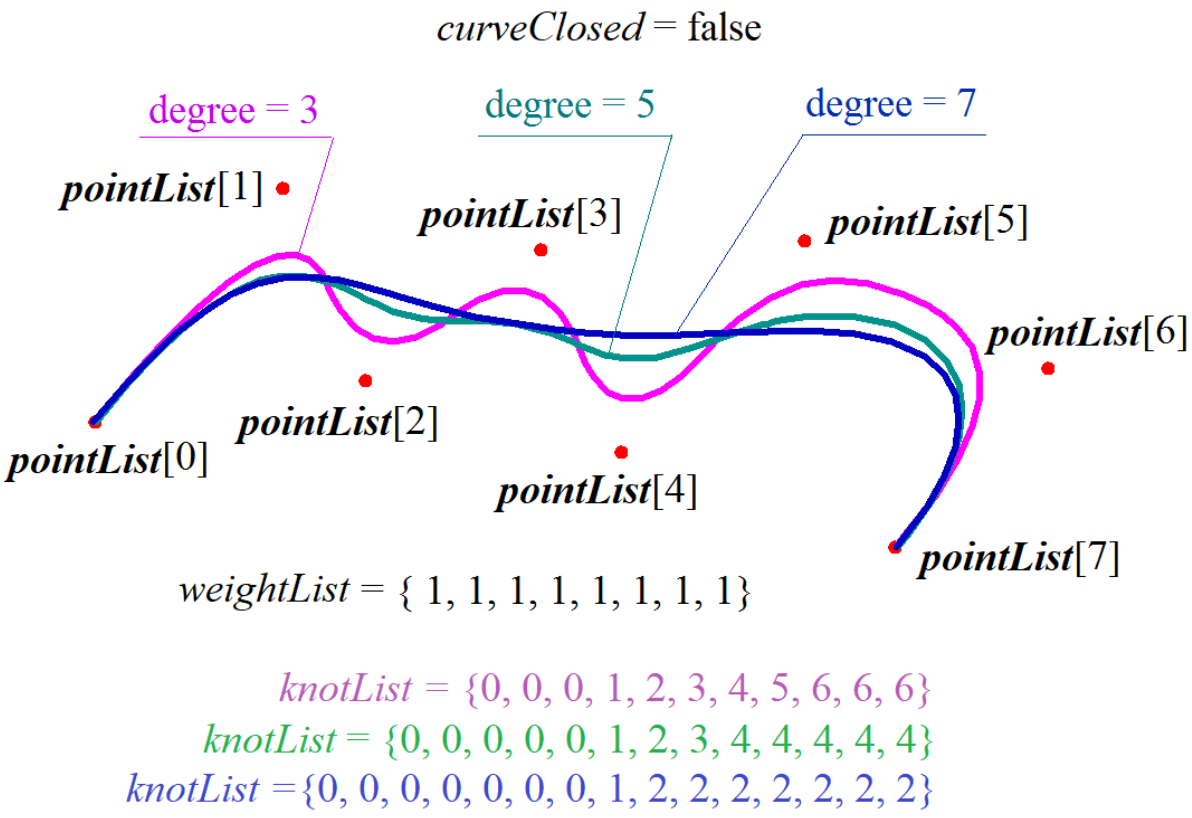


Fig. 4

Figure 5 shows three open NURBS curves of the fourth order having different control point weights:

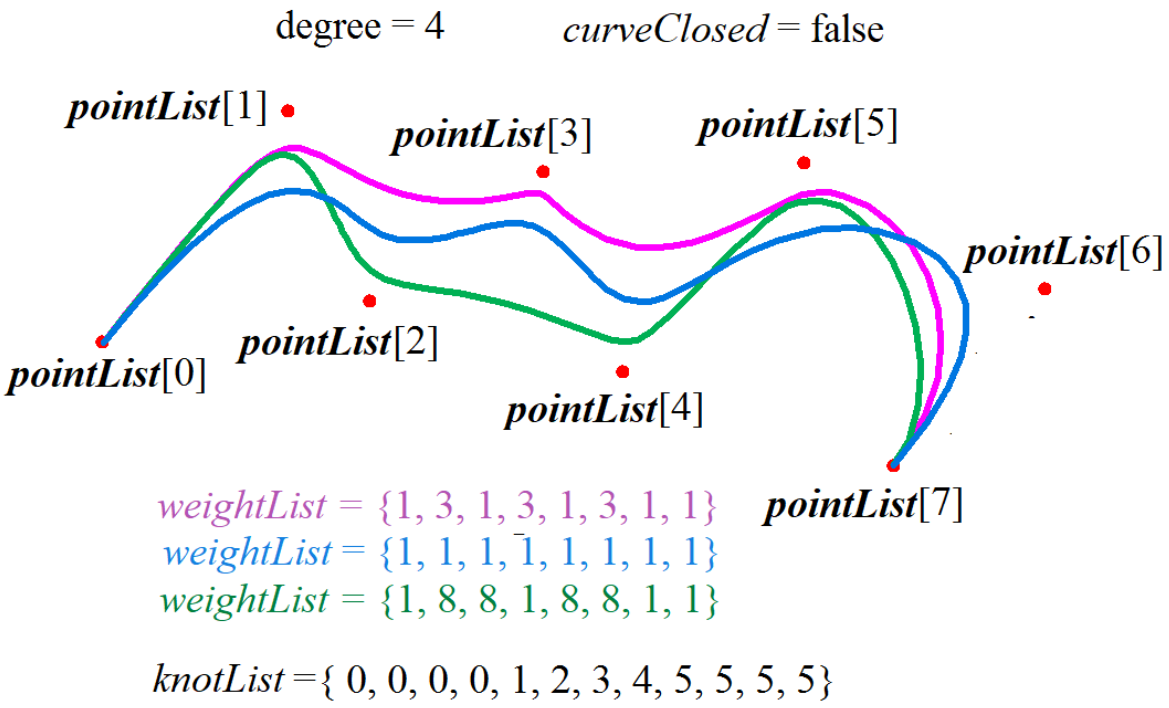


Fig. 5

Figure 6 shows an open NURBS curve of the third order whose shape matches a circular arc. The distance between control points **pointList**[0] and **pointList**[1] equals the distance between control points **pointList**[1] and **pointList**[2], and the weight **weightList**[1] of the middle control point equals the weight of the end control points multiplied by the cosine of half the arc angle.

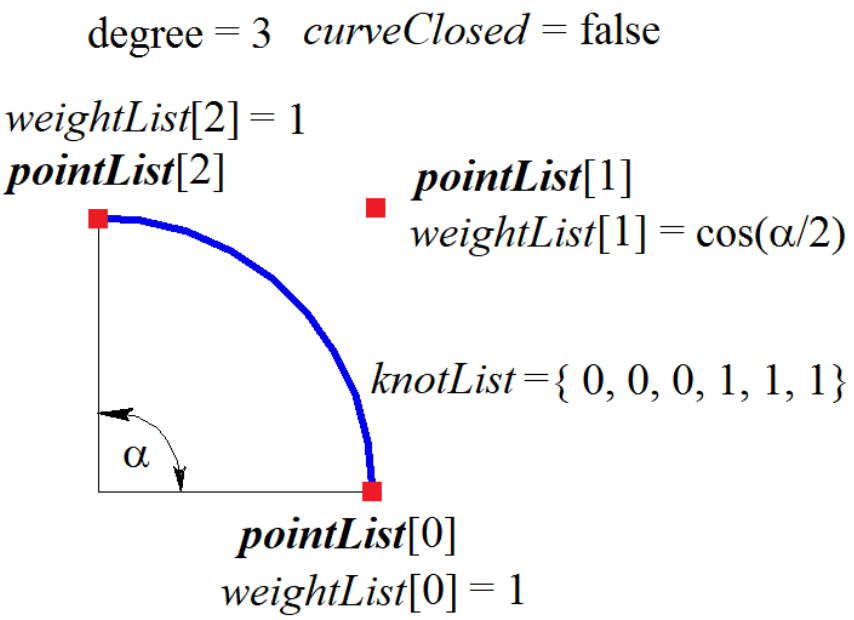


Fig. 6

The function

```
MbResultType NurbsCopy( const MbCurve &  curve,
                              MbCurve *& result );
```

constructs a NURBS copy of a given two-dimensional curve.

The input parameter of the function is the source curve **curve**.

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

The function constructs a NURBS curve whose shape copies that of the source curve. For most curve types, the NURBS copy exactly matches the source curve in shape. In cases where it is impossible to exactly reproduce the source curve's shape, the NURBS copy approximates the source curve.

In the test application *test.exe*, construction of a two-dimensional NURBS curve is performed using the menu command "Create→Curve in space→By points→NURBS".

3.3.4 Constructing a Regular Polygon

The function

```
MbResultType RegularPolygon( const MbCartPoint &   centre,
                             const MbCartPoint &   point,
                                   size_t          vertexCount,
                                   bool            describe,
                                   MbCurve *&      result );
```

constructs a closed two-dimensional polyline representing a regular polygon inscribed in or circumscribed about a specified circle.

Input parameters of the function are:

- **centre** - the center of the circle circumscribed about or inscribed in the regular polygon;
- **point** - a point on the circle;
- **vertexCount** - the number of polygon vertices;
- **describe** - a flag determining the polygon's position relative to the circle.

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

The function returns an operation result code from the enumeration *MbResultType* (see *mb_operation_result.h*).

The function is declared in the file *action_curve.h*.

Parameters **centre** and **point** define the circle in which the polygon with *vertexCount* vertices will be inscribed (**describe** = *false*) or about which the polygon with **vertexCount** vertices will be circumscribed (**describe** = *true*).

The **vertexCount** parameter defines the number of vertices of the regular polygon:

- when **vertexCount** == 0, a circle with center **centre** passing through **point** is constructed;
- when **vertexCount** == 1, a line segment with endpoints at **centre** and **point** is constructed;
- when **vertexCount** == 2, a rectangle with sides parallel to the global axes and opposite vertices at **centre** and **point** is constructed, Fig. 1:

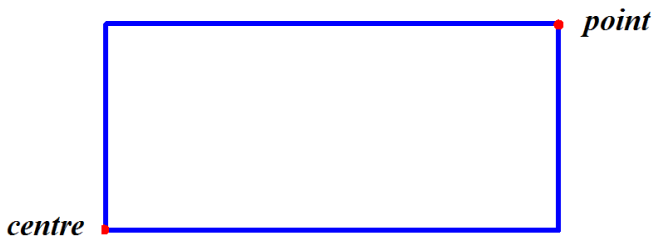


Fig. 1

- when **vertexCount** >= 3, a regular polygon with the specified number of sides is constructed.

The constructed curve is a closed polyline described in the paragraph Two-dimensional polyline MbPolyline. Figure 2 shows a regular polygon inscribed in a circle, and Fig. 3 shows a regular polygon circumscribed about a circle:

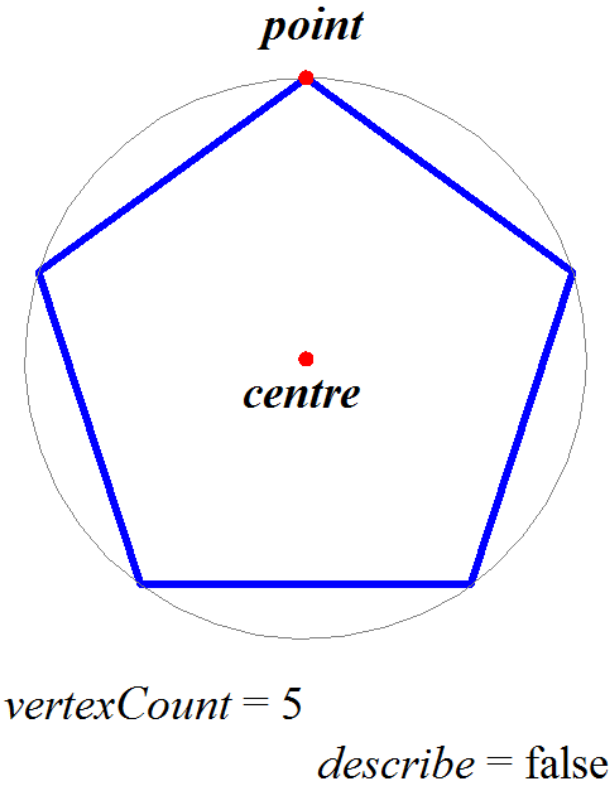


Fig. 2

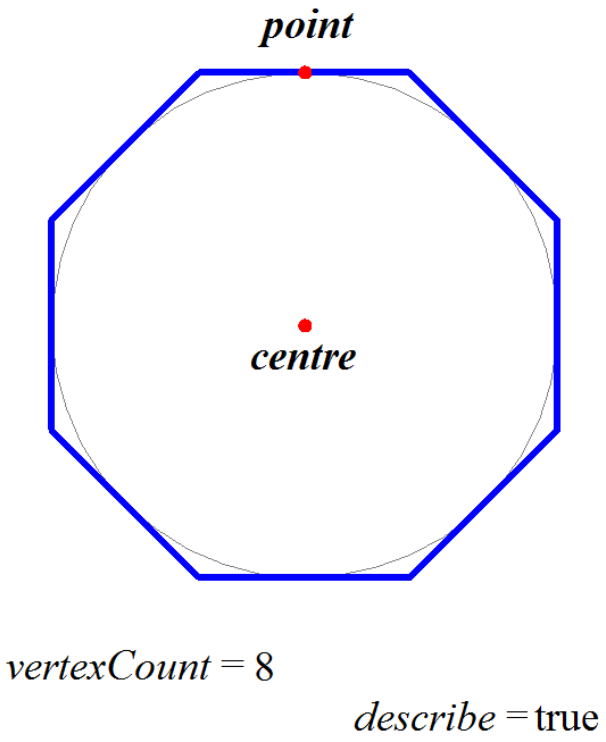


Fig. 3

In the test application *test.exe*, construction of a regular polygon is performed via the menu command “Create→Curve on plane→By points→Polygon”.

3.3.5 Constructing Two-Dimensional Cosine Wave

The function

```
MbResultType Cosinusoid( const MbCartPoint & point0,
                         const MbCartPoint & point1,
                               MbCartPoint & point2,
                               double        phase,
                               double        waveLength,
                               MbCurve *&    result );
```

constructs a cosine wave (two-dimensional harmonic curve).

Input parameters of the function are as follows:

- **point0** — an origin of the local coordinate system;
- **point1** — a point located on the X axis of the local coordinate system;
- **point2** — a point defining the Y axis of the local coordinate system;
- **phase** — a phase shift of the harmonic curve (in radians);
- **waveLength** — a wavelength of the harmonic curve.

The output parameter of the function is the pointer **result** to the constructed two-dimensional curve of type MbCurve representing the cosine wave.

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

The function is declared in *action_curve.h* file.

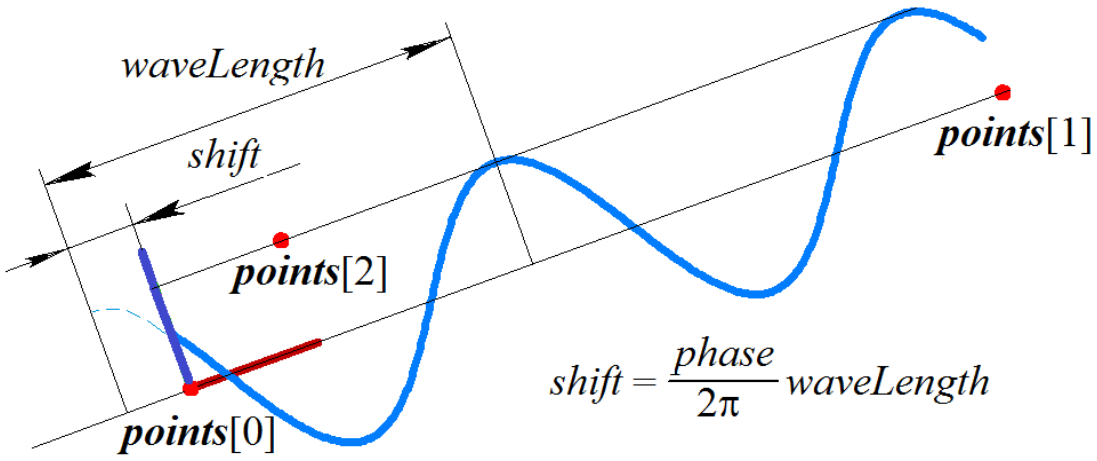
point0, **point1** and **point2** points define the local coordinate system of the cosine wave. **point0** will be the origin of the local coordinate system of the cosine wave, **axisX** of the local coordinate system will be directed from **point0** to **point1**, **axisY** of the local coordinate system will be orthogonal to **axisX** and directed from **point0** to **point2**.

point0, **point1** and **point2** points should not lie along a single straight line.

Two-dimensional cosine waves are described in Item MbCosinusoid Two-Dimensional Cosine Wave.

The amplitude of the cosine wave is equal to the length of the projection of the vector (**point2** – **point0**) onto the Y axis **axisY**. The wavelength of the cosine wave is equal to **waveLength**. The number of waves of the cosine wave is determined by the distance between the points **point0** and **point1**.

phase defines the phase shift of the cosine wave:



3.3.6 Constructing Two-Dimensional Compound Curve

The function

```
MbResultType CreateContour( MbCurve   & curve,
                            MbContour *& result );
```

constructs a compound curve (contour) based on an original curve.

curve original curve is an input parameter of the function.

The output parameter of the function is the pointer **result** containing a constructed curve.

If successful, the function returns rt_Success, otherwise it returns an error code from *MbResultType*.

The function is declared in the *action_curve.h* file.

The function 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 *MbContour Two-Dimensional Contour*.

The function

```
MbResultType AddCurveToContour( MbCurve   &  curve,
                                MbContour &  contour,
                                bool         toEnd,
                                double       epsilon = METRIC_REGION );
```

modifies a compound curve by adding another curve.

The input parameters of the function are:

- **curve** — a curve to be added;
- **contour** — a compound curve to be modified;
- **toEnd** — the flag indicating the position to add the curve;
- **epsilon** — the point coincidence tolerance when adding the curve.

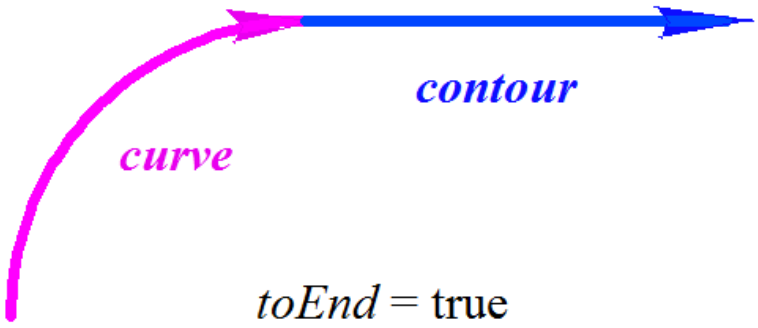
The output parameter of the function is the modified curve **contour**.

Upon successful completion, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType_* enumeration.

The function is declared in the *action_curve.h* file.

The function modifies the compound curve **contour** by adding the curve **curve** to the beginning or end of the compound curve.

If **toEnd** = *true*, the curve **curve** will be added to the end of the compound curve **contour**; if **toEnd** = *false*, the curve **curve** will be added to the beginning of the compound curve **contour**. The connection points of the modified curve and the added curve must coincide within the tolerance **epsilon**:



If the added curve **curve** is a compound curve, its components will be added to the **contour** curve.

3.3.7 Constructing Surface and Plane Intersection Curves

The function

```
void SurfaceSection( const MbSurface &         surface,
                     const MbPlacement3D &     place,
                           RPArray<MbCurve> &  result,
                           VERSION             version = Math::DefaultMathVersion() );
```

constructs intersection curves of a surface and a plane.

The input parameters of the function are:

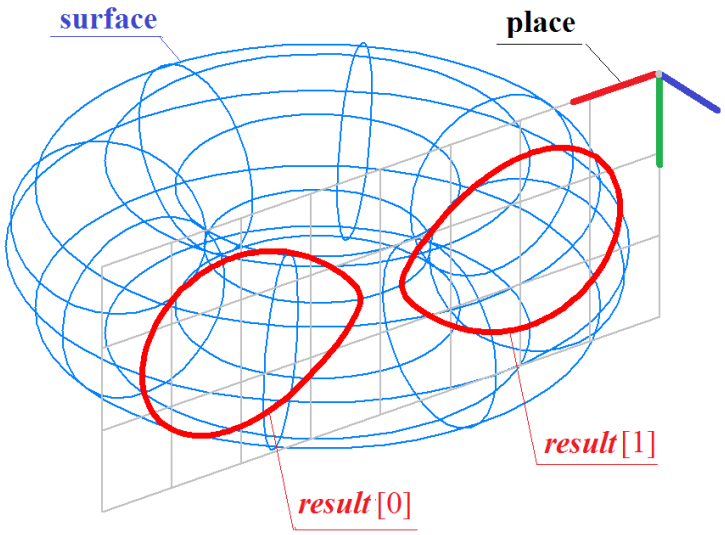
- **surface** — a surface;
- **place** — the local coordinate system of the plane;
- **version** — an execution version (optional).

The output parameter of the function is the set of constructed two-dimensional curves **result** on the surface and two-dimensional curves on the plane.

The function has no return value.

The function is declared in the file *action_curve.h*.

The function constructs the intersection of the surface **surface** with the XY plane of the local coordinate system **place**. The curves **result** are constructed in the XY plane of the local coordinate system **place**. Figure shows an example of constructing two-dimensional intersection curves of a torus surface and the XY plane of a local coordinate system:



3.3.8 Constructing a Two-Dimensional Face Edge Curve

The function

```
MbResultType FaceBoundSegment( const MbFace &     face,
                                     size_t       loopInd,
                                     size_t       edgeInd,
                               const MbSurface &  surface,
```

(continues on next page)

(continued from previous page)

```
                                    VERSION     version,
                                    MbCurve *&  result );
```

constructs a projection of a face edge onto a surface.

The input parameters of the function are:

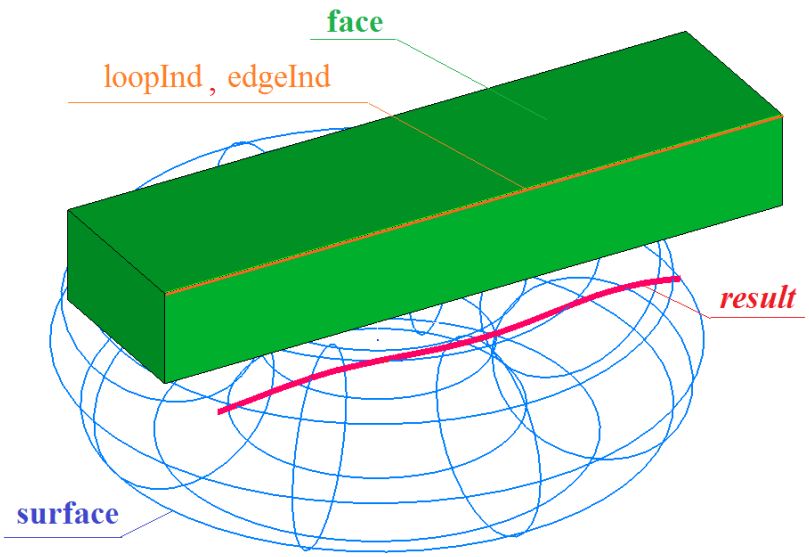
- **face** — a face;
- **loopInd** — an index of the loop in the face containing the edge to project;
- **edgeInd** — an index of the edge to project within the loop;
- **surface** — a projection surface;
- **version** — a construction version.

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

If successful, the function returns rt_Success; otherwise, it returns an error code from the enumeration *MbResultType*.

The function is declared in the file *action_curve.h*.

The function constructs the projection of the face edge **face** onto the surface **surface**. The edge in the face is specified by the face loop index **loopInd** and the edge index within the loop **edgeInd**. The curve **result** is constructed in the parameter space of the surface **surface**. Figure shows an example of constructing the projection of a face edge onto a given surface:



In the test application *test.exe*, constructing curves of surface-plane intersection is performed using the menu command “Create→Curve in space→Based on a solid→Edges projections on a face”.

3.3.9 Projecting a Curve onto a Surface

The function

```
MbResultType SurfaceBoundContour( const MbSurface &  surface,
                                  const MbCurve3D &  spaceCurve,
                                        VERSION      version,
                                        MbContour *& result );
```

constructs a two-dimensional curve in the parameter space of a surface for a space curve lying on that surface.

The input parameters of the function are:

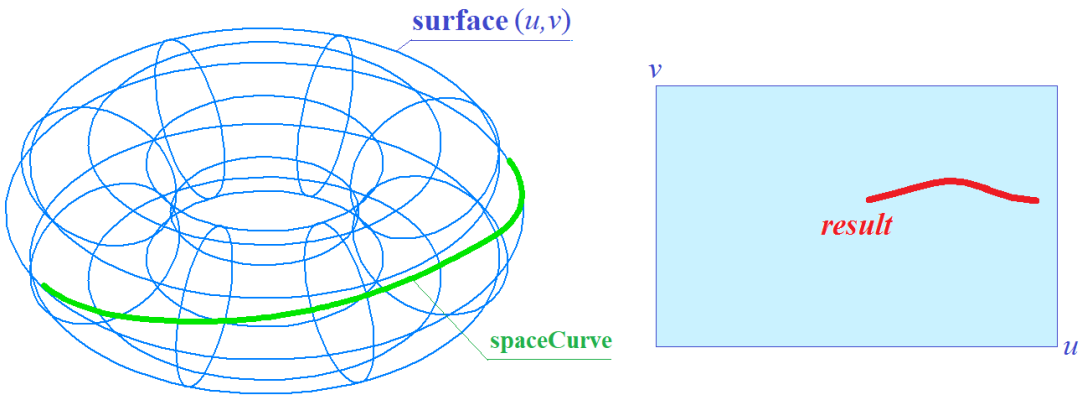
- **surface** — a surface;
- **spaceCurve** — a space curve;
- **version** — a construction version.

The output parameter of the function is the constructed composite curve **result**.

Upon successful completion, the function returns rt_Success; otherwise, it returns an error code from the enumeration *MbResultType*.

The function is declared in the file *action_curve.h*.

The space curve must lie on the surface **surface**. The curve **result** is constructed in the parameter space of the surface **surface**. Figure shows an example of projecting the space curve **spaceCurve** onto the given surface:



3.4 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 models and wireframe models. In some cases curves act as reference objects to position other elements of a geometric model.

All curves are the MbCurve3D class inheritors and are described in section *CURVES*. The curves can be constructed by directly calling the corresponding constructors or using the methods described in this section.

3.4.1 Constructing a Line and a Segment

The function

```
MbResultType Line( const MbCartPoint3D & point1,
                   const MbCartPoint3D & point2,
                         MbCurve3D *&    result );
```

constructs a straight line through two non-matched points in space.

The input parameters of the function are:

- **point1** — the first point through which the line passes;
- **point2** — the second point through which the line passes.

The output parameter of the function is the constructed straight line **result**.

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

The function is declared in the *action_curve3d.h* file.

The **point1** parameter defines the starting point of the line, corresponding to the zero parameter value of the line. The vector starting at **point1** and ending at **point2** defines the line direction, as shown in Fig. 1. The derivative of the line represents a unit-length vector.

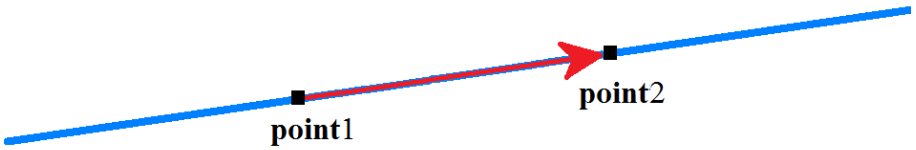


Fig. 1

The straight line is described in the section *Straight Line MbLine3D*.

The function

```
MbResultType Segment( const MbCartPoint3D & point1,
                      const MbCartPoint3D & point2,
                            MbCurve3D *&    result );
```

constructs a line segment between two non-matched points in space.

The input parameters of the function are:

- **point1** — the starting point of the segment;
- **point2** — the ending point of the segment.

The output parameter of the function is the constructed line segment **result**.

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

The function is declared in the *action_curve3d.h* file.

The **point1** parameter defines the starting point of the segment, corresponding to the zero parameter value of the line. The **point2** parameter defines the ending point of the segment, corresponding to the line parameter value of one, as shown in Fig. 2:



Fig. 2

The line segment is described in the section *Straight Line Segment MbLineSegment3D*.

3.4.2 Constructing a Circle, an Ellipse, and Their Arcs

The function

```
MbResultType Arc( const MbCartPoint3D &           centre,
                  const SArray<MbCartPoint3D> &   points,
                        bool                      curveClosed,
                        double                    angle,
                        double &                  a,
                        double &                  b,
                        MbCurve3D *&              result );
```

constructs an elliptical arc; in the special case it constructs a circular arc.

The input parameters of the function are:

- **centre** – the center of the ellipse (circle).
- **points** – a set of arc points (the **points** container can contain 0 or 2 elements).
- **curveClosed** – the flag indicating whether the curve is cyclically closed.
- **angle** – the span angle of the elliptical arc (in radians).
- **a** – the length of the major semi-axis of the ellipse (computed when the **points** set is non-empty).
- **b** – the length of the minor semi-axis of the ellipse (computed when the **points** set is non-empty).

The output parameter of the function is the constructed ellipse (circle) or its arc **result**.

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration.

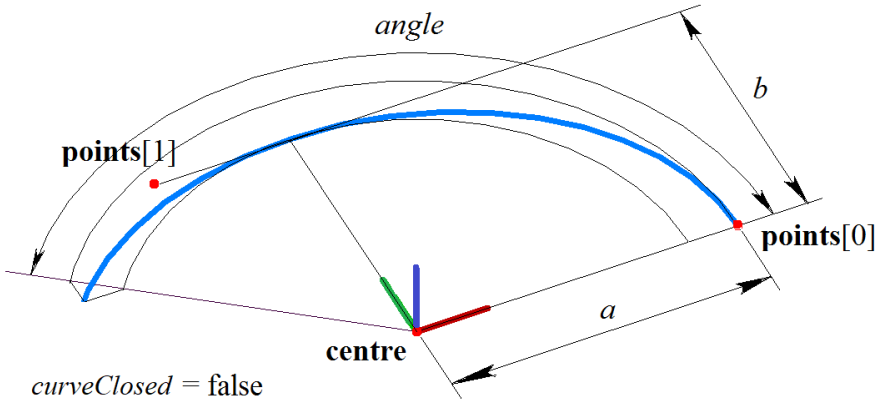
The function is declared in the file *action_curve3d.h*.

The curve can be constructed either from given points or from scalar parameters.

The **centre** parameter defines the center point of the ellipse.

The set of points **points** can be empty; in this case, the lengths of the ellipse semi-axes **a** and **b** must be non-zero. If the set **points** contains two elements, the points **centre**, **points**[0] and **points**[1] define the plane of the **axisX** and **axisY** of the ellipse's local coordinate system: - the **axisX** is directed from the center to the point **points**[0]; - the **axisY** is orthogonal to **axisX** and directed from the center toward the point **points**[1].

The distance between the points **centre** and **points**[0] determines the length of the first semi-axis of the ellipse **a**, while the distance between the point **centre** and the projection of the point **points**[1] onto the **axisY** determines the length of the second semi-axis of the ellipse *b*:



The elliptical arc is described in section *MbArc3D Elliptical Arc*.

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

The **curveClosed** parameter determines the cyclic closedness of the curve. If **curveClosed** = *false*, the **angle** parameter must be non-zero; it defines the span angle of the elliptical arc in parametric units.

In the special case when **a** = **b**, the function constructs a circle (**curveClosed** = *true* or **angle** = 0) or a circular arc (**curveClosed** = *false*, **angle** > 0, and **angle** < 2π).

3.4.3 Construction of a Regular Polygon

The function

```
MbResultType RegularPolygon( const MbCartPoint3D &  centre,
                             const MbCartPoint3D &  point,
                             const MbVector3D &     axisZ,
                                   size_t           vertexCount,
                                   bool             describe,
                                   MbCurve3D *&     result )
```

constructs a regular polygon inscribed in a specified circle or circumscribed about the circle.

The function parameters are:

- **centre** — the center of the circle circumscribed about the equilateral polygon or inscribed in the equilateral polygon;
- **point** — a point on the circle;
- **axisZ** — the vector perpendicular to the circle plane;
- **vertexCount** — the number of vertices of the polygon;
- **describe** — the flag indicating a circumscribed or an inscribed polygon (*true* — a circumscribed polygon, *false* — an inscribed polygon).

The output parameter of the function is the constructed polyline curve **result**.

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

The function is declared in the *action_curve3d.h* file.

The **centre**, **point** and **axisZ** parameters determine the circle in which the polygon with **vertexCount** vertices is inscribed (**describe** = *false*) or about which the polygon with **vertexCount** vertices is circumscribed (**describe** = *true*). The center of the circle is located at point **centre**, the circle passes through **point**, and the axis of the circle is parallel to the **axisZ** vector.

The **vertexCount** parameter determines the number of vertices of the regular polygon:

- if **vertexCount** == 0, a circle centered at **centre** and passing through **point** is constructed;
- if **vertexCount** == 1, a line segment with endpoints at **centre** and **point** is constructed;
- if **vertexCount** == 2, a rectangle with sides parallel to the global axes and with opposite vertices at **centre** and **point** is constructed;
- if **vertexCount** >= 3, a regular polygon with the specified number of sides is constructed, inscribed in the circle (**describe** == *false*) or circumscribed about the circle (**describe** == *true*) centered at **centre** and passing through **point**.

The constructed curve is a closed polyline described in section *MbPolyline3D Polyline* (for **vertexCount** >= 3). Fig. 1 shows a regular polygon inscribed in a circle, and Fig. 2 shows a regular polygon circumscribed about a circle:

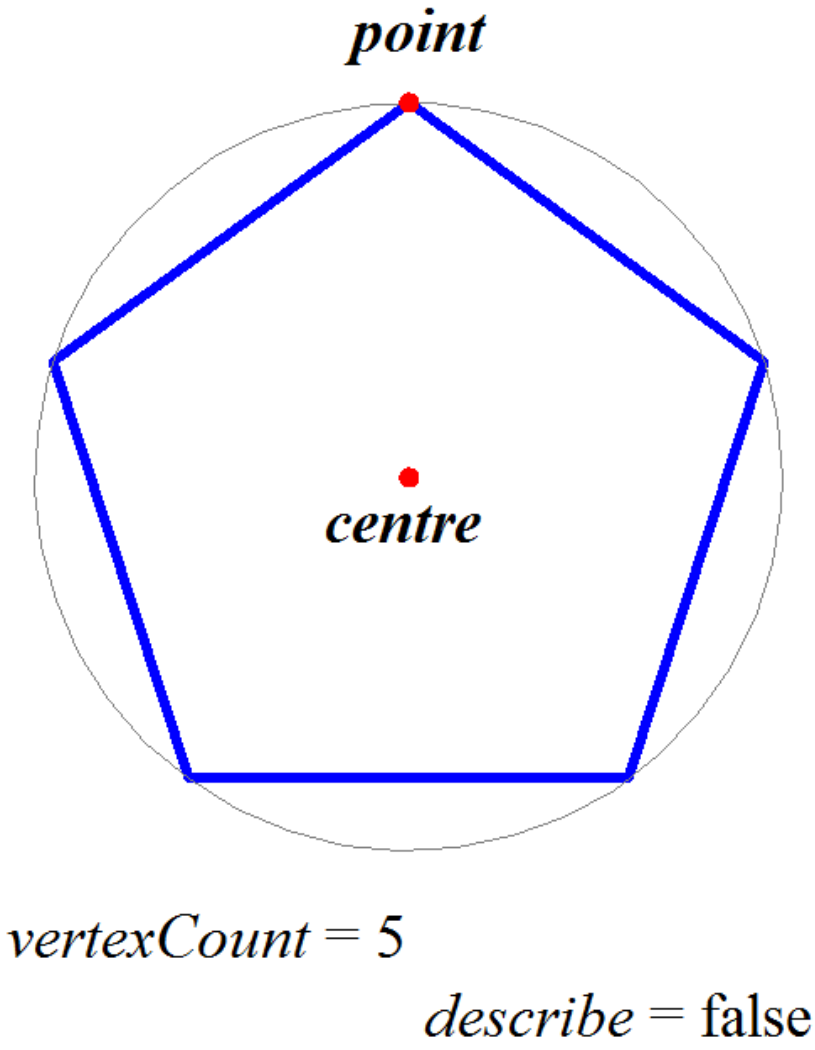


Fig. 1

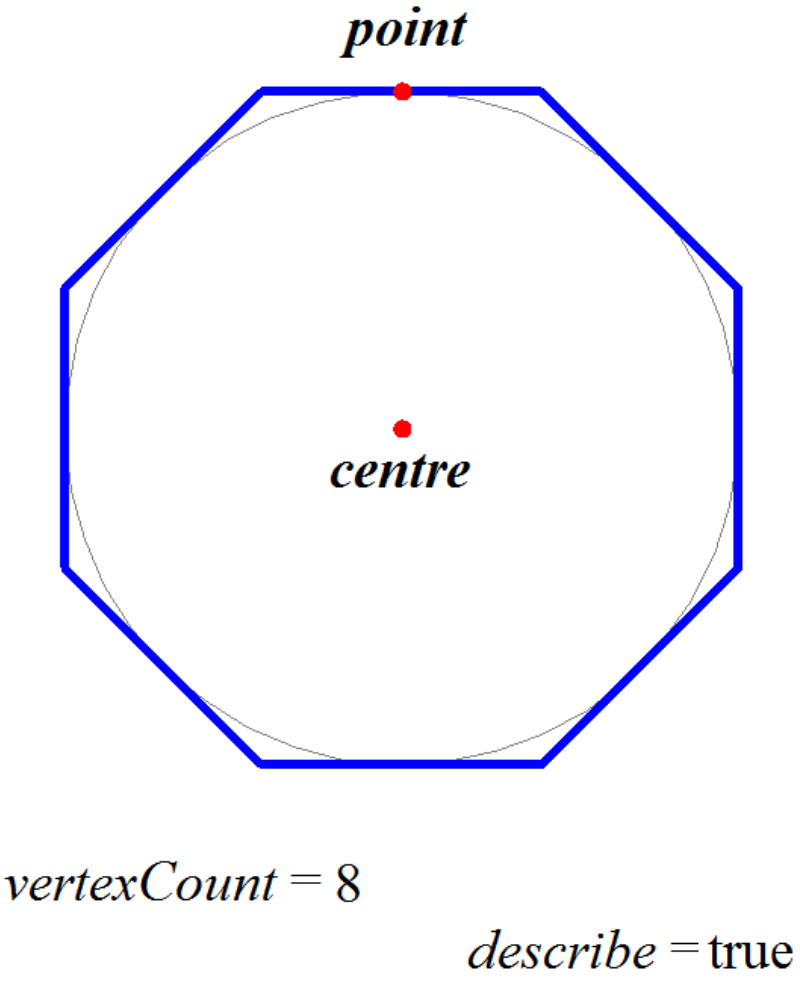


Fig. 2

3.4.4 Constructing a Compound Curve

The function

```
MbResultType CreateContour( MbCurve3D   &  curve,
                            MbContour3D *& result )
```

constructs a compound curve based on a single source curve.

The input parameter of the function is the source curve **curve**.

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

The output parameter of the function is the constructed compound curve (contour) **result**.

If successful, **result** returns a pointer to a new MbContour3D object; the caller is responsible for deleting this object.

The function is declared in the *action_curve3d.h* file.

The function creates the compound curve **result** based on the source curve **curve**. If the source curve **curve** is a compound curve, the constituent elements of the source curve are placed into the curve **result**. A compound curve is described in subsection *Contour MbContour3D*.

The function

```
MbResultType AddCurveToContour( MbCurve3D & curve,
                                MbCurve3D & contour,
                                bool        toEnd )
```

modifies a compound curve by adding a curve.

The input parameters of the function are:

- **curve** is the curve to be added;
- **contour** is the compound curve to be modified;
- **toEnd** is a flag that specifies where to add the curve.

The output parameter of the function is the modified curve **contour**.

If successful, the function returns rt_Success; otherwise, it returns an error code from the enumeration *MbResultType*.

The function is declared in the *action_curve3d.h* file.

The function modifies the compound curve **contour** by adding the curve **curve** to the beginning or to the end of the compound curve. If **toEnd** = *true*, the curve **curve** is added to the end of the compound curve **contour**; if **toEnd** = *false*, the curve **curve** is added to the beginning of the compound curve **contour**. The joint points of the curve being modified and the curve being added must coincide, see Fig. 1:

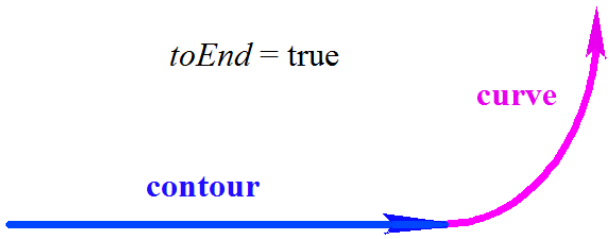


Fig. 1

If the curve **curve** being added is a compound curve, the constituent elements of the curve being added are added to the curve **contour**.

The function

```
MbResultType CreateContours( RPArray<MbCurve3D> &   curves,
                             double                 metricEps,
                             RPArray<MbContour3D> & result,
                             bool                   onlySmoothConnected = false,
                             VERSION                version =␣
↪Math::DefaultMathVersion() )
```

constructs compound curves from a set of source curves (original curves).

The input parameters of the function are:

- **curves** is a set of source curves;
- **metricEps** is a capture radius for joining the source curves;
- **onlySmoothConnected** is a flag to use smooth joining of the source curves;
- **version** is an operation version.

The output parameter of the function is a set of constructed compound curves **result**.

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

The function is declared in the *action_curve3d.h* file.

The function creates compound curves based on the source curves **curves** and stores them in the set **result**.

If the flag **onlySmoothConnected** = *true*, all the constructed compound curves are smooth, that is, the curves have equally directed tangents at the joint points, but the derivative length at the joint points may change abruptly. If any source curve in the set **curves** is a compound curve, the constituent elements of such a source curve are added to a compound curve.

Degenerate curves, null-pointer curves and duplicate curves are removed from the input array if they have no owner.

The function

```
MbResultType CreateContours( RPArray<MbCurve3D> & curves,
                             double               metricAcc,
                             bool                 onlySmoothConnected = false,
                             VERSION              version =␣
↪Math::DefaultMathVersion() )
```

constructs contours from a set of curves with removal of degenerate curves (on original curves).

The input parameters of the function are:

- **curves** is the set of source curves (it is modified);
- **metricAcc** is the capture radius for joining the source curves;
- **onlySmoothConnected** is a flag that specifies that only smoothly joined segments are used in a contour;
- **version** is the operation version.

The output parameter of the function is the set **curves**, which, after the execution, contains the constructed compound curves.

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

The function is declared in the *action_curve3d.h* file.

The function automatically removes from the array **curves** all degenerate curves (with zero length or with coincident start and end points for non-closed curves), null-pointer curves and duplicate curves that have no owner. The resulting compound curves are stored back into the array **curves**, replacing the original elements.

3.4.5 Curve Extension

The function

```
MbResultType ExtendCurve( const MbCurve3D &                    sourceCurve,
                          const MbCurveExtensionParameters3D & parameters,
                                SPtr<MbWireFrame> &            resFrame );
```

extends the given curve in accordance with the specified parameters.

The function parameters are:

- **sourceCurve** - the source (extendable) curve;
- **parameters** - the extension parameters (for the definition of the MbCurveExtensionParameters3D class and its constructors, see *op_curve_parameter.h*);
- **resFrame** - the operation result. It is a wireframe, covering the source curve completely. Additionally, it includes an "extending segment". The wireframe contains the MbExtendCurveCreator creator declared in the *cr_extending_curve.h* file.

The function is declared in the *action_curve3d.h* file.

In the test application test.exe, curve extension is performed via the menu command "Modify→Curve in space→Extend".

It is recommended to wrap the original curve in a wireframe and call the ExtendWireFrame() function, detailed in subsection *Wireframe Extension*.

3.4.6 Wireframe Extension

The function

```
MbResultType ExtendWireFrame(       c3d::WireFrameSPtr &           wireframe,
                                    MbeCopyMode                    sameEdges,
                              const MbCurveExtensionParameters3D & params,
                                    c3d::WireFrameSPtr &           result );
```

extends the given wireframe, thereby generalizing the curve extension operation.

The function parameters are:

- **wireframe** - the source (extendable) wireframe (it is recommended to pass a normalized wireframe);
- **sameEdges** - the copy mode of the input wireframe;
- **params** - the extension parameters;
- **result** - the resulting wireframe.

The function is declared in the *action_wireframe.h* file.

The MbCurveExtensionParameters3D class inherits the properties of the MbCurveExtensionParameters class. The **params** object defines the following group of wireframe extension parameters:

- start extension parameters - an object of the MbCurveExtensionEnds class or one of its derived classes (MbCurveExtensionLinear, MbCurveExtensionCircular, MbCurveExtensionNatural);
- end extension parameters - an object of the MbCurveExtensionEnds class or its derived classes, which is set independently of the start parameters;

- the closure permission flag (**allowClosure**), which determines whether the curve is allowed closing during extension. In the course of extension (if the extension length is sufficiently large), the curve may become closed. This is not always desirable, since a gap may need to be preserved between the endpoints. To prevent closure, set **allowClosure** = *false*. The gap value is determined by internal algorithms and is not available for modification by the user. If the extended curve has reached its maximum length (either the curve became closed, or the maximum permissible gap between the endpoints has been achieved), one of the following warnings is issued: *rt_CurveClosedAtStart*, *rt_CurveClosedAtEnd*, or *rt_CurveClosedBothSides* (see Table).
- the along-surface extension flag (**alongSurface**), which determines whether the surface curve (*MbSurfaceCurve*) must remain on its surface. The difference is illustrated in Fig. 1, which shows a surface of revolution with an arbitrary surface curve on it (blue) and two variants of arc extension:
  - if the curve is extended as a spatial curve (**alongSurface** = *false*), the extended segment (purple) is a circular arc, but may leave the surface;
  - when extending along the surface (**alongSurface** = *true*), the curve always remains on the surface, but the shape of the extension segment (orange) depends on the surface geometry. For non-surface curves when **alongSurface** = *true*, the error code *rt_IncorrectData* is returned.

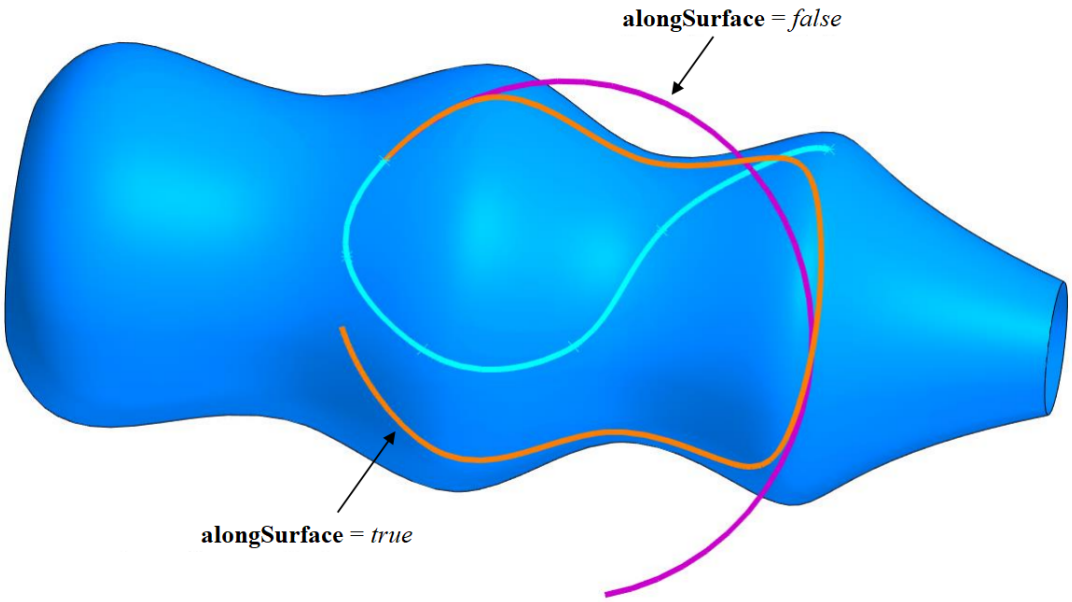


Fig. 1

- the operation name maker (an object of the MbSNameMaker class), used for naming the created wireframe.

In most cases, when extending along a surface, it is not allowed the extended curve to go beyond the boundaries. If it occurs, the operation returns the error *rt_BeyondLimitsExtension* (see Table). Going beyond the boundaries is permitted for MbElementarySurface and for periodic surfaces on the boundaries that correspond to the periodic direction.

The MbCurveExtensionEnds class describes the extension parameters for an individual end-

point of the wireframe. The following terms are used in the documentation:

- **extension length** — the metric length by which the wireframe is extended from the selected endpoint;
- **extension method** — a value of the MbeCurveExtensionWays enumeration that defines the extension method (linear, circular arc, or natural continuation of the original wireframe).

Three different extension methods are implemented in the geometric kernel:

1. The **linear** extension method (MbeCurveExtensionWays::cew_Linear) is performed by restoring the tangent at the given point of the curve and laying off a segment of the specified length along it. This method ensures smoothness of at least G^1 at the junction with the extension segment. The parameters for this extension method are set via objects of the MbCurveExtensionLinear class.
2. The **circular arc** extension method (MbeCurveExtensionWays::cew_Circular) is performed by constructing the osculating circle at the endpoint being extended and taking its arc of the specified length. This method ensures smoothness of at least G^2. The parameters for this extension method are set via objects of the MbCurveExtensionCircular class.
3. The **natural continuation** extension method (MbeCurveExtensionWays::cew_CommonNatural) extends the curve as a natural continuation of the original curve (the shape it had immediately before the extension operation). The result is the same curve, only with a greater length. However, in some cases extension is performed by creating a contour and attaching to the original curve a NURBS segment that ensures continuity of at least G^3 at the junction joint. The parameters for this extension method are set via objects of the MbCurveExtensionNatural class.

The MbCurveExtensionLinear, MbCurveExtensionCircular, and MbCurveExtensionNatural classes are derived from MbCurveExtensionEnds and allow setting the extension parameters specific to each particular method. The differences between the extension methods are illustrated in Fig. 2:

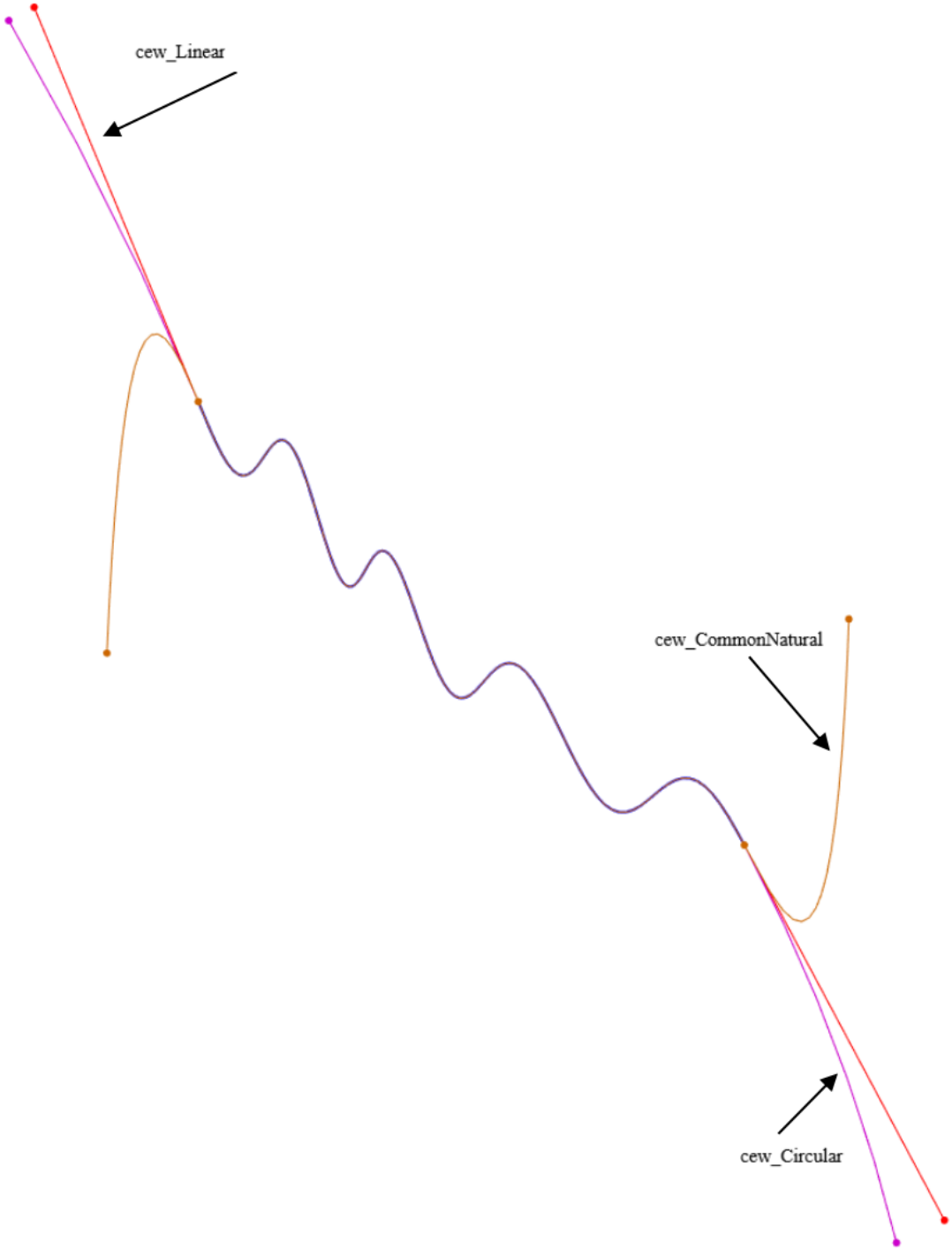


Fig. 2

There are no restrictions on the combinations of extension options for individual endpoints. The extension of each endpoint is performed independently of the other. For example, Fig. 3 shows an elliptic arc extended at the start by 30 units using the linear method and at the end by 60 units using the circular arc method.

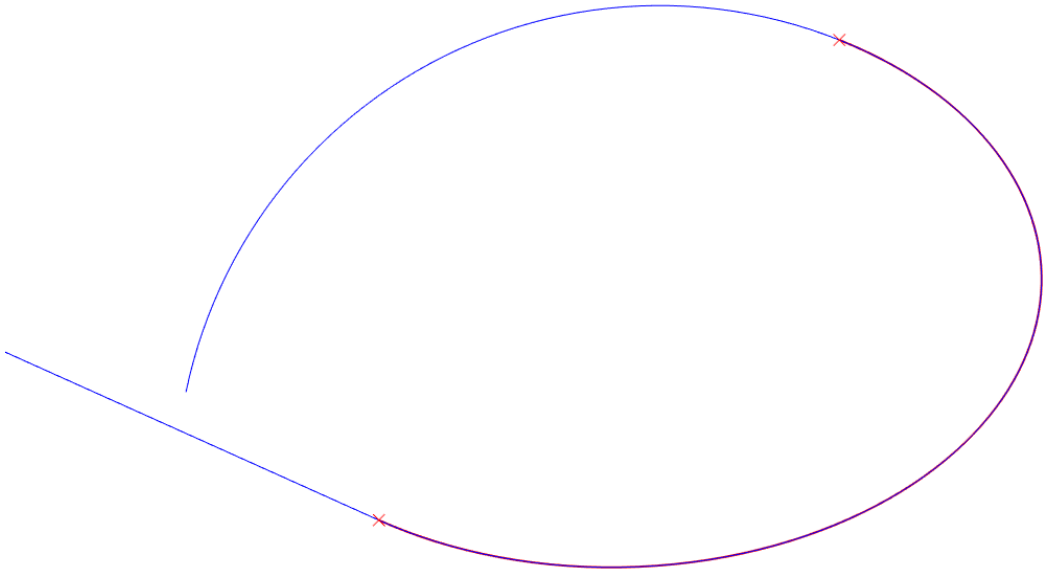


Fig. 3

The ExtendWireFrame() function returns the operation result code as shown in Table:

Operation Result Codes

For critical errors, the pointer to the resulting wireframe is always null; for non-critical errors, the wireframe is created, but its geometry may not satisfy the requirements.

The following is a typical sequence of steps for calling the wireframe extension algorithm:

1. Include the required header files:

```
#include <op_wireframe_parameter.h>
#include <action_wireframe.h>
```

2. As an example, extend a wireframe at the end by 100 units using the circular arc method:

```
double extendLength = 100.0;    // Extension length
```

3. Initialize the extension parameters for each individual endpoint using objects of the MbCurveExtensionEnds class. If extension of a particular endpoint is not required, its parameters should be initialized using the default constructor.

```
// No extension at the start. Use the default constructor.
MbCurveExtensionEnds     paramStart;

// Extend at the end using the circular arc method.
MbCurveExtensionCircular paramEnd( extendLength );
```

4. Initialize the remaining extension parameters and include in them the endpoint extension objects created in step 3.

```
bool allowClosure = false;  // Closure permission flag
bool alongSurface = false;  // Along-surface extension flag
MbSNameMaker  name;                  // Operation name maker
MbCurveExtensionParameters3D opParameters( paramStart,
                                           paramEnd,
                                           allowClosure,
                                           alongSurface,
                                           name );
```

5. Call the API function that performs the extension:

```
SPtr<MbWireFrame> scrWireFrame;      // A pre-prepared non-null wireframe

SPtr<MbWireFrame> result( nullptr ); // Extension result
MbResultType      res = ::ExtendWireFrame( scrWireFrame,
                                           cm_Copy,
                                           opParameters,
                                           result );
```

The resulting wireframe contains the MbExtendCurveCreator builder (a derived class of MbCreator).

In the test application test.exe, wireframe extension is performed via the menu command “Create→Wire frame→Based on wire frame→Extended wire frame”.

3.5 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 section *SURFACES*. The surfaces can be constructed by directly calling the corresponding constructors or using the methods described in this section.

3.5.1 Elementary Surface Construction

The function

```
MbResultType ElementarySurface( const MbCartPoint3D & point0,
                                const MbCartPoint3D & point1,
                                const MbCartPoint3D & point2,
                                      MbeSpaceType    surfaceType,
                                      MbSurface *&    result );
```

constructs an elementary surface.

The input parameters of the function are:

- **point0** - a point that defines the origin of the local coordinate system of the surface.
- **point1** - a point that defines the direction of the **axisX** or **axisZ** axis (depending on the surface type) of the local coordinate system and the height of the surface.
- **point2** - a point that defines the direction of the **axisY** axis of the local coordinate system.
- **surfaceType** - a surface type.

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

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration.

The function is declared in the *action_surface.h* file.

The **surfaceType** parameter defines the type of the surface to be created. The control points **point0**, **point1** and **point2** define the local coordinate system and dimensions of the elementary surface. The point **point0** defines the origin of the local coordinate system. Table shows the surface type, the axis of the local coordinate system, and the quantity defined by the point **point1** for the creation of a surface of the **surfaceType** type:

surfaceType	**Surface type**	The point **point1** defines
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**, major radius

The point **point1** defines the height of a cylindrical surface, the height of a conical surface, and the major radius of a toroidal surface. The point **point2** defines the radius of a cylindrical surface, the radius of a conical surface and a cone half-angle, the radius of a sphere, and the minor radius of a toroidal surface.

When constructing a plane, the points **point0**, **point1**, and **point2** define the local coordinate system of the plane with the origin at the point **point0**. The **axisX** axis of the local coordinate system of the plane is directed from the point **point0** to the point **point1**; the **axisY** axis of the local coordinate system of the plane is orthogonal to the **axisX** axis and directed toward the point **point2**. The domain of the first parameter is equal to twice the distance between the points **point0** and **point1**. The domain of the second parameter is equal to twice the distance between the point **point0** and the projection of the point **point2** onto the **axisY** axis, Fig. 1:

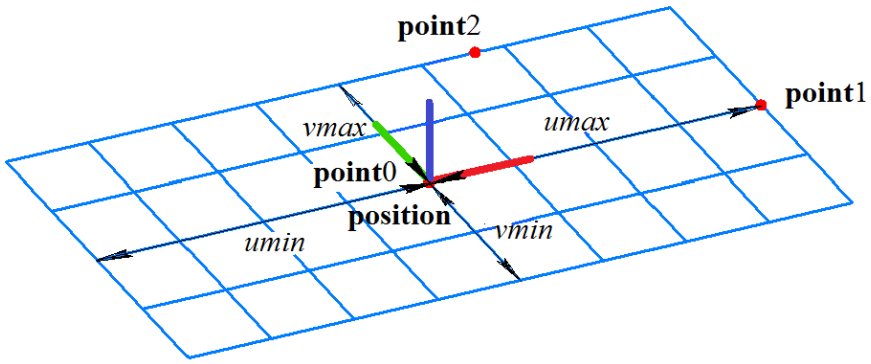


Fig. 1

When constructing a cylindrical surface, the point **point0** defines the center of the bottom base of the cylinder, where the origin of the local coordinate system is located. The point **point1** defines the center of the top base of the cylinder. The **axisZ** axis of the local coordinate system of the surface is directed from the point **point0** to the point **point1**. The point **point2** together with the previous points defines the plane of the **axisX** and **axisZ** axes of the local coordinate system of the surface. The distance between the points **point0** and **point1** defines the height

of the cylindrical surface; the distance from the **axisZ** axis to the point **point2** defines the radius of the cylindrical surface, Fig. 2:

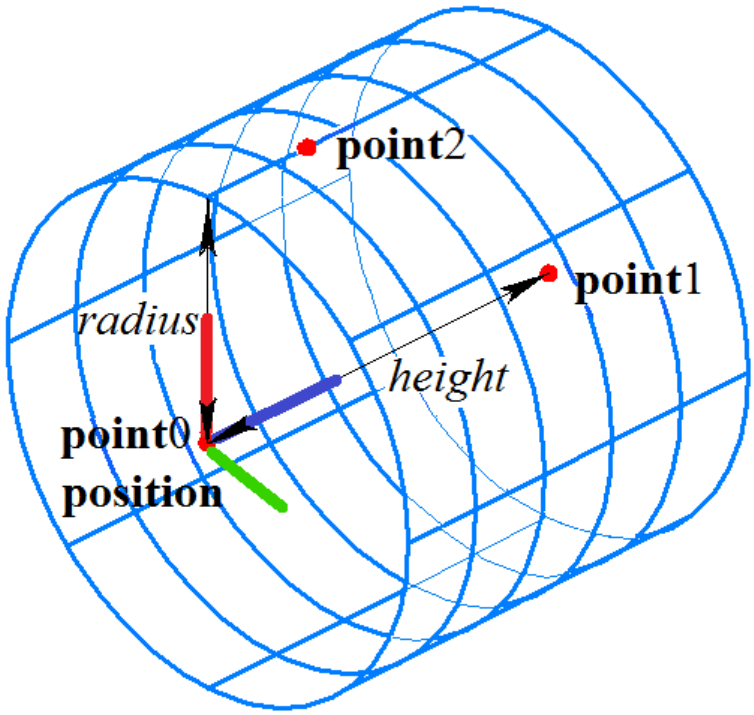


Fig. 2

The cylindrical surface is described in the section *Cylindrical Surface MbCylinderSurface*.

When constructing a conical surface, the point **point0** defines the apex of the cone, where the origin of the local coordinate system is located. The point **point1** defines the center of the cone base and the direction of the **axisZ** axis of the local coordinate system of the cone. The point **point2** together with the previous points defines the plane of the **axisX** and **axisZ** axes of the local coordinate system of the surface. The distance between the points **point0** and **point1** defines the height of the conical surface. The point **point2** defines the cone half-angle, when the point **point2** lies on the conical surface, Fig. 3:

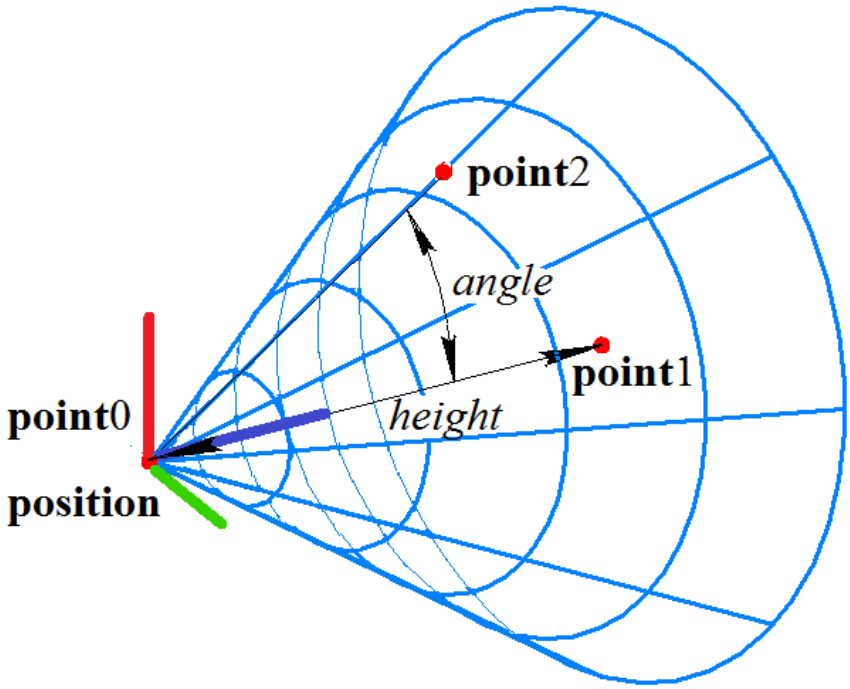


Fig. 3

The conical surface is described in the section *Conical Surface MbConeSurface*.

When constructing a spherical surface, the point **point0** defines the center of the sphere, where the origin of the local coordinate system is located. The **axisZ** axis of the local coordinate system of the surface is directed from the point **point0** to the point **point1**. The point **point2** together with the previous points defines the plane of the **axisX** and **axisZ** axes of the local coordinate system of the surface. The distance between the points **point0** and **point2** defines the radius of the spherical surface, Fig. 4:

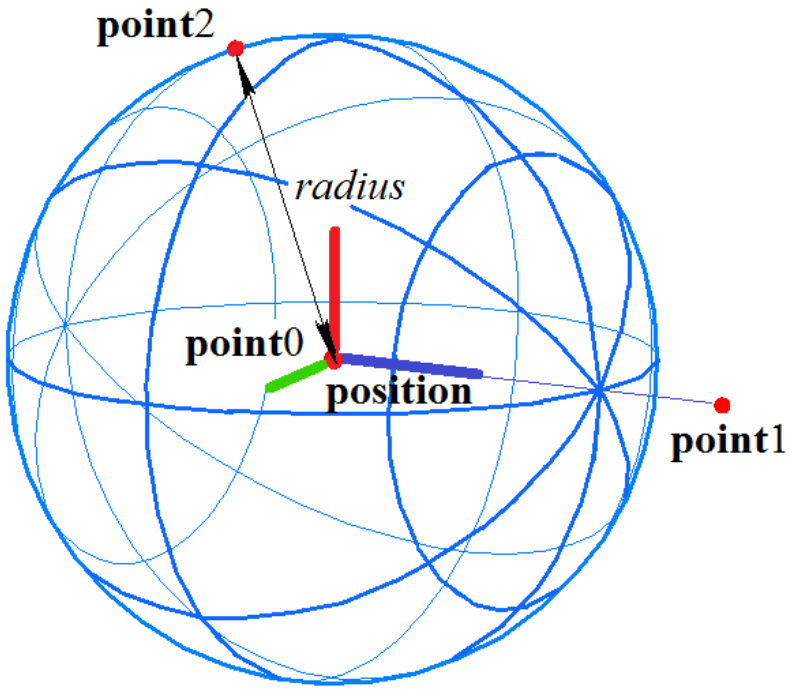


Fig. 4

The spherical surface is described in the section *Spherical Surface MbSphereSurface*.

When constructing a toroidal surface, the point **point0** defines the center of the torus, where the origin of the local coordinate system is located. The **axisX** axis of the local coordinate system of the surface is directed from the point **point0** to the point **point1**. The point **point2** together with the previous points defines the plane in which the **axisX** and **axisZ** axes of the local coordinate system of the surface lie. The distance between the points **point0** and **point1** defines the major radius of the torus; the distance between the points **point1** and **point2** defines the minor radius of the torus, Fig. 5:

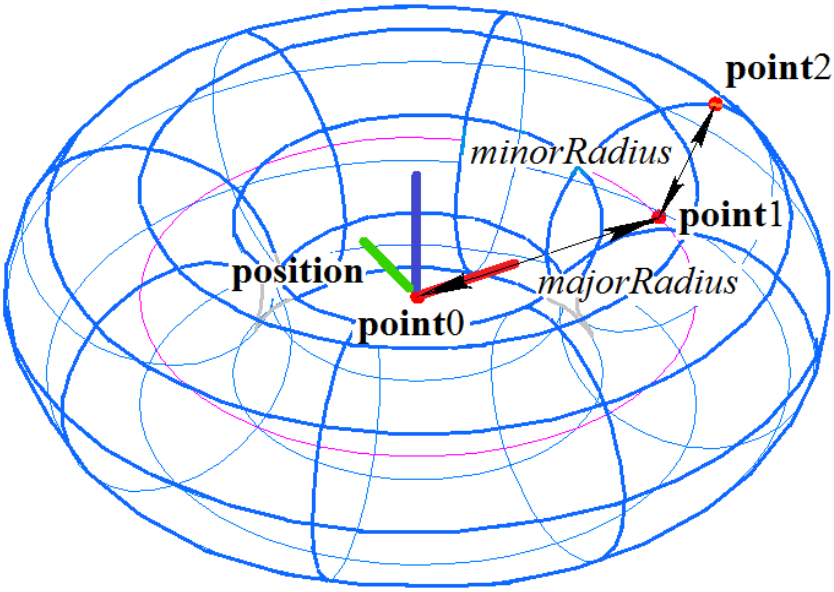


Fig. 5

The toroidal surface is described in the section *Toroidal Surface MbTorusSurface*.

In the test application *test.exe*, the construction of an elementary surface from the given points is performed by the menu command «Create→Surface→Elementary».

3.5.2 NURBS Surface Construction

The function

```
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 NURBS surface by corner control points.

The input parameters of the function are:

- **pUMinVMin** is a left lower corner point of the surface;
- **pUMaxVMin** is a right lower corner point of the surface;
- **pUMaxVMax** is a right upper corner point of the surface;
- **pUMinVMax** is a left upper corner point of the surface;
- **uCount** is a number of control points along the first parameter (horizontally);
- **vCount** is a number of control points along the second parameter (vertically);

- **uDegree** is an order of B-splines along the first parameter;
- **vDegree** is an order of B-splines along the second parameter.

The output parameter of the function is the constructed spline surface **result**.

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration.

The function is declared in the *action_surface.h* file.

This function constructs a NURBS surface whose control points are located at the nodes of a quadrilateral grid with **uCount** columns and **vCount** rows. The order of the surface *B*-splines along the first parameter is equal to **uDegree**, and the order of the surface *B*-splines along the second parameter is equal to **vDegree**. The control points of the surface are calculated under the requirement of uniform distribution and coincidence of the corner points with the points **pUMinVMin**, **pUMaxVMin**, **pUMaxVMax**, **pUMinVMax**, see Fig. 1:

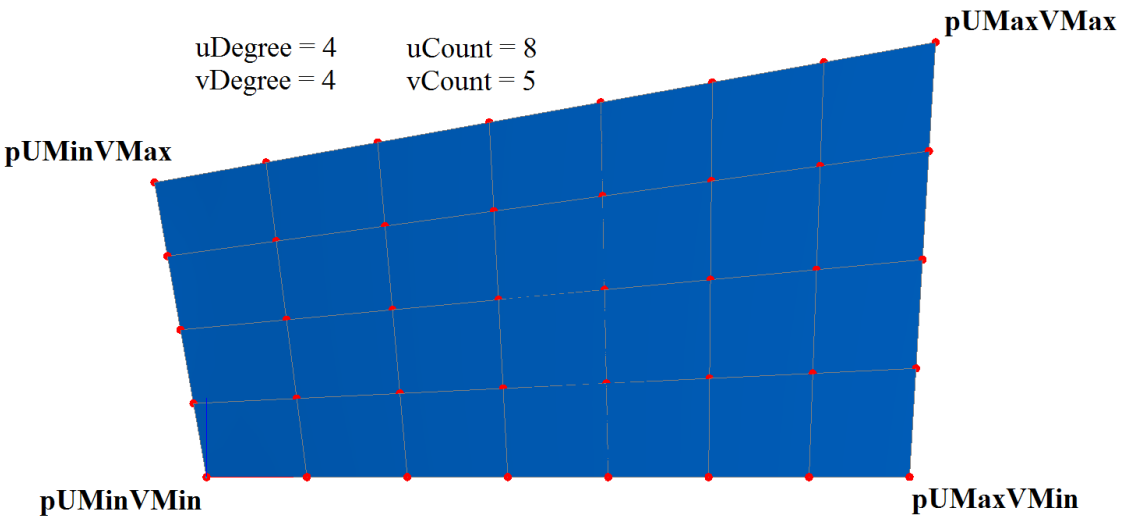


Fig. 1

The weights of all control points are equal to one. The constructed surface is intended for further modification. For the function parameters the following inequalities must hold: **uCount** ≥ **uDegree** and **vCount** ≥ **vDegree**. The NURBS surface is described in subsection *NURBS Surface MbSplineSurface*.

The function

```
MbResultType SplineSurface( const SArray<MbCartPoint3D> & pointList,
                            const SArray<double> &        weightList,
                                  size_t                  uCount,
                                  size_t                  vCount,
                                  size_t                  uDegree,
                            const SArray<double> &        uKnotList,
                                  bool                    uClosed,
                                  size_t                  vDegree,
                            const SArray<double> &        vKnotList,
                                  bool                    vClosed,
                                  MbSurface *&            result );
```

constructs a NURBS surface by control points and their weights.

The input parameters of the function are:

- **pointList** is a set of control points conventionally arranged as **vCount** rows with **uCount** points in each row;
- **weightList** is a set of weights of control points consistent with the set of control points;
- **uCount** is a number of control points along the first parameter (in each row);
- **vCount** is a number of control points along the second parameter (the number of rows);
- **uDegree** is an order of B-splines along the first parameter;
- **uKnotList** is a knot vector of the first parameter;
- **uClosed** is a cyclic closedness of the surface along the first parameter;
- **vDegree** is an order of B-splines along the second parameter;
- **vKnotList** is a knot vector of the second parameter;
- **vClosed** is a cyclic closedness of the surface along the second parameter.

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

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration.

The function is declared in the *action_surface.h* file.

This function constructs a NURBS surface with control points **pointList**. The control points of the **pointList** set are divided into conventional rows and columns. Each row contains **uCount** control points from the **pointList** set, and the number of rows is equal to **vCount**. Thus, the **pointList** set must contain **uCount**·**vCount** control points. The order of the surface *B*-splines along the first parameter is equal to **uDegree**, and the order of the surface *B*-splines along the second parameter is equal to **vDegree**. For the function parameters the following inequalities must hold: **uCount** ≥ **uDegree** and **vCount** ≥ **vDegree**. The NURBS surface is described in subsection *NURBS Surface MbSplineSurface*.

The sets **weightList**, **uKnotList**, **vKnotList** may be empty.

If the **weightList** set is empty, the weights of the control points are equal to one.

If the **uKnotList** or **vKnotList** sets are empty, the knot vectors are generated automatically.

If the **weightList** set is not empty, it must be consistent with the set of control points **pointList**.

If the **uKnotList** and **vKnotList** sets are not empty, they must contain a certain number of elements: for **uClosed** = *false*, the **uKnotList** knot vector must contain **uCount** + **uDegree** elements; for **uClosed** = *true*, the **uKnotList** knot vector must contain **uCount** + 2 · **uDegree** - 1 elements; for **vClosed** = *false*, the **vKnotList** knot vector must contain **vCount** + **vDegree** elements; for **vClosed** = *true*, the **vKnotList** knot vector must contain **vCount** + 2 · **vDegree** - 1 elements. Fig. 2 shows the arrangement of control points for constructing a spline surface in the shape of a torus:

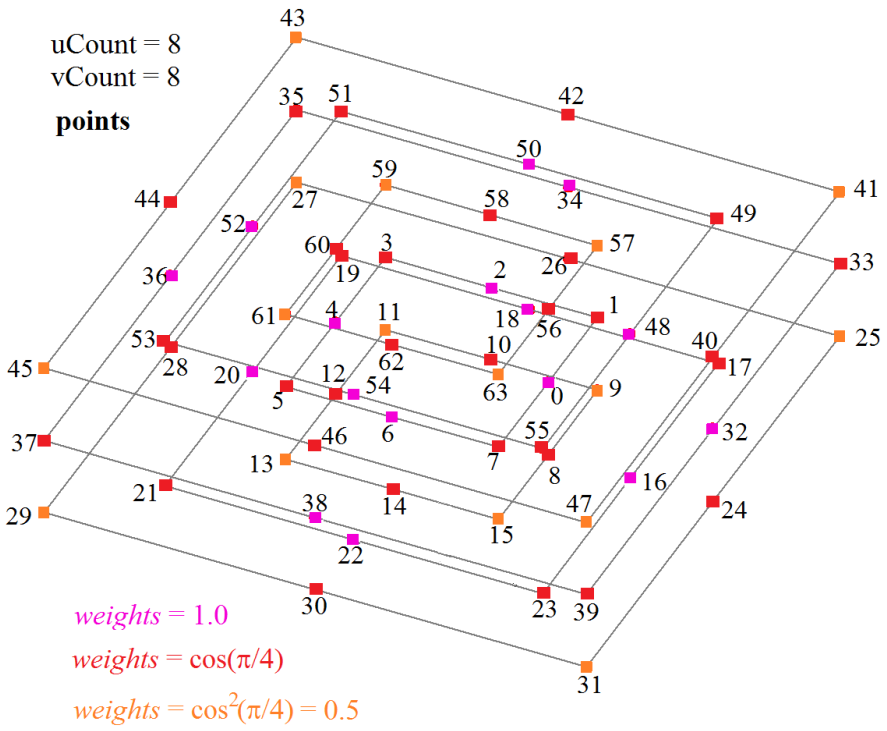


Fig. 2

Fig. 3 shows the spline surface in the shape of a torus constructed on the above control points:

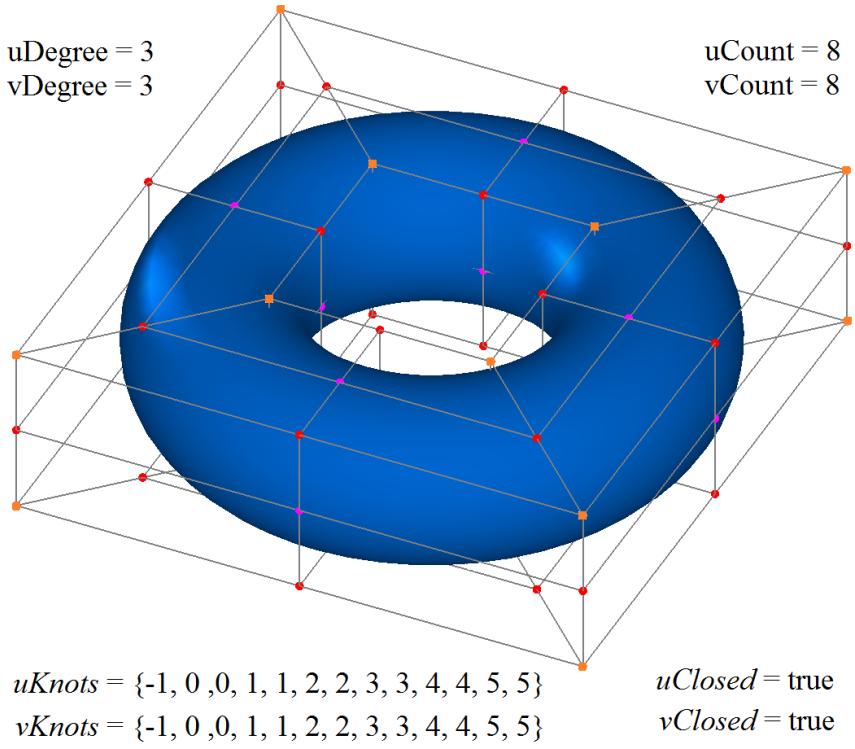


Fig. 3

Below there is an example in the C++ programming language that implements construction of a spline surface in the shape of a torus by this function (the center of the torus coincides with the origin of the global coordinate system, and the torus axis is directed along the global Z-axis):

```
void AddTorusPoints( double r, double z,
SArray<MbCartPoint3D> & pointList, double w,
SArray<double> & weightList )
{
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 );
pointList.Add( p ); weightList.Add( wI );
p.Add( toY, r ); pointList.Add( p ); weightList.Add( wA );
p.Add( toX,-r ); pointList.Add( p ); weightList.Add( wI );
p.Add( toX,-r ); pointList.Add( p ); weightList.Add( wA );
p.Add( toY,-r ); pointList.Add( p ); weightList.Add( wI );
p.Add( toY,-r ); pointList.Add( p ); weightList.Add( wA );
p.Add( toX, r ); pointList.Add( p ); weightList.Add( wI );
p.Add( toX, r ); pointList.Add( p ); weightList.Add( wA );
}

void GetTorusPoints( double majorR, double minorR,
SArray<MbCartPoint3D> & pointList,
SArray<double> & weightList )
{
```

(continues on next page)

(continued from previous page)

```
double zLavel( 0.0 );
double w0( 1.0 ), wA = ::cos( M_PI_4 );
::AddTorusPoints( majorR - minorR, zLavel, pointList, w0, weightList );
::AddTorusPoints( majorR - minorR,-minorR, pointList, wA, weightList );
::AddTorusPoints( majorR ,-minorR, pointList, w0, weightList );
::AddTorusPoints( majorR + minorR,-minorR, pointList, wA, weightList );
::AddTorusPoints( majorR + minorR, zLavel, pointList, w0, weightList );
::AddTorusPoints( majorR + minorR, minorR, pointList, wA, weightList );
::AddTorusPoints( majorR , minorR, pointList, w0, weightList );
::AddTorusPoints( majorR - minorR, minorR, pointList, wA, weightList );
}

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> pointList( 64, 1 );
SArray<double> weightList( 64, 1 );
SArray<double> uKnotList( 13, 1 );
SArray<double> vKnotList( 13, 1 );
size_t uDegree( 3 ), vDegree( 3 );
size_t uCount( 8 ), vCount( 8 );
bool uClosed( true ), vClosed( true );
::GetTorusPoints( majorR, minorR, pointList, weightList );
::GetTorusKnots( 0.0, 4.0, uKnotList );
::GetTorusKnots( -2.0, 2.0, vKnotList );
res = ::SplineSurface( pointList, weightList, uCount, vCount, uDegree, uKnotList,
uClosed, vDegree, vKnotList, vClosed, result );
}
return ( res == rt_Success ) ? result : NULL;
}
```

The function

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

constructs a NURBS copy of a given surface.

The input parameters of the function are:

- **surface** is the source surface;
- **version** is the construction version.

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

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration.

The function is declared in the *action_surface.h* file.

This function constructs a NURBS surface whose shape reproduces the shape of the source surface. The NURBS surface is described in subsection *NURBS Surface MbSplineSurface*. For most surface types the NURBS copy coincides in shape completely with the source surface. If it is impossible to reproduce the shape of the source surface exactly, the NURBS copy approximates the source surface.

If the source surface is a surface with arbitrary borders MbCurveBoundedSurface, which is described in subsection *Surface with Arbitrary Borders MbCurveBoundedSurface*, this function constructs a surface with arbitrary borders whose base surface is a NURBS copy of the base surface of the source surface.

In the test application test.exe, construction of a NURBS surface is performed by the menu command «Create → Surface → Based on points → NURBS».

3.5.3 Construction of an extrusion surface

The function

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

constructs an extrusion surface.

The input parameters of the function are:

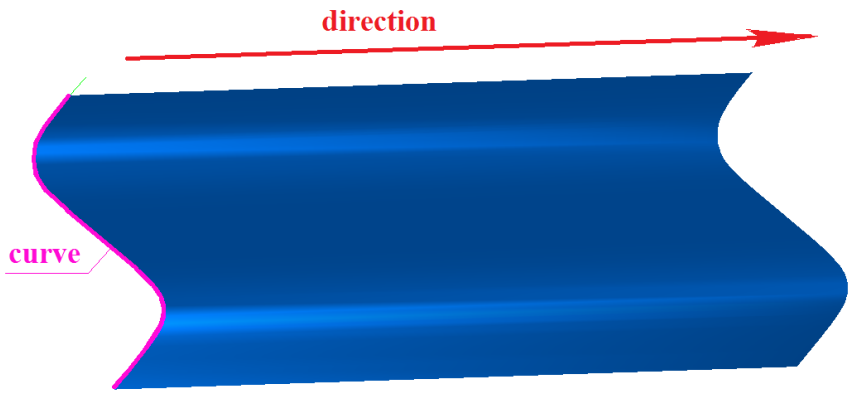
- **curve** — a generatrix curve;
- **direction** — an extrusion vector that defines a direction and a length;
- **simplify** — a flag for simplifying a surface, if it is possible (for example, reducing it to a simpler analytical type when it is allowed).

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

If successful, the function returns rt_Success; otherwise, it returns a result code from the MbResultType enumeration (see *mb_operation_result.h*).

The function is declared in the *action_surface.h* file.

An extrusion surface is one of the motion surfaces obtained by moving a generatrix curve along a given trajectory (a directing curve). For an extrusion surface, the trajectory is a straight-line segment defined by the **direction** vector. The extrusion surface is described in *MbExtrusionSurface extrusion surface*. Figure shows a surface constructed by extrusion of a curve along a given vector:



In the test application *test.exe*, construction of an extrusion surface is performed by the menu command “Create → Surface → Based on curves → Extrusion”.

3.5.4 Construction of a Revolution Surface

The function

```
MbResultType RevolutionSurface( const MbCurve3D &       curve,
                                const MbCartPoint3D &   origin,
                                const MbVector3D &      axis,
                                      double            angle,
                                      bool              simplify,
                                      MbSurface *&      result );
```

constructs a revolution surface.

The input parameters of the function are:

- **curve** is a generatrix curve;
- **origin** is a point defining the position of the rotation axis;
- **axis** is a direction of the rotation axis;
- **angle** is a rotation angle;
- **simplify** is a flag for simplifying the resulting surface (if possible, an elementary surface is created, for example, a sphere, a torus, a cylinder, or a cone instead of a general revolution surface).

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

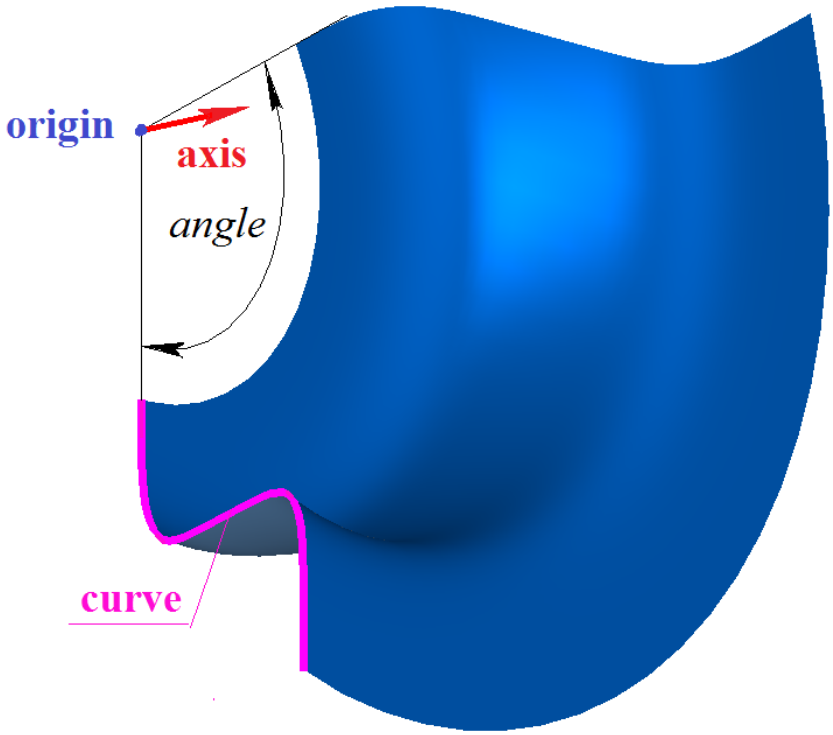
If successful, the function returns rt_Success; otherwise, it returns a result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_surface.h* file.

A revolution surface is obtained by moving the generatrix curve along an arc of a circle whose center lies on an axis passing through the **origin** point in the direction of the **axis** vector, and whose sweep angle is equal to **angle**.

If the **simplify** flag is set to *true*, the function tries to create a simplified (elementary) surface. For example, when a straight line is revolved about an axis, a cylindrical, conical, or planar surface can be created; when an arc of a circle is revolved, a spherical or toroidal surface can be created.

A revolution surface is described in subsection *Revolution surface MbRevolutionSurface*. Figure shows a surface constructed by revolving a curve about an axis through a given angle:



In the test.exe application, construction of a revolution surface is performed by the menu command:

"Create → Surface → Based on curves → Revolution".

3.5.5 Construction of a Surface Based on a Curve Grid

The function

```
MbResultType MeshSurface( const RPArray<MbCurve3D> & uCurveList,
                          const RPArray<MbCurve3D> & vCurveList,
                                MbSurface *&         result );
```

constructs a surface based on two families of curves.

The input parameters of the function are:

- **uCurveList** is a family of curves along the U-direction;
- **vCurveList** is a family of curves along the V-direction.

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

If successful, the function returns rt_Success; otherwise, it returns a result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_surface.h* file.

To construct a surface, each curve from the **uCurveList** set must intersect all curves from the **vCurveList** set, and, respectively, each curve from the **vCurveList** set must intersect all curves from the **uCurveList** set. The constructed surface passes through the curves of the **uCurveList** set and the **vCurveList** set.

A surface constructed on a network of curves is described in subsection *MbMeshSurface surface based on a network of curves*. Figs. 1 and 2 show two sets of curves and the surface constructed on these curves:

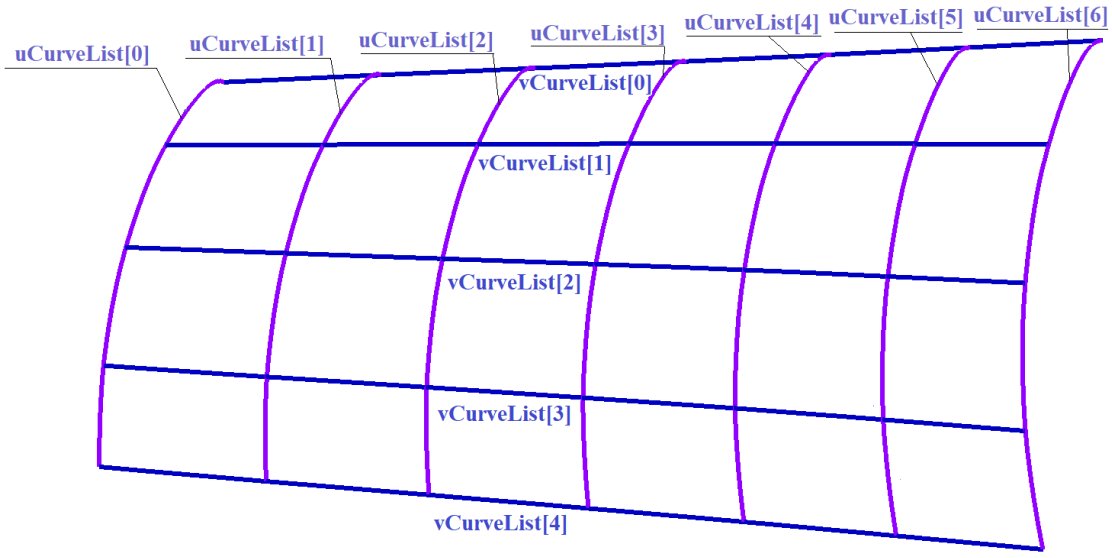


Fig. 1

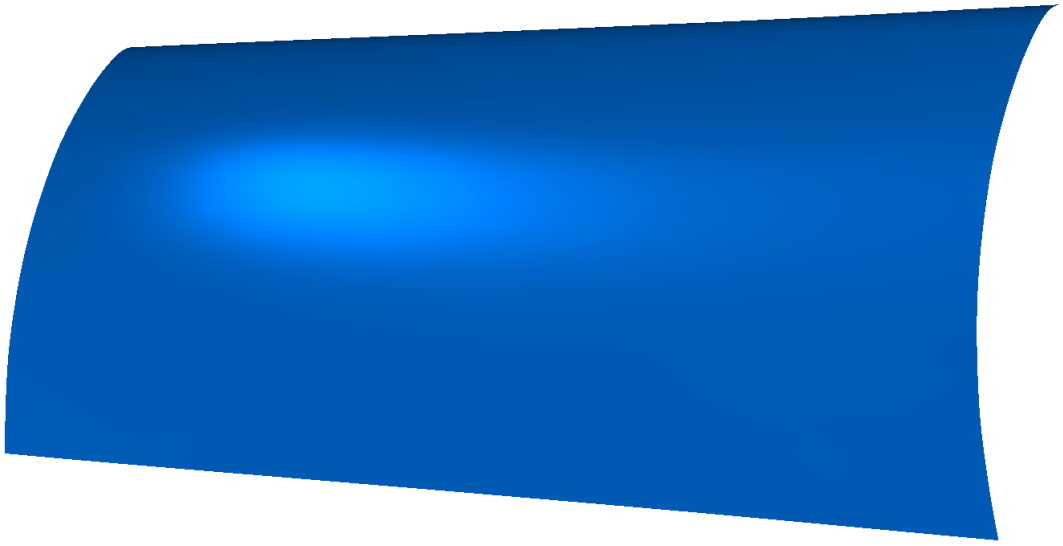


Fig. 2

In the test application *test.exe*, construction of a surface based on a curve grid is performed by

the menu command “Create → Surface → Based on curves → Based on a mesh of curves”.

3.5.6 Construction of a Surface with Arbitrary Borders

The function

```
MbResultType BoundedSurface(       MbSurface &          surface,
                            const  RPArray<MbCurve> &   boundList,
                                   MbSurface *&         result );
```

constructs a surface with a given border.

The input parameters of the function are:

- **surface** – a source surface;
- **boundList** – a set of two-dimensional closed curves in the parameter space (the (u,v) plane).

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

If successful, the function returns rt_Success; otherwise, it returns a result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_surface.h* file.

By default, surfaces have a rectangular parameter domain. The initial parameter domain can be changed by defining a border of the domain with the two-dimensional closed curves **boundList**. If there are more than one curve in the **boundList** set, then the first curve of the **boundList** set must describe an outer border, and the remaining curves must lie inside the first curve of the set. The curves of the **boundList** set may go beyond the initial parameter domain, if there are no singular points (poles, parametrization breaks) there. If the set of borders is empty, then a closed composite curve is constructed along the initial border of the parameter domain of the **surface** surface.

A surface with arbitrary borders is described in subsection *Surface with Arbitrary Borders MbCurveBoundedSurface*. Fig. 1 shows a surface and two closed curves on it, and Fig. 2 shows the result of constructing a surface with borders of the shape of these curves:

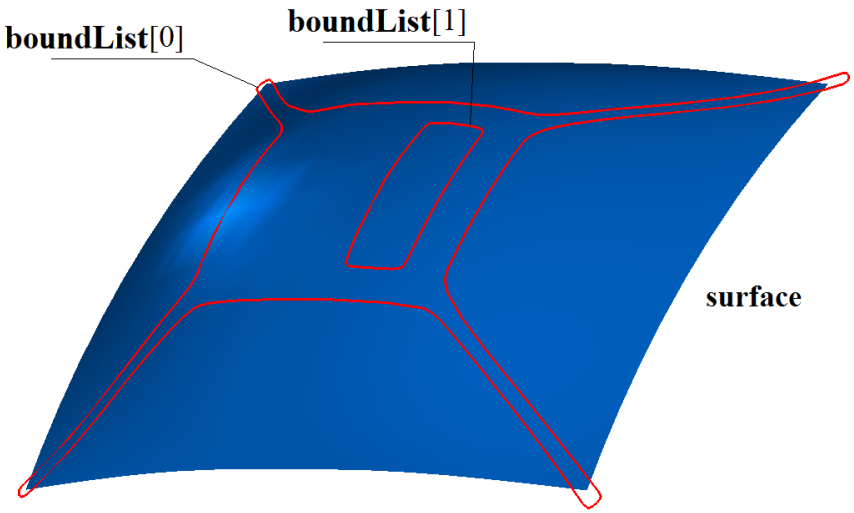


Fig. 1

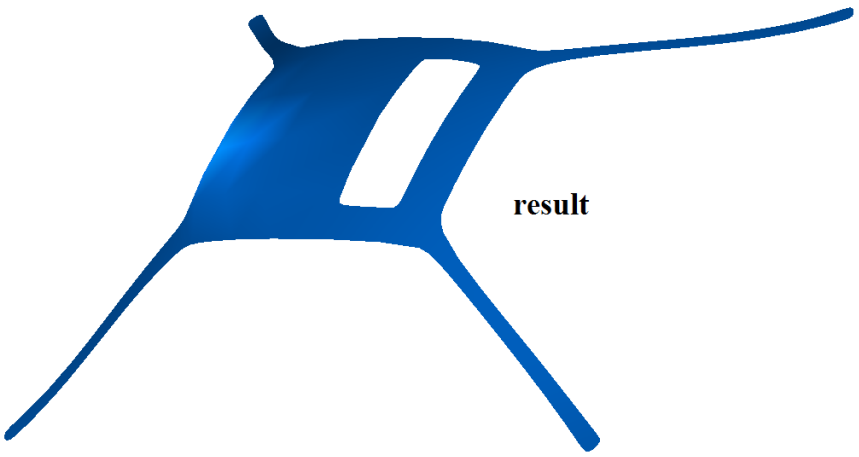


Fig. 2

3.6 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.

3.6.1 Constructing a Modified Solid

The function

```
MbResultType ModifiedSolid(       MbSolid &                 solid,
                                  MbeCopyMode               sameShell,
                            const MbModifiedSolidParams &   params,
                                  MbModifiedSolidResults &  results );
```

Depending on the modification parameters MbModifiedSolidParams the function performs one of the following actions:

1. Removal of the selected faces from the solid without adjacent fillets (*dmm_RemoveFaces*).

   Removal of the selected faces together with adjacent fillets (*dmm_RemoveFacesWithAdjacentFillets*).

   Removal of closed cylindrical, conical, toroidal, spherical faces, revolution and extrusion faces whose radius is less than a given value and without adjacent fillets (*dmm_RemoveBelowRadius*).

   Removal of closed cylindrical, conical, toroidal, spherical faces, revolution and extrusion faces whose radius is less than a given value together with adjacent fillets (*dmm_RemoveBelowRadiusWithAdjacentFillets*).

2. Creation of a solid from selected faces with an environment (*dmm_Create*).

3. Translation of the selected faces with an environment relative to the remaining faces of the solid (*dmm_MoveFacesByVector*) or without adaptation of adjacent fillets (*dmm_MoveFacesByVectorNoAdaptation*). The translation direction and distance are defined by the vector **direction**.
4. Replacement of the selected faces of the solid with offset faces (translation along the normal, radius change) (*dmm_OffsetFaces*). The offset distance is defined by the parameter **value**.
5. Change of the radius of the selected fillet faces by a given value (*dmm_FilletsRadiusChange*). The value of the radius change is defined by the parameter **value**.
6. Replacement of the selected faces of the solid with deformable faces (conversion to NURBS) for editing (*dmm_ConvertFacesToNurbs*).
7. Removal of the selected fillet faces of the solid (*dmm_PurifyFillets*).

   Removal of the selected fillet faces together with all smoothly connected fillets (*dmm_PurifyFilletsChains*).

   Removal of all fillets of the solid whose radius is less than a given value (*dmm_PurifyFilletsBelowRadius*).
8. Merging of edge vertices and removal of edges (*dmm_DeleteEdges*).
9. Replacement of smoothly joined faces with a single face (*dmm_UniteFaces*).
10. Rotation of the selected faces with adaptation of adjacent fillets (*dmm_RotateFaces*) or without it (*dmm_RotateFacesNoFilletAdaptation*). The rotation angle is defined by the parameter **angle**, the rotation axis is defined by the axis **axis**.
11. Replacement of the selected faces of the solid (*dmm_ReplaceFaces*).
12. Setting the radii of the selected fillet faces for a chain of smoothly joined fillets (*dmm_FilletsRadiusSet*) and setting the radii of the selected fillet faces (*dmm_FilletsRadiusSetSingle*). The new radius is defined by the parameter **value**.

The input parameters of the function are:

- **solid** – the initial solid;
- **sameShell** – the mode of copying of the initial solid;
- **params** – the modification parameters.

The output parameter of the function is the modification results structure **results**, which contains the modified solid and additional information about the operation.

If successful, the function returns rt_Success; otherwise it returns an error code from the enumeration MbResultType (see *mb_operation_result.h*).

Possible error codes:

rt_Success – no errors;

rt_Error – unknown error;

rt_IncorrectData – incorrect data;

rt_ParameterError – wrong parameter.

The function is declared in the file *action_direct.h*.

When the object **params** with modification parameters is created, one of the following constructors is used (see *op_direct_mod_parameter.h*).

Constructors without a faces set

- **type** – the modification type. The following types are available for this constructor: *dmm_OffsetFaces*, *dmm_FilletsRadiusChange*;
- **value** – the offset distance for *dmm_OffsetFaces* or the radius change value for *dmm_FilletsRadiusChange*;
- **names** – the name maker of the operation.

```
MbModifiedSolidParams(       MbeModifyingMode  type,
                       const MbSNameMaker &    names,
                             double            value );
```

- **type** – the modification type. The following types are available for this constructor: *dmm_RotateFaces*, *dmm_RotateFacesNoFilletAdaptation*;
- **axis** – the rotation axis;
- **angle** – the rotation angle;
- **names** – the name maker of the operation.

```
MbModifiedSolidParams(       MbeModifyingMode  type,
                       const MbSNameMaker &    names,
                       const MbAxis3D &        axis,
                             double            angle );
```

- **type** – the modification type. The following types are available for this constructor: *dmm_MoveFacesByVector*, *dmm_MoveFacesByVectorNoAdaptation*;
- **direction** – the translation direction;
- **names** – the name maker of the operation.

```
MbModifiedSolidParams(       MbeModifyingMode  type,
                       const MbSNameMaker &    names,
                       const MbVector3D &      direction );
```

Constructors with a faces set

- **modType** – the modification type. The following types are available for this constructor: *dmm_UniteFaces*, *dmm_Create*, *dmm_ConvertFacesToNurbs*, *dmm_PurifyFillets*;
- **names** – the name maker of the operation;
- **faces** – the list of faces to be modified.

```
MbModifiedSolidParams(       MbeModifyingMode   modType,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces );
```

- **type** – the modification type. The following types are available for this constructor: *dmm_OffsetFaces*, *dmm_FilletsRadiusChange*, *dmm_RemoveBelowRadius*, *dmm_RemoveBelowRadiusWithAdjacentFillets*, *dmm_PurifyFilletsBelowRadius*;

- **names** – the name maker of the operation;
- **faces** – the list of faces to be modified;
- **value** – the offset distance for *dmm_OffsetFaces*, the radius change value for *dmm_FilletsRadiusChange*, the maximum radius value for *dmm_Remove* and *dmm_Purify*.

```
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                             double             value )
: MbModifiedSolidParams( type, names, value )
{
  AddFaces( faces );
}
```

- **type** – the modification type. The following types are available for this constructor: *dmm_RotateFaces*, *dmm_RotateFacesNoFilletAdaptation*;
- **names** – the name maker of the operation;
- **faces** – the list of faces to be modified;
- **axis** – the rotation axis;
- **angle** – the rotation angle.

```
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                       const MbAxis3D &         axis,
                             double             angle )
: MbModifiedSolidParams( type, names, axis, angle )
{
  AddFaces( faces );
}
```

- **type** – the modification type. The following types are available for this constructor: *dmm_MoveFacesByVector*;
- **names** – the name maker of the operation;
- **faces** – the list of faces to be modified;
- **direction** – the shift direction.

```
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                       const MbVector3D &       direction )
: MbModifiedSolidParams( type, names, direction )
{
  AddFaces( faces );
}
```

- **type** – the modification type. The following types are available for this constructor: *dmm_ConvertFacesToNurbs*;
- **names** – the name maker of the operation;
- **faces** – the list of faces to be modified;
- **surfaces** – the replacement surfaces (may be an empty array).

```
template<class SurfArray>
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                       const SurfArray &        surfaces )
: MbModifiedSolidParams( type, names )
{
  AddFaces( faces );
  AddSurfaces( surfaces );
}
```

- **type** – the modification type. The following types are available for this constructor: *dmm_ReplaceFaces*;
- **names** – the name maker of the operation;
- **replacedFaces** - the list of faces to be replaced;
- **replacementSurfaces** – the list of replacement surfaces;
- **surfaceParts** – the list of surface part numbers that are used when several replacement options are possible;
- **sameSurfaces** – whether to use the source surfaces (true) or their copies (false);
- **offsets** – the list of offsets from the replacement surfaces.

```
template<class SurfArray> MATH_FUNC_EX
MbModifiedSolidParams(       MbeModifyingMode     type,
                       const MbSNameMaker &       names,
                       const c3d::ItemIndices &   replacedFaces,
                       const SurfArray &          replacementSurfaces,
                             bool                 sameSurfaces,
                       const c3d::NumbersVector & surfaceParts,
                       const c3d::DoubleVector &  offsets );
```

- **modType** – the modification type. The following types are available for this constructor: *dmm_DeleteEdges*;
- **names** – the name maker of the operation;
- **edges** – the list of edges to be modified;
- **beginOrEnd** – the flag specifying whether the beginning or the end of the corresponding edge is to be deleted.

```
MbModifiedSolidParams(       MbeModifyingMode         modType,
                       const MbSNameMaker &           names,
                       const c3d::EdgeFacesIndices & edges,
                       const c3d::BoolVector &       beginOrEnd )
: MbModifiedSolidParams( modType, names )
{
  AddEdges( edges, beginOrEnd );
}
```

The parameter **solid** contains the initial solid to be processed.

The parameter **sameShell** controls the data transfer from the initial solid **solid** to the constructed solid **results** and can take one of four values:

- *cm_Same* = 0 (the solid is not copied);
- *cm_KeepHistory* (the initial solid and its copy share base surfaces, vertices and faces that are not changed by the operation);
- *cm_KeepSurface* (the initial solid and its copy share base surfaces);
- *cm_Copy* (the initial solid and its copy do not share any data).

The enumeration MbeCopyMode is described in detail in *Copying a Set of Faces*.

The parameter **params** describes the modification parameters that are defined by the modification type and by additional parameters depending on the constructor used.

The modification type is defined by the parameter **_modificationType**, which can take one of the values of the enumeration MbeModifyingMode (see *op_direct_mod_parameter.h*).

For the modification types *dmm_RemoveFaces* and *dmm_RemoveFacesWithAdjacentFillets* the function removes the faces **faces** from a copy of the solid **solid** and, using the environment, processes the place of the removed faces while keeping the solid closed. If the faces **faces** are connected to the other faces by fillets, then in the *dmm_RemoveFacesWithAdjacentFillets* mode the adjacent fillet faces are automatically added to the set of faces to be removed.

For the modification types *dmm_RemoveBelowRadius* and *dmm_RemoveBelowRadiusWithAdjacentFillets* the function removes closed cylindrical, conical, toroidal, spherical faces as well as revolution and extrusion faces whose radius is less than or equal to the given radius value. In the *dmm_RemoveBelowRadiusWithAdjacentFillets* mode these faces are removed together with adjacent fillets. Such a setting allows to remove small through or blind holes automatically without explicitly listing all corresponding faces.

Fig. 1 shows an initial solid, and Fig. 2 shows the solid obtained by removing the specified faces of the initial solid:

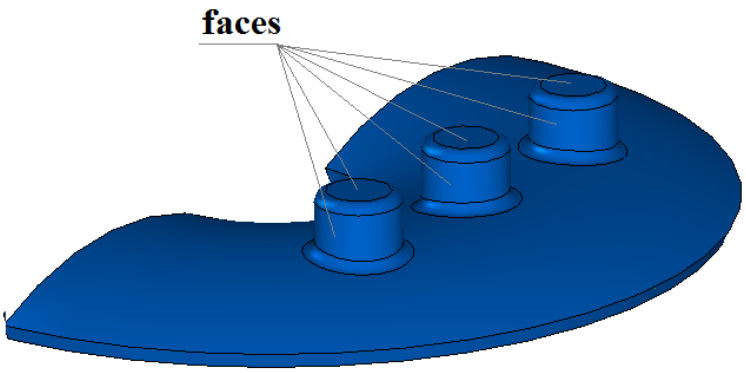


Fig. 1

_modificationType = *dmm_RemoveFacesWithAdjacentFillets*

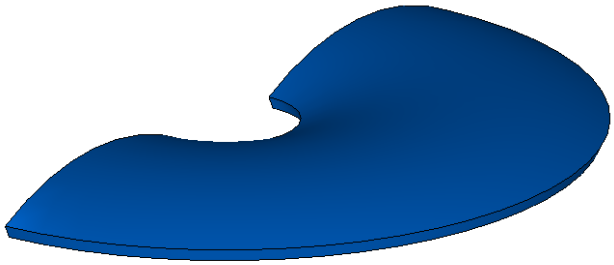


Fig. 2

The same result can be obtained by selecting all the specified faces together with the adjacent fillets and using the *dmm_RemoveFaces* type.

For the modification type *dmm_Create* the function creates a new solid from copies of the faces **faces**. If the faces **faces** are connected to other faces by fillets, the adjacent fillets are removed first and then a new solid is created from the faces **faces**. The new solid is closed because it contains both the original faces **faces** and faces constructed on the base surfaces that are adjacent to the faces **faces**. Additional faces are required to eliminate boundary edges.

Fig. 3 shows an initial solid, and Fig. 4 shows the solid constructed from the specified faces of the initial solid. The solid in Fig. 3 consists of five faces: three faces from the set **faces** and two faces based on the cylindrical surface that supported the two removed fillets:

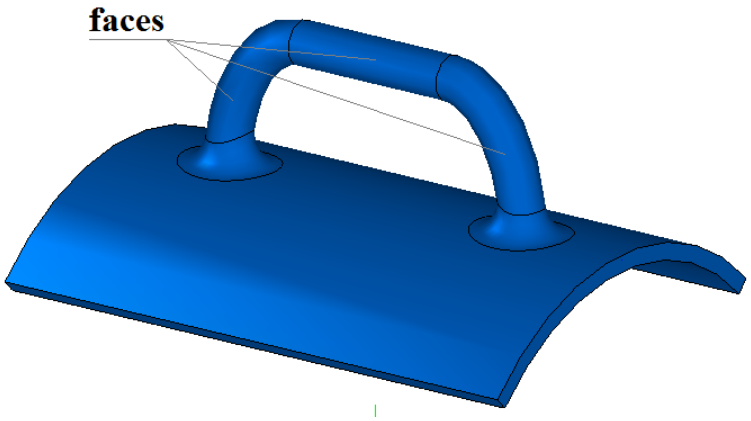


Fig. 3

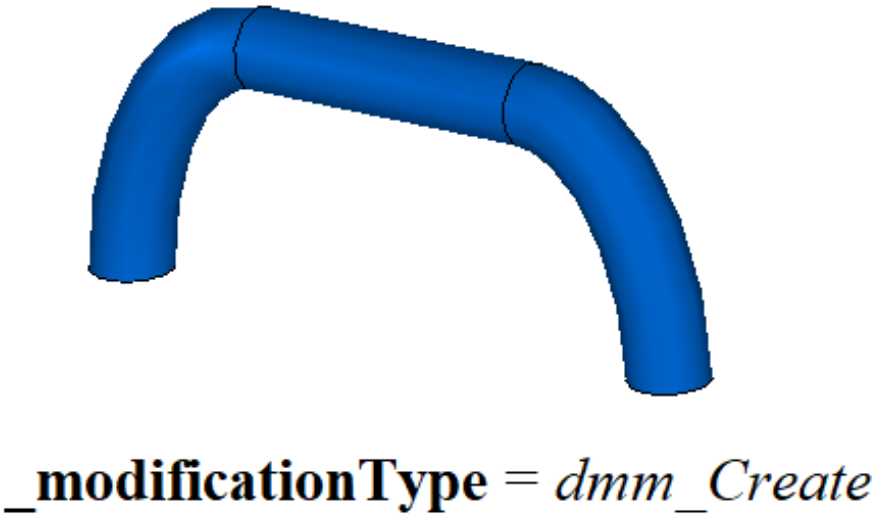
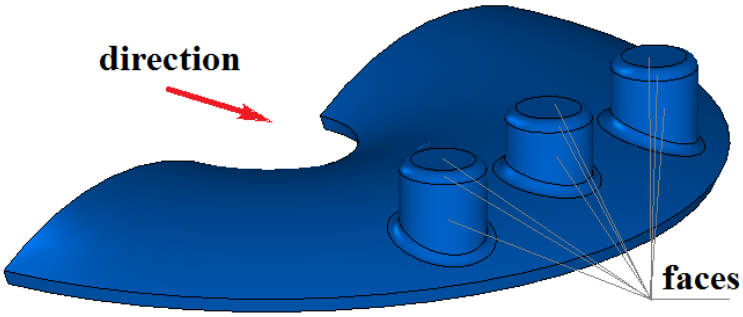


Fig. 4

For the modification types *dmm_MoveFacesByVector* and *dmm_MoveFacesByVectorNoAdaptation* the function translates the faces **faces** relative to the remaining faces in a copy of the solid **solid**. The translation is performed in the direction of the vector **direction** by a distance equal to its length. In the *dmm_MoveFacesByVector* mode the adjacent fillets are adapted; in the *dmm_MoveFacesByVectorNoAdaptation* mode the fillets are not rebuilt.

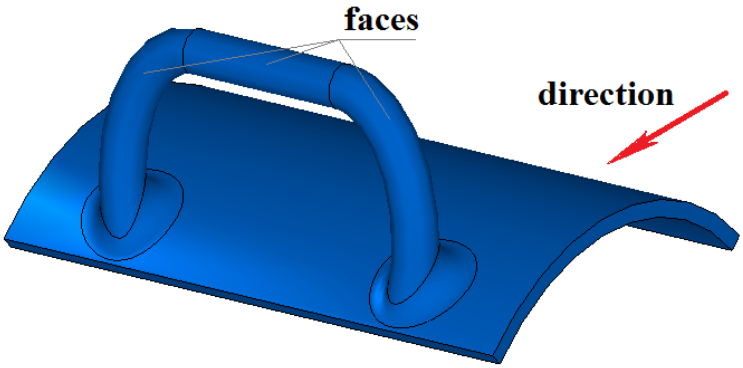
If the faces **faces** are connected to other faces by fillets, then in the mode with adaptation the fillets are removed first, then the faces **faces** are translated, and finally the fillets are restored in their new positions.

Fig. 5 and Fig. 6 present solids obtained by translating the specified faces of the initial solids shown in Fig. 1 and Fig. 3, respectively:



_modificationType = *dmm_MoveFacesByVector*

Fig. 5



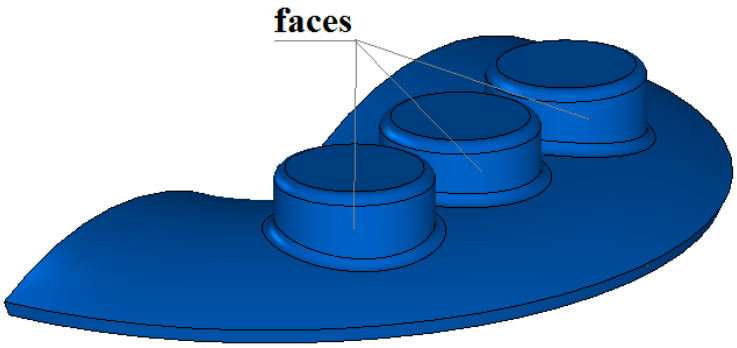
_modificationType = *dmm_MoveFacesByVector*

Fig. 6

For the modification type *dmm_OffsetFaces* the function replaces the faces **faces** in a copy of the solid **solid** with offset faces. The offset faces are translated along the normal by a distance defined by the parameter **value**. If **value** is positive, the translation is performed towards the outer normals; if it is negative, the translation is performed in the opposite direction.

If the faces **faces** are connected to other faces by fillets, the fillets are removed first, then the faces **faces** are replaced with offset faces, and finally the fillets are restored in their new positions.

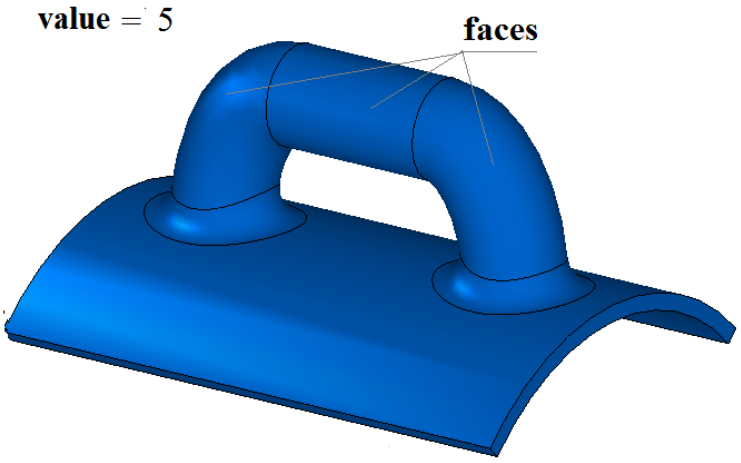
Fig. 7 and Fig. 8 present solids obtained by replacing the specified faces with offset faces. The initial solids are shown in Fig. 1 and Fig. 3, respectively:



value = 10

_modificationType = *dmm_OffsetFaces*

Fig. 7



_modificationType = *dmm_OffsetFaces*

Fig. 8

If the parameter **value** is negative, the offset faces are translated by a negative distance. Fig. 9 and Fig. 10 present solids obtained by replacing the specified faces with offset faces with negative offset. The initial solids are shown in Fig. 5 and Fig. 6, respectively:

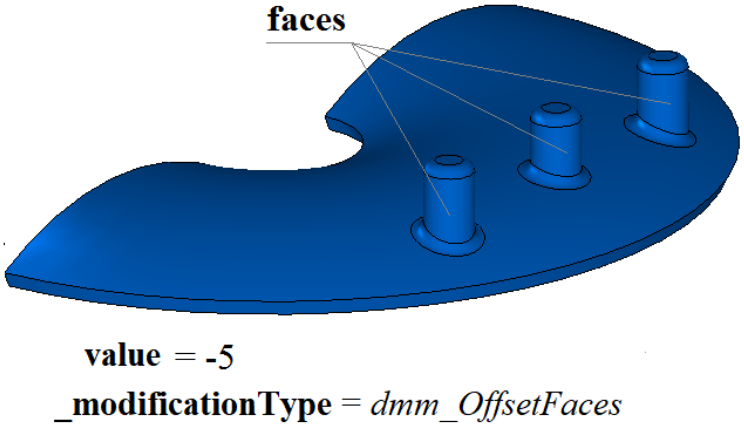


Fig. 9

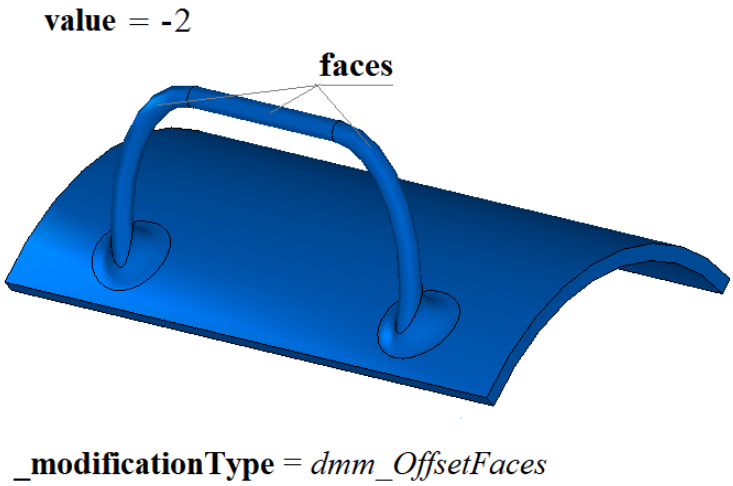


Fig. 10

For the modification type *dmm_FilletsRadiusChange* the function changes the radii of the fillet faces **faces** in a copy of the solid **solid**. The radius of each fillet face is increased or decreased by the value defined by the parameter **value**.

A positive value of **value** increases the fillet radius, a negative value decreases it. At the same time, the solid remains closed and the continuity of connection with adjacent faces is preserved as far as possible for the given geometry.

Fig. 11 and Fig. 12 show solids obtained by changing the radius of the specified fillet faces. The initial solids are shown in Fig. 1 and Fig. 3, respectively:

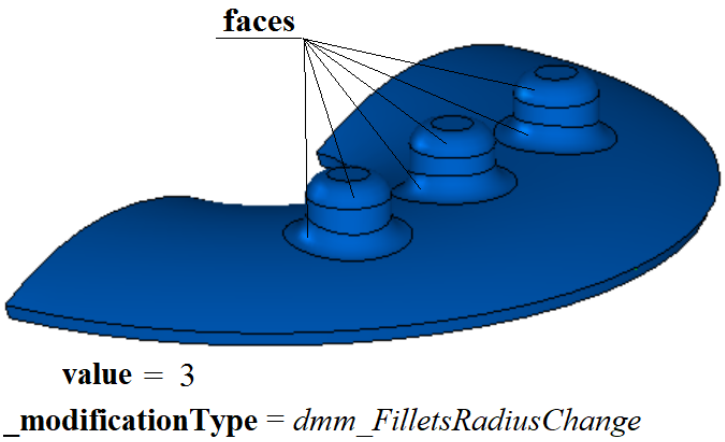


Fig. 11

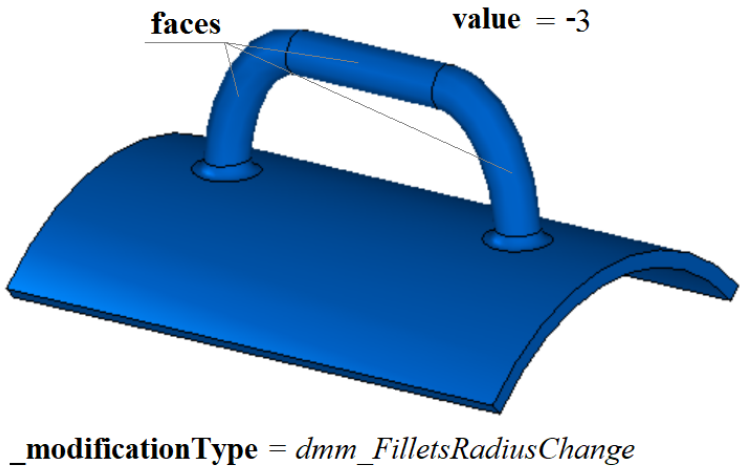


Fig. 12

For the modification type *dmm_ConvertFacesToNurbs* the function replaces the faces **faces** in a copy of the solid **solid** with deformable faces (NURBS-surfaces), which makes it possible to edit these faces by control points.

If the faces **faces** are connected to other faces by fillets, the adjacent fillets are removed first, then the faces **faces** are replaced with deformable faces based on the corresponding NURBS-surfaces, and finally the fillets are restored in their new positions.

In Fig. 13 an initial solid is shown, and in Fig. 14 the solid obtained by replacing the specified face of the initial solid with a deformable face is shown; the control points of the deformable face are also shown. The deformable face is constructed on the basis of a NURBS-surface:

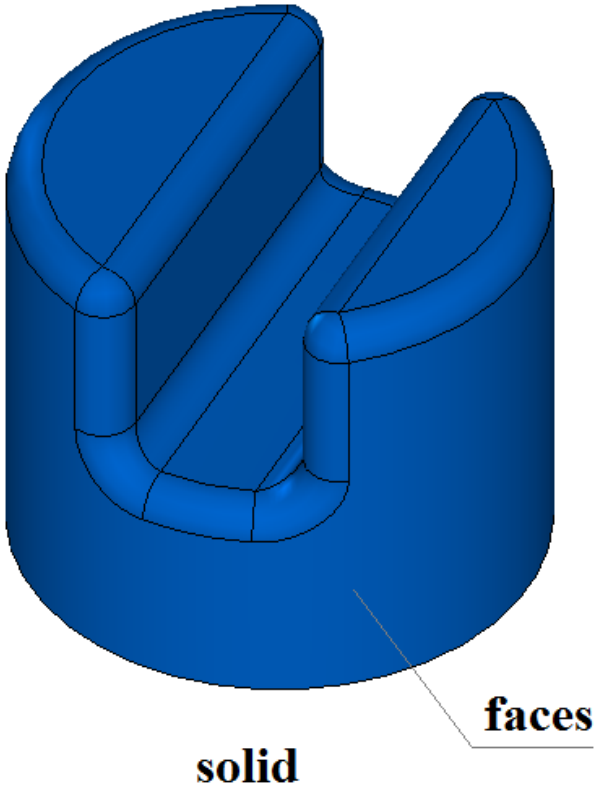
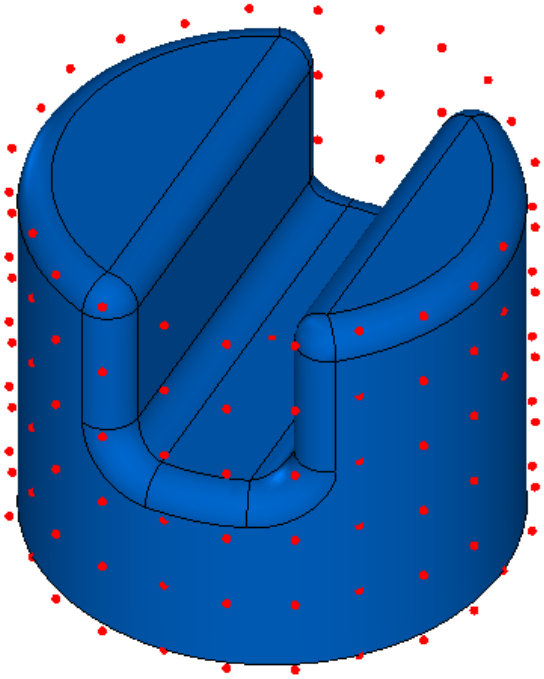


Fig. 13



_modificationType =

dmm_ConvertFacesToNurbs

Fig. 14

Fig. 15 shows a deformable solid with moved control points of the deformable face. Fig. 16 presents the result of deforming the solid:

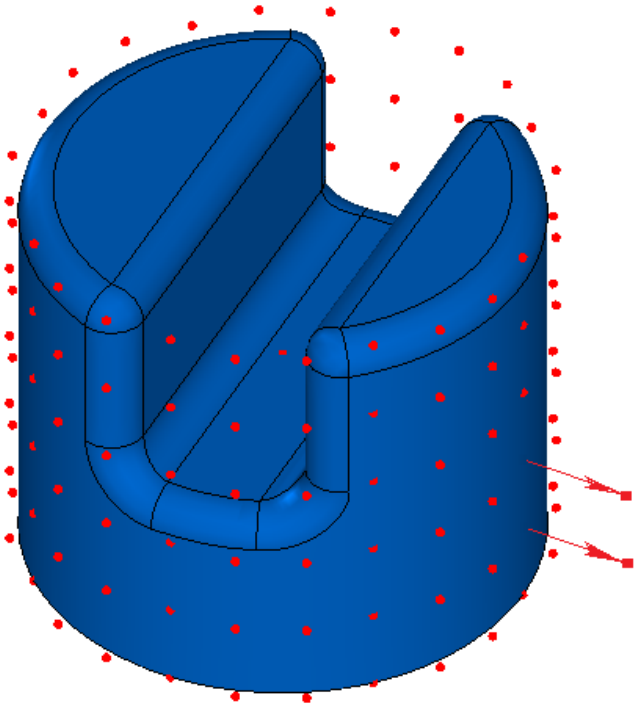
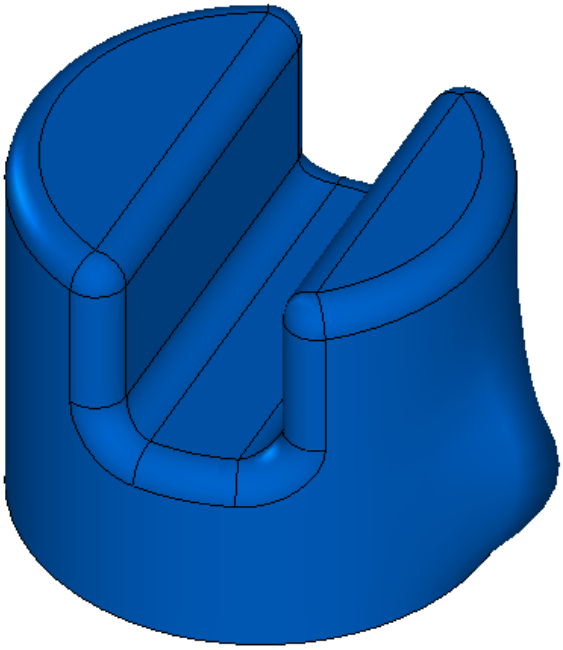


Fig. 15

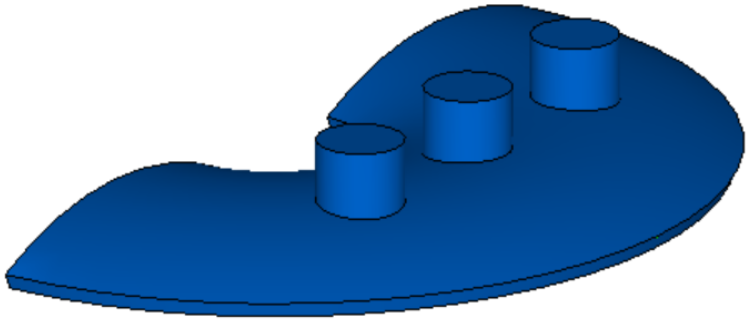


result

Fig. 16

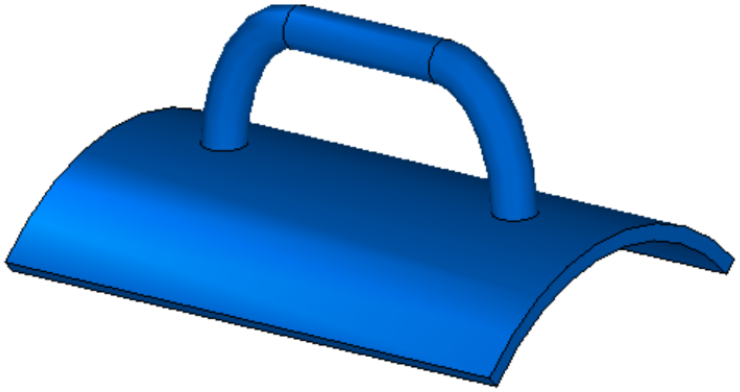
For the modification types *dmm_PurifyFillets* and *dmm_PurifyFilletsChains* the function removes the fillet faces **faces** in a copy of the solid **solid** and, using the environment, processes the place of the removed faces while keeping the solid closed. In the *dmm_PurifyFilletsChains* mode all fillets that are smoothly connected to the selected faces are additionally removed.

Fig. 17 and Fig. 18 present solids obtained by removing the specified fillet faces. The initial solids are shown in Fig. 1 and Fig. 3, respectively:



_modificationType = *dmm_PurifyFillets*

Fig. 17



_modificationType = *dmm_PurifyFillets*

Fig. 18

The function ModifiedSolid() operates under the condition that the topology of the edited solid remains unchanged.

The function ModifiedSolid() adds to the history of the constructed solid the builder MbFaceModifiedSolid, which contains all data required to perform the operation. The builder MbFaceModifiedSolid is declared in the file *cr_modified_solid.h*.

The test application *test.exe* constructs a modified solid by using the menu commands "Create → Solid → By direct editing".

3.6.2 Constructing a Deformable Body

Construction of a deformable body is performed by a user in several sequential steps, each of which is implemented by a separate function. The goal is to obtain the body result in which the selected faces of the initial body solid are replaced with deformable faces.

1. At the first step, the faces are prepared for deformation: a parameter object of type NurbsValues is created and one of the functions ModifiedNurbsItem is called, either the variant with an array of faces **faces** or the variant with a single face **face**. As a result, the body **result** is obtained, in which the specified faces of the initial body **solid** are replaced with NURBS-surfaces suitable for further modification.
2. If necessary, for a selected face **face** of the body **result** the function GetControlSurface is called and a copy of the base NURBS-surface of the face **face** is obtained, which can be modified independently of the body.
3. If the shape of a face is to be controlled through its control points, the function FaceControlPoints is called for the face **face** to obtain the matrix of control points and their weights. The user changes the coordinates of the control points and/or their weights according to the required deformation.
4. After the modifications have been made, the deformation is applied to the body using one of the functions NurbsModification. In the variant with a surface, the modified surface **faceSurface** and the fixed points are passed. In the variant with control points, the modified control points, the weights and the fixed points are passed. In both cases, the output is the modified body **result** with a changed shape of the selected face.

The function

```
MbResultType ModifiedNurbsItem(      MbSolid &          solid,
                                     MbeCopyMode        sameShell,
                               const NurbsValues &      p,
                               const RPArray<MbFace> &  faces,
                               const MbSNameMaker &     names,
                                     MbSolid *&         result )
```

replaces the selected faces of the **solid** with deformable faces for further editing.

The input parameters of the function are:

- **solid** – the initial solid;
- **sameShell** – the mode of copying of the initial solid;
- **p** – an object containing transformation parameters;
- **faces** – the faces of the solid to be replaced;
- **names** – a name maker of the operation.

The output parameter of the function is the constructed solid **result**.

If successful, the function returns rt_Success; otherwise it returns an error code from the enumeration *MbResultType* (see *mb_operation_result.h*).

The function is declared in the file *action_direct.h*.

When the object **p** with modification parameters is created, the following constructor of parameters is used (see *op_shell_parameter.h*):

- **ud, vd** – the order of the NURBS copy in the *u*- and *v*-directions;
- **uc, vc** – the number of control points in the *u*- and *v*-directions;
- **umin, umax, vmin, vmax** – parameter ranges in *u* and *v* for deforming the face;
- **uapprox, vapprox** – flags that allow an approximating surface to be constructed instead of an exact one.

```
NurbsValues( size_t ud, size_t uc, double umin, double umax, bool uapprox,
             size_t vd, size_t vc, double vmin, double vmax, bool vapprox )
: uParameters( ud, uc, umin, umax, uapprox )
, vParameters( vd, vc, vmin, vmax, vapprox )
{}
```

The object **p** stores two sets of parameters of type MbNurbsParameters: **uParameters** and **vParameters**. The parameters **ud**, **uc**, **umin**, **umax**, **uapprox** define the fields of **uParameters**, and the parameters **vd**, **vc**, **vmin**, **vmax**, **vapprox** define the fields of **vParameters**.

The considered function copies the solid **solid** (depending on the parameter **sameShell**) and prepares the copy of the solid for further deformation of the specified faces of the initial solid.

The parameter **solid** contains the initial solid in which the specified faces will be replaced with deformable faces based on NURBS-surfaces.

The parameter **sameShell** controls the transfer of faces, edges and vertices from the initial solid **solid** to the constructed solid **result** and can take one of four values:

- *cm_Same* = 0 (the solid is not copied);
- *cm_KeepHistory* (the initial shell and its copy share base surfaces, vertices and faces that are not changed by the operation);

- *cm_KeepSurface* (the initial solid and its copy share base surfaces);
- *cm_Copy* (the initial solid and its copy do not share any data).

The enumeration MbeCopyMode is declared in the file *mb_enum.h* and is described in detail in *Copying a Set of Faces MbFaceShell*.

The parameter **faces** contains the set of faces of the solid **solid** that will be replaced with deformable faces based on NURBS-surfaces in the constructed solid **result**.

The parameter **names** is used for naming new faces, edges and vertices and for versioning of the operation.

The function

```
MbResultType ModifiedNurbsItem(      MbSolid &       solid,
                                     MbeCopyMode     sameShell,
                               const NurbsValues &   params,
                               const MbFace &        face,
                               const MbSNameMaker &  names,
                                     MbSolid *&      result )
```

replaces a single face of the solid with a deformable face for further editing. This function differs from the one described above only in the fourth parameter, which contains not a set of faces but a single face **face** to be replaced.

The function is declared in the file *action_direct.h*.

The constructed body **result** may look similar to the original body **solid**, but unlike the original body it has NURBS surfaces available for modification. Fig. 1 shows an initial cylindrical solid **solid** whose side face is a cylinder MbCylinderSurface and the solid **result** with a side face based on a NURBS-surface. The control points of the NURBS-surface are also shown in Figure.

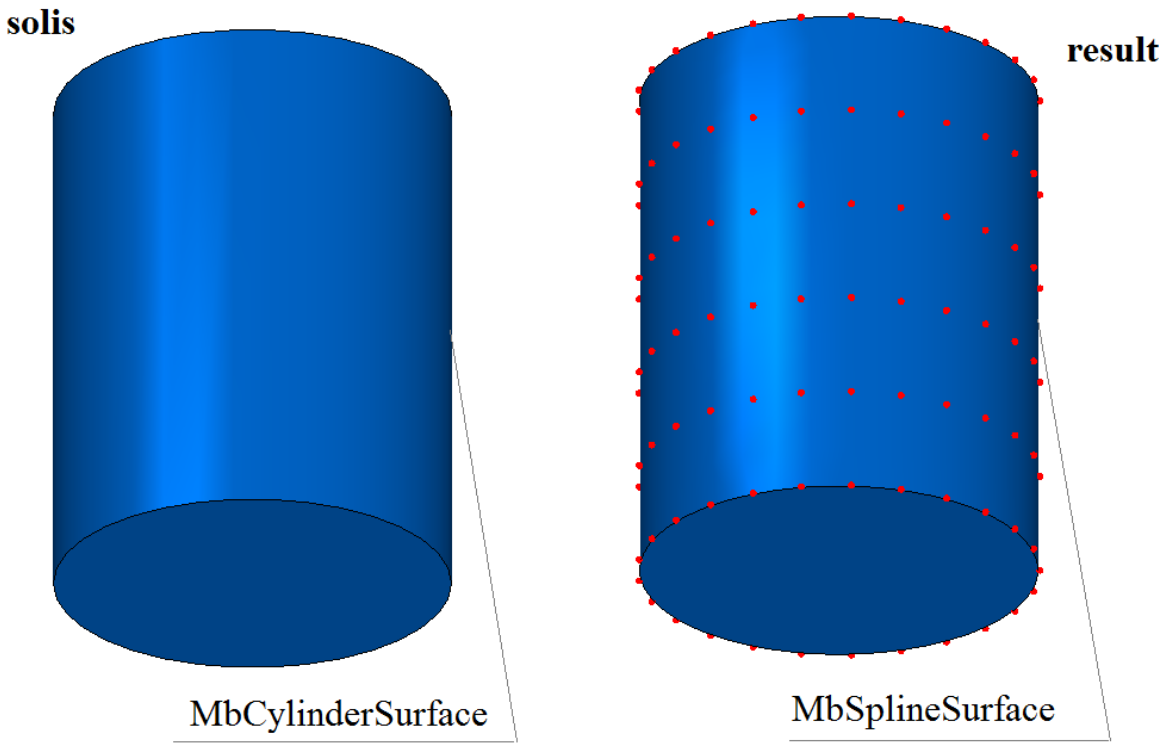


Fig. 1

The functions ModifiedNurbsItem() add to the history of the constructed solid the builder MbModifiedNurbsItem, which contains all data required to perform the operation. The builder MbModifiedNurbsItem is declared in the file *cr_modified_nurbs_.h*.

The function

```
MbSurface * GetControlSurface( const MbFace & face )
```

creates a deformable NURBS-surface of the face **face** for further editing.

The input parameter of the function is the initial face **face**.

The function is declared in the file *action_direct.h*.

The function returns a non-null pointer to a surface if the face **face** is based on a NURBS-surface. This requirement is satisfied after successful execution of the functions ModifiedNurbsItem(). The function returns a pointer to a copy of the base surface of the face **face**, not to the base surface of the face itself. The control points and their weights of the returned surface can then be modified.

The function

```
MbResultType FaceControlPoints( const MbFace &                face,
                                Array2<MbCartPoint3D> & controlPoints,
                                Array2<double> &        result )
```

returns the set of control points of the surface of a face and the set of their weights.

The input parameter of the function is the initial face **face**.

The output parameters of the function are:

- **controlPoints** – the control points of the face;
- **result** – the weights of the control points.

The function is declared in the file *action_direct.h*.

The function returns rt_Success if the face **face** is based on a NURBS-surface; otherwise it returns an error code from the enumeration *MbResultType* (see *mb_operation_result.h*). By changing the positions of the control points **controlPoints** and changing the weights in **result**, one can change the shape of the surface. The considered function should be used after successful execution of the functions ModifiedNurbsItem() described above.

The control points **controlPoints** are passed as a rectangular matrix whose rows correspond to the first parameter of the NURBS-surface and whose columns correspond to the second parameter of the NURBS-surface. That is, the points lying in one row of the rectangular matrix **controlPoints** are arranged along a coordinate line of the surface with a fixed second parameter of the NURBS-surface, and the points lying in one column of the rectangular matrix **controlPoints** are arranged along a coordinate line of the surface with a fixed first parameter of the NURBS-surface. In Fig. 2 the control points are shown against the background of the coordinate lines of the surface.

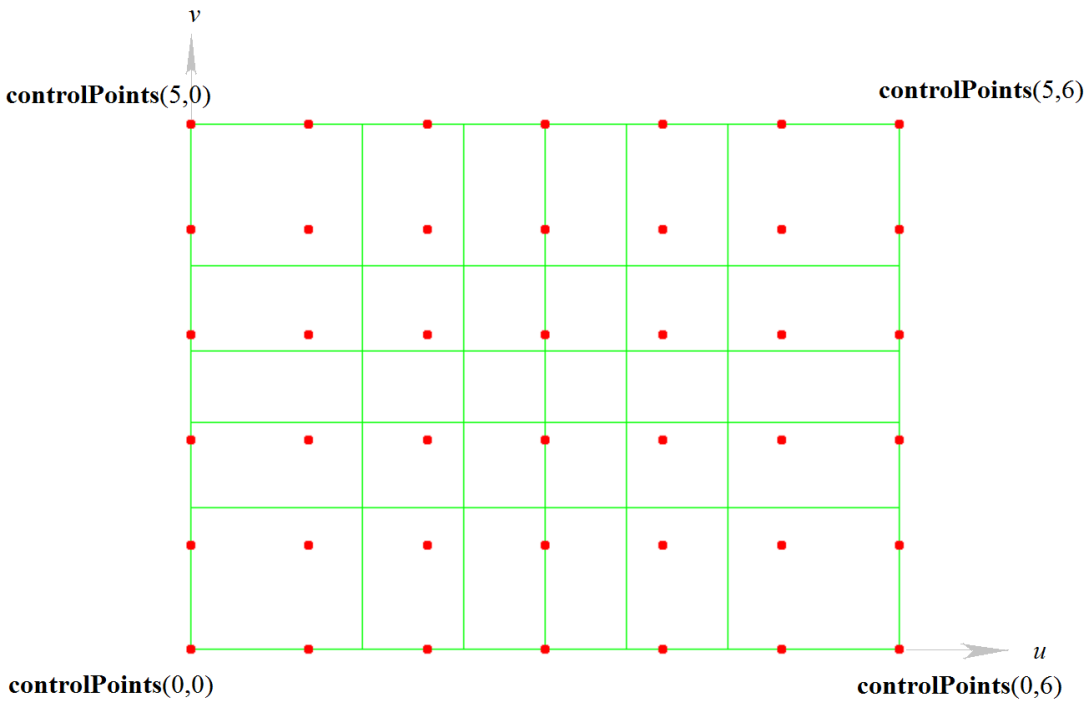


Fig. 2

The rectangular matrix **result** of weights corresponds to the rectangular matrix of control points **controlPoints**.

Fig. 3 shows the face **face** and the control points **controlPoints** of its surface leftward and the edited control points **controlPoints** for subsequent modification of the face **face** on the right.

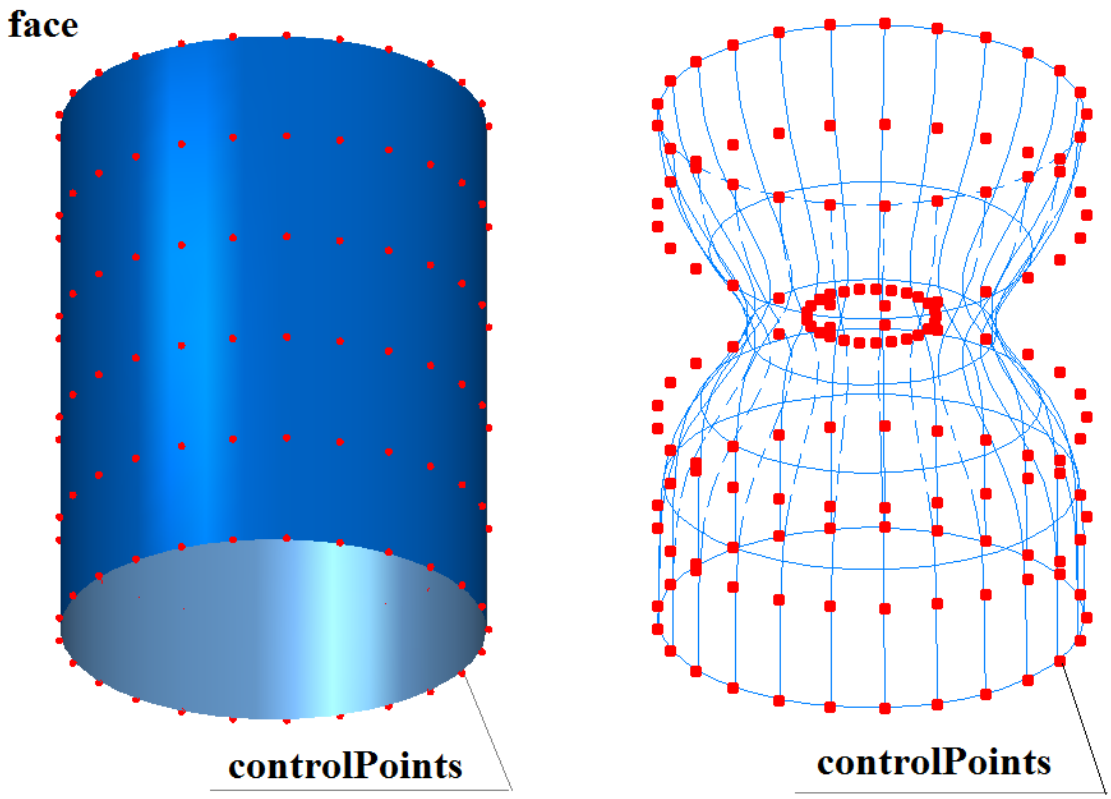


Fig. 3

The function

```
MbResultType NurbsModification(       MbSolid &        solid,
                                      MbeCopyMode      sameShell,
                                      MbFace *         face,
                                      MbSurface &      faceSurface,
                                      Array2<bool> &   fixedPoints,
                                const MbSNameMaker &   names,
                                      MbSolid *&       result )
```

deforms the specified face of the initial solid by substituting the given surface into the modified face.

The input parameters of the function are:

- **solid** – the initial solid;
- **sameShell** – the mode of copying of the initial solid;
- **face** – the face of the solid to be modified;
- **faceSurface** – the new NURBS-surface of the face;
- **fixedPoints** – the fixed points (if the **fixedPoints** pointer is equal to *nullptr*, all control points are considered movable);
- **names** – a name maker of the operation.

The output parameter of the function is the constructed modified solid **result**.

If successful, the function returns rt_Success; otherwise it returns an error code from the enumeration *MbResultType* (see *mb_operation_result.h*).

The function is declared in the file *action_direct.h*.

The considered function copies the solid **solid** and substitutes the given surface **faceSurface** into the copy of the face **face** of the solid **solid**.

The enumeration MbeCopyMode is described in detail in *Copying a Set of Faces MbFaceShell*.

The surface **faceSurface** must be a NURBS-surface. The parameter **fixedPoints** is a rectangular matrix whose rows and columns correspond to the rows and columns of the matrix of control points of the NURBS-surface **faceSurface**. The elements of the matrix with the value *true* define fixed control points of the NURBS-surface.

The function

```
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 initial solid using the given control points and their weights.

The input parameters of the function are:

- **solid** – the initial solid;
- **sameShell** – the mode of copying of the initial solid;
- **face** – the face of the solid to be modified;
- **controlPoints** – the given control points of the face to be modified;
- **weights** – the weights of the control points;
- **fixedPoints** – the fixed points (if the **fixedPoints** pointer is equal to *nullptr*, all control points are considered movable);
- **names** – a name maker of the operation.

The output parameter of the function is the constructed solid **result**.

If successful, the function returns rt_Success; otherwise it returns an error code from the enumeration *MbResultType* (see *mb_operation_result.h*).

The function is declared in the file *action_direct.h*.

The considered function should be used after successful execution of the functions ModifiedNurbsItem(), GetControlSurface() and FaceControlPoints() described above. The functions ModifiedNurbsItem() prepare the faces for modification.

Fig. 4 shows an initial cylindrical solid **solid** whose side face is based on a NURBS-surface and whose control points are to be modified. Also Fig. 4 shows the solid **result** with a modified side face; the control points **controlPoints** of this face are shown in the figure.

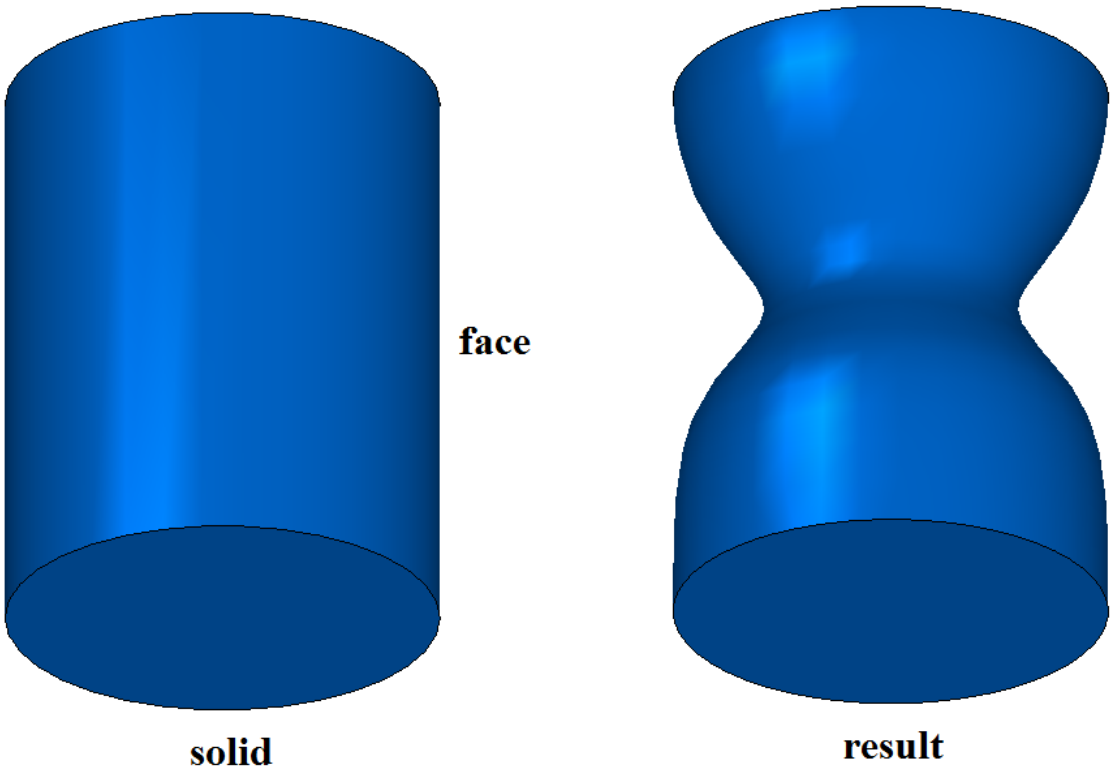


Fig. 4

The functions NurbsModification() add to the history of the constructed solid the builder MbNurbsModification, which contains all data required to perform the operation. The builder MbNurbsModification is declared in the file *cr_modified_nurbs_.h*.

3.6.3 Constructing a Deformable Block

The function

```
MbResultType NurbsBlockSolid( const MbPlacement3D &     place,
                                    double              ax,
                                    double              ay,
                                    double              az,
                                    bool                outDir,
                              const MbSNameMaker &      names,
                                    SimpleName          name,
                                    NurbsBlockValues &  param,
                                    MbSolid *&          result )
```

constructs a solid in the form of a rectangular parallelepiped with deformable faces.

The input parameters of the function are:

- **place** – the local coordinate system;
- **ax** – the size along the X axis;

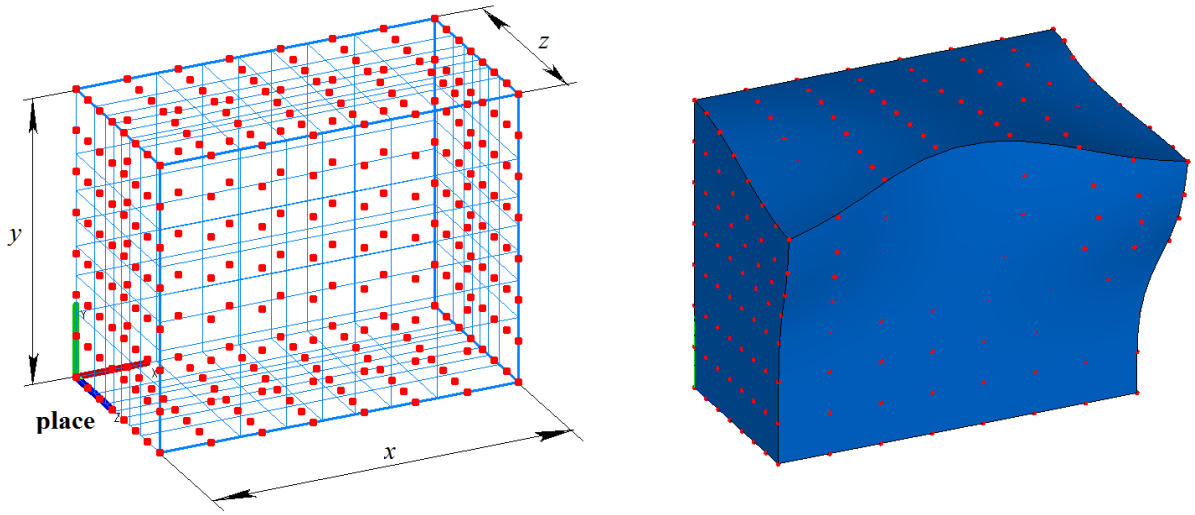
- **ay** – the size along the Y axis;
- **az** – the size along the Z axis;
- **outDir** – the orientation of face normals outward from the solid (*true* – normals point outward);
- **names** – the operation name maker;
- **name** – the main name;
- **param** – the face construction parameters (see NurbsBlockValues struct in the file *op_shell_parameter.h*).

The output parameter of the function is the solid **result** built on NURBS surfaces.

The function is declared in the file *action_direct.h*.

The function constructs a deformable solid in the form of a rectangular parallelepiped whose dimensions in the local coordinate system **place** are defined by the parameters **ax**, **ay**, **az**. All faces of the parallelepiped are built on the basis of planar NURBS surfaces whose orders and numbers of control points are determined by the parameter **param**.

Figure shows the parallelepiped **result** constructed by the NurbsBlockSolid() function and its control points. The figure on the right shows the same parallelepiped after deformation by moving the face control points:



The NurbsBlockSolid() function adds the MbNurbsBlockSolid builder to the construction history of the resulting solid. The builder stores all data required to perform the operation. The MbNurbsBlockSolid builder is declared in the file *cr_nurbs_block_solid.h*.

3.6.4 Constructing a Smoothed Surface

The function

```
MbResultType SplineSurfaceSmoothing( const MbSplineSurface &   surface,
                                     size_t                    udegree,
                                     size_t                    vdegree,
                                     MbSplineSurface *&        result );
```

smooths a copy of the source surface.

The input parameters of the function are:

- **surface** - the source spline surface;
- **udegree** - the new order of the spline surface in the first surface parameter;
- **vdegree** - the new order of the spline surface in the second surface parameter.

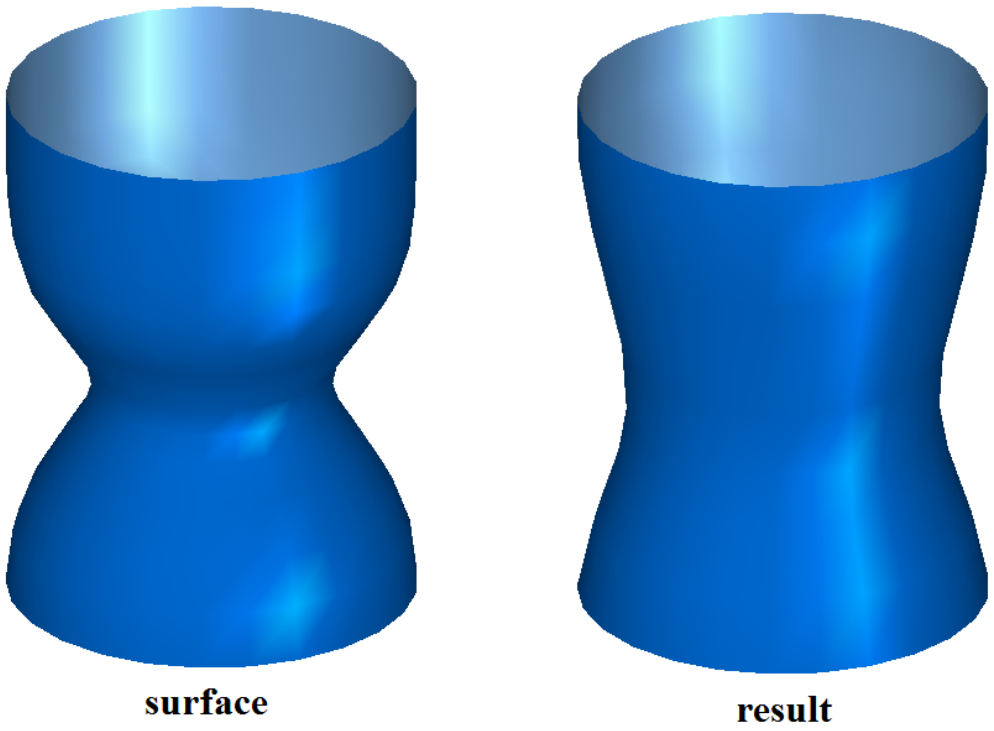
The output parameter of the function is the smoothed copy of the source surface **result**.

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the file *action_direct.h*.

The function smoothes the source surface **surface** without changing its spline orders or the number of control points. The smoothing is controlled by the **udegree** and **vdegree** parameters, which must be greater than the corresponding spline orders of the source surface.

Figure shows the source NURBS surface **surface** and the smoothed surface **result**:



3.6.5 Constructing a Transformed Body

The function

```
MbResultType TransformedSolid(      MbSolid &           solid,
                                    MbeCopyMode         sameShell,
                              const TransformValues &   p,
                              const MbSNameMaker &      names,
                                    MbSolid *&          result )
```

performs a transformation of a source body by a given matrix calculated from the bounding cube.

The input parameters of the function are:

- **solid** is the source body.
- **sameShell** is a mode of copying the source body.
- **p** is the transformation parameters.
- **names** is a name maker of the operation.

The output parameter of the function is the modified body **result**.

If succeessful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the *action_direct.h* file.

When creating the **p** object with the transformation parameters, one of the following constructors is used (see *op_shell_parameter.h*):

Default constructor:

```
TransformValues()
: matrix()
, fixedPoint()
, useFixed( false )
, isotropy( false )
{}
```

Constructor by a matrix:

```
TransformValues( const MbMatrix3D & m )
: matrix( m )
, fixedPoint()
, useFixed( false )
, isotropy( false )
{}
```

Constructor by a matrix and a fixed transformation point:

```
TransformValues( const MbMatrix3D & m, const MbCartPoint3D & f, bool fix = false,␣
↪bool iso = false )
: matrix( m )
, fixedPoint( f )
, useFixed( fix )
, isotropy( iso )
{}
```

Constructor by a fixed transformation point and scale factors along axes:

```
TransformValues( double sX, double sY, double sZ, const MbCartPoint3D & fP );
```

Copy constructor:

```
TransformValues( const TransformValues & other )
: matrix    ( other.matrix )
```

(continues on next page)

(continued from previous page)

```
, fixedPoint( other.fixedPoint )
, useFixed  ( other.useFixed )
, isotropy  ( other.isotropy )
{}
```

The function scales the body by a matrix with equal or different scale factors along the axes of the global coordinate system.

The **solid** parameter contains the source body to be processed.

The **sameShell** parameter controls data copying from the source body **solid** to the resulting body **result** (see *Copying_a_Set_of_Faces*).

The **p** parameter contains a transformation matrix **p**.*matrix* and values **p**.*fixedPoint*, **p**.*useFixed*, **p**.*isotropy*, which are used for calculating the transformation matrix. The TransformValues transformation parameter is declared in the *op_shell_parameter.h* file.

The **names** parameter is used for naming body elements.

The transformation matrix is calculated from the bounding cube of the source body **solid** and from a displacement of one of the points of the bounding cube by the method:

```
bool MbCube::CalculateMatrix(      size_t           pIndex,
                             const MbCartPoint3D &  point,
                             const MbCartPoint3D &  fixedPoint,
                                   bool             useFixed,
                                   bool             isotropy,
                                   MbMatrix3D &     matrix ) const;
```

where

- **pIndex** is the index of a point of the bounding cube to be displaced;
- **point** is a space point to which the point of the bounding cube with index **pIndex** shall be connected;
- **fixedPoint** is a fixed transformation point (used if **useFixed** = *true*);
- **useFixed** is a flag for using a fixed point;
- **isotropy** is a flag for equal transformation scale factors;
- **matrix** is the calculated transformation matrix.

Figure 1 shows the numbering of the points of the bounding cube of the **solid** body. The vertices of the bounding cube have indices 0–7, the midpoints of the edges of the bounding cube have indices 8–19, and the centers of the faces of the bounding cube have indices 20–25.

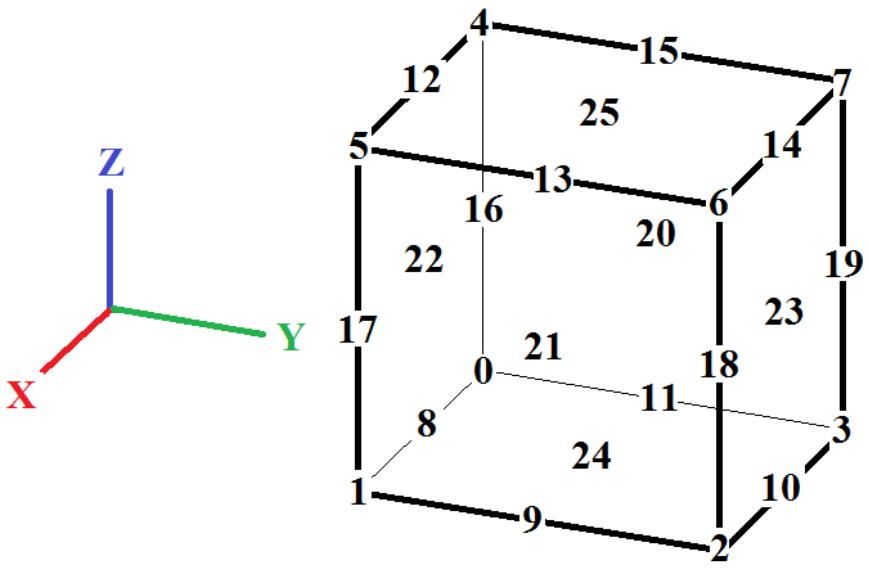


Fig. 1

The transformation matrix **matrix** is calculated from a deformed cube obtained by moving the point of the bounding cube with index **pIndex** to the space point **point**.

If **useFixed** = *true*, the fixed transformation point is the **fixedPoint** point. If **useFixed** = *false*, the fixed transformation point is the point of the bounding cube opposite to the point with index **pIndex**. The following pairs of points of the bounding cube are considered 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*, the transformation scale factors along all axes are equal and proportional to the displacement vector of the point of the bounding cube with index **pIndex**. If **isotropy** = *false*, the transformation scale factors are proportional to the projections of the displacement vector of the point of the bounding cube with index **pIndex** onto the edges of the bounding cube.

The values **p**.*fixedPoint*, **p**.*useFixed*, **p**.*isotropy* and **p**.*matrix* are used as the **fixedPoint**, **useFixed**, **isotropy** and **matrix** parameters of the CalculateMatrix() method.

Figure 2 shows the source body **solid**, the points of the bounding cube, and the space point **point** to which the specified point of the bounding cube shall be moved, and Figure 3 shows the transformed body **result** constructed by the function under consideration with the parameters **useFixed** = *false* and **isotropy** = *false*.

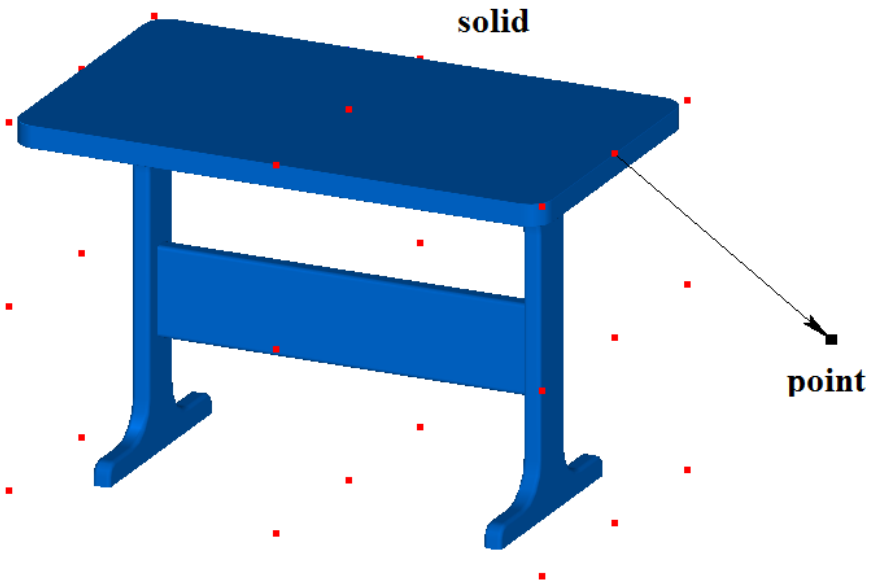


Fig. 2

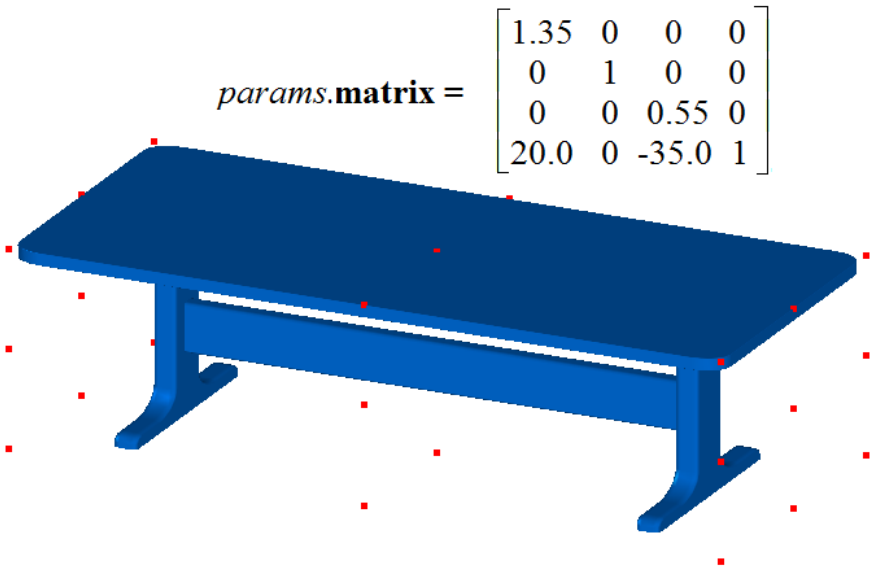


Fig. 3

A body can also be transformed by a given matrix by the Transform() method, which is overloaded in all geometric objects including MbSolid, but in this case the body will not contain any information about the performed operations.

The TransformedSolid() function adds the MbTransformedSolid builder to the history of the resulting body. The MbTransformedSolid builder is declared in the *cr_transformed_solid.h* file.

The test application *test.exe* constructs a transformed body by the menu command “Create → Body → By direct editing → Transformation of bounding box”.

3.7 SHEET METAL BODY CONSTRUCTION METHODS

The C3D Geometric Kernel supports functions that permit to construct models of sheet metal structures. There are the following sheet body construction functions: 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.

3.7.1 Constructing a Sheet Body

The function

```
MbResultType CreateSheetSolid( const MbPlacement3D &            placement,
                                     RPArray<MbContour> &       contours,
                                     bool                       unbended,
                               const MbSheetMetalValues &       params,
                                     RPArray<MbSNameMaker> *    nameMakers,
                                     RPArray<MbSMBendNames> &   resultBends,
                                     MbSolid *&                 result );
```

constructs a sheet metal solid by extrusion of planar contours.

The input parameters of the function are:

- **placement** – the local coordinate system;
- **contours** – the contours of the sheet metal solid defined in the XY plane of the local coordinate system;
- **unbended** – a flag for constructing the solid in the unbent state;
- **params** – the construction parameters;
- **nameMakers** – the construction name makers.

The output parameters of the function are the names of the sheet faces of the constructed bends, written into **resultBends**, and the solid **result** itself.

If successful, the function returns rt_Success; otherwise, it returns an error code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is declared in the file *action_sheet.h*.

If the contours are closed, the sheet metal solid is constructed with the specified thickness along the given contour. Among the closed contours, one contour must be the outer contour containing all the remaining inner contours. Cutouts are formed along the inner contours.

If the contours are open, the sheet metal solid is constructed by extrusion of closed contours over the specified distances. These closed contours are derived from the open contours by rounding the corners between line segments using the radii specified in **params** or **resultBends**. The resulting contours are given a sheet metal thickness in one direction or the other. The thickening direction is specified in the operation parameters **params**. If a contour already contains an arc, the arc must join the neighboring segments smoothly.

Sheet metal solids with bends can only be constructed from open contours, and the **unbended** flag is meaningful only in this case. If the **resultBends** array is empty, the bends are constructed using the uniform parameters specified in **params**, and the names of the sheet faces of the

constructed bends are not recorded. If the **resultBends** array is populated, the number of its elements must match the number of bends being formed.

Bend parameters are looked up as follows. Each contour in the **contours** array corresponds to a name maker in the **nameMakers** array, and each name maker contains an array of contour segment names. If a bend is formed at the junction of two line segments, the parameters of that bend are located in the element of the **resultBends** array whose **segName1** and **segName2** fields contain the names of those segments. If a bend is formed at the position of an arc, the parameters for it are found in the element of the **resultBends** array in which **segName1** holds the name of the arc and **segName2** equals *SIMPLENAME_MAX*.

Fig. 1 shows a sheet metal solid constructed from several closed sketches:

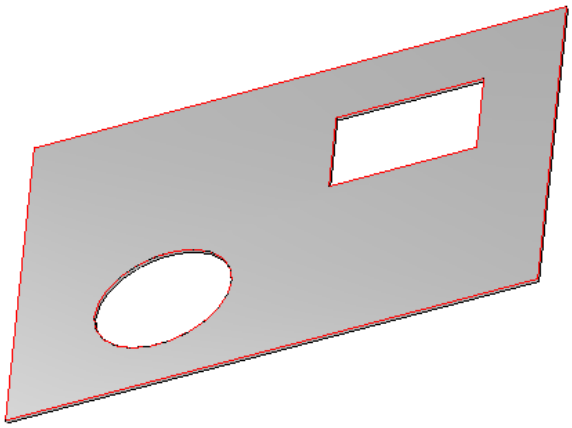


Fig. 1

Fig. 2 shows a sheet metal solid constructed from an open sketch containing an arc and a pair of line segments joining at an angle:

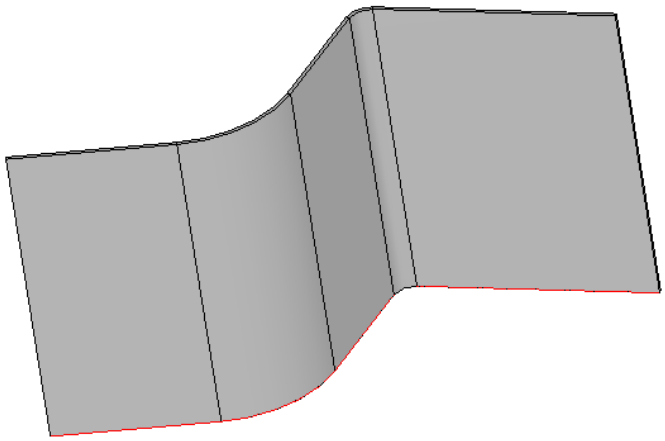


Fig. 2

The CreateSheetSolid() method adds the MbSheetMetalSolid builder in the newly constructed body log containing all data required to execute the operation. The MbSheetMetalSolid builder is declared in the *cr_sheet_metal_solid.h* file.

3.7.2 Pairing Sheet Body Bends

The function

```
bool CollectBends( const MbFaceShell &                faceShell,
                   const RPArray<MbFace> &            innerFaces,
                   const RPArray<MbFace> &            outerFaces,
                         RPArray<MbSheetMetalBend> & result );
```

groups sheet body bend faces into pairs.

The input parameters of the function are:

- **faceShell** - the sheet body shell containing the bend faces
- **innerFaces** - the inner bend faces
- **outerFaces** - the outer bend faces

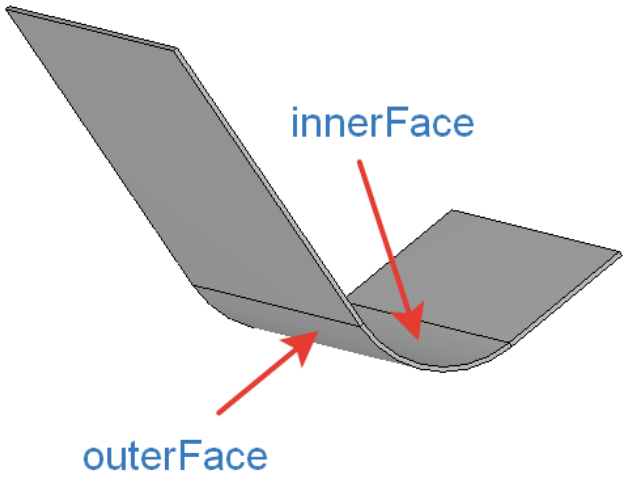
The output parameter of the function is the array **result** of detected face pairs forming the bends.

If successful, the function returns *true*; otherwise, it returns *false*.

The function is declared in the *action_sheet.h* file.

The function searches, for each bend, a pair of faces (an inner and an outer one) in the unordered sets **innerFaces** and **outerFaces** and, for the pairs found, creates bend objects that are added to the **result** array.

Figure shows the inner and outer faces of a bend:



3.7.3 Checking Whether a Face Can Be Fixed

The function

```
bool IsSuitableForFixed( const MbFace & face );
```

checks whether a face can be used as a fixed face for bending or unbending a sheet body bend.

The function returns *true* if the face can be selected as a fixed face for bending and unbending a sheet body; otherwise, it returns *false*.

The function is declared in the *action_sheet.h* file.

3.7.4 Searching for Faces for a Curve

The function

```
void FindCurveFaces( const RPArray<MbFace> &  faces,
                     const MbCurve3D &        curve,
                           RPArray<MbFace> &  result );
```

searches faces on which the specified straight-line curve lies.

The input parameters of the function are:

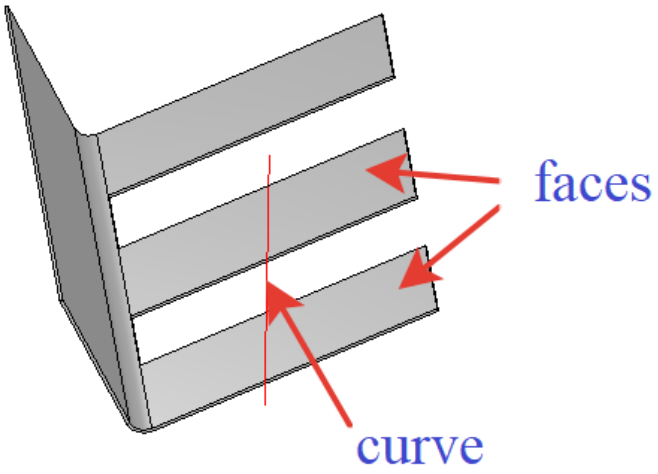
- **faces** - the set of faces for the search
- **curve** - the straight-line curve that may lie on some faces of the **faces** set

The output parameter of the function is the set of faces **result** on which the **curve** lies.

The function does not return a value.

The function is declared in the *action_sheet.h* file.

Figure shows two sheet faces lying under a line segment:



3.7.5 Searching for a Sheet Body Face

The function

```
MbFace * FindSheetFace( const MbCurveEdge & edge );
```

searches for the top or bottom sheet face of a body that contains the given edge.

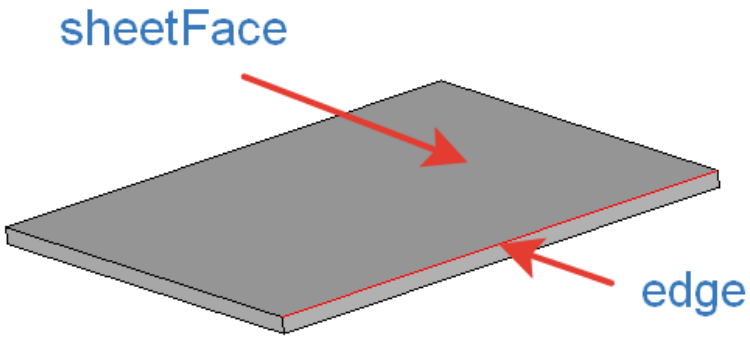
The input parameter of the function is the unoriented edge **edge** of a sheet face.

If successful, the function returns a pointer to the detected sheet face; otherwise, it returns *NULL*.

The function is declared in the *action_sheet.h* file.

The search for the top or bottom sheet face of the body is performed among the two faces adjacent along the edge **edge**.

Figure shows the sheet face adjacent to an edge:



3.7.6 Searching for a Mate Face of a Sheet Body Bend

The function

```
MbFace * FindPairBendFace( const MbFace & face );
```

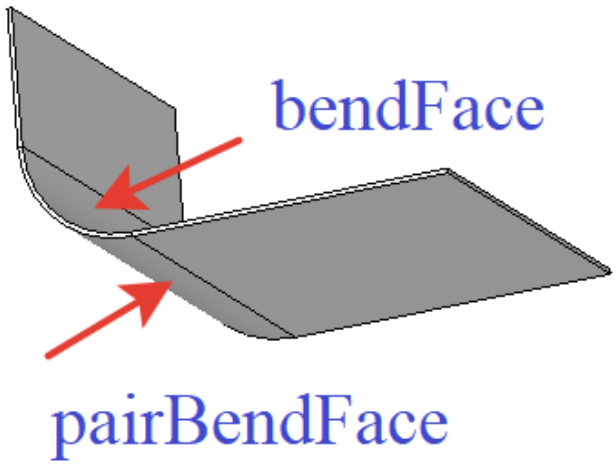
searches for the bend face of a sheet body that is paired with the given one.

The input parameter of the function is the sheet body bend face **face**.

If successful, the function returns a pointer to the detected face; otherwise, it returns *NULL*.

The function is declared in the *action_sheet.h* file.

Figure shows a bend face **bendFace** and its mate face **pairBendFace**:



3.7.7 Searching for a Flat Sheet Body Face

The function

```
MbFace * GetPairPlanarFaceByEdge( const MbCurveEdge & edge,
                                  const double        begDistance,
                                  const double        endDistance );
```

searches for a mate flat face of a sheet body by the given edge and distances from its endpoints.

The input parameters of the function are:

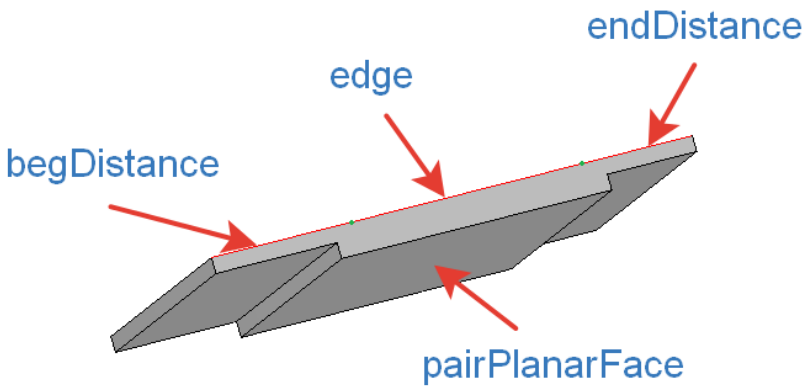
- **edge** – the edge used for the search.
- **begDistance** – the distance from the start of the edge.
- **endDistance** – the distance from the end of the edge.

If successful, the function returns a pointer to the detected flat sheet face; otherwise, it returns *NULL*.

The function is declared in the *action_sheet.h* file.

The function searches for a mate face of a sheet body for the sheet face that contains the edge **edge**. It is used in the bend-on-edge operation to determine the sheet thickness at the bend adhesive area. The method is applied for multi-thickness sheet bodies in situations when the selected sheet face has several mate faces located at different distances from it. Positive values of **begDistance** and **endDistance** specify an offset outward along the edge, and negative values of **begDistance** and **endDistance** specify an offset inward along the edge.

Figure shows the flat sheet face **pairPlanarFace** found by the edge and the offsets from its endpoints:



3.7.8 Searching for a Mate Face in a Sheet Body

The function

```
MbFace * GetPairPlanarFaceByCurve( const MbFace    & sheetFace,
                                   const MbCurve3D & curve );
```

searches for a flat mate face of a sheet body by a straight-line curve **curve** lying on the face **sheetFace**.

The input parameters of the function are:

- **sheetFace** – a flat face of the sheet body.
- **curve** – a straight-line curve lying on the sheet face.

If the function succeeds, it returns a pointer to the detected face; otherwise, it returns *NULL*.

The function is declared in the *action_sheet.h* file.

The mate face search function for a sheet body is an auxiliary function for the bend-by-line construction function. It is used for multi-thickness sheet bodies when the selected sheet face has several mate faces located at different distances from it. Fig. 1 shows a flat sheet face found by the opposite face and the curve lying on it.

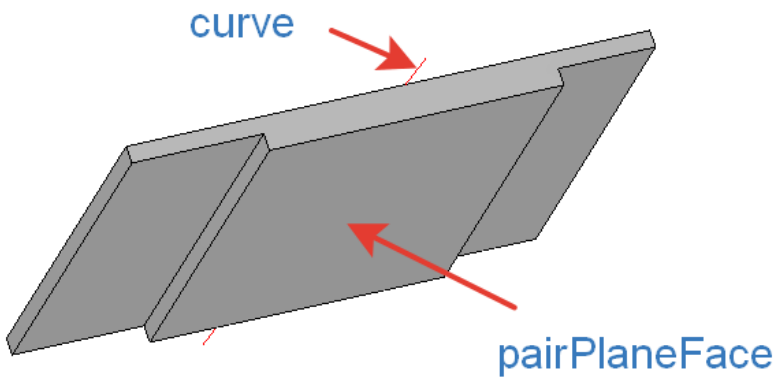


Рис. 1

The function

```
MbFace * GetPairPlanarFaceByContour( const MbFaceShell &             shell,
                                     const MbFace &                  sheetFace,
                                     const MbPlacement3D &           place,
                                     const RPArray<const MbCurve> & segments );
```

searches for a flat mate face of a sheet body by a set of contour segments.

The input parameters of the function are:

- **shell** – the sheet body shell.
- **sheetFace** – a face of the sheet body.
- **place** – the local coordinate system lying on the face **sheetFace**.
- **segments** – curves lying in the XY plane of the local coordinate system **place**.

If successful, the function returns a pointer to the detected face; otherwise, it returns *NULL*.

The function is declared in the *action_sheet.h* file.

The mate face search function for a sheet body is an auxiliary function for sheet body construction functions based on contours. It is used for multi-thickness sheet bodies when the selected sheet face has several mate faces located at different distances from it. Fig. 2 shows a flat sheet face found by the opposite face and the contour lying on it.

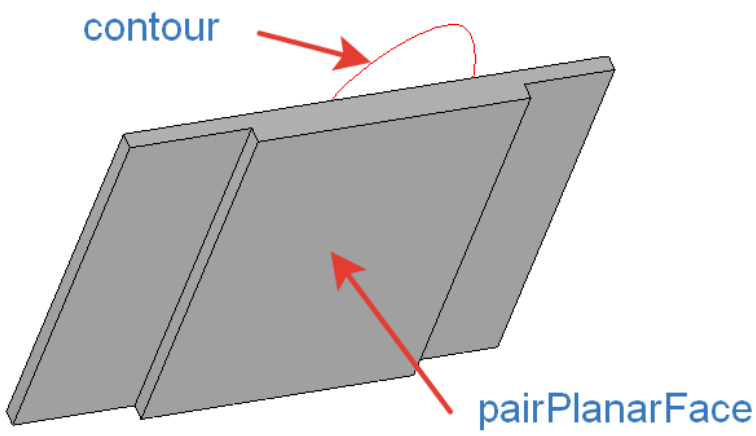


Рис. 2

The function

```
MbFace * GetPairPlanarFace( const MbFaceShell * faceShell,
                            const MbFace &       sheetFace );
```

searches for a mate flat face by a given sheet body face.

The input parameters of the function are:

- **faceShell** – the sheet body shell.
- **sheetFace** – a sheet body face.

If successful, the function returns a pointer to the detected face; otherwise, it returns *NULL*.

The function is declared in the *action_sheet.h* file.

The search is first performed through the edges of the outer loop of the face **sheetFace**; if this fails, the vertices of this loop are used, and if the face is still not found, all adjacent faces or all faces of the **faceShell** set are checked. In the latter case, preference is given to faces that are located closer. Fig. 3 shows a flat sheet face **pairsheetFace** found by the opposite flat sheet face **sheetFace**.

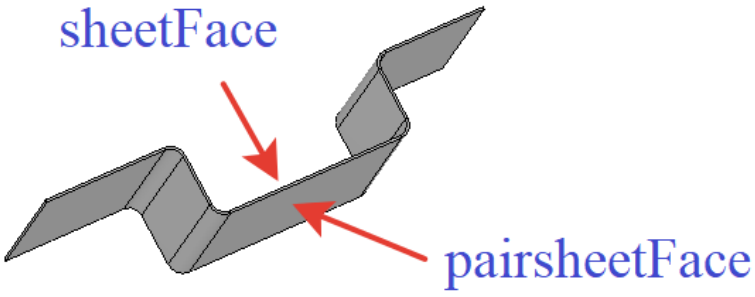


Рис. 3

3.7.9 Determining the Distance Between Similar Faces

The function

```
double GetDistanceIfSameAndOpposite( const MbFace & face1,
                                     const MbFace * face2 );
```

determines the distance between a pair of similar faces of a sheet body.

The input parameters of the function are:

- **face1** – the first face.
- **face2** – the second face.

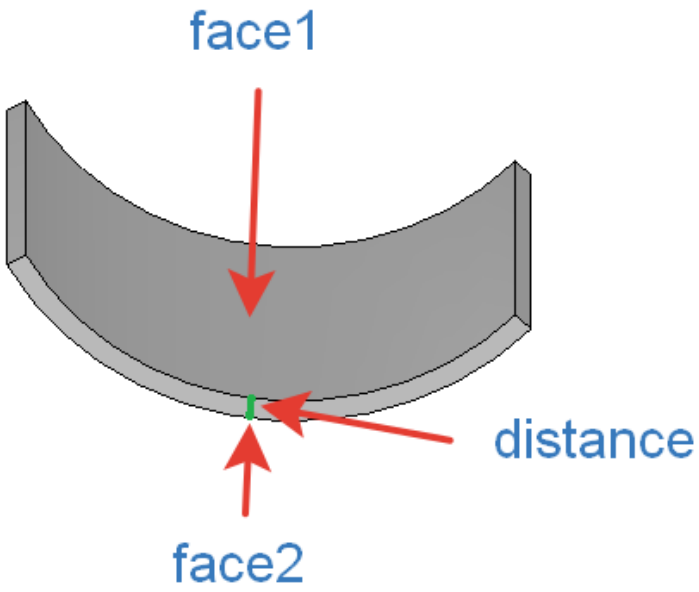
If the function succeeds, it returns the distance between the faces; otherwise, it returns zero.

The function is declared in the *action_sheet.h* file.

Pairs of planar, cylindrical, and conical faces are considered similar, and in the case of a ruled bend, ruled faces paired with offset faces are also considered similar, provided that their normals are collinear and oppositely directed.

The distance is positive if the face **face2** lies on the side opposite to the normal direction of the face **face1**, and negative otherwise.

Figure shows a pair of similar faces and the distance between them:



3.7.10 Searching for Similar Bends

The function

```
void GetSimilarCylindricBends( const MbFaceShell &                shell,
                               RPArray<MbSheetMetalBend> & bends );
```

searches for similar bends.

The input parameters of the function are the sheet body shell **shell** and the array of initial bends **bends** for which similar bends must be found.

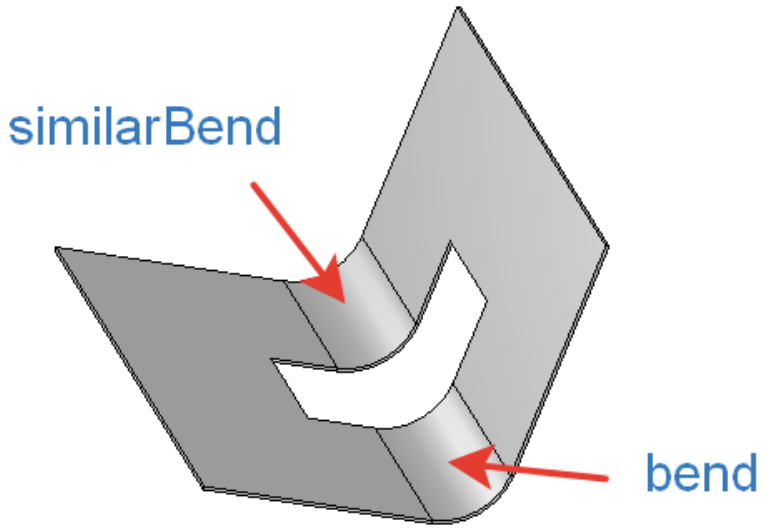
The output parameter of the function is the **bends** array extended with the detected similar bends.

The function does not return a value.

The function is declared in the *action_sheet.h* file.

In the set of sheet body faces, the function searches for bent cylindrical, conical, or ruled bends that must be added to the bends from **bends** to form groups of bends that can be unbent only together.

Figure shows a pair of similar bends that can be bent or unbent only simultaneously:



3.7.11 Searching for a Tangency Point in a Sheet Body Bend

The function

```
bool CalculateTangentPoint( const MbFace &      face,
                            const MbPlane &     plane,
                                  MbCartPoint & tangentPoint );
```

computes a tangency point for bending or unbending a sheet body.

The input parameters of the function are:

- **face** – the sheet body face containing the tangency point.
- **plane** – the tangent plane.
- **tangentPoint** – the tangency point.

The output parameter of the function is the tangency point **tangentPoint**.

If successful, the function returns *true*; otherwise, it returns *false*.

The function is declared in the *action_sheet.h* file.

The point **tangentPoint** can lie inside the face ($0.0 \le x \le 1.0$ and $0.0 \le y \le 1.0$) or outside it. In the first case, the tangency point is converted into the coordinates of the surface lying under the face; in the second case, one of the tangency points of the surface lying under the face **face** and the plane **plane** is found.

3.7.12 Searching for a Bend Centerline

The function

```
bool CalculateConicAxisLine( const MbFace &        face,
                                   MbLineSegment & axisLineSegment );
```

computes the centerline of an unbent conic bend.

The input parameter of the function is the face **face** of the unbent conic bend of a sheet body.

The output parameter of the function is the centerline **axisLineSegment**.

If the function succeeds, it returns *true*; otherwise, it returns *false*.

The centerline **axisLineSegment** is defined in the parametric coordinates of the planar face **face**.

The function is declared in the *action_sheet.h* file.

3.8 AUXILIARY METHODS

For constructing solids, parameters are required which in some cases must be calculated, as well as auxiliary elements. In this section, the functions that perform auxiliary calculations and constructions are listed.

3.8.1 Calculating the Depth of an Extrusion Body or the Angle of a Revolution Body

The function

```
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 revolution angle for subsequent construction of a body by extruding or revolving a generatrix curve.

The input parameters of the function are:

- **sweptData** – data of the generatrix curve.
- **axis** – the axis of revolution (used for the revolution calculation when **rotation** == *true*).
- **direction** – the extrusion direction (used for the extrusion calculation when **rotation** == *false*).
- **rotation** – the flag specifying the calculation type: *true* for revolution, *false* for extrusion.
- **operationDirection** – the direction of the subsequent operation: forward (*true*) or backward (*false*).
- **point** – the point up to which the generatrix curve is to be revolved or extruded.

The output parameter of the function is **value**, which is the extrusion depth or the revolution angle.

If the calculation is successful, the function returns *true*; otherwise it returns *false*.

The function is declared in the *action_solid.h* file.

The function is an auxiliary one for the *ExtrusionSolid*, *RevolutionSolid*, *ExtrusionResult*, and *RevolutionResult* functions.

3.8.2 Determining the Curve Image for Extrusion or Rotation

The function

```
GetSweptImagePosition( const MbCurve3D &    generatrix,
                       const MbSurface &    surface,
                       const MbVector3D &   direction,
                       const MbAxis3D &     axis,
                       const bool           rotation,
                             MbCartPoint &  imagePosition,
                             MbResultType & resType );
```

determines the position of the image of a generatrix curve point on a surface for subsequent construction of a body by extruding or revolving the generatrix curve up to the given surface.

The input parameters of the function are:

- **generatrix** – the generatrix curve.
- **surface** – the surface up to which the curve is to be extruded or revolved.
- **direction** – the extrusion direction (used for the extrusion calculation).
- **axis** – the axis of revolution (used for the revolution calculation).
- **rotation** – the flag specifying the calculation type: *true* for revolution and *false* for extrusion.

The output parameters of the function are:

- **imagePosition** – the point on the surface where the image of the generatrix curve lies.
- **resType** – the operation result code: if the function succeeds, it returns rt_Success; otherwise it returns an error code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is an auxiliary one for the ExtrusionSolid(), RevolutionSolid(), ExtrusionResult(), and RevolutionResult() functions.

3.8.3 Determining Extrusion or Rotation Parameters

The function

```
void GetRangeToSurface( const MbSurface &    surface,
                        const MbCurve3D &    curve,
                        const MbVector3D &   direction,
                        const MbAxis3D &     axis,
                        const bool           rotation,
                        const bool           operationDirection,
                        const MbCartPoint &  imagePosition,
                              double         range,
                              MbRect &       rectOnSurface,
                              MbResultType & resType );
```

computes the extrusion depths in the forward and backward directions or the revolution angles in the forward and backward directions for subsequent construction of a body by extruding or revolving a generatrix curve up to the given surface, and also determines the bounding box of the curve image.

The input parameters of the function are:

- **surface** – the surface up to which the curve is to be extruded or revolved;
- **curve** – the generatrix curve;
- **direction** – the extrusion direction (used for the extrusion calculation);
- **axis** – the axis of revolution (used for the revolution calculation);
- **rotation** – the flag specifying the calculation type: *true* for revolution and *false* for extrusion;
- **operationDirection** – the direction of the subsequent operation: forward (*true*) or backward (*false*);
- **imagePosition** – the point on the surface where the image of the generatrix curve lies.

The output parameters of the function are:

- **range** – the distances to the surface: range[0] is in the backward direction and range[1] is in the forward direction;
- **rectOnSurface** – the bounding rectangle of the curve image on the surface;
- **resType** – the operation result code: if the function succeeds, it returns rt_Success; otherwise it returns an error code from the MbResultType enumeration (see *mb_operation_result.h*).

The function is an auxiliary one for the ExtrusionSolid(), RevolutionSolid(), ExtrusionResult() and RevolutionResult() functions.

3.8.4 Determining the Orientation of the Generatrix

The function

```
double AreaSign( const MbCurve3D &  curve,
                 const MbAxis3D &   axis,
                 const MbVector3D & direction,
                       bool         rotation )
```

computes the area of the projection of a curve onto a virtual coordinate plane to determine the orientation of a generatrix curve when subsequently constructing a body by extruding or revolving the generatrix curve. An open generatrix curve is closed by a segment for the calculation.

The input parameters of the function are:

- **curve** – the generatrix curve;
- **axis** – the axis of revolution (used for the revolution calculation);
- **direction** – the extrusion direction (used for the extrusion calculation);
- **rotation** – the flag specifying the operation type: *true* for revolution and *false* for extrusion.

The output of the function is the projection area of the curve.

The function is an auxiliary one for the ExtrusionSolid(), RevolutionSolid(), ExtrusionResult() and RevolutionResult() functions.

3.8.5 Determining the Orientation of the Cutting Surface

The function

```
void AnalyzeSurfaceRelationToSweptOperation(const MbSurface &     cuttingSurface,
                                            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 a cutting surface relative to the solid which is to be constructed by extruding or revolving a generatrix curve up to a specified surface.

The input parameters of the function are:

- **cuttingSurface** – the cutting surface with respect to which the extrusion or revolution operation is analyzed;
- **imagePosition** – the point on the surface where the image of the generatrix curve lies;
- **curve** – the generatrix curve;
- **direction** – the extrusion direction (used for the extrusion calculation);
- **axis** – the axis of revolution (used for the revolution calculation);
- **rotation** – the flag specifying the operation type: *true* for revolution and *false* for extrusion;
- **operationDirection** – the direction of the subsequent operation: forward (*true*) or backward (*false*).

The output parameters of the function are:

- **relativeSense** – the orientation of the surface relative to the extrusion or revolution operation;
- **resType** – the operation result code: if the function succeeds, it returns rt_Success; otherwise it returns an error code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is an auxiliary one for the ExtrusionSolid(), RevolutionSolid(), ExtrusionResult(), and RevolutionResult() functions.

3.8.6 Orientation of Curves of a Swept Body

The function

```
MbResultType EvolutionNormalize( const MbSurface &           surface,
                                 const RPArray<MbContour> &  contours,
                                 const MbCurve3D &           guide,
                                 const EvolutionValues &     parameters,
                                       MbAxis3D &            axis,
                                       double &              angle,
                                       VERSION               version,
                                       MbMatrix3D *          matrix = nullptr );
```

determines the orientation of a generating contour and a guide curve for constructing a kinematic (evolution) body. It also computes a transformation matrix that moves the section to the nearest end of the guide curve.

The input parameters of the function are:

- **surface** – the surface for a kinematic operation;
- **contours** – the generatrix contours;
- **guide** – the guide curve;
- **parameters** – the operation parameters;
- **version** – the operation version.

The output parameters of the function are:

- **axis** – the rotation axis of the generatrix contour (the positive direction is defined by the right-hand rule with respect to **axis**);
- **angle** – the rotation angle of the generatrix contour;
- **matrix** – a pointer to the transformation matrix of the section to the nearest end of the guide curve (if the pointer is equal to *nullptr*, the matrix is not computed).

If the function succeeds, it returns rt_Success; otherwise, it returns a result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function is an auxiliary one for the EvolutionSolid() and EvolutionResult() functions.

3.8.7 Copying the Guiding Curve of a Sweep Body

The function

```
MbCurve3D *TrimClosedSpine( const MbCurve3D & curve,
                                  double      t )
```

constructs a copy of a closed curve with the start point defined by the parameter **t**.

The input parameters of the function are:

- **curve** — the guiding curve;
- **t** — the curve parameter.

If successful, the function returns the constructed copy of the curve with the start point at the given parameter value; otherwise, it returns null.

The function is auxiliary for the EvolutionSolid() and EvolutionResult() functions.

3.8.8 Constructing a rib

The function

```
MbResultType RibElement( const MbSolid &               solid,
                         const MbRibSolidParameters &  params,
                               MbSolid &               result )
```

constructs a rib for the source solid.

The input parameters of the function are:

- **solid** — the source solid;

- **params** — the rib parameters.

The output parameter of the function is the constructed solid **result**.

If successful, the function returns rt_Success; otherwise, it returns a result code from the *MbResultType* enumeration (see *mb_operation_result.h*).

The function constructs a separate rib without merging it with the source solid.

3.8.9 Checking a Curve for Ruled Body Construction

The function

```
void CheckRuledCurve( const MbCurve3D &  curve0,
                      const MbCurve3D &  curve1,
                            bool &       isInverted,
                            bool &       isShifted,
                            VERSION      version )
```

checks the second curve for consistency with the first curve for creation of an open ruled body and determines whether it is necessary to invert and/or shift the beginning of the second curve for correct construction of the ruled body.

The input parameters of the function are:

- **curve0** — the first curve;
- **curve1** — the second curve;
- **version** — the version of the operation.

The output parameters of the function are:

- **isInverted** — a flag that is set to *true* on output if the second curve must be used in the reverse direction with respect to the original one during construction;
- **isShifted** — a flag that is set to *true* on output if the beginning of parametrization of the second curve must be shifted (reparametrized) during construction in order to be aligned with the first curve.

The function has no return value.

The function is declared in the *action_shell.h* file.

The function is an auxiliary for the RuledShell() function that constructs an open ruled body by two curves.

3.8.10 Checking Curve Parameters for Constructing a Ruled Body

The function

```
bool CheckRuledParams( const MbCurve3D &          curve,
                       SArray<double>   &         params,
                       bool                       isAscending )
```

checks the curve parameters and normalizes the parameters of a closed curve.

The input parameters of the function are:

- **curve** — the curve;
- **params** — the set of the curve parameters;

- **isAscending** — the flag that specifies ascending order of the set parameters.

The output parameter of the function is the set of the curve parameters **params**.

If the parameter array is successfully normalized, the function returns *true*; otherwise, it returns *false*.

The function is declared in the *action_shell.h* file.

The function is auxiliary for the RuledShell() function that constructs a ruled open shell.

3.8.11 Checking a Curve for Joint Body Construction

The function

```
void CheckJoinedCurve( const MbCurve3D &  curve1,
                       const MbCurve3D &  curve2,
                             bool &       isInverted1,
                             bool &       isShifted1,
                             VERSION      version )
```

checks the second curve for consistency with the first curve for creation of an open joint body and determines whether it is necessary to invert and/or shift the beginning of the second curve for correct construction of the joint body.

The input parameters of the function are:

- **curve1** — the first curve;
- **curve2** — the second curve;
- **version** — the version of the operation.

The output parameters of the function are:

- **isInverted1** — a flag that is set to *true* on output if the second curve must be used in the reverse direction with respect to the original one during construction;
- **isShifted1** — a flag that is set to *true* on output if the beginning of parametrization of the second curve must be shifted (reparametrized) during construction in order to be aligned with the first curve.

The function has no return value.

The function is declared in the *action_shell.h* file.

The function is an auxiliary function for the JoinShell() function that constructs an open joint body by two curves.

3.8.12 Checking Curve Parameters for Joint Body Construction

The function

```
bool CheckJoinedParams( const MbCurve3D &        curve,
                              SArray<double> &   params,
                              bool               isAscending )
```

checks the validity of the curve parameters and, if necessary, normalizes the parameters of a closed (periodic) curve to the base parameter range.

The input parameters of the function are:

- **curve** — the curve;
- **params** — a set of the curve parameters;
- **isAscending** — a flag that defines the expected order of parameters in the array; if the value is *true*, the parameters must be (and after normalization remain) ordered in ascending order.

The output parameter of the function is the set of the curve parameters **params**, which is modified if necessary (normalized and, when the ascending flag is set, reordered).

If the array of parameters is successfully checked and (if necessary) normalized, the function returns *true*; if the parameters are invalid and cannot be normalized, the function returns *false*.

The function is declared in the *action_shell.h* file.

The function is an auxiliary function for the JoinShell() function that constructs an open joint body.

3.8.13 Constructing a Curve from a Set of Edges

The function

```
template <typename ConstEdgeVector, typename OrientsVector>
MbCurve3D * CreateJoinedCurve( const ConstEdgeVector &  edges,
                               const OrientsVector &    orients,
                               const MbMatrix3D &       matr,
                                     MbResultType &     res );
```

constructs a curve from a set of edges lying on a joint surface for subsequent use in the JoinShell operation.

The input parameters of the function are:

- **edges** — a set of edges (a container of constant pointers to edges);
- **orients** — a container of Boolean flags that define directions of using the corresponding edges from **edges** (the container size must match the number of edges);
- **matr** — a transformation matrix of edges (the edges are preliminarily transformed by this matrix before constructing the curve).

The output parameter of the function is the value **res** of the MbResultType enumeration.

If the function succeeds, it returns a pointer to the constructed curve; otherwise it returns a null pointer.

The function is declared in the *action_shell.h* file.

3.8.14 Determining Curve Crossing Points

The function

```
ptrdiff_t CurveCurveCrossing( const MbCurve3D &       curve1,
                              const MbCurve3D &       curve2,
                                    SArray<double> &  result1,
                                    SArray<double> &  result2,
                                    double            epsilon = Math::metricRegion );
```

finds crossing points of two spatial curves and returns the parameters of these points on each curve. Crossing points are such points of the curves that the segment connecting these points is perpendicular to the tangents of both curves.

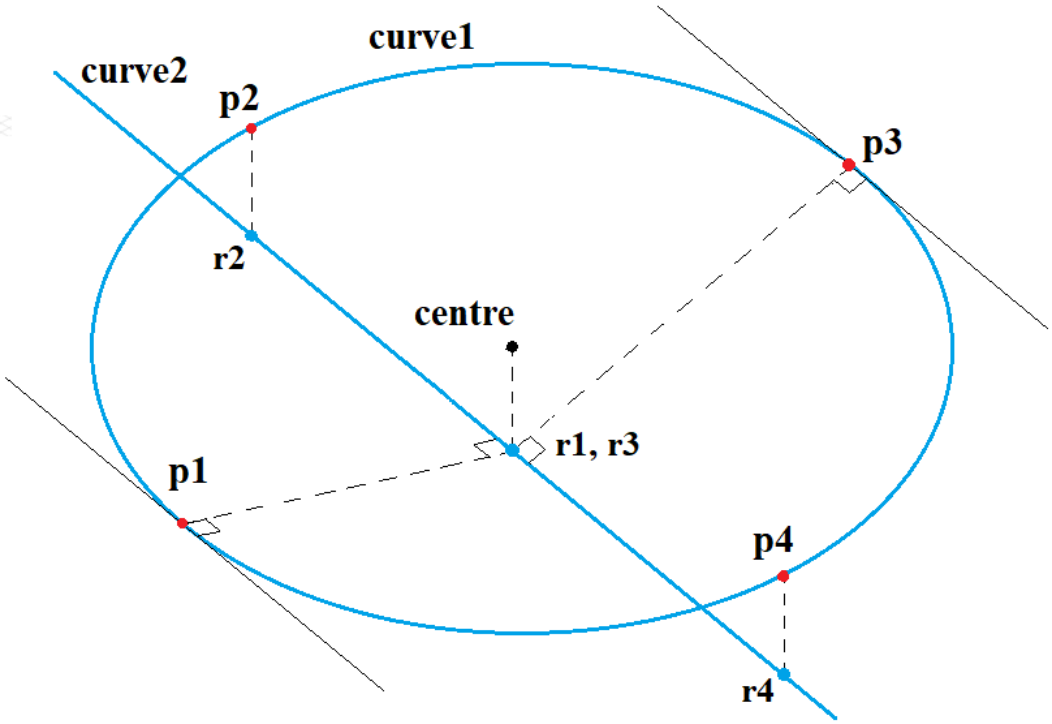
Function parameters:

- **curve1** – the first curve;
- **curve2** – the second curve;
- **result1** – parameters of crossing points on the first curve;
- **result2** – parameters of crossing points on the second curve;
- **epsilon** – accuracy of solution search.

The return value is the number of found crossings.

Parameters of the found crossing points for the first curve are stored in the **result1** array, and for the second curve in the **result2** array. In the special case of a geometric intersection of the curves, the segment between the crossing points degenerates to zero length, and the crossing points become intersection points. In addition to proper crossings, the CurveCurveCrossing() function returns crossing points of the curves for which the segment connecting these points is perpendicular to the tangents to the curves at these points.

For example, in figure the CurveCurveCrossing() function returns four pairs of intersection parameters: on the circle **curve1** they correspond to points p1, p2, p3, and p4, and on the segment **curve2** to points r1, r2, r3, and r4.



3.9 ELEMENTARY OBJECTS

Elementary objects are C3D geometric kernel entities that describe such mathematical constructs as a vector, a point, an axis, a local coordinate system, a transformation matrix, a bounding box, and a bounding rectangle. Elementary objects have simple data structures. They serve as a tool and building material for more complex geometric objects and are used by all modules of the geometric kernel.

3.9.1 Vector in three-dimensional space MbVector3D

The MbVector3D class is declared in the *mb_vector3d.h* file.

The class supports basic vector operations (addition, subtraction, multiplication by a scalar, dot and cross products, normalization) and serves as a basic building block for describing directions, normals, and displacements in three-dimensional space.

A MbVector3D vector describes a displacement or a direction in three-dimensional space and is defined by three components *x*, *y*, *z* in a Cartesian coordinate system.

To denote vectors in three-dimensional space, we will use one or several lowercase Latin letters in boldface, and to combine vector components we will enclose them in square brackets, for example,

$$\mathbf{v} = [x, y, z]. \tag{3.1}$$

A MbVector3D vector is not bound to any point in space and therefore has no method that translates it in space.

3.9.2 Radius Vector of a Point in Three-Dimensional Space MbCartPoint3D

The MbCartPoint3D class is declared in the *mb_cart_point3d.h* file.

The MbCartPoint3D radius vector (a Cartesian point) describes the position of a point in three-dimensional space and is defined by three coordinates *x*, *y*, *z* in a Cartesian coordinate system. The radius vector represents a transformation that maps the origin of the Cartesian coordinate system to the point in space with the given coordinates in this Cartesian coordinate system.

To denote points in three-dimensional space, we will use one or several lowercase Latin letters in boldface, and to combine point coordinates we will enclose them in square brackets, for example,

$$\mathbf{p} = [x, y, z]. \tag{3.2}$$

The MbCartPoint3D radius vector, unlike the MbVector3D vector, is bound to the origin of the chosen coordinate system. Coordinates of the MbCartPoint3D radius vector and components of the *MbVector3D* vector are changed in a different way when switching from one coordinate system to another and when changing the position in space by means of the following methods:

MbCartPoint3D & Transform(const *MbMatrix3D* &),

MbCartPoint3D & Rotate(const MbAxis3D &, double angle),

MbCartPoint3D & Move(const *MbVector3D* &).

These methods return a reference to the point itself after the transformation.

3.9.3 Homogeneous Vector in Three-Dimensional Space MbHomogenius3D

The MbHomogeneous3D class is declared in the *mb_homogenius3d.h* file.

The extended radius vector MbHomogeneous3D describes the position of a point in three-dimensional space and is defined by four coordinates *x*, *y*, *z*, *w*. The fourth coordinate is called the weight. The extended radius vector is used when computing the radius vector of *B*-curves and *B*-surfaces constructed on the basis of *B*-splines. The coordinates x_w, y_w, z_w, *w* of the extended radius vector MbHomogeneous3D are related to the coordinates *x*, *y*, *z* of the radius vector by the relations:

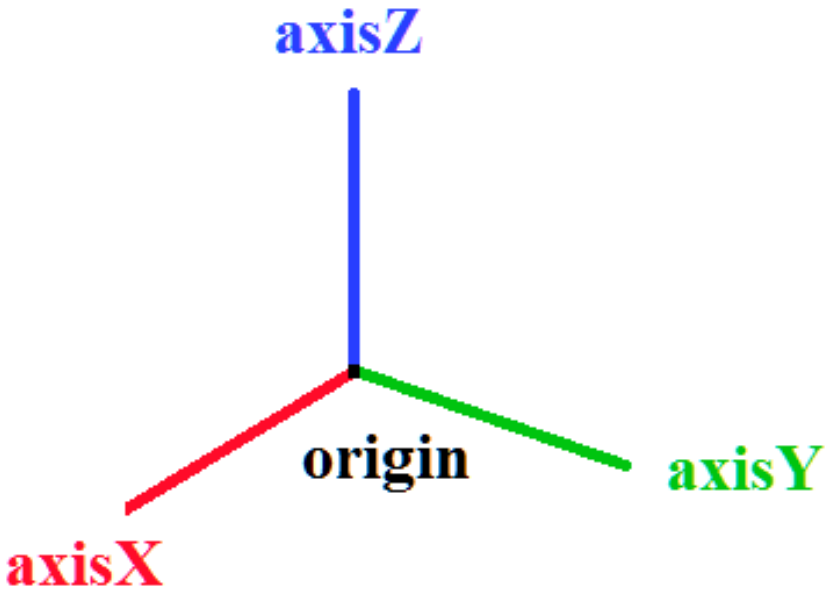
$$x = \frac{x_w}{w}, \quad y = \frac{y_w}{w}, \quad z = \frac{z_w}{w}.$$

In operations of multiplication by the extended *MbMatrix3D* matrix, it can be assumed that vectors and points also have the fourth coordinate: for a *MbVector3D* vector it is equal to zero, and for a *MbCartPoint3D* radius vector it is equal to one.

3.9.4 Local Coordinate System MbPlacement3D

The MbPlacement3D class is declared in the *mb_placement3d.h* file.

The local coordinate system in three-dimensional space MbPlacement3D is defined by the initial point **origin** and three non-coplanar vectors **axisX**, **axisY**, **axisZ**:



In most cases the coordinate system is right-handed, and the system vectors are orthonormal. By means of transformations the coordinate system can become left-handed and non-orthonormal. The state of a local coordinate system can be queried by the following methods:

- bool IsLeft() — whether the coordinate system is left-handed;
- bool IsRight() — whether the coordinate system is right-handed;
- bool IsTranslation() — whether the origin **origin** of the coordinate system is shifted;

- bool IsRotation() — whether the coordinate system is rotated with respect to the global one;
- bool IsOrt() — whether the coordinate system is marked as orthogonal by the internal flag;
- bool IsSingle() — whether the coordinate system coincides with the global one;
- bool IsNormal() — whether the coordinate system is orthonormal;
- bool IsOrthogonal() — whether the coordinate system is orthogonal;
- bool IsCircular() — whether the coordinate system is orthogonal with the **axisX** and **axisY** axes of equal length, so that a circle remains a circle;
- bool IsIsotropic() — whether the coordinate system is orthogonal with the **axisX**, **axisY**, **axisZ** axes of equal length, so that objects are not distorted and only scaled;
- bool IsAffine() — whether the coordinate system is affine (allows translation, scaling, and skewing of axes).

The local coordinate system is Cartesian. A point in the Cartesian coordinate system is defined by three coordinates *x*, *y*, *z*. A local coordinate system can act as a cylindrical or a spherical coordinate system.

The local coordinate system MbPlacement3D can be used to define a cylindrical coordinate system. In this case the origin of the cylindrical system coincides with the origin of the Cartesian system, the Oz-axis coincides with the Z-axis of the Cartesian system, and the main plane coincides with the XY-plane.

A point in a cylindrical coordinate system is defined by three coordinates *r*, *f*, *z*:

- *r*, *f* are the polar coordinates of the projection of the point onto the main plane;
- *r* specifies the length of the radius vector projection;
- *f* specifies the polar angle of the radius vector projection, *z* specifies the distance from the point to the main plane.

When the local coordinate system MbPlacement3D is used as a spherical coordinate system, a point is defined by three coordinates *r*, *f*, *w*:

- *r* specifies the distance from the origin to the point (the length of the radius vector);
- *f* specifies the angle between the projection of the radius vector onto the plane and a ray lying in this plane (longitude);
- *w* specifies the angle between the radius vector and the normal to the plane of the spherical coordinate system (polar distance).

The plane of the spherical coordinate system coincides with the XY-plane of the Cartesian system, and the longitude *f* is measured from the X-axis towards the positive direction of the Y-axis.

3.9.5 Extended Matrix in Three-Dimensional Space MbMatrix3D

The MbMatrix3D class is declared in the *mb_matrix3d.h* file.

In three-dimensional space, the MbMatrix3D matrix describes a transformation from one coordinate system to another. The matrix has the size four by four. Let a local affine coordinate system be constructed in the chosen coordinate system with the origin at the point **r** with coordinates r_1, r_2, r_3 and with basis vectors **a** = [a_1 a_2 a_3], **b** = [b_1 b_2 b_3], **c** = [c_1 c_2 c_3]. The vectors **a**, **b**, **c** must be linearly independent, but they may be non-orthogonal to each other and may

have arbitrary length. The MbMatrix3D matrix of transformation of coordinates from the local system to the chosen coordinate system has the form:

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

To denote extended matrices in three-dimensional space, we will use uppercase Latin letters in boldface, for example **M**. Note that each basis vector **a**, **b**, **c** and the origin point **r** of the local coordinate system corresponds to its own row in the matrix of transformation from the local system to the chosen coordinate system.

The MbMatrix3D matrix is an extended matrix and works with homogeneous radius vectors and homogeneous vectors MbHomogeneous3D in three-dimensional space.

When transforming a *MbCartPoint3D* radius vector by the MbMatrix3D matrix, a fourth coordinate equal to one must be added to the point. Let a point with coordinates x_1, x_2, x_3 in the local coordinate system have coordinates p_1, p_2, p_3 in the chosen coordinate system. Then, by means of the extended MbMatrix3D matrix, the coordinates are related by

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

Note that a three-dimensional radius vector is multiplied by the extended MbMatrix3D matrix on the right.

When transforming an *MbVector3D* vector by the MbMatrix3D matrix, a fourth component equal to zero must be added to the vector. Let a vector with components y_1, y_2, y_3 in the local coordinate system have components v_1, v_2, v_3 in the chosen coordinate system. Then, by means of the extended MbMatrix3D matrix, the components are related by

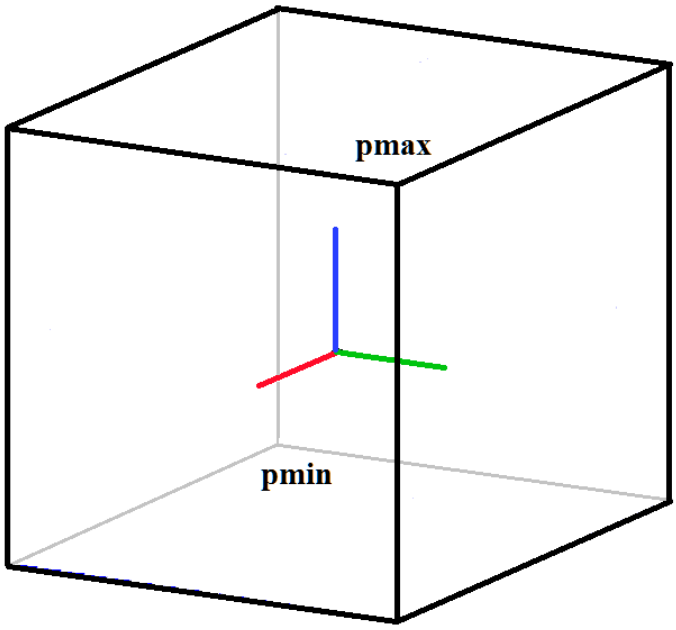
$$\begin{bmatrix} v_1 & v_2 & v_3 & 0 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & y_3 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

Note that a three-dimensional vector is multiplied by the extended MbMatrix3D matrix on the right.

3.9.6 Bounding Box in Three-Dimensional Space MbCube

The MbCube class is declared in the *mb_cube.h* file.

The MbCube bounding box describes the extents of an extended object (a curve, a surface, a solid, or several solids) in three-dimensional space and is defined by two points **pmin** and **pmax**. The faces of the bounding box are parallel to the planes of the coordinate system in which the cube is defined. The points **pmin** and **pmax** describe two opposite vertices of the bounding box that have the smallest and the largest coordinates, respectively:



If the extents of an extended object are not defined, then the bounding box is considered empty and **pmin** = [10^{-300} 10^{-300} 10^{-300}], **pmax** = [-10^{-300} -10^{-300} -10^{-300}]. For an empty bounding box **pmin** > **pmax**, and the IsEmpty() method returns *true*.

3.9.7 MbRect1D Univariate Dimension

The MbRect1D class is declared in the *mb_rect1d.h* file.

The MbRect1D univariate dimension describes a one-dimensional area, for example, a curve parameter domain, and it is defined by two numbers: *zmin* and *zmax*:



The *zmin* and *zmax* numbers define the start and the end boundary of the area. If a one-dimensional area is not defined, then the univariate dimension is considered empty and *zmin* > *zmax*. For an empty univariate dimension, the IsEmpty() method returns *true*.

3.9.8 MbVector in Two-Dimensional Space

The MbVector() class is declared in the *mb_vector.h* file.

The MbVector() vector describes a displacement or a direction in two-dimensional space and is defined by two components *x* and *y* in a Cartesian coordinate system.

To denote vectors in two-dimensional space, we will use one or several lowercase Latin letters in bold italic typeface, and we will enclose the vector components in square brackets, for example,

$$\mathbf{v} = [x, y]. \tag{3.3}$$

The MbVector() vector is not bound to points in space, and therefore it does not have a method that moves it in space.

3.9.9 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(□), *ay* = sin(□), where □ is the angle between the normalized vector and x-axis of the coordinate system.

3.9.10 Point Radius Vector in Two-Dimensional Space MbCartPoint

The MbCartPoint() class is declared in the *mb_cart_point.h* file.

The radius vector of MbCartPoint() (a Cartesian point) defines the position of a point in two-dimensional space and is specified by two coordinates *x* and *y* in a Cartesian coordinate system. The radius vector describes the transformation that moves the origin of the Cartesian coordinate system to the point in space with the given coordinates in this Cartesian coordinate system. To denote points in two-dimensional space we will use one or several lowercase Latin letters in bold italics, and to combine the point coordinates we will enclose them in square brackets, for example,

$$\mathbf{point} = [x\ y]. \tag{3.4}$$

Unlike a vector, a radius vector is bound to the origin. The coordinates of the radius vector MbCartPoint() and the components of the MbVector vector change in different ways when switching from one coordinate system to another and when changing the position in space, which is implemented in the methods:

void Transform(const *MbMatrix* &);

void Rotate(const MbCartPoint & , double angle);

void Move(const MbVector &).

3.9.11 Homogeneous Vector in Two-Dimensional Space MbHomogeneous

The MbHomogeneous class is declared in the *mb_homogeneous.h* file.

The extended radius vector MbHomogeneous() defines the position of a point in two-dimensional space and is specified by three coordinates *x*, *y*, and *w*. The third coordinate is called a weight. The extended radius vector is used when computing the radius vector of two-dimensional *B*-curves constructed on the basis of *B*-splines. The coordinates x_w, y_w, *w* of the extended radius vector MbHomogeneous are related to the coordinates *x*, *y* of the radius vector *MbCartPoint* by the relations:

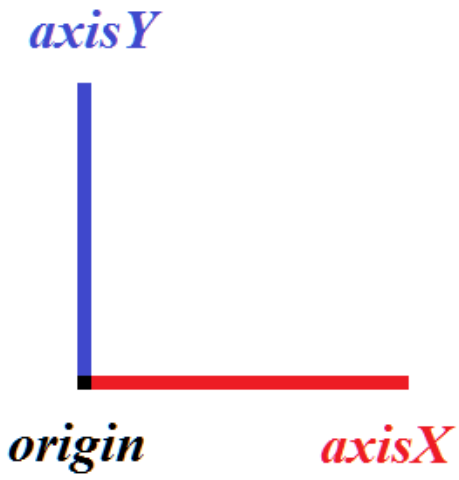
$$x = \frac{x_w}{w}; \quad y = \frac{y_w}{w}.$$

In operations of multiplication by the extended matrix *MbMatrix* it can be assumed that two-dimensional vectors and points also have the third coordinate: for a *MbVector* vector it is equal to zero, and for a MbCartPoint() radius vector it is equal to one.

3.9.12 Local Coordinate System MbPlacement

The MbPlacement() class is declared in the *mb_placement.h* file.

The local coordinate system MbPlacement() is used to define the position and orientation of objects in two-dimensional space and to construct transformation matrices MbMatrix. The local coordinate system MbPlacement() in two-dimensional space is described by the origin point **origin** and two non-parallel vectors **axisX** and **axisY**:



In most cases the coordinate system is right-handed and its vectors are orthonormal. By applying transformations the coordinate system may become left-handed and non-orthonormal. The state of the local coordinate system can be queried by the methods:

bool IsLeft() – whether the coordinate system is left-handed;

bool IsSingle() – whether the coordinate system coincides with the one in which it is defined;

bool IsNormal() – whether the coordinate system is orthonormal;

bool IsCircular() – whether the coordinate system is orthogonal with the axes **axisX** and **axisY** of equal length, in which a circle remains a circle;

bool IsIsotropic() – whether the coordinate system is orthogonal with the axes **axisX** and **axisY** of equal length, in which objects are not distorted but only scaled;

bool IsAffine() – whether the coordinate system is affine (otherwise the system is orthonormal).

The local coordinate system is Cartesian.

3.9.13 Extended Matrix in Two-Dimensional Space MbMatrix

The MbMatrix class is declared in the *mb_matrix.h* file.

In a two-dimensional space, the MbMatrix matrix describes a transformation from one coordinate system to another. The matrix has the size three by three. Let a local affine coordinate system be constructed in the chosen coordinate system with the origin at the point r with coordinates r_1, r_2 and with the basis vectors a = [a_1 a_2], b = [b_1 b_2]. The vectors a and b must not be collinear, but they may be non-orthogonal to each other and may have arbitrary lengths. The MbMatrix matrix of the coordinate transformation from the local system to the chosen coordi-

nate system has the form:

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

To denote extended matrices in the two-dimensional space we will use uppercase Latin letters in bold italics, for example M. Note that each basis vector a, b and the origin point r of the local coordinate system correspond to their own row in the transformation matrix from the local system to the chosen coordinate system.

The MbMatrix matrix is extended and works with homogeneous radius vectors and homogeneous vectors *MbHomogeneous* in a two-dimensional space.

When transforming a *MbCartPoint* radius vector by the MbMatrix matrix, a third coordinate equal to one must be added to the point. Let a point with coordinates x_1, x_2 in the local coordinate system have coordinates p_1, p_2 in the chosen coordinate system, then with the extended MbMatrix matrix the coordinates are related by

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

Note that a two-dimensional radius vector is multiplied by the extended MbMatrix matrix on the right.

When transforming an MbVector vector by the MbMatrix matrix, a third component equal to zero must be added to the vector. Let a vector with components y_1, y_2 in the local coordinate system have components r_1, r_2 in the chosen coordinate system, then with the extended MbMatrix matrix the components are related by

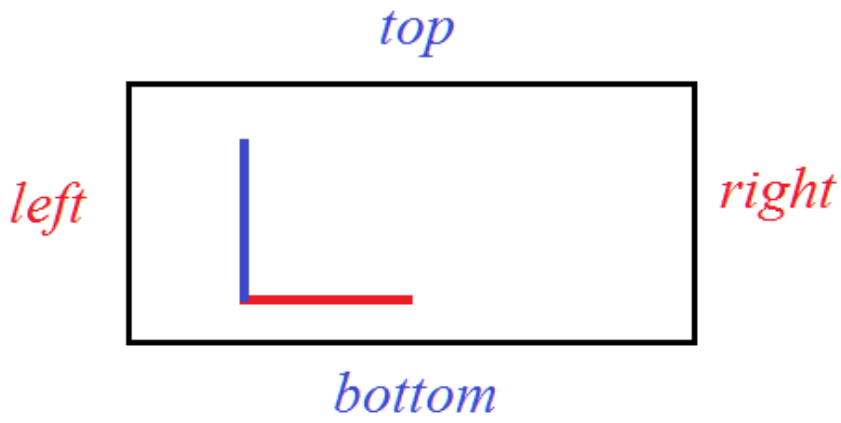
$$\begin{bmatrix} r_1 & r_2 & 0 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}.$$

Note that a two-dimensional vector is multiplied by the extended MbMatrix matrix on the right.

3.9.14 Bounding Rectangle in Two-Dimensional Space MbRect

The MbRect class is declared in the *mb_rect.h* file.

The MbRect bounding rectangle describes the dimensions of an extended object (a curve or several curves) in two-dimensional space and is defined by four numbers: *left*, *right*, *bottom*, *top*. The sides of the bounding rectangle are parallel to the axes of the coordinate system in which the rectangle is defined. The *left* and *right* numbers describe the minimum and maximum abscissa of the bounding rectangle, and the *bottom* and *top* numbers describe the minimum and maximum ordinate of the bounding rectangle:



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

3.10 GEOMETRICAL OBJECTS

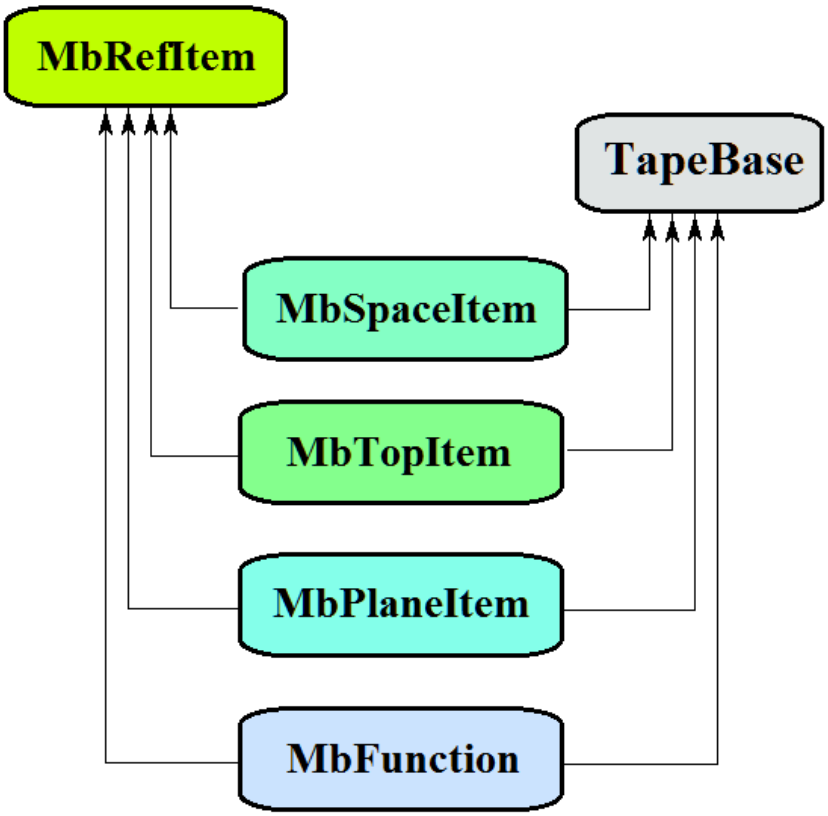
Geometric objects describe the shape of modeled entities. They include curves, surfaces, solids, as well as topological entities that represent geometric properties independent of quantitative characteristics and capture the continuous connectivity of points in three-dimensional space. Geometric objects can be three-dimensional or two-dimensional. Two-dimensional objects are used when working in the parameter domains of surfaces and in planes of three-dimensional local coordinate systems. This section describes the base classes of geometric objects.

3.10.1 MbRefItem Reference Counter

The MbRefItem class is declared in the *reference_item.h* file.

The MbRefItem class is characterized by the number of its owners *useCount* and represents a counter of objects that own the given object.

The geometric objects of the C3D kernel are divided into three groups: two-dimensional geometric objects, three-dimensional geometric objects and topological objects. Geometric objects of these three groups are derived from the MbRefItem and TapeBase classes:



The TapeBase class provides stream writing and reading for its child class.

The following objects are derived from the MbRefItem and TapeBase classes:

MbSpaceItem is the base abstract class of three-dimensional geometric objects;

MbTopItem is the base abstract class of topological objects;

MbPlaneItem is the base abstract class of two-dimensional geometric objects;

MbFunction is the base abstract class of scalar functions.

The reference counter ensures correct work of classes and methods that contain pointers to geometric objects. If a class stores a pointer to a geometric object in its data, then in its constructor it must increase the reference counter of the geometric object by one using the AddRef() method, and in its destructor it must call the Release() method of the geometric object, which decreases the reference counter of the geometric object by one and, when the reference counter reaches zero, deletes the geometric object.

The DecRef() method decreases the reference counter of the geometric object by one. It is intended for internal use in the implementation where a decrease of the counter without ownership release is explicitly required.

The MbRefItem class is processed by the duplication registrar MbRegDuplicate and by the transformation registrar MbRegTransform.

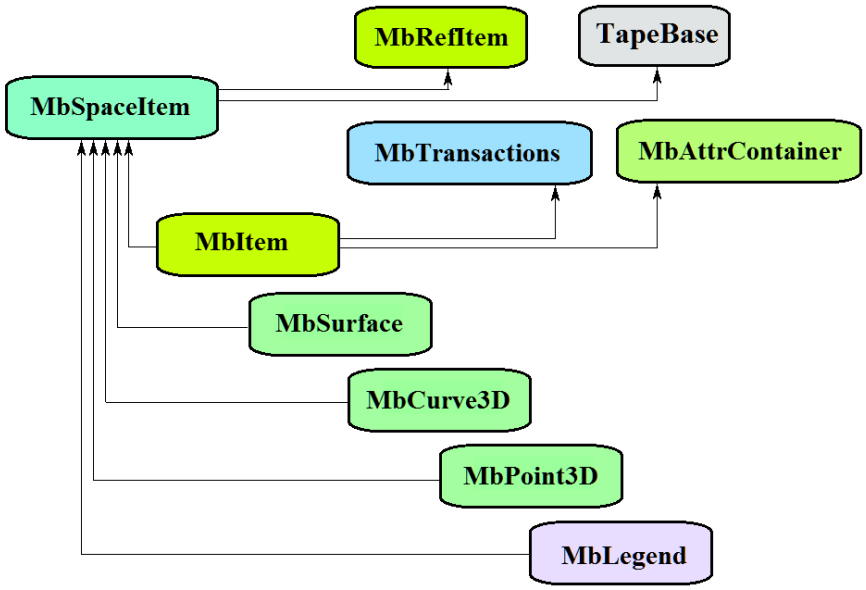
The RefType() method returns the registration type of the object that uses the reference counter.

3.10.2 Three-Dimensional Geometric Object MbSpaceItem

The MbSpaceItem class is declared in the *space_item.h* file.

The MbSpaceItem class is derived from the *MbRefItem* and TapeBase classes and is the base class for three-dimensional geometric objects.

The three-dimensional geometric objects of the C3D kernel include points, curves, surfaces, auxiliary objects and geometric model objects:



The following families of three-dimensional geometric objects are derived from the MbSpaceItem class:

MbPoint3D is a point.

MbCurve3D is a curve.

MbSurface is a surface.

MbLegend is an auxiliary geometric object.

MbItem is a geometric model object.

The main methods of three-dimensional geometric objects are:

```
virtual void Move( const MbVector3D &      to,
                         MbRegTransform *  iReg = nullptr );

virtual void Rotate( const MbAxis3D &       axis,
                           double           angle,
                           MbRegTransform * iReg = nullptr );

virtual void Transform( const MbMatrix3D &     matr,
                              MbRegTransform * iReg = nullptr );
```

These methods perform geometric transformations of the object. When a transformation is applied, the MbRegTransform registrar is used to prevent repeated transformations of shared

nested objects. If an object stores pointers or references to other objects, then the nested objects are also transformed. The registrar must be used when several related objects, which reference common nested objects, are transformed sequentially. If the transformation is performed without the registrar, shared nested objects can be transformed multiple times.

In addition, all geometric objects have methods that provide copying, checking for equality, checking for the possibility to make objects equal, and making objects equal:

```
// Create a copy of the object using the registrar
MbSpaceItem & Duplicate( MbRegDuplicate * iReg = nullptr );

// Determine whether the objects are equal
virtual bool IsSame( const MbSpaceItem & other,
                     double              accuracy = LENGTH_EPSILON );

// Determine whether the objects are similar
virtual bool IsSimilar( const MbSpaceItem & item );

// Make the objects equal if they are similar
virtual bool SetEqual( const MbSpaceItem & item );
```

When a copy is created, the MbRegDuplicate registrar is used to prevent repeated copying of shared nested objects. If an object stores pointers or references to other objects, then the nested objects are also copied. The registrar must be used when several related objects, which reference common nested objects, are copied sequentially. If copying is performed without the registrar, multiple copies of the same nested object may be created instead of a single shared copy.

To identify the type, geometric objects are provided with the following methods:

```
MbeSpaceType IsA();
MbeSpaceType Type();
MbeSpaceType Family();
```

These methods return a value of the enumeration of three-dimensional geometric object types.

The following methods

```
MbProperty & CreateProperty( MbePrompt name );

void GetProperties( MbProperties & properties );

void SetProperties( MbProperties & properties );
```

provide access to and editing of the internal data of geometric objects.

The GetProperties() method appends the object data to the **properties** collection in the form of descendants of the MbProperty class.

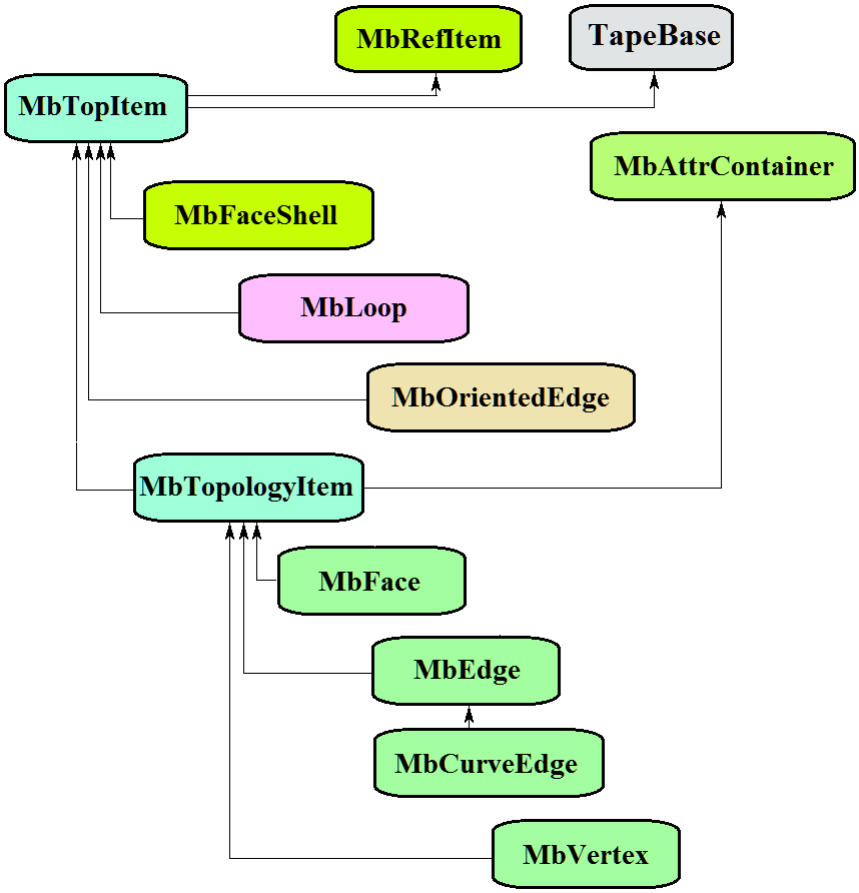
The CalculateWire(double sag, *MbMesh* & **mesh**) method builds a polygonal copy of the geometric object which is used for visualization.

3.10.3 Topological Object MbTopItem

The MbTopItem class is declared in the *topology_item.h* file.

The MbTopItem class is derived from the *MbRefItem* and TapeBase classes and is the base class for topological objects. Among the topological objects, the class of named topological objects *MbTopologyItem* is distinguished. It is derived from the MbTopItem and MbAttributeContainer classes. The topology object proxy MbTopologyProxy is derived from MbTopologyItem and is used in naming algorithms. The *MbTopologyItem* class is also declared in the *topology_item.h* file.

The C3D geometric kernel operates on the topological objects shown in Figure:



The following topological objects are derived from the MbTopItem class:

MbFaceShell – a set of faces;

MbLoop – a cycle of edges forming a face boundary;

MbOrientedEdge – an oriented edge of a cycle;

MbTopologyItem – a named topological object.

The following objects are derived from the named topological object MbTopologyItem:

MbVertex – a vertex;

MbEdge – an edge;

MbFace – a face.

The edge has a descendant *MbCurveEdge* that describes a smooth junction of two faces or a face boundary.

The interaction of named topological objects with attributes is provided by the attribute container MbAttributeContainer.

The main methods of named topological objects are:

```
virtual void Transform( const MbMatrix3D &     matr,
                              MbRegTransform * iReg = nullptr );
virtual void Move( const MbVector3D &     to,
                         MbRegTransform * iReg = nullptr );
virtual void Rotate( const MbAxis3D &       axis,
                           double           angle,
                           MbRegTransform * iReg = nullptr );
```

These methods perform transformations of a topological object and operations with its name and attributes. When a transformation is performed, the MbRegTransform registrar is used to prevent multiple transformations of nested objects. If an object contains pointers or references to other objects, then the nested objects are also transformed. The registrar must be used when several related objects are transformed sequentially and their relationship is defined by pointers or references to common objects. Performing transformations without a registrar may lead to multiple transformations of common nested objects.

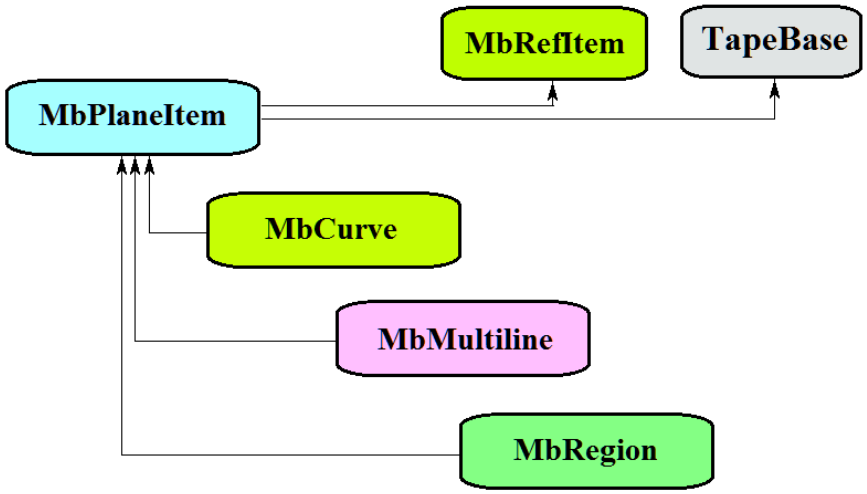
To identify their type, topological objects are equipped with the IsA() method, which returns a type from the enumeration of topological objects MbeTopologyType.

3.10.4 Two-Dimensional Geometric Object MbPlaneItem

The MbPlaneItem class is declared in the *plane_item.h* file.

The MbPlaneItem class is derived from the *MbRefItem* and TapeBase classes and is the parent class for all two-dimensional geometric objects.

The two-dimensional geometric objects of the C3D kernel include curves, multiline and region:



The following families of two-dimensional geometric objects are derived from MbPlaneItem:

MbCurve – two-dimensional curve;

MbMultiline – multiline;

MbRegion – region.

The main methods of two-dimensional geometric objects are:

```
virtual void Move( const MbVector &         to,
                         MbRegTransform *   iReg        = nullptr,
                   const MbSurface *        newSurface  = nullptr );

virtual void Rotate( const MbCartPoint &     pnt,
                     const MbDirection &     angle,
                           MbRegTransform *  iReg       = nullptr,
                     const MbSurface *       newSurface = nullptr ) = 0;

virtual void Transform( const MbMatrix &         matr,
                              MbRegTransform *   iReg       = nullptr,
                        const MbSurface *        newSurface = nullptr );
```

serving the transformation of a two-dimensional geometric object. During the transformation, the registrator MbRegTransform is used to prevent multiple transformation of nested objects. If an object contains pointers or references to other objects, then the nested objects are also transformed. The registrator must be used if it is necessary to sequentially transform several interrelated objects whose connection is determined by the presence of pointers or references to common objects in them. When transforming without a registrator, one can obtain multiple transformation of common nested objects.

In addition, all geometric objects have methods providing duplication, checking for equality, checking the possibility of making them equal, and making objects equal:

```
MbPlaneItem & Duplicate( MbRegDuplicate * iReg = nullptr );

bool IsSame( const MbPlaneItem & item,
                   double        accuracy = LENGTH_EPSILON );

bool IsSimilar( const MbPlaneItem & item );

bool SetEqual( const MbPlaneItem & item );
```

When creating a copy, the registrator MbRegDuplicate is used to prevent multiple copying of nested objects. If an object contains pointers or references to other objects, then the nested objects are also copied. The registrator must be used if it is necessary to sequentially copy several interrelated objects whose connection is determined by the presence of pointers or references to common objects in them. When copying without a registrator, one can obtain a set of copies of the same nested object instead of a single copy of it.

For type identification, geometric objects are equipped with methods

```
MbePlaneType IsA();

MbePlaneType Type();

MbePlaneType Family();
```

returning the type from the enumeration of two-dimensional geometric objects.

The methods

```
MbProperty & CreateProperty( MbePrompt name );

void GetProperties( MbProperties & properties );

void SetProperties( MbProperties & properties );
```

provide retrieval and editing of internal data of geometric objects. The GetProperties() method adds the object data to the **properties** set in the form of classes derived from MbProperty.

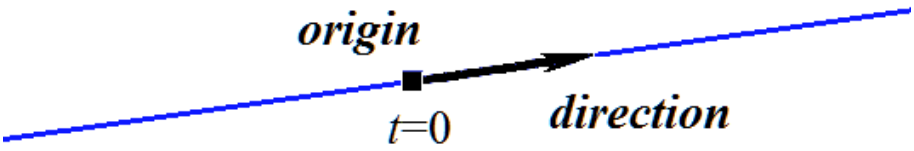
3.11 TWO-DIMENSIONAL CURVES

Two-dimensional curves are used to define surface parameter domains, to construct planar sketches, to build three-dimensional curves on surfaces, curves of surface intersections, and projections of three-dimensional curves onto surfaces and planes of local coordinate systems. Many two-dimensional curves are organized in the same way as three-dimensional curves, with the only difference that two-dimensional points and vectors are used instead of three-dimensional ones. Vectors, position vectors of points, and matrices in a two-dimensional space will be denoted by Latin letters set in bold italic type.

3.11.1 Two-Dimensional Line MbLine

The MbLine class is declared in the *cur_line.h* file.

The two-dimensional straight line MbLine is defined by the start point MbCartPoint **origin** and the direction vector MbDirection **direction**:



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

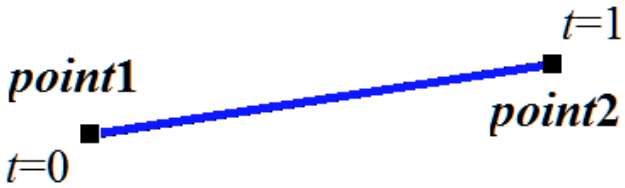
$$\mathbf{r}(t) = \mathbf{origin} + t \cdot \mathbf{direction}.$$

The straight line behaves as an infinite object although its data has boundary values of the parameter t_{min} and t_{max}. Note that unlike other curves, in the methods for calculating the radius vector and its derivatives the line does not adjust the parameter *t* when it goes beyond the limit values t_{min} and t_{max}.

3.11.2 Two-Dimensional Line Segment MbLineSegment

The MbLineSegment class is declared in the *cur_line_segment.h* file.

The two-dimensional line segment MbLineSegment is defined by the start point *MbCartPoint* **point1** and the end point *MbCartPoint* **point2**:



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

$$\mathbf{r}(t) = (1 - t) \cdot \mathbf{point1} + t \cdot \mathbf{point2}$$

The parameter domain of the segment lies between zero and one. The parameter t_{min} = 0 corresponds to the start point of the segment **point1**, and the parameter t_{max} = 1 corresponds to the end point of the segment **point2**.

3.11.3 MbPointCurve Two-Dimensional Curve-Point

The MbPointCurve class is declared in the *cur_point_curve.h* file.

A two-dimensional point curve is defined by a point *MbCartPoint* **point**, the minimum curve parameter **tmin**, the maximum curve parameter **tmax**, and the curve closedness flag **closed**.

In the method

```
PointOn( double & t, MbCartPoint & r)
```

the radius vector of the curve **r** is described by the vector function **r**(*t*) = **point**.

The curve parameter **t** takes values on the interval **tmin** ≤ **t** ≤ **tmax**. The curve can be periodic. The following inequality must hold for the parameters: **tmin** < **tmax**.

A two-dimensional point curve is used together with another two-dimensional curve to describe the intersection of surfaces, one of which has a singular point, for example, a pole. The parameters **tmin**, **tmax**, **closed** of the point curve correspond to the parameters of the two-dimensional curve used together with the point curve.

3.11.4 Two-Dimensional Projection Curve MbProjCurve

The MbProjCurve class is declared in the *cur_projection_curve.h* file.

A two-dimensional projection curve is described by a space curve *MbCurve3D** **spaceCurve**, a surface *MbSurface** **surface** and a two-dimensional curve MbCurve* **curve**. The curve also has some additional data that is optional and is used to speed up the curve methods.

A two-dimensional projection curve represents the projection of the space curve **spaceCurve** onto the surface **surface**, which is approximately described by the two-dimensional curve **curve** within the definition domain of the surface parameters. The definition domains of the parameters of the curves **spaceCurve** and **curve** coincide. The two-dimensional curve **curve** is usually a spline whose control points are obtained by projecting points of the space curve **spaceCurve** onto the surface **surface**. The parametrization of the curve **curve** is consistent with the parametrization of the space curve at the control points. The two-dimensional curve **curve** may lie outside the definition domain of the surface parameters.

In the method

```
PointOn( double & t, MbCartPoint & r)
```

the radius vector of the curve **r** is described by the vector function

$$\mathbf{r}(t) = [u \quad v],$$

where *u*, *v* are the parameters of the projection of the point **spaceCurve**(*t*) onto the surface **surface**.

The PointOn() method corrects the parameter *t* if it is outside the curve domain. The _PointOn() method is used to calculate a point without correcting the parameter.

An initial approximation of the parameters *u* and *v* is computed by the method

```
curve->PointOn(t, point),
```

u = **point**.x, *v* = **point**.y.

Then the parameters *u* and *v* are refined by an iterative method using the 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 the partial derivatives of the radius vector of the surface, which are calculated by the methods

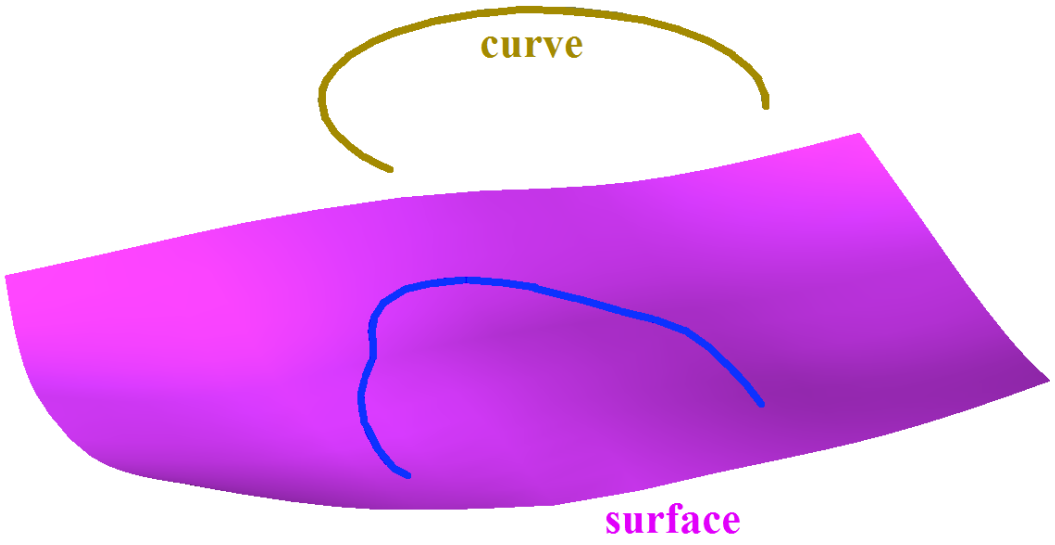
```
surface->_DeriveU(u, v, deriveU)
```

and

```
surface->_DeriveV(u, v, deriveV),
```

respectively.

A projection curve is shown in the figure:



The projection curve is used for an exact description of the projection of a space curve onto a surface.

3.11.5 Two-Dimensional Contour MbContour

The MbContour class is declared in the *cur_contour.h* file.

The two-dimensional contour MbContour is described by a set of RPArray<MbCurve> **segments** of curves joined to each other and by the curve closedness attribute **closed**.

A two-dimensional contour is a composite curve. Unlike other curves, a contour can have breaks. The curves forming a contour will be called **segments**. The following conditions are satisfied for contour segments: the beginning of each subsequent segment coincides with the end of the previous segment. For a periodic contour, the beginning of the first segment coincides with the end of the last segment. If a contour contains only one segment of one of the following types, it is considered nonperiodic: MbLine, MbLineSegment, MbCosinusoid.

To check the closedness and continuity of contour points with a specified tolerance, use the IsClosedContinuousC0() method. The method checks the coincidence of the first and last contour points, as well as the coincidence of the end of each segment with the beginning of the next one. The check does not require smooth joining of segments. By default, the tolerance 5.0 * PARAM_NEAR is used.

The initial value of the contour parameter is zero: $t_{\text{min}} = 0$. The parametric length of a contour is equal to the sum of the parametric lengths of its constituent segments:

$$t_{\text{max}} = \sum (w_{i\text{max}} - w_{i\text{min}}) ,$$

where $w_{\text{i min}}$ and $w_{\text{i max}}$ are the minimum and maximum parameter values of the *i*-th segment. When calculating the radius-vector of a contour point, the working segment and the value of its local parameter are first determined from the parameter value. Then, the radius-vector of the working segment is calculated, which serves as the contour radius-vector.

In the method

```
PointOn( double &       t,
         MbCartPoint &  r )
```

the curve radius-vector **r** is described by the vector function

$$\mathbf{r}(t) = \mathbf{segments}[k](w_k),$$

where **segments**[*k*](w_{k}) is the working contour segment with index *k*, and w_{k} is the working segment parameter equal to

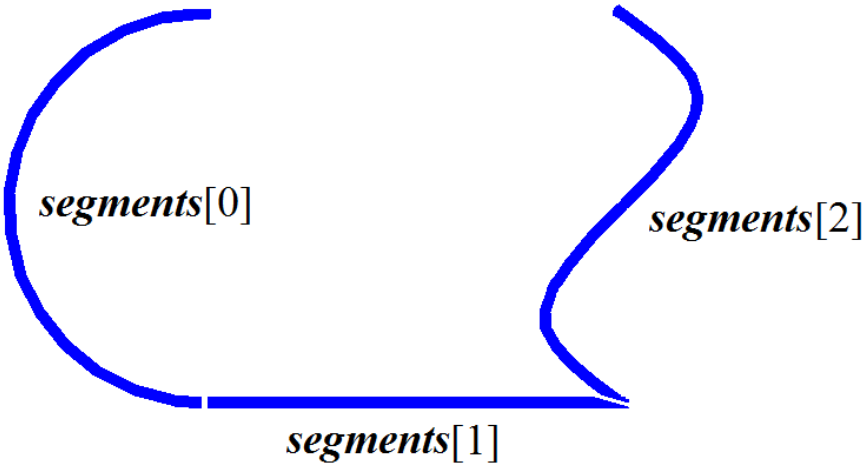
$$w_k = w_{k\text{min}} + t - \sum_{i=0}^{k-1} (w_{i\text{max}} - w_{i\text{min}}) .$$

The segment with index *k* is determined from the contour parameter *t* according to the condition

$$\sum_{i=0}^{k-1} (w_{i\text{max}} - w_{i\text{min}}) \le t < \sum_{i=0}^{k} (w_{i\text{max}} - w_{i\text{min}}) ,$$

where $w_{\text{i min}}$ and $w_{\text{i max}}$ are the minimum and maximum parameter values of the *i*-th segment.

The contour is shown in the figure:



Other two-dimensional contours must not be used as segments of a two-dimensional contour. If a contour needs to be constructed based on other contours, the latter must be considered as a set of their constituent curves rather than as individual curves.

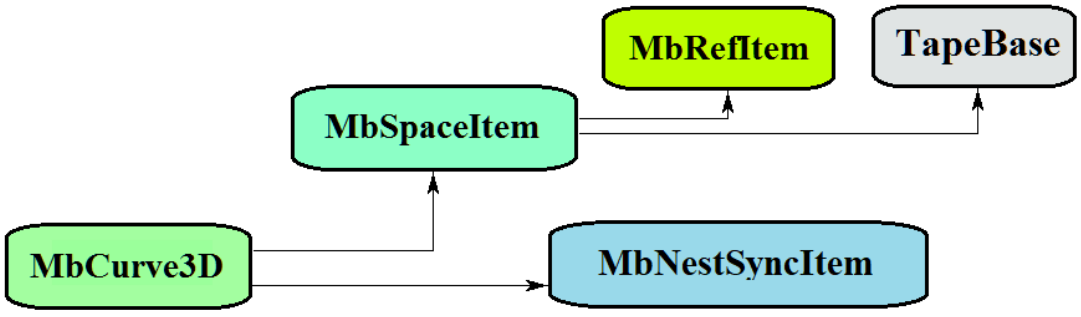
3.12 CURVES

Curves are representatives of the *MbSpaceItem* three-dimensional geometric objects family. All curves have a common parent class *MbCurve3D*. The C3D geometric kernel uses curves that are constructed using analytical functions, from a set of points, based on curves, and based on surfaces. Curves are used to construct surfaces as well as auxiliary elements of the geometric model. We'll use **bold** Latin letters to designate radius vectors of points, and matrices in three-dimensional space.

3.12.1 MbCurve3D Curve

The MbCurve3D class is declared in the *curve3d.h* file.

The MbCurve3D curve is derived from the *MbSpaceItem* and MbNestSyncItem classes:



A curve is an abstract class. The C3D geometric kernel implements the following curves derived from the MbCurve3D class:

MbLine3D – a straight line;

MbLineSegment3D – a line segment;

MbArc3D – an elliptical arc;

MbPolyline3D – a polyline;

MbNurbs3D – a B-curve (NonUniform Rational B-Spline);

MbBezier3D – a composite Bezier curve;

MbHermit3D – a Hermite curve;

MbCubicSpline3D – a cubic spline;

MbOffsetCurve3D – an offset curve;

MbTrimmedCurve3D – a trimmed curve;

MbReparamCurve3D – a reparameterized curve;

MbConeSpiral – a conical spiral;

MbCurveSpiral – a spiral with a straight axis and a variable radius;

MbCrookedSpiral – a spiral with an axis in the form of a planar curve;

MbBridgeCurve3D – a Hermite spline joining two curves;

MbContour3D – a contour (a composite curve);

MbPlaneCurve – a planar curve in space;

MbSurfaceCurve – a curve on a surface;

MbSilhouetteCurve – a surface silhouette curve;

MbContourOnSurface – a contour on a surface;

MbContourOnPlane – a contour on a plane;

MbSurfaceIntersectionCurve – a surface intersection curve.

The MbCurve3D curve is a vector function

$$\text{curve}(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix}$$

of a scalar parameter t taking values within the interval [t_{min}, t_{max}]. A curve is a continuous mapping of a part of the numerical axis into three-dimensional space. The range of the curve parameter is the interval [t_{min}, t_{max}] in one-dimensional space. The coordinates $x(t)$, $y(t)$, $z(t)$ of the curve point **curve**(t) are single-valued continuous functions of the parameter t.

The boundary values t_{min} and t_{max} of the parameter domain are returned by the curve methods double GetTMin() and double GetTMax(), respectively.

A curve is called periodic if there exists $p > 0$ such that **curve**($t \pm kp$) = **curve**(t), where k is an integer.

The methods

```
bool IsPeriodic()
```

and

```
bool IsClosed()
```

of a periodic curve return *true*.

The method

```
double GetPeriod()
```

of a periodic curve or a curve that can be extended to a periodic one returns the period *p*. The parameter domain of a periodic curve always lies within a single period.

A curve is considered periodic if:

- The start and end points of the curve coincide. The following 3D curve types are not considered periodic when their start and end points coincide, for example, in the case of a degenerate curve; the IsClosed() and IsPeriodic() methods always return *false* for them:
  - *MbLine3D*;
  - *MbLineSegment3D*;
  - MbSpiral and derived classes;
  - *MbBridgeCurve3D*;
  - *MbContour3D*, if it contains only one segment of one of the types listed above.
- The derivatives at the start and end points coincide. If a curve can inherently have breaks (a contour or a polyline), the derivatives are allowed not to coincide. For a Bezier curve, the derivatives must coincide in direction but may differ in magnitude.

To check whether the start and end points of a curve coincide with a specified tolerance, use the IsTouch() method, which determines coincidence regardless of the smoothness of the closure.

The primary curve method is

```
void PointOn( double &        t,
              MbCartPoint3D & r )
```

It returns the radius-vector **r** of the curve point for a given parameter *t*.

The methods

```
void FirstDer( double &      t,
               MbVector3D & rt )
```

```
void SecondDer( double &       t,
                MbVector3D & rtt )
```

```
void ThirdDer( double &        t,
               MbVector3D & rttt )
```

return the first $\mathbf{r}_t$, second $\mathbf{r}_{tt}$, and third $\mathbf{r}_{ttt}$ derivatives, respectively, of the curve radius-vector for a given parameter *t*. These methods correct the curve parameter when it is outside the parameter domain (except for the MbLine3D line). When the curve parameter *t* is outside the interval $[t_{min}, t_{max}]$, non-periodic curves shift the parameter *t* to the nearest boundary t_{min} or t_{max}, while periodic curves add or subtract the required number of periods.

The method

```
void _PointOn( double          t,
               MbCartPoint3D & r )
```

returns the radius-vector **r** of the curve point for a given parameter *t* both within and outside the curve parameter domain. In the general case, a non-periodic curve outside its parameter domain is extended along the tangent at the boundary point. The exceptions are periodic curves, the MbArc3D arc, MbSpiral spirals, the MbCharacterCurve3D symbolic curve, and the MbTrimmedCurve3D trimmed curve within its base curve. Periodic curves outside their parameter domain are extended cyclically. The methods

```
void _FirstDer( double &      t,
                MbVector3D & rt )
```

```
void _SecondDer( double &       t,
                 MbVector3D & rtt )
```

```
void _ThirdDer( double &        t,
                MbVector3D & rttt )
```

return the first $\mathbf{r}_t$, second $\mathbf{r}_{tt}$, and third $\mathbf{r}_{ttt}$ derivatives, respectively, of the curve radius-vector for a given parameter *t* both within and outside the curve domain.

Curves override the following methods of a three-dimensional geometric object:

methods for transforming a geometric object:

```
void Move( const MbVector3D &      to,
                 MbRegTransform *  iReg = nullptr )
```

```
void Rotate( const MbAxis3D &       axis,
                   double           angle,
                   MbRegTransform * iReg = nullptr )
```

```
void Transform( const MbMatrix3D &     matr,
                      MbRegTransform *  iReg = nullptr )
```

methods for copying objects, checking equality and similarity, and making objects equal:

```
MbSpaceItem & Duplicate( MbRegDuplicate * iReg = nullptr )
```

```
bool IsSame( const MbSpaceItem & other,
                   double        accuracy = LENGTH_EPSILON )
```

```
bool IsSimilar( const MbSpaceItem & item )
```

```
bool SetEqual( const MbSpaceItem & item )
```

methods that return a type from the geometric object enumeration:

```
MbeSpaceType IsA()
```

```
MbeSpaceType Type()
```

```
MbeSpaceType Family()
```

methods for retrieving and editing the internal object data:

```
MbProperty & CreateProperty( MbePrompt name )
```

```
void GetProperties( MbProperties & properties )
```

```
void SetProperties( MbProperties & properties )
```

the method that builds a polygonal copy of a geometric object:

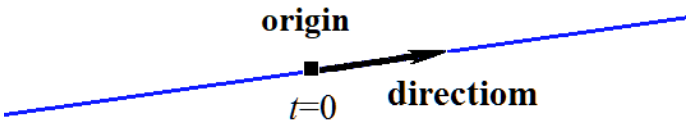
```
void CalculateMesh( const MbStepData & stepData,
                    const MbFormNote & note,
                          MbMesh &     mesh ) const;
```

All curves, except for *MbContour3D*, *MbContourOnSurface*, and *MbContourOnPlane*, usually do not have breaks. *MbContour3D*, *MbContourOnSurface*, and *MbContourOnPlane* are composite curves and can have breaks at the junctions of their constituent segments.

3.12.2 MbLine3D Straight Line

The MbLine3D class is declared in the *cur_line_3d.h* file.

The MbLine3D straight line is described by the start point *MbCartPoint3D* **origin** and the direction vector *MbVector3D* **direction**.



In the PointOn(double & *t*, MbCartPoint3D & **r**) method, the radius vector **r** of the line is described by the vector function

$$\mathbf{r}(t) = \mathbf{origin} + t \cdot \mathbf{direction}.$$

The straight line behaves as an infinite object, although in its data it has boundary values of the parameter *tmin* and *tmax*. Note that, unlike other curves, in the methods for calculating the radius vector and its derivatives the line does not adjust the parameter *t* when it goes beyond the limiting values *tmin* and *tmax*.

3.12.3 MbLineSegment3D Straight-Line Segment

The MbLineSegment3D class is declared in the *cur_line_segment_3d.h* file.

The MbLineSegment3D straight-line segment is defined by the start point *MbCartPoint3D* **point1** and the end point *MbCartPoint3D* **point2**.



In the method

```
PointOn( double & t, MbCartPoint3D & r )
```

the radius vector **r** of the segment is described by the vector function

$$\mathbf{r}(t) = (1 - t) \cdot \mathbf{point1} + t \cdot \mathbf{point2}.$$

The domain of the segment parameter lies within the range from zero to one. The parameter t_{min} = 0 corresponds to the start point **point1** of the segment, and the parameter t_{max} = 1 corresponds to the end point **point2** of the segment.

3.12.4 MbArc3D Elliptical Arc

The MbArc3D class is declared in the *cur_arc_3d.h* file.

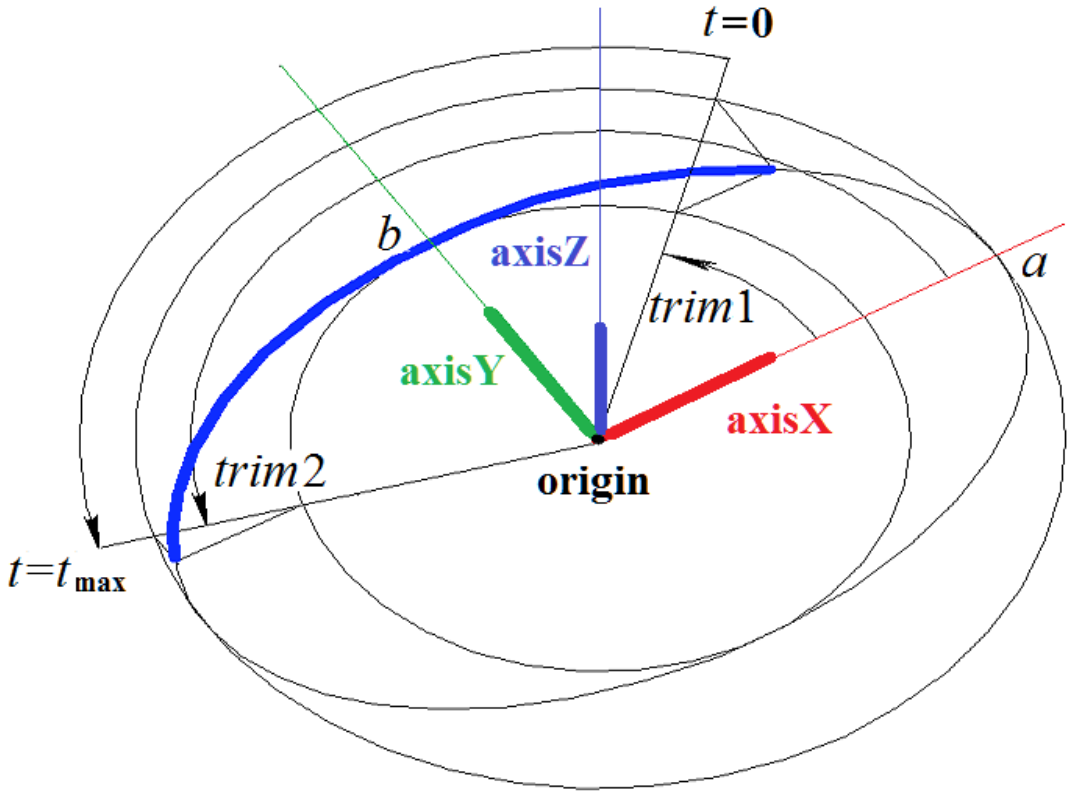
The elliptical arc is a inheritor of the MbCurve3D curve. The MbArc3D elliptical arc is defined by two radii *a* and *b* and two angles *trim1* and *trim2* specified in the local coordinate system *MbPlacement3D* **position**.

The angles *trim1* and *trim2* are measured along the arc in the direction of motion from the **position.axisX** vector to the **position.axisY** vector. The angles *trim1* and *trim2* will be called trimming parameters. The trimming parameter values equal to zero and 2π correspond to the point on the coordinate axis **position.axisX**. The curve parameter *t* takes values on the interval 0 ≤ *t* ≤ *trim2* – *trim1*. The curve may be periodic. For a periodic curve, *trim2* – *trim1* = 2π.

In the PointOn(double & *t*, MbCartPoint3D & **r**) method, the radius vector **r** of the curve is described by the vector function

$$\mathbf{r}(t) = \mathbf{position}.\mathbf{origin} + a\cos(trim1 + t)\,\mathbf{position}.\mathbf{axisX} + b\sin(trim1 + t)\,\mathbf{position}.\mathbf{axisY}.$$

The elliptical arc is shown in the figure:



The radii of the curve must be greater than zero: *a* > 0, *b* > 0. The trimming parameters must satisfy the inequalities *trim1* < *trim2*.

The local coordinate system **position** may be either right-handed or left-handed.

3.12.5 Polyline MbPolyline3D

The MbPolyline3D class is declared in the *cur_polyline3d.h* file.

The polyline is derived from the MbPolyCurve3D curve. The MbPolyline3D polyline is defined by the number of segments *segmentsCount*, the set of control points SArray<*MbCartPoint3D*> **pointList** and the curve periodicity flag *closed*.

The curve passes through the set of points **pointList**[*i*], *i* = 0,..., *segmentsCount*, at parameter values *t* = 0,..., *segmentsCount*. If *closed* = true, then the curve contains a segment connecting the last point of the set **pointList**[*segmentsCount* - 1] with the first point **pointList**[0]. The curve parameter *t* takes values on the interval 0 ≤ *t* ≤ *segmentsCount*.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector of the curve **r** is defined by a vector function

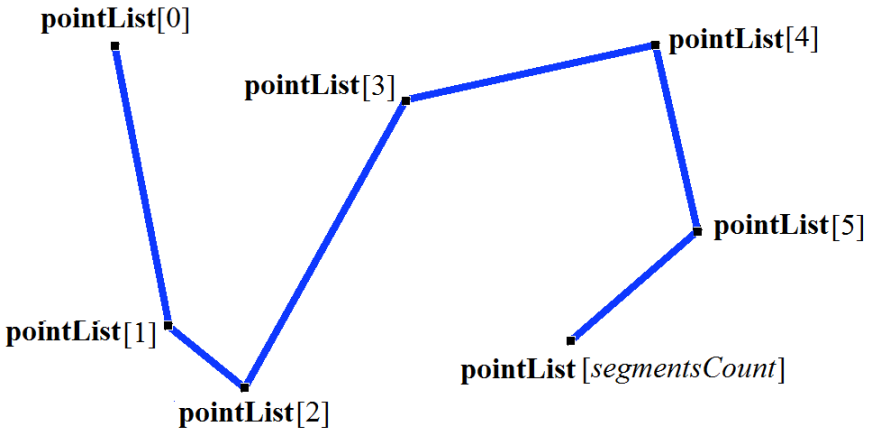
$$\mathbf{r}(t) = \mathbf{pointList}[i] \cdot (1 - w) + \mathbf{pointList}[i + 1] \cdot w,$$

where

$$w = \frac{t - t_i}{t_{i+1} - t_i},$$

and $t_i \le t \le t_{i+1}$. The polyline is the simplest curve constructed from a set of points. It consists of segments that consecutively connect the control points.

The polyline is shown in the figure:



The curve may be periodic. The period of a periodic curve is equal to *segmentsCount*. The derivatives of the curve at the control points (for integer parameter values) lose continuity in length and direction. The derivatives of the curve at the control points have a specific direction and length. The polyline has a number of useful properties: it requires a minimum of computations, and the projection of a polyline is also a polyline.

3.12.6 MbNurbs3D NURBS Curve

The MbNurbs3D class is declared in the *cur_nurbs3d.h* file.

The NURBS curve takes its name from the initials of the words NonUniform Rational B-Spline. The curve is derived from the MbPolyCurve3D curve. The curve is defined by the set of control points SArray<*MbCartPoint3D*> **pointList**, the set of weights of the control points *weights*, the knot vector *knots*, the spline order *degree*, the curve form parameter *form*, and the curve periodicity flag *closed*. The curve also has some additional data that are optional and used to speed up the execution of its methods.

The curve is constructed based on B-splines. The knot vector *knots* is a nondecreasing sequence of real numbers and defines the domain of the curve parameter and the curve shape. The curve form parameter *form* generally takes the *ncf_Unspecified* value; in special cases it stores information about the original curve from which the NURBS copy was obtained. The NURBS curve order *degree* is equal to the order of divided differences used to compute the B-splines. Let the knot vector contain *knotsCount* elements and the set of control points contain *pointsCount* elements. For a non-periodic NURBS curve, the numbers of elements in the sets are related by *knotsCount* = *pointsCount* + *degree*. For a periodic NURBS curve, the numbers of elements in the sets are related by *knotsCount* = *pointsCount* + 2 *degree* – 1.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector of the curve **r** is defined by the vector function

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

where $N_j^{degree}(t)$ are B-splines of order *degree* for the *j*-th control point **pointList**[*j*].

The NURBS curve is shown in Fig. 1:

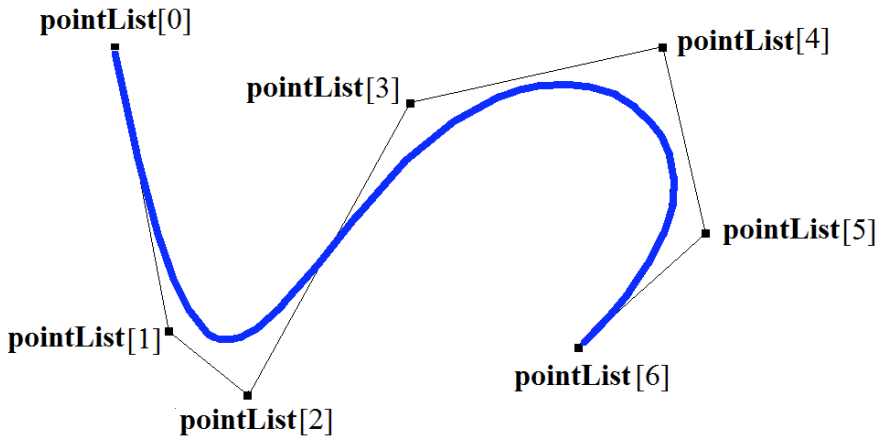


Fig. 1

The curve parameter *t* takes values on the interval *tmin* ≤ *t* ≤ *tmax*, where *tmin* = *knots* [*degree* – 1], *tmax* = *knots* [*knotsCount* – *degree*]. The curve may be periodic. A periodic NURBS curve is shown in Fig. 2:

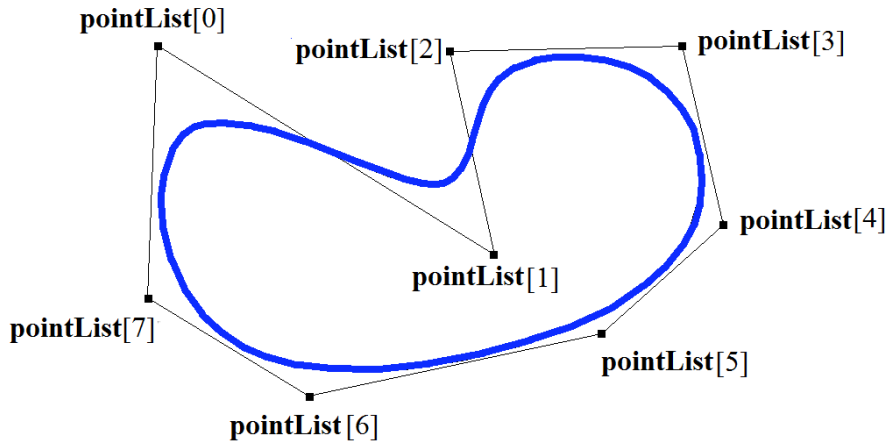


Fig. 2

The NURBS curve shape depends on the control point locations, the control point weights, and the knot vector values. In the general case, a NURBS curve does not pass through the set of points **pointList**[i], *i* = 0,..., **pointsCount** – 1. For an open NURBS curve to pass through the end control points, the first *degree* elements and the last *degree* elements of the knot vector *knots* must coincide. The larger the weight of a control point, the closer the curve passes to this point, all other conditions being equal.

Each curve can construct its own NURBS copy using the virtual method NurbsCurve(const MbNurbsParameters & *tParameters*).

3.12.7 MbHermit3D Hermite Curve

The MbHermit3D class is declared in the *cur_hermit3d.h* file.

The Hermite curve is a descendant of the MbPolyCurve3D curve. The curve is described by a set of control points SArray<MbCartPoint3D> **pointList**, a set of curve derivatives at the control points SArray <MbVector3D> **vectorList**, a set of curve parameter values at the control points *tList*, the number *splinesCount* of Hermite cubic splines, and a curve periodicity flag *closed*. The curve also has some additional data that is optional and serves to accelerate the curve methods.

The Hermite curve at the parameter value *tList[i]*, *i* = 0, 1, ..., *splinesCount*, passes through the control point **pointList**[*i*] and has the derivative **vectorList**[*i*] at that point. The curve is constructed from *splinesCount* third-degree Hermite splines that are smoothly joined together. Each cubic Hermite spline describes a curve segment between two adjacent control points. Each cubic Hermite spline is defined by two endpoints and two curve derivatives at these points.

When computing the radius vector of a point on the Hermite curve, the index *i* of the working segment (the number of the cubic Hermite spline) is first determined from the curve parameter value *t* under the condition *tList[i]* ≤ *t* ≤ *tList[i+1]*. The curve radius vector is computed as the radius vector of the found segment for its local parameter *w*, which is determined from *tList[i]* and *tList[i+1]*.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the curve radius vector **r** is described by the vector function of the found segment for its local parameter *w*:

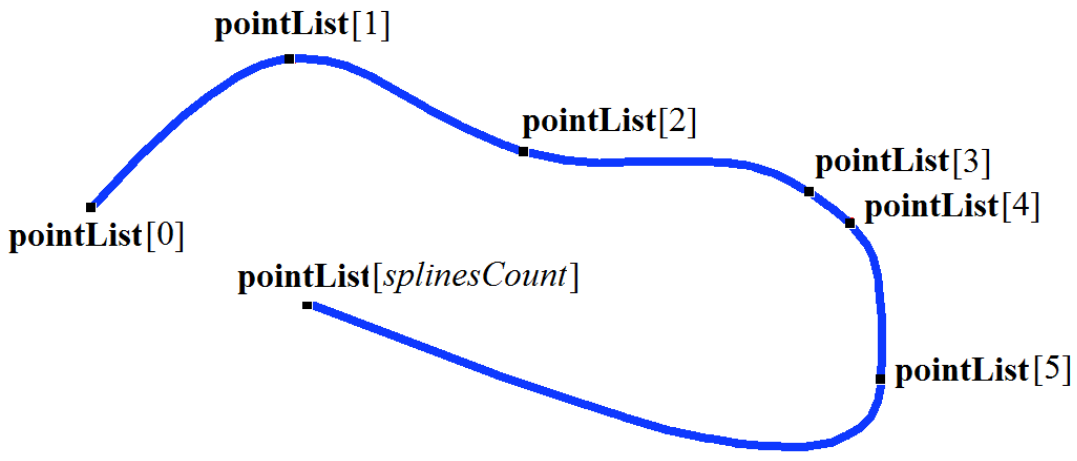
$$r(t) = (1-3w^2+2w^3)\mathbf{pointList}[i]+(3w^2-2w^3)\mathbf{pointList}[i+1]+\big((w-2w^2+w^3)\mathbf{vectorList}[i]+(-w^2+w^3)\mathbf{vectorList}[i \tag{3.5}$$

where

$$w = \frac{t - tList[i]}{tList[i+1] - tList[i]},$$

and *tList[i]* ≤ *t* ≤ *tList[i+1]*.

The Hermite curve is shown in the figure:



The curve parameter *t* takes values on the interval *tmin* ≤ *t* ≤ *tmax*, where *tmin* = *tList[0]*, *tmax* = *tList[splinesCount]*. The curve can be periodic.

The curve shape depends on the arrangement of the control points, on the curve derivatives at the control points, and on the set *tList* of parameter values at the control points. When constructing the curve from control points only, the curve parameter values at the control points

tList[i] (*i* = 0, 1, ..., *splinesCount*) vary proportionally to the distance between points, and the derivatives **vectorList**[*i*] (*i* = 1, 2, ..., *splinesCount-1*) are computed by constructing a parabola passing through three adjacent points **pointList**[*i-1*], **pointList**[*i*], **pointList**[*i+1*] at the corresponding parameter values *tList[i−1]*, *tList[i]*, *tList[i+1]* and computing the parabola derivative at the middle point.

3.12.8 MbBezier3D Bezier Composite Curve

The MbBezier3D class is declared in the *cur_bezier3d.h* file.

The composite Bézier curve is a derived class of MbPolyCurve3D. The curve is defined by a set of control points SArray<MbCartPoint3D> **pointList**, by the number **splinesCount** of Bézier curves, and by the periodicity flag of the curve **closed**. The curve also has some additional data that is not mandatory and is used to speed up the work of the curve methods.

The curve is constructed based on **splinesCount** third-degree Bézier curves that are smoothly joined to each other. Each Bézier curve is defined by four control points and passes only through two end points. The composite curve is used to construct a spline passing through given points. The given points serve as the junction points of the third-degree Bézier curves. A pair of internal control points of each third-degree Bézier curve is determined from the condition of smooth joining of the curve with the neighboring curves. For the composite curve, the number of control points is equal to 3(*splinesCount* + 1). For a non-periodic composite curve, the first **pointList** and the last control points are not used.

On each third-degree Bézier curve, the parameter of the composite curve increases by one. When calculating the radius vector of a point of the composite curve, first, by the value of the curve parameter *t*, the number of the working segment (the number of the third-degree Bézier curve) is determined as the largest integer not exceeding *t*. Let the number of the third-degree Bézier curve be *n*. Then the fractional part of the parameter *w* = *t* - *n* is determined. The radius vector of the composite curve is calculated as the radius vector of the found segment for its local parameter *w*.

In the PointOn(double & *t*, MbCartPoint3D & **r**) method, the radius vector of the curve **r** is described by the vector function of the found segment for its local parameter *w*:

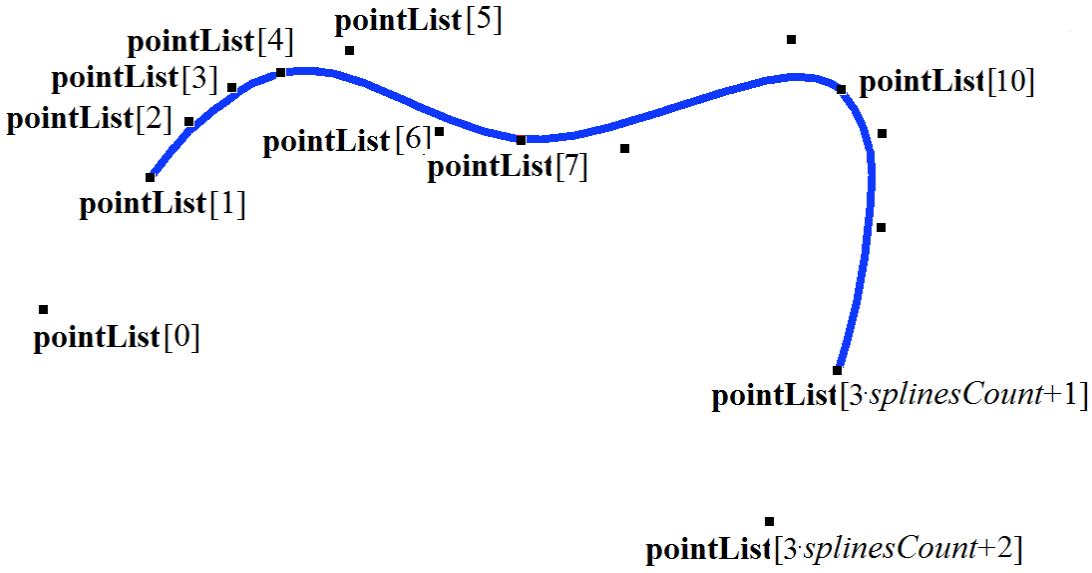
$$r(t) = \sum_{j=0}^{3} \frac{3!}{j!(3-j)!} \cdot w^j (1-w)^{3-j} \cdot \mathbf{pointList}[3n+j],$$

where w = t - n, n ≤ t ≤ n + 1, 0 ≤ w ≤ 1,

$$B_j^3(w) = \frac{3!}{j!(3-j)!} \cdot w^j (1-w)^{3-j}$$

are the third-degree Bernstein functions for the j-th control point, j = 0, 1, 2, 3, of **pointList**[*3n + j*] of the found segment with number *n*.

The composite Bézier curve is shown in the figure:



The curve parameter *t* takes values on the interval 0 ≤ *t* ≤ *splinesCount*. The curve can be periodic. The period of a periodic curve is equal to **splinesCount**.

For integer values of the parameter, the curve passes through control points; for example, at parameter *t* = *n* the curve passes through the control point **pointList**[3*n*], *n* = 0, 1, ..., **splinesCount**. The derivatives of the curve at the junction points of the third-degree Bézier curves (for integer values of the parameter) lose continuity in length.

3.12.9 Cubic Spline MbCubicSpline3D

The MbCubicSpline3D class is declared in the **cur_cubic_spline3d.h** file.

The cubic spline is a derived curve of MbPolyCurve3D. The curve is defined by an array of control points SArray<*MbCartPoint3D*> **pointList**, an array of the second derivatives of the curve at the control points SArray<*MbVector3D*> **vectorList**, an array of the curve parameter values at the control points *tList*, the maximum index value of the parameter array *splinesCount*, and the flag of curve periodicity *closed*. The curve also contains some additional data that is optional and is used to speed up the curve methods.

For the cubic spline, at the parameter value *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, the curve passes through the control point **pointList**[*i*] and has the second derivative **vectorList**[*i*] at this point. The curve is constructed so that when moving from **pointList**[*i*]to **pointList**[*i* + 1], the second derivative of the radius vector of the curve changes linearly from **vectorList**[*i*] to **vectorList**[*i* + 1].

When computing the radius vector of a point on the composite curve, first, by the curve parameter value *t* the index *i* of the active segment is determined from the condition *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1]. The radius vector of the curve is then computed from **pointList**[*i*], **pointList**[*i* + 1], **vectorList**[*i*], **vectorList**[*i* + 1] of the found segment for the local parameter *w*, which is determined by *tList*[*i*] and *tList*[*i* + 1].

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector **r** of the curve is defined by the vector function

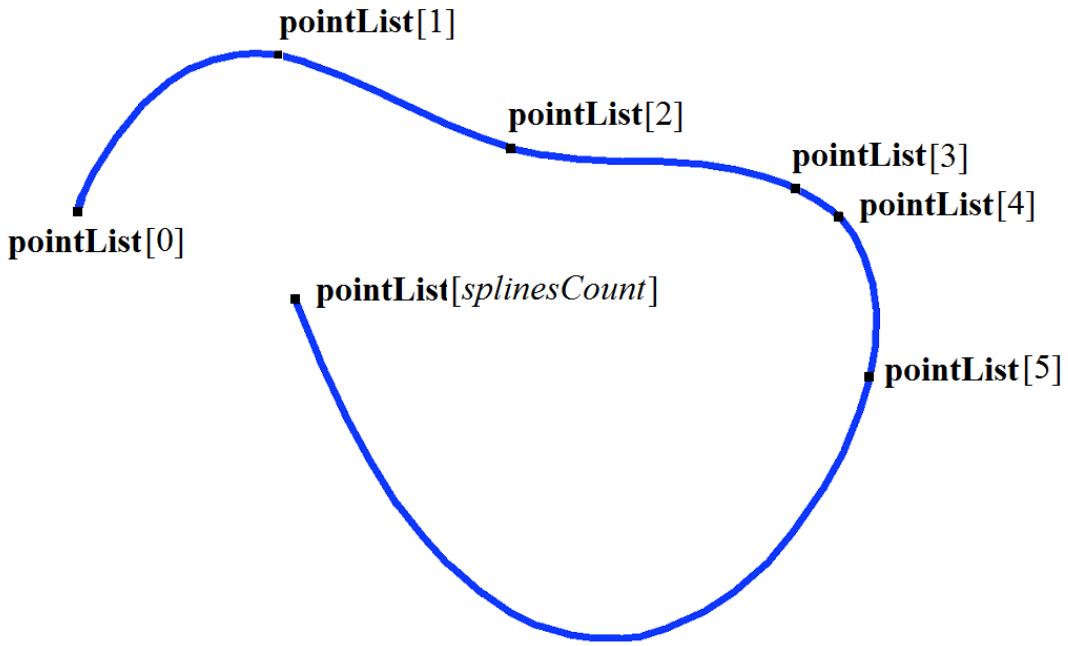
$$r(t) = (1-w)\mathbf{pointList}[i] + w\mathbf{pointList}[i+1] + \left((-2w+3w^2-w^3)\mathbf{vectorList}[i] + (-w+w^3)\mathbf{vectorList}[i+1]\right)\frac{(tList}{}$$

where

$$w = \frac{t - tList[i]}{tList[i+1] - tList[i]},$$

and *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1].

A cubic spline constructed through the same control points as the Hermite composite curve is shown in the figure:



The curve parameter *t* takes values on the interval *tmin* ≤ *t* ≤ *tmax*, where *tmin* = *tList*[0], *tmax* = *tList*[*splinesCount*]. The curve can be periodic.

The shape of the curve depends on the arrangement of the control points and on the *tList* array of parameter values at the control points. When the curve is constructed only from the control points, the curve parameter values at the control points *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, vary proportionally to the distance between the points, and the second derivatives **vectorList**[*i*], *i* = 1, 2, ..., *splinesCount* – 1 are computed by solving a system of equations.

3.12.10 Trimmed Curve MbTrimmedCurve3D

The MbTrimmedCurve3D class is declared in the *cur_trimmed_curve3d.h* file.

A trimmed curve is defined by the base curve *MbCurve3D* * **basisCurve**, the start parameter of trimming of the base curve **trim1**, the end parameter of trimming of the base curve **trim2**, and the flag of coincidence of the directions of the base curve and the trimmed curve **sense**.

The trimmed curve coincides with the base curve on the segment defined by the **trim1** and **trim2** parameters but can have the opposite direction. If **sense** = 1, then **trim1** < **trim2** and the trimmed curve coincides in direction with the base curve. If **sense** = -1, then **trim2** < **trim1** and the trimmed curve is directed opposite to the base curve.

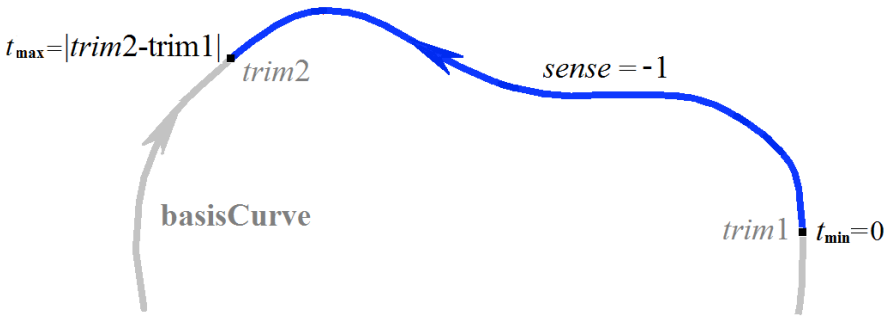
In the method

```
PointOn( double &          t,
         MbCartPoint3D   & r)
```

the radius vector of the curve **r** is described by the vector function

$$\mathbf{r}(t) = \mathbf{basisCurve}(trim1 + sense \cdot t).$$

The trimmed curve is shown in the figure:



The curve parameter *t* takes values on the segment

$$0 \le t \le sense \cdot (trim2 - trim1).$$

The trimmed curve can theoretically be used to change the direction of a curve, but it is better to use the Inverse() method.

The trimmed curve can be used to change the position of the start point of a periodic curve. For this, the base curve must be periodic and **trim2** = **trim1** + **period**. In this case the trimmed curve will also be periodic.

Another trimmed curve must not be used as a base curve for a trimmed curve; the base curve of the latter must be used with the corresponding recalculation of trimming parameters.

Each curve can build its own trimmed copy using the virtual method

```
Trimmed( double t1, double t2, int sense).
```

3.12.11 Reparametrized Curve MbReparamCurve3D

The MbReparamCurve3D class is declared in the *cur_reparam_curve3d.h* file.

A reparametrized curve is defined by the base curve MbCurve3D **basisCurve**, the start parameter *tmin*, the end parameter *tmax*, and the derivative *dt* of the base curve parameter with respect to the parameter of the reparametrized curve.

The reparametrized curve completely coincides with the base curve but has a different parameter range.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector of the curve **r** is described by the vector function

$$\mathbf{r}(t) = \mathbf{basisCurve}(v(t)),$$

where

$$v(t) = b_{\min}\frac{trim2 - t}{trim2 - trim1} + b_{\max}\frac{t - trim1}{trim2 - trim1},$$

bmin, *bmax* are the boundary values of the domain of the base curve parameter.

The curve parameter *t* takes values on the segment *tmin* ≤ *t* ≤ *tmax*.

The reparametrized curve coincides with the base curve but has a different parameter domain. A curve with a modified parameter length is used to match the parameter ranges of two curves. For example, if it is required that a line segment and an arc have the same parameter ranges, then a reparametrized curve is constructed based on one of these curves with the parameter range taken from the other curve.

Another reparametrized curve must not be used as a base curve for a reparametrized curve; the base curve of the latter must be used instead.

3.12.12 Equidistant Curve MbOffsetCurve3D

The MbOffsetCurve3D class is declared in the *cur_offset_curve3d.h* file.

An equidistant curve is defined by the base curve *MbCurve3D* **basisCurve** and the offset vector *MbVector3D* **offset**. The curve also has some additional data that is optional and is used to speed up the curve methods.

The base curve **basisCurve** is a MbSpine object, which constructs a moving local coordinate system along the curve, with the first coordinate axis directed along the tangent to the curve. The offset vector **offset** defines the displacement of the start point of the base curve to the start point of the equidistant curve. The offset vector **offset** is orthogonal to the tangent vector of the base curve at the start point. In the moving local coordinate system, the displacement of all points of the base curve to the corresponding points of the equidistant curve is equal to the offset vector and orthogonal to the tangent vector of the base curve at the current point.

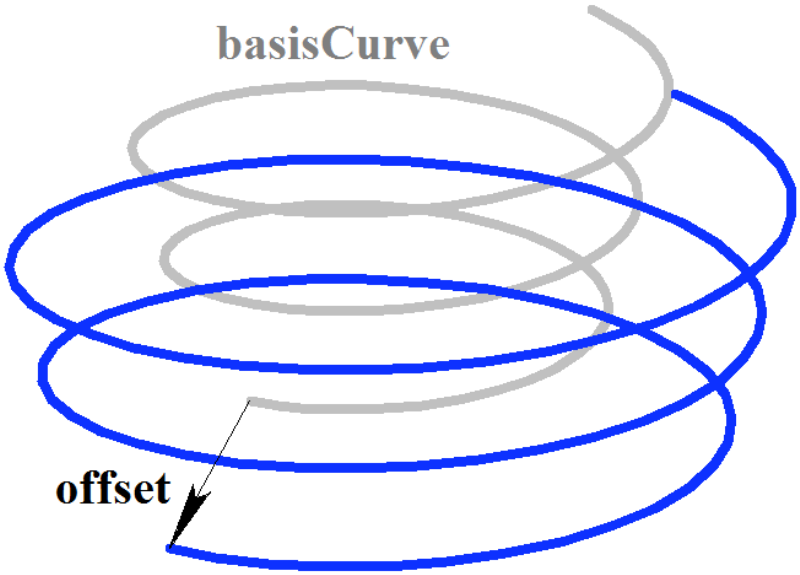
The radius vector of a point of the equidistant curve is computed as follows. For a given parameter of the base curve, a point on the spine curve and a local coordinate system with the origin at this point and the first coordinate axis directed along the tangent to the curve at this point are computed. Then the rotation matrix of the local coordinate system during its movement from the start point of the base curve to the given point is computed. A copy of the offset vector is transformed by this rotation matrix, and the computed point of the base curve is shifted by the resulting vector.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector of the curve **r** is described by the vector function

$$\mathbf{r}(t) = \mathbf{basisCurve}(t) + \mathbf{offset} \cdot \mathbf{A}(t),$$

where **A**(t) is the rotation matrix of the local coordinate system during its movement from the start point of the base curve to the given point.

The equidistant curve to a conical spiral is shown in the figure:



The parameter range of the equidistant curve coincides with the parameter range of the base curve.

Another equidistant curve must not be used as the base curve for an equidistant curve; the spine curve of the latter must be used with the corresponding recalculation of the offset vector.

3.12.13 MbCharacterCurve3D Character Curve

The MbCharacterCurve3D class is declared in the *cur_character_curve3d.h* file.

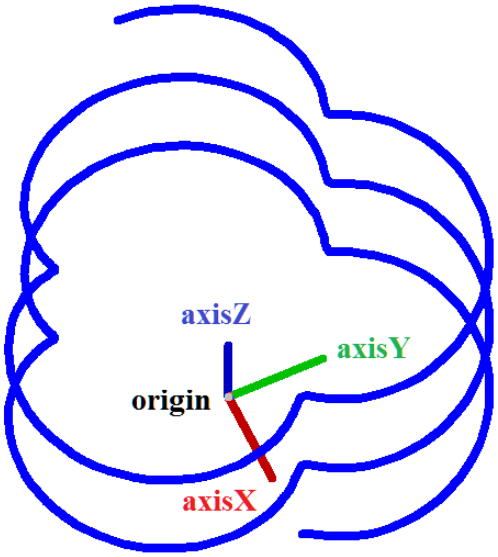
A symbolic curve is defined by the coordinate functions *xFunction*, *yFunction*, *zFunction*, the local coordinate system *MbPlacement3D* **position** of the functions, the transformation matrix **transform**, the boundary values of the parameter domain *tmin* and *tmax*, the periodicity flag of the curve *closed*, and the type of the coordinate system (Cartesian, cylindrical, spherical) *coordinateType* in which the coordinate functions are specified. The curve also has some additional data that is optional and is used to speed up the curve methods.

The coordinate functions *xFunction*(t), *yFunction*(t), *zFunction*(t) of the symbolic curve are scalar functions of the common parameter *t* and are given as symbolic expressions. For each symbolic expression, lexical analysis is performed and a tree is built that computes the value of the symbolic expression for a given parameter and the derivatives of the symbolic expression with respect to the parameter. The curve parameter *t* takes values on the segment $tmin \le t \le tmax$.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector of the curve **r** is described by the vector function

$$\mathbf{r}(t) = [xFunction(t)\ yFunction(t)\ zFunction(t)].$$

The character curve is shown in Fig. 1:



$$xFunction(t)=100*\cos(t)-20*\cos(4*t)$$

$$yFunction(t)=100*\sin(t)-20*\sin(4*t)$$

$$zFunction(t)=t*10$$

Fig. 1

The curve can be periodic. The symbolic expressions on the curve domain must represent continuous, finite, and single-valued functions.

The parser of input expressions recognizes basic mathematical operations with respect to priorities and parentheses: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), and unary minus (-). Operations are performed on variables, constants, and functions.

The parser recognizes the names of basic mathematical functions listed in Table 1:

Function name	Description
sin	Sine
cos	Cosine
tan	Tangent
sind	Sine, input argument in degrees
cosd	Cosine, input argument in degrees
tand	Tangent, input argument in degrees
asin	Arc sine
acos	Arc cosine
atan	Arc tangent
asind	Arc sine, output value in degrees
acosd	Arc cosine, output value in degrees
atand	Arc tangent, output value in degrees
sqrt	Square root
exp	Exponential
ln	Natural logarithm
lg	Decimal logarithm
deg	Function converting radians to degrees
rad	Function converting degrees to radians
abs	Absolute value

Unnamed constants (numbers) can be specified as floating-point numbers (a string of digits

separated by a dot) or as integers.

Named constants can be used in symbolic expressions and are listed in Table 2:

Constant name	Description
M_PI	π — ratio of the circle circumference to its diameter
M_PI_2	$\pi / 2$
M_PI_4	$\pi / 4$
M_SQRT2	$\sqrt{2}$
M_E	e — base of the natural logarithm
M_FI	Φ — golden ratio
M_RADDEG	$180 / \pi$ — conversion factor from radians to degrees
M_DEGRAD	$\pi / 180$ — conversion factor from degrees to radians

The name of a variable in an analytical expression can be a sequence of string and numeric literals (the first character is a string literal) that does not coincide with function names or constant names.

Constraints for creating a symbolic curve:

- Maximum variable length — 512 characters.
- Maximum expression length — 2048 characters.
- No operations of integration, differentiation, or limit evaluation.
- No simplification of mathematical expressions (for example, fraction reduction).

Due to these constraints, correct evaluation of an analytical expression can be obtained only for relatively simple mathematical expressions and functions. For example, the value of $sin(x)/x$ at zero may be computed incorrectly.

Below is a code example demonstrating how to create a symbolic three-dimensional curve.

```
MbCharacterCurve3D * CreateAnalyticalCurve3D()
{
    MbCharacterCurve3D * curve( nullptr );

    // String representations of the 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 analytical 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 ) );
```

(continues on next page)

(continued from previous page)

```
    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 );
        MbVector3D    axisY( 0.0, 1.0, 0.0 );

        MbPlacement3D place( point, axisX, axisY );     // Local coordinate system.

        MbeLocalSystemType3D cs = ls_CartesianSystem;   // Type of the local␣
↪coordinate system.

        // Creating a symbolic three-dimensional curve.
        curve = new MbCharacterCurve3D( *x, *y, *z, cs, place, t1, t2 );
    }

    return curve;
}
```

3.12.14 MbConeSpiral Conical Spiral

The MbConeSpiral class is declared in the *cur_cone_spiral.h* file.

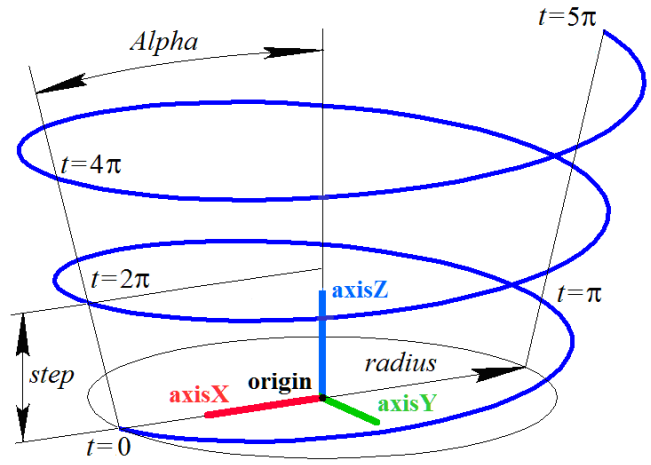
The conical spiral is derived from the MbSpiral curve. The spiral is defined by the local coordinate system *MbPlacement3D* **position**, the radius *radius*, the tangent of the cone angle *tgAlpha*, the spiral pitch divided by 2π, *step_2pi*, and two boundary parameters of the spiral *tmin* and *tmax*. The curve also has some additional data that is optional and is used to speed up the curve methods.

The spiral axis coincides with the **position.axisZ** axis of the local coordinate system. The *tgAlpha* parameter is equal to the tangent of the angle between the spiral axis and the cone generatrix of the spiral. The *tmin* and *tmax* parameters are angles measured from the **position.axisX** vector to the **position.axisY** vector. Angle values that are multiples of 2π correspond to points of the curve in the XZ plane of the local coordinate system. The curve parameter *t* takes values on the segment *tmin* ≤ *t* ≤ *tmax*.

In the PointOn(double & *t*, MbCartPoint3D & **r**) method, the radius vector of the curve **r** is described by the vector function

$$\mathbf{r}(t) = \mathbf{position}.\mathbf{origin} + (radius + t \cdot tgAlpha \cdot step\_2pi)\,(\cos(t) \cdot \mathbf{position}.\mathbf{axisX} + \sin(t) \cdot \mathbf{position}.\mathbf{axisY}) + (t \cdot step$$

The conical spiral is shown in the figure:



The curve radius must be greater than zero: *radius* > 0. For the boundary parameters, the inequality *tmin* < *tmax* must hold. The curve cannot be periodic.

The local coordinate system **position** can be either right-handed or left-handed. For a cylindrical spiral, *tgAlpha* = 0.

3.12.15 MbCurveSpiral Variable Radius Spiral

The MbCurveSpiral class is declared in the *cur_curve_spiral.h* file.

The variable radius spiral is derived from the MbSpiral curve. The spiral is defined by the local coordinate system *MbPlacement3D* **position**, the two-dimensional curve **curve** specifying the radius law, the spiral pitch *step*, and two boundary parameters of the spiral *tmin* and *tmax*. The curve also has some additional data that is optional and is used to speed up the curve methods.

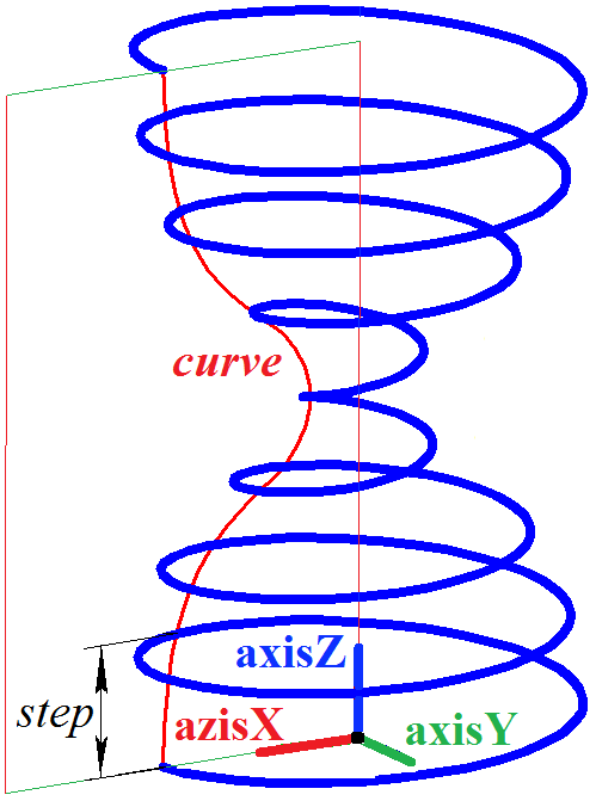
The spiral axis coincides with the **position.axisZ** axis of the local coordinate system. The two-dimensional curve **curve** lies in the XZ plane of the local coordinate system and defines the radius variation law of the spiral. The **position.axisZ** axis serves as the abscissa axis, and the **position.axisX** axis serves as the ordinate axis of the two-dimensional space of the **curve**. The origin of the two-dimensional coordinate system of the **curve** coincides with the origin of the local coordinate system **position**. The spiral radius is equal to the ordinates of points of the two-dimensional curve **curve**. The *tmin* and *tmax* parameters are angles measured from the **position.axisX** vector to the **position.axisY** vector. Angle values that are multiples of 2π correspond to points of the curve in the XZ plane of the local coordinate system.

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

$$\mathbf{r}(t) = \mathbf{position.origin} + radius(t)\,(\cos(t)\cdot\mathbf{position.axisX} + \sin(t)\cdot\mathbf{position.axisY}) + \left(t\cdot\frac{step}{2\pi}\right)\mathbf{position.axisZ},$$

where *radius*(*t*) is the local radius. The local radius *radius*(*t*) is computed as follows. For a given parameter *t*, the abscissa *t*·*step* / 2π of the required two-dimensional point of the curve is determined. Then the intersection point of the **curve** and the vertical line intersecting the abscissa axis at *t*·*step* / 2π is found. The ordinate of the two-dimensional intersection point of **curve** and the vertical line is equal to the required spiral radius *radius*(*t*). The local radius is equal to the distance from the local abscissa axis to the intersection point of the vertical line with the **curve** in the two-dimensional space of the XZ plane of the local coordinate system **position**.

The variable radius spiral is shown in the figure:



The **curve** must lie above the abscissa axis and must not intersect the abscissa axis of its two-dimensional coordinate system. The **curve** must not have vertical tangents. For the boundary parameters, the inequality *tmin* < *tmax* must hold. The curve cannot be periodic.

The local coordinate system **position** can be either right-handed or left-handed.

3.12.16 MbCrookedSpiral Spiral with Curved Planar Axis

The MbCrookedSpiral class is declared in the *cur_crooked_spiral.h* file.

A crooked spiral with a planar curved axis is a derived curve of MbSpiral. The spiral is defined by a local coordinate system *MbPlacement3D* **position**, a two-dimensional curve MbCurve **curve** that defines the spiral axis, the spiral radius *radius*, the spiral step *step*, and two boundary parameters of the spiral *tmin* and *tmax*. The curve also has some additional data that are optional and serve to speed up the curve methods.

The two-dimensional curve **curve** lies in the XZ plane of the local coordinate system **position** and defines the spiral axis. The **position.axisZ** axis serves as the abscissa axis, and the **position.axisX** axis serves as the ordinate axis of the two-dimensional space of the **curve** curve. The origin of the two-dimensional coordinate system of the **curve** curve coincides with the origin of the local coordinate system **position**. The spiral radius is constant. The *tmin* and *tmax* parameters are angles and are measured from the **position.axisX** vector to the **position.axisY** vector. Angle values that are multiples of 2π correspond to points of the curve in the XZ plane of the local coordinate system.

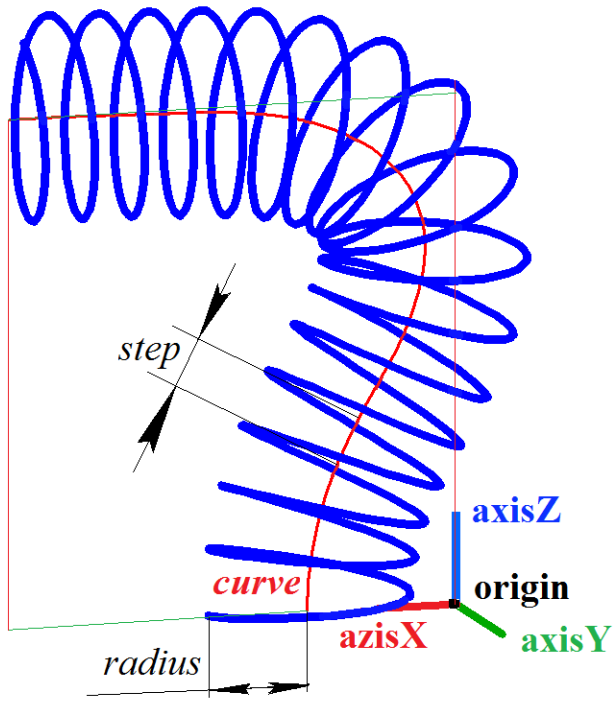
In the PointOn(double & *t*, MbCartPoint3D & **r**) method, the radius vector **r** of the curve is de-

scribed by the vector function

$$\mathbf{r}(t) = \mathbf{position.origin} + \Big((\mathbf{point}.y + \mathbf{radius} \cdot \cos(t) \cdot \mathbf{normal}.y) \cdot \mathbf{position.axisX} \Big) + \Big(\mathbf{radius} \cdot \sin(t) \cdot \mathbf{position.ax}$$
$$\Big((\mathbf{point}.x + \mathbf{radius} \cdot \cos(t) \cdot \mathbf{normal}.x) \cdot \mathbf{position.a}$$

where **point** is a point of the two-dimensional curve computed by the **curve**-> PointOn(*t*, **point**) method, and **normal** is the normal of the two-dimensional curve computed by the **curve**-> Normal(*t*, **normal**) method.

A variable-radius spiral is shown in the figure:



The minimum curvature radius of the **curve** curve must not be less than the spiral radius. For the boundary parameters, the inequality *tmin* < *tmax* must hold. The curve cannot be periodic.

The local coordinate system **position** can be either right-handed or left-handed.

3.12.17 MbBridgeCurve3D Bridge Curve

The MbBridgeCurve3D class is declared in the *cur_bridge3d.h* file.

The bridge curve is derived from the *MbCurve3D* curve and is defined by two MbCurve3D curves **curve1** and **curve2**, point parameters **param1** and **param2** of these curves, flags **sense1** and **sense2** indicating whether the directions of derivatives of the bridge curve and the curves being connected coincide, and two boundary parameters **tmin** and **tmax** of the bridge curve.

The bridge curve is constructed using the constructor

```
MbBridgeCurve3D( MbCurve3D & c1, double t1, bool s1,
                 MbCurve3D & c2, double t2, bool s2,
                 double _tmin = 0.0, double _tmax = 1.0 );
```

The bridge curve is intended for smoothly connecting two specified points of the **curve1** and **curve2** curves. It is a cubic Hermite spline constructed from two end points and curve derivatives at these points. The curve parameter **t** ranges from **tmin** to **tmax**:

$$\textbf{tmin} \leq \textbf{t} \leq \textbf{tmax}$$

In the method

```
PointOn( double & t, MbCartPoint3D & r )
```

the radius vector **r** of the curve is described by the vector function

$$r(t) = (1 - 3w^2 + 2w^3) \cdot point1 + (3w^2 + 2w^3) \cdot point2 + \Big(\big(w - 2w^2 + w^3\big) \cdot derive1 + \big(-w^2 + w^3\big) \cdot derive2 \Big) \cdot (tmax - tmin$$

where

$$w = \frac{t - tmin}{tmax - tmin}$$

is the relative parameter value; **point1** is a point of the **curve1** curve calculated using the PointOn(**param1**, **point1**) method; **point2** is a point of the **curve2** curve calculated using the PointOn(**param2**, **point2**) method; **derive1** and **derive2** are derivatives of the bridge curve at its end points. The **derive1** and **derive2** vectors are parallel to the derivatives of the curves being connected. The lengths of the **derive1** and **derive2** vectors are equal to the distance between the end points divided by **tmax** – **tmin**. Fig. 1 shows the bridge curve:

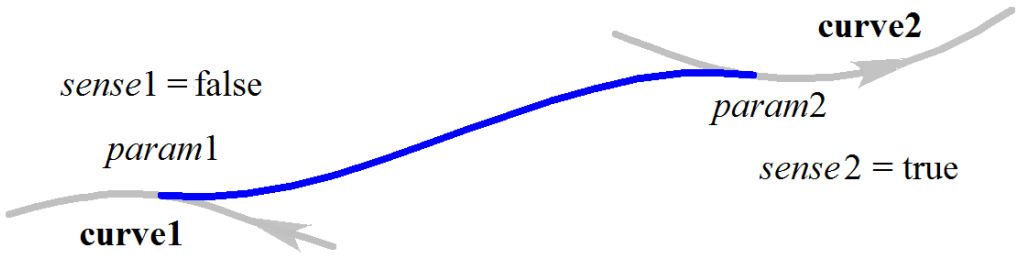


Fig. 1

The shape of the curve depends on the positions of the end points and on the directions of the curves being connected at these points. The boundary parameters must satisfy the condition **tmin** < **tmax**. The curve cannot be periodic.

3.12.18 MbContour3D Contour

The MbContour3D class is declared in the *cur_contour3d.h* file.

The MbContour3D contour is defined by a set of RPArray<MbCurve3D> **segments** of curves joined one after another and by the *closed* flag indicating whether the curve is periodic.

The contour is a composite curve. Unlike other curves, a contour may have breaks. The curves that form the contour are called segments. The segments of the contour satisfy the following conditions: the start point of each subsequent segment coincides with the end point of the previous segment. For a periodic contour, the start point of the first segment coincides with the

end point of the last segment. If the contour contains only one segment of one of the following types, it is considered non-periodic: MbLine3D, MbLineSegment3D, MbSpiral and classes derived from it, MbBridgeCurve3D. To check whether the start and end points of the contour coincide within a specified tolerance, you must use the IsTouch() method, which determines coincidence regardless of the smoothness of closure. In the general case, at the junctions of segments, the derivatives of the contour are discontinuous in length and direction.

The initial value of the contour parameter is zero: $t_{min} = 0$. The parametric length of the contour is equal to the sum of the parametric lengths of its segments:

$$t_{max} = \sum (w_{imax} - w_{imin}),$$

where w_{imin} and w_{imax} are the minimum and maximum values of the parameter of the *i*-th segment. When computing the radius vector of a point on the contour, the working segment and the value of its local parameter are first determined from the parameter value, then the radius vector of the working segment is computed and used as the radius vector of the contour.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector **r** of the curve is described by the vector function

$$\mathbf{r}(t) = \mathbf{segments}[k](w_k),$$

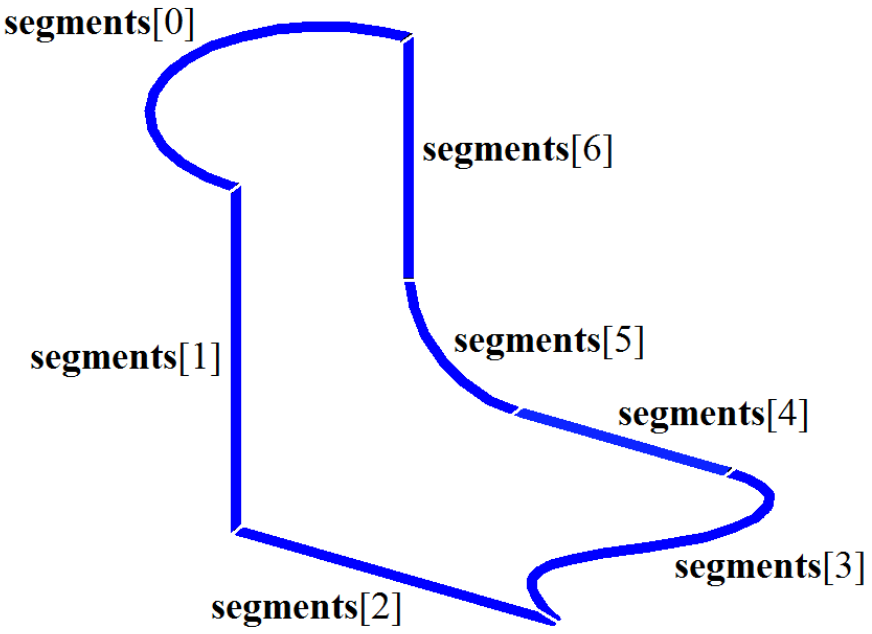
where **segments**[k](w_k) is the working contour segment with index *k*, and w_k is the parameter of the working segment, equal to

$$w_k = w_{kmin} + t - \sum_{i=0}^{k-1} (w_{imax} - w_{imin}).$$

The segment with index *k* is determined from the contour parameter *t* by the condition

$$\sum_{i=0}^{k-1} (w_{imax} - w_{imin}) \le t < \sum_{i=0}^{k} (w_{imax} - w_{imin}),$$

where w_{imin} and w_{imax} are the minimum and maximum values of the parameter of the *i*-th segment. The contour is shown in the figure:



Segments of the contour must not themselves be other contours. If a contour needs to be constructed based on other contours, the latter must be treated as a collection of their constituent curves rather than as single curves.

The MbContour3D contour is the most general type of curve.

3.12.19 MbPlaneCurve Plane Curve

The MbPlaneCurve class is declared in the *cur_plane_curve.h* file.

The MbPlaneCurve plane curve is defined by a local coordinate system *MbPlacement3D* **position** and a two-dimensional curve MbCurve **curve** lying in the XY plane of the local coordinate system.

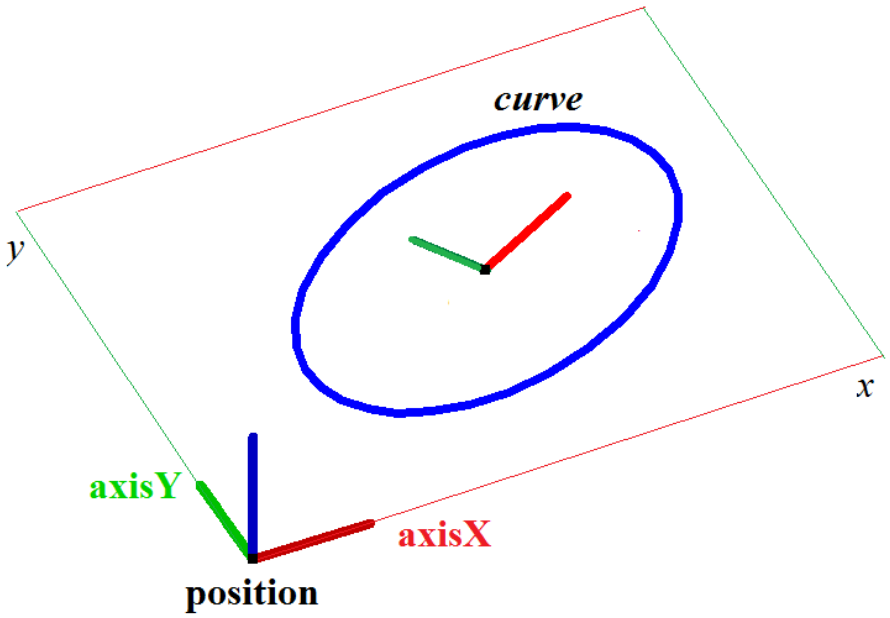
A plane curve represents a mapping of a curve in the two-dimensional space of the XY plane of the local coordinate system into three-dimensional space.

In the PointOn(double & *t*, *MbCartPoint3D* & **r**) method, the radius vector **r** of the curve is described by the vector function

$$\mathbf{r}(t) = \mathbf{position}.\mathbf{origin} + (\mathbf{point}.x \cdot \mathbf{position}.\mathbf{axisX}) + (\mathbf{point}.y \cdot \mathbf{position}.\mathbf{axisY}),$$

where **point** is a point of the two-dimensional curve computed by the **curve**-> PointOn(*t*, **point**) method.

A plane curve as part of a two-dimensional ellipse and a local coordinate system is shown in the figure:



The domain of the parameters and the periodicity of the plane curve coincide with the parameter domain and periodicity of the two-dimensional **curve**.

3.12.20 MbSurfaceCurve Curve on Surface

The MbSurfaceCurve class is declared in the *cur_surface_curve.h* file.

A curve on surface MbSurfaceCurve is defined by a surface *MbSurface* **surface**, a two-dimensional curve MbCurve **curve** in the surface parameter space, and a Boolean flag of curve periodicity **closed**. The curve also contains some additional data that is optional and is used to speed up the work of the curve methods.

A curve on a surface represents a mapping of a curve from the two-dimensional surface parameter space into three-dimensional space. The two-dimensional curve *curve* may lie outside the domain of the surface parameters. The parameter domain of the surface curve coincides with the parameter domain of the two-dimensional curve **curve**.

In the PointOn(double & *t*, MbCartPoint3D & *r*) method, the radius vector of the curve *r* is described by the vector function

r(*t*) = *surface*(*point.x*, *point.y*),

where *point* is a point of the two-dimensional curve that is computed by the *curve* → PointOn(*t*, *point*) method. The *x* and *y* coordinates of the two-dimensional point **point** serve as the *u* and *v* parameters of the surface *surface*(*u*, *v*). The curve on the surface is constructed by introducing a dependence of the surface parameters *u* and *v* on a common parameter *t*: *u* = *u*(*t*), *v* = *v*(*t*). This dependence is defined by the two-dimensional curve *curve*.

A curve on a surface is shown in Fig. 1:

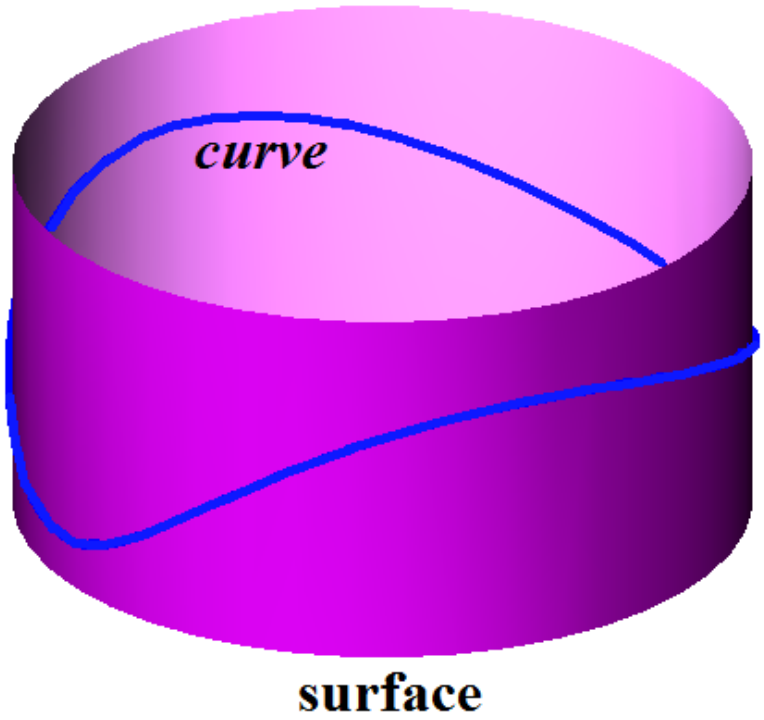


Fig. 1

The two-dimensional curve in the domain of the surface parameters is shown in Fig. 2:

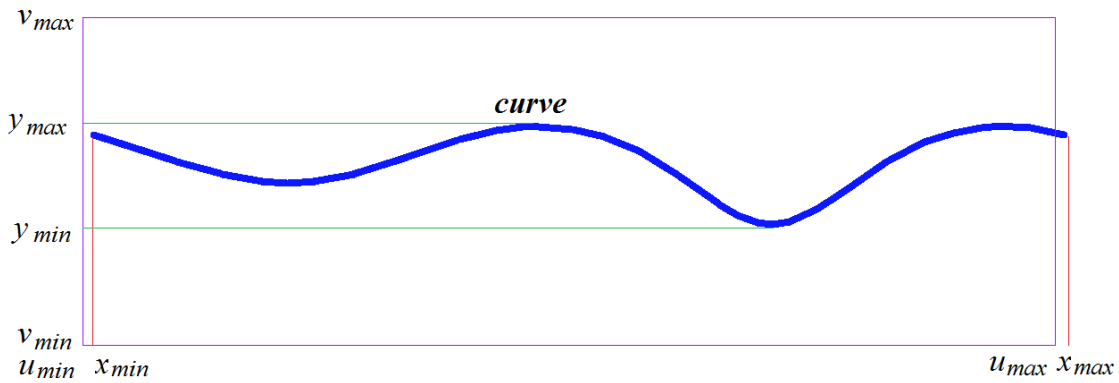


Fig. 2

The derivative of a surface curve is computed as a composite function

$$\frac{dr(t)}{\text{dt}} = \frac{\partial \text{surface}(u,v)}{\partial u}\text{derive}.x + \frac{\partial \text{surface}(u,v)}{\partial v}\text{derive}.y,$$

where *derive* is the derivative of the two-dimensional curve that is computed by the *curve* → FirstDer(*t*, *derive*) method. The derivative of the curve on the surface lies in the tangent plane of the surface constructed at the point under consideration.

A curve on a surface can be periodic if the two-dimensional curve *curve* is periodic or if the surface *surface* is periodic and the curve *curve* has matching derivatives at its ends and the end points of the curve are shifted by the corresponding period of the surface that is periodic in the first or second parameter *surface*.

The surface *surface* of a curve on a surface can be any surface except for a surface bounded by curves *MbCurveBoundedSurface*. If a curve on a surface bounded by curves needs to be constructed, then the base surface *MbCurveBoundedSurface* is used as the surface.

3.12.21 MbSilhouetteCurve Silhouette Curve

The MbSilhouetteCurve class is declared in the *cur_silhouette_curve.h* file.

A silhouette curve of a surface MbSilhouetteCurve is a derived class of a curve on surface *MbSurfaceCurve*. A silhouette curve is defined by a surface *MbSurface* **surface**, a two-dimensional curve MbCurve **curve** in the surface parameter space, a Boolean flag of curve periodicity **closed**, a Boolean flag of perspective *perspective*, a view vector **eye**, and the curve type **species**. The curve also contains some additional data that is not mandatory and is used to speed up the work of the curve methods.

A silhouette curve is a curve on the surface *surface* that separates the part of the surface visible from the observation point from the part of the surface invisible from the observation point. If *perspective = true*, then the observation point is defined by the view vector **eye**. If *perspective = false*, then the observation point is located at infinity and the view vector *eye* defines the direction from the observation point to the surface. The normal of the surface *surface* at the silhouette curve is orthogonal to the straight line connecting this point on the surface with the observation point.

In the particular case when an exact silhouette curve of the surface can be constructed, the curve type *species* takes the value *cbt_Ordinary*. For example, for a sphere the silhouette curve is a circle. In the particular case, an exact spatial curve *exactCurve* is constructed, which exactly

describes the silhouette of the surface and is used to calculate the radius vector of the silhouette curve and its derivatives.

In the general case, the curve type *species* takes the value *cbt_Specific*, the two-dimensional curve *curve* is a spline and approximates the silhouette of the surface. In the general case, a point of the silhouette curve is calculated by an iterative method that uses the two-dimensional curve *curve* as an initial approximation.

The parameter domain of the silhouette curve of the surface coincides with the parameter domain of the two-dimensional curve *curve*.

In the PointOn(double & *t*, MbCartPoint3D & *r*) method, the radius vector of the curve *r* is described by the vector function

$$\mathbf{r}(t) = \mathbf{surface}(u, v),$$

where *u*, *v* are the coordinates of a two-dimensional point whose initial approximation is calculated by the *curve* → PointOn(*t*, *point*) method, *u* = *point.x*, *v* = *point.y*. Then the parameters *u* and *v* are refined by an iterative method that uses the equation

$$\mathbf{vector} \cdot \mathbf{n}(u, v) = 0,$$

where *n(u, v)* is the surface normal that is calculated by the *surface* → Normal(*u*, *v*, *n*) method, and **vector** is the view vector (for an observation point at infinity **vector** = **eye**).

A silhouette curve of a torus surface is shown in Fig. 1:

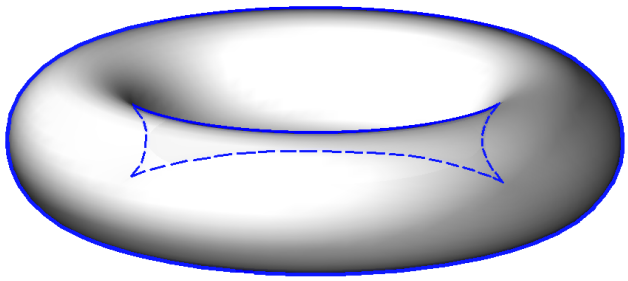


Fig. 1

A silhouette curve of the torus surface from another direction is shown in Fig. 2:

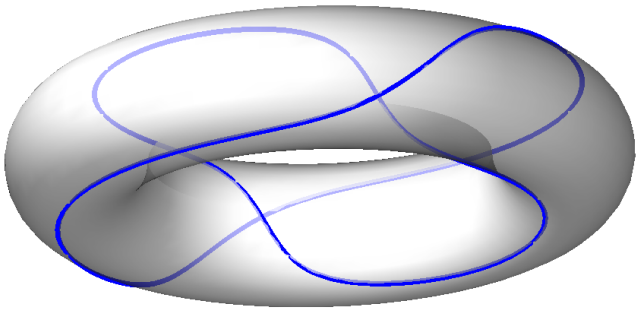


Fig. 2

When crossing the silhouette curve, the scalar product of the surface normal vector and the view vector changes sign. A silhouette curve is always either closed or starts and ends at the surface boundaries. A silhouette curve is used to construct projections of the silhouette of a curvilinear surface onto a plane.

3.12.22 MbContourOnSurface Contour on Surface

The MbContourOnSurface class is declared in the *cur_contour_on_surface.h* file.

A contour on surface MbContourOnSurface is defined by a surface *MbSurface* **surface** and a two-dimensional contour *MbContour* **contour** in the surface parameter space. The curve also contains some additional data that is not mandatory and is used to speed up the work of the curve methods.

A contour on a surface is a composite curve, therefore it may have breaks at the junction points of the segments of the two-dimensional contour. A contour on the surface represents a mapping of a contour from the two-dimensional surface parameter space into three-dimensional space. The two-dimensional contour **contour** may lie outside the domain of the surface parameters. The parameter domain of the contour on the surface coincides with the parameter domain of the two-dimensional contour **contour**.

In the PointOn(double & *t*, MbCartPoint3D & *r*) method, the radius vector of the curve *r* is described by the vector function

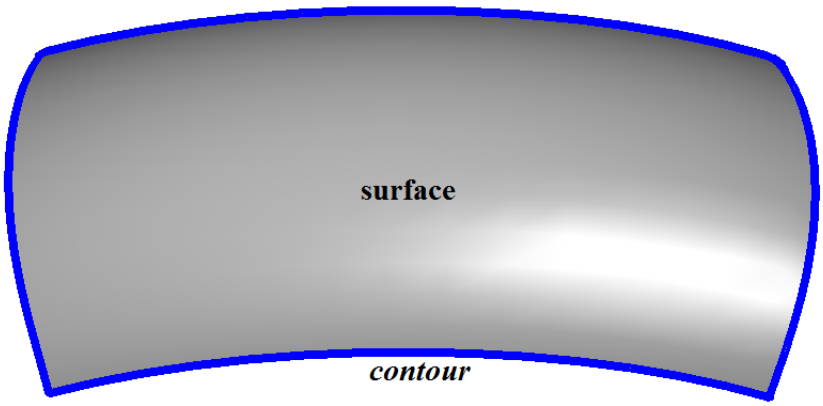
$$\mathbf{r}(t) = \mathbf{surface}(\mathbf{point}.x, \mathbf{point}.y),$$

where *point* is a point of the two-dimensional contour that is calculated by the *contour* → PointOn(*t*, *point*) method. The *x* and *y* coordinates of the two-dimensional point *point* serve as the *u* and *v* parameters of the surface *surface*(*u*, *v*). A contour on the surface is constructed by introducing a dependence of the surface parameters *u* and *v* on a common parameter *t*: *u* = *u*(*t*), *v* = *v*(*t*). This dependence is defined by the two-dimensional contour *contour*. The derivative of a contour on a surface is computed as a composite 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 the derivative of the two-dimensional contour that is calculated by the *contour* → FirstDer(*t*, *derive*) method. The derivative of the contour on the surface lies in the tangent plane of the surface constructed at the point under consideration.

A contour on the surface is shown in the figure:



A contour on a surface can be periodic if the two-dimensional contour *contour* is periodic or if the surface **surface** is periodic and the end points of the contour **contour** are shifted by the corresponding period of the surface that is periodic in the first or second parameter **surface**.

A periodic contour on a surface is usually used to describe the boundary of this surface.

The surface *surface* of a contour on a surface can be any surface except for a surface bounded by curves *MbCurveBoundedSurface*. If a contour on a surface bounded by curves needs to be constructed, then the base surface *MbCurveBoundedSurface* is used as the surface.

3.12.23 MbContourOnPlane Contour on Plane

The MbContourOnPlane class is declared in the *cur_contour_on_plane.h* file.

A contour on a plane MbContourOnPlane is a derived class of *MbContourOnSurface*. A contour on a plane is defined by a plane *MbSurface* **surface** and a two-dimensional contour *MbContour* **contour** in the plane parameter space. The curve also contains some additional data that is not mandatory and is used to speed up the work of the curve methods.

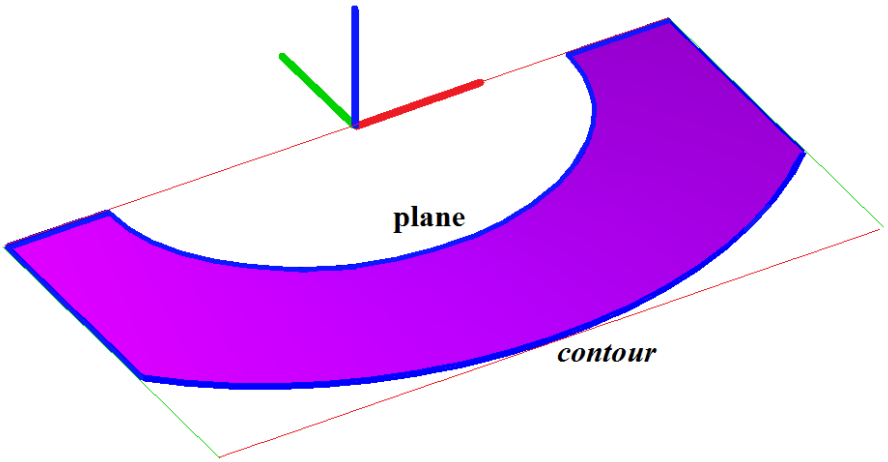
A contour on a plane is a composite curve, therefore it may have breaks at the junction points of the segments of the two-dimensional contour. A contour on a plane represents a mapping of a contour from the two-dimensional plane parameter space into three-dimensional space. The two-dimensional contour **contour** may lie outside the domain of the plane parameters. The parameter domain of the contour on a plane coincides with the parameter domain of the two-dimensional contour **contour**.

In the PointOn(double & *t*, MbCartPoint3D & *r*) method, the radius vector of the curve *r* is described by the vector function

$$\mathbf{r}(t) = \mathbf{position.origin} + (\mathbf{point}.x\,\mathbf{position.axisX}) + (\mathbf{point}.y\,\mathbf{position.axisY}),$$

where **position** is the local coordinate system of the plane **surface**, and **point** is a point of the two-dimensional contour that is calculated by the *contour* → PointOn(*t*, *point*) method.

A contour on a plane is shown in the figure:



A contour on a plane can be periodic if the two-dimensional contour **contour** is periodic or if the plane **surface** is periodic and the end points of the contour **contour** are shifted by the corresponding period of the plane that is periodic in the first or second parameter **surface**.

A periodic contour on a plane is usually used to describe the boundary of this plane.

A contour on a plane is similar to a contour on a surface but has higher computation speed.

3.12.24 MbSurfaceIntersectionCurve Surface Intersection Curve

The MbSurfaceIntersectionCurve class is declared in the *cur_surface_intersection.h* file.

A surface intersection curve MbSurfaceIntersectionCurve is defined by two curves MbSurfaceCurve **curveOne** and *MbSurfaceCurve* **curveTwo** on the intersecting surfaces, a construction parameter **buildType**, and a tolerance **tolerance**. The curve also contains some additional data that is not mandatory and is used to speed up the work of the curve methods.

The curves **curveOne**(*t*) and **curveTwo**(*t*) have the same domain of the parameter *t* and coincide in space with a certain known accuracy. The **buildType** parameter of the intersection curve specifies the curve type and carries information about the way of computing the radius vector of a point on the curve. The **buildType** parameter takes the values *cbt_Specific*, *cbt_Ordinary*, *cbt_Boundary*, *cbt_Tolerant*. Fig. 1 shows two surfaces and their intersection curve:

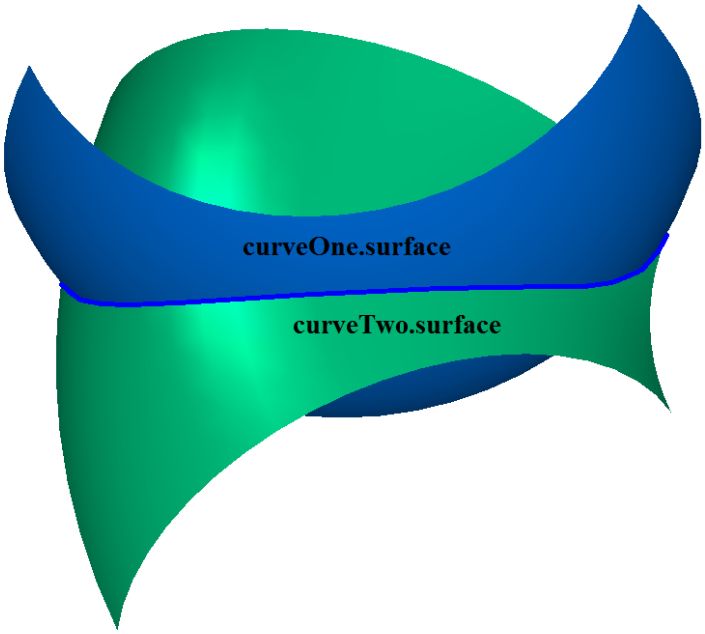


Fig. 1

Fig. 2 and 3 show the curves on the surfaces from which the intersection curve is constructed.

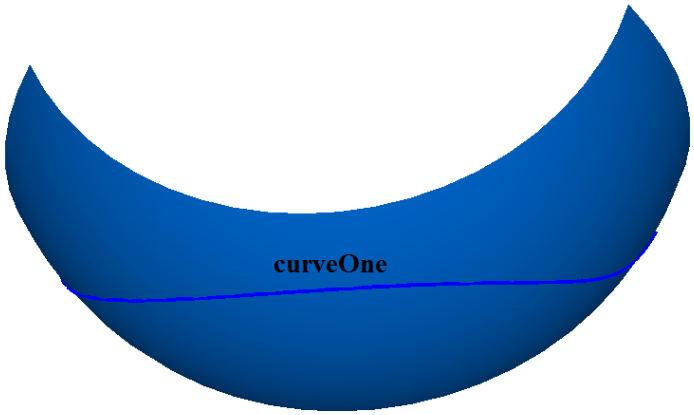


Fig. 2

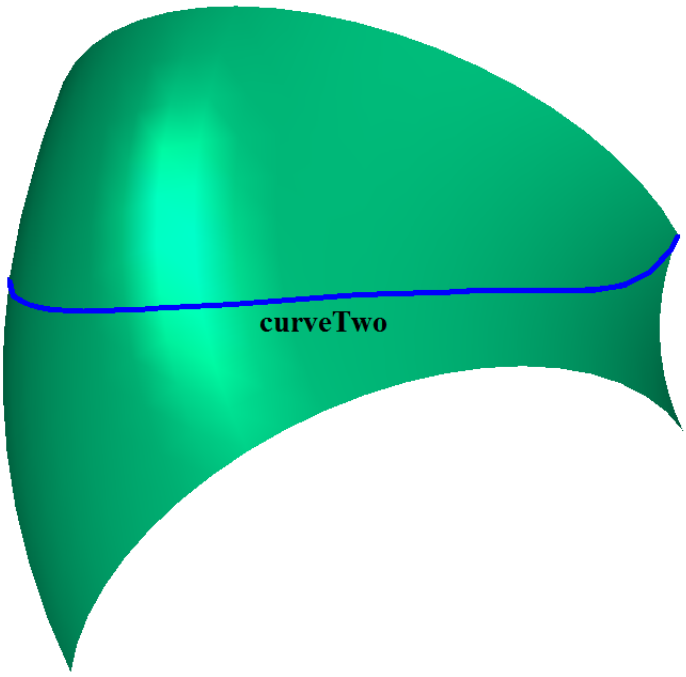


Fig. 3

In the general case, the intersection curve has the *cbt_Specific* type; the two-dimensional curves **curveOne.curve** and **curveTwo.curve** are splines and approximate the intersection of the surfaces *curveOne.surface* and *curveTwo.surface*. The splines on the surfaces have consistent control points. At the control points, the splines on the surfaces **curveOne** and **curveTwo** coincide and have the same parameter values. On the segments between the spline control points, the MbSurfaceIntersectionCurve also returns the exact value of the point radius vector. In the general case, a point of the surface intersection curve is computed by an iterative method that uses the two-dimensional curves *curveOne*.*curve* and *curveTwo*.*curve* as an initial approximation.

In particular cases, the intersection curve has the *cbt_Ordinary*, *cbt_Boundary*, or *cbt_Tolerant* type, and a point of the surface intersection curve is computed as the arithmetic mean of the radius vectors of the curves **curveOne**(*t*) and **curveTwo**(*t*).

If *buildType = cbt_Ordinary*, then the MbSurfaceIntersectionCurve curve exactly describes the intersection of the surfaces, and the curves **curveOne**(*t*) and **curveTwo**(*t*) coincide in space. An example of such a curve is the intersection curve of a plane and a cylindrical surface whose axis is orthogonal to the plane, see Fig. 4.

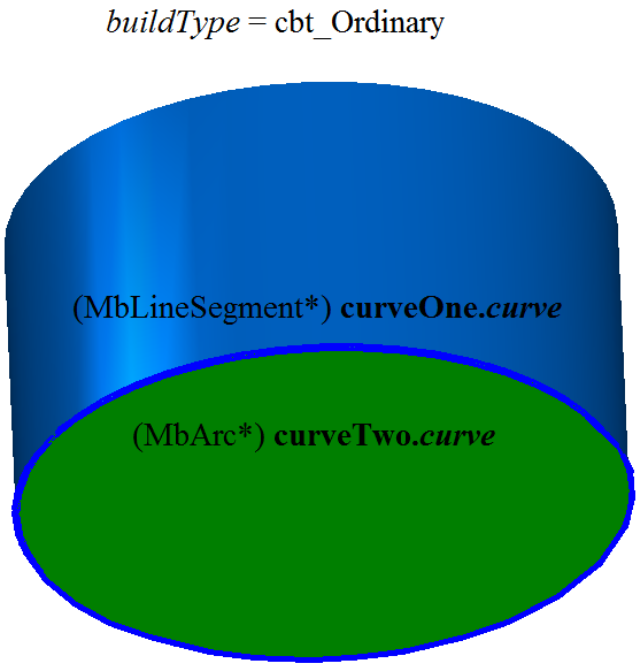


Fig. 4

On the plane, **curveOne**.*curve* is a circle, and on the cylindrical surface, **curveTwo**.*curve* is a segment whose parametric length is equal to the parametric length of the two-dimensional curve on the plane. The equality of the parametric lengths is achieved by constructing, on the basis of the segment, a reparameterized curve MbReparamCurve.

If *buildType = cbt_Boundary*, then the MbSurfaceIntersectionCurve curve describes a surface boundary, see Fig. 5. For such a curve, the equalities *curveOne*.*curve* = *curveTwo*.*curve* and *curveOne.surface* = *curveTwo.surface* hold.

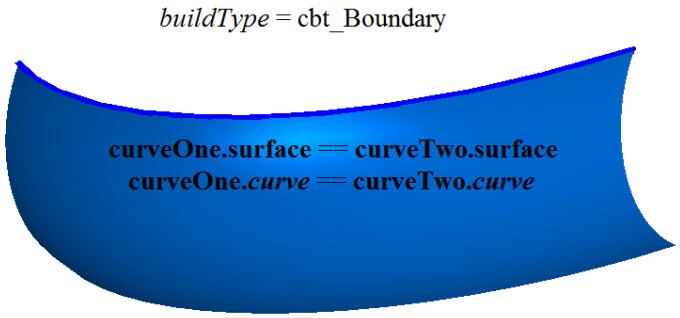


Fig. 5

If *buildType = cbt_Tolerant*, then the MbSurfaceIntersectionCurve curve describes the intersection of surfaces approximately. The curves *curveOne*(*t*) and **curveTwo**(*t*) coincide in space with accuracy **tolerance**. Such curves are constructed when another construction is impossible, for example, when it is necessary to intersect two surfaces that touch each other not exactly but with some “noise”.

In the PointOn(double & *t*, MbMatrix3D & *r*) method, the radius vector of the curve *r* is described by the vector function

$$\mathbf{r}(t) = 0.5\,(\mathbf{curveOne.surface}(u_1, v_1) + \mathbf{curveTwo.surface}(u_2, v_2))\,,$$

where u_1, v_1 are the coordinates of a two-dimensional point whose initial approximation is computed by the *curveOne*.*curve* → PointOn(*t*, *point1*) method, u_1 = *point1.x*, v_1 = *point1.y*, and u_2, v_2 are the coordinates of a two-dimensional point whose initial approximation is computed by the *curveTwo*.*curve* → PointOn(*t*, *point2*) method, u_2 = *point2.x*, v_2 = *point2.y*. Then, in the general case (*buildType = cbt_Specific*), the parameters u_1, v_1, u_2, v_2 are refined by an iterative method that uses the equations

$$\mathbf{curveOne.surface}(u_1, v_1) = \mathbf{plane}(x, y),$$

$$\mathbf{curveTwo.surface}(u_2, v_2) = \mathbf{plane}(x, y),$$

where **plane** is a plane perpendicular to the segment connecting two nearest control points of the intersection curve. The intersection curve in the general case and the control points used to construct the curves **curveOne** and **curveTwo** are shown in Fig. 6.

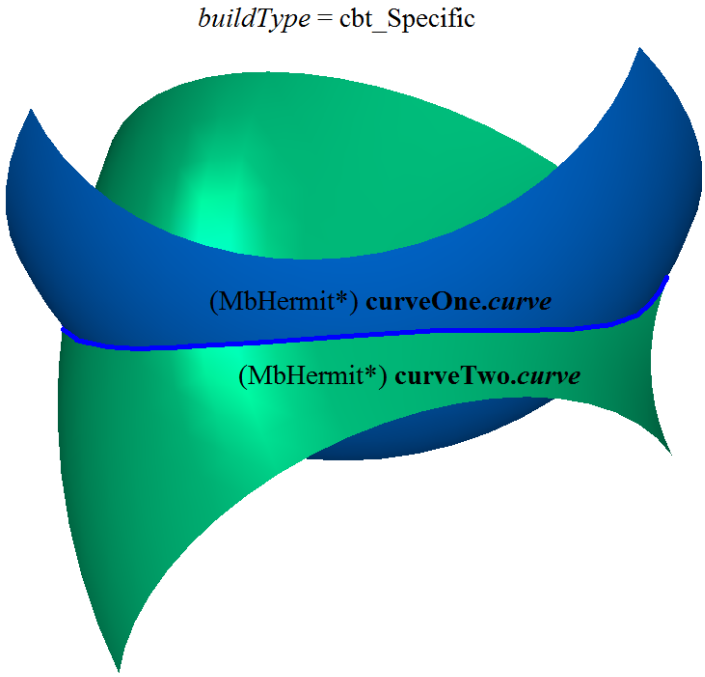


Fig. 6

The parameter domain of the intersection curve coincides with the domain of the common parameter of the curves **curveOne** and **curveTwo**. A surface intersection curve may be periodic.

Among its data, a surface intersection curve contains a three-dimensional curve **spaceCurve**, which coincides with the intersection curve with accuracy **tolerance**. The **spaceCurve** curve is used to construct planar projections of edges. The **spaceCurve** curve is an auxiliary object and is computed only when necessary.

3.13 SURFACES

Surfaces belong to the family of *MbSpaceItem* three-dimensional geometric objects. Surfaces play a major role in construction of a geometric model and 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 will use **bold** Roman font to designate vectors, radius vectors of points, and matrices in three-dimensional space.

3.13.1 MbSurface Surface

The MbSurface class is declared in the *surface.h* file.

The MbSurface surface is an inheritor of the *MbSpaceItem* class, Fig. 1.

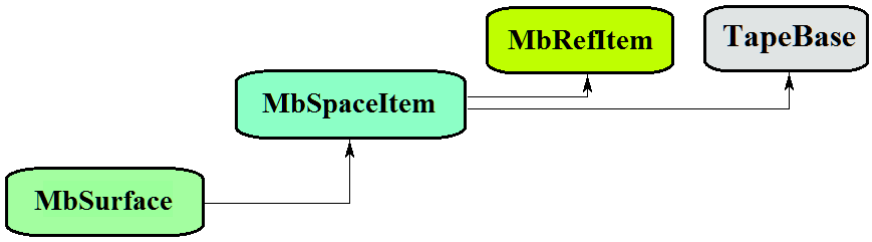


Fig. 1

The surface is an abstract class. The C3D geometric kernel implements the following surfaces, which are inheritors of the MbSurface class:

- *MbPlane* — plane,
- *MbCylinderSurface* — cylindrical surface,
- *MbConeSurface* — conical surface,
- *MbSphereSurface* — spherical surface,
- *MbTorusSurface* — toroidal surface,
- *MbExtrusionSurface* — extrusion surface,
- *MbRevolutionSurface* — revolution surface,
- *MbExpansionSurface* — plane-parallel kinematic surface,
- *MbSpiralSurface* — spiral surface,
- *MbEvolutionSurface* — swept surface,
- *MbExactionSurface* — swept surface with adaptation,
- *MbSectorSurface* — sectorial surface,
- *MbRuledSurface* — ruled surface,
- *MbLoftedSurface* — surface based on a family of curves,
- *MbElevationSurface* — surface based on a family of curves and a guiding curve,
- *MbCornerSurface* — surface based on three curves,
- *MbCoverSurface* — surface based on four curves,
- *MbCoonsPatchSurface* — bicubic Coons surface,
- *MbMeshSurface* — surface based on a network of curves,
- *MbJoinSurface* — joint surface,
- *MbSplineSurface* — NURBS surface (NonUniform Rational B-Spline surface),
- *MbOffsetSurface* — equidistant surface,
- *MbChamferSurface* — chamfer surface,
- *MbFilletSurface* — fillet surface,
- *MbChannelSurface* — fillet surface with variable radius,
- *MbCurveBoundedSurface* — surface with arbitrary borders.

The MbSurface surface is a vector function

$$\mathbf{surface}(u,v) = \begin{bmatrix} x(u,v) & y(u,v) & z(u,v) \end{bmatrix}$$

of two scalar parameters *u* and *v* taking values on a two-dimensional connected domain Ω. The surface is a continuous mapping of the two-dimensional connected domain Ω into three-dimensional space. The domain Ω is described in a two-dimensional Cartesian coordinate system. In the particular case, the domain Ω is a rectangle, and the surface parameters take values within $u_{min} \le u \le u_{max}$, $v_{min} \le v \le v_{max}$. In the general case, the domain Ω is described by two-dimensional curves. The coordinates *x*(*u*, *v*), *y*(*u*, *v*), *z*(*u*, *v*) of the surface point **surface**(*u*, *v*) are single-valued continuous functions of the parameters *u* and *v*.

The boundary values u_{min}, u_{max}, v_{min}, v_{max} of the parameter domain are returned by the surface methods double GetUMin(), double GetUMax(), double GetVMin(), double GetVMax() respectively.

A surface is called periodic with respect to the first parameter if there exists $p_u > 0$ such that **surface**(*u* ± *k*p_u, *v*) = **surface**(*u*, *v*), where *k* is an integer. A surface is called periodic with respect to the second parameter if there exists $p_v > 0$ such that **surface**(*u*, *v* ± *k*p_v) = **surface**(*u*, *v*), where *k* is an integer. The domain of a periodic surface parameter lies within one period for the corresponding parameter.

The method bool IsUClosed() of a surface periodic with respect to the first parameter returns *true*.

The method bool IsVClosed() of a surface periodic with respect to the second parameter returns *true*.

The method double GetUPeriod() of a surface periodic with respect to the first parameter, or of a surface that can be extended to a periodic one, returns the period p_u. The method double GetVPeriod() of a surface periodic with respect to the second parameter, or of a surface that can be extended to a periodic one, returns the period p_v. The domain of a periodic surface parameter always lies within one period.

Let us introduce the notation

$$s_u = \frac{\partial \mathbf{surface}(u,v)}{\partial u};$$

$$s_v = \frac{\partial \mathbf{surface}(u,v)}{\partial v};$$

$$s_{uu} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial u^2};$$

$$s_{vv} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial v^2};$$

$$s_{uv} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial u \partial v} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial v \partial u};$$

$$s_{uuu} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial u^3};$$

$$s_{uuv} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial u \partial u \partial v};$$

$$s_{uvv} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial u \partial v \partial v};$$

$$s_{vvv} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial v^3}$$

for the partial derivatives of the surface with respect to its parameters.

The primary method of the surface is

void PointOn(double & *u*, double & *v*, *MbCartPoint3D* & **s**).

It returns the radius vector **s**(*u*, *v*) of the surface point for the given parameters *u* and *v*.

The methods

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* & $\mathbf{s}_{vvv}$)

return the derivatives $\mathbf{s}_u$, $\mathbf{s}_v$, $\mathbf{s}_{uu}$, $\mathbf{s}_{uv}$, $\mathbf{s}_{vv}$, $\mathbf{s}_{uuu}$, $\mathbf{s}_{uuv}$, $\mathbf{s}_{uvv}$, $\mathbf{s}_{vvv}$ of the surface radius vector for the given parameters *u* and *v*, respectively. The listed methods adjust the surface parameters when they go beyond the domain (the exception is the plane *MbPlane*). When the parameter *u* goes beyond the interval $[u_{min}, u_{max}]$, surfaces that are non-periodic with respect to the first parameter shift the parameter *u* to the nearest boundary u_{min} or u_{max}, while surfaces that are periodic with respect to the first parameter add or subtract the required number of periods. When the parameter *v* goes beyond the interval $[v_{min}, v_{max}]$, surfaces that are non-periodic with respect to the second parameter shift the parameter *v* to the nearest boundary v_{min} or v_{max}, while surfaces that are periodic with respect to the second parameter add or subtract the required number of periods.

The method

void _PointOn(double *u*, double *v*, MbCartPoint3D & **s**)

returns the radius vector **s**(*u*, *v*) of the surface point for the given parameters *u* and *v* both within the surface parameter domain and beyond it. Each non-periodic surface is extended beyond the parameter domain according to its own law. In the absence of such a law (in the general case), a non-periodic surface is extended beyond the parameter domain along the tangent to the corresponding boundary point of the surface.

The methods

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* & $\mathbf{s}_{vvv}$)

return the derivatives $\mathbf{s}_u$, $\mathbf{s}_v$, $\mathbf{s}_{uu}$, $\mathbf{s}_{uv}$, $\mathbf{s}_{vv}$, $\mathbf{s}_{uuu}$, $\mathbf{s}_{uuv}$, $\mathbf{s}_{uvv}$, $\mathbf{s}_{vvv}$ of the surface radius vector for the given parameters *u* and *v* both within the surface domain and beyond it.

Surfaces override the following methods of a three-dimensional geometric object:

methods for geometric object transformation,

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 for copying, checking for coincidence, checking for the possibility of making objects coincident, and making objects coincident,

MbSpaceItem & Duplicate(MbRegDuplicate * iReg = NULL),

bool IsSame(const *MbSpaceItem* & **item**),

bool IsSimilar(const *MbSpaceItem* & **item**),

bool SetEqual(const *MbSpaceItem* & **item**),

methods returning a type from the geometric object enumeration,

MbeSpaceType IsA(),

MbeSpaceType Type(),

MbeSpaceType Family(),

methods for retrieving and editing the internal data of an object,

MbProperty & CreateProperty(MbePrompt name),

void GetProperties(MbProperties & *properties*),

void SetProperties(MbProperties & *properties*),

a method that fills the polygonal copy of a geometric object,

void CalculateWire(double sag, *MbMesh* & **mesh**).

Most surfaces have a rectangular parameter domain. Among all surfaces, *MbCurveBoundedSurface* stands out as a universal surface. *MbCurveBoundedSurface* has curvilinear boundaries and may have arbitrary cutouts inside. MbCurveBoundedSurface is built on the basis of any surface with a rectangular parameter domain.

3.13.2 MbPlane Plane

The MbPlane class is declared in the *surf_plane.h* file.

The MbPlane plane belongs to the group of elementary surfaces MbElementarySurface. The plane is defined by the XY plane of the local coordinate system MbPlacement3D **position**. The first parameter of the plane is measured along the vector **position.axisX**, and the second parameter of the plane is measured along the vector **position.axisY**. The domain of the plane is described by the boundary parameter values *umin*, *umax* and *vmin*, *vmax*, Fig. 1:

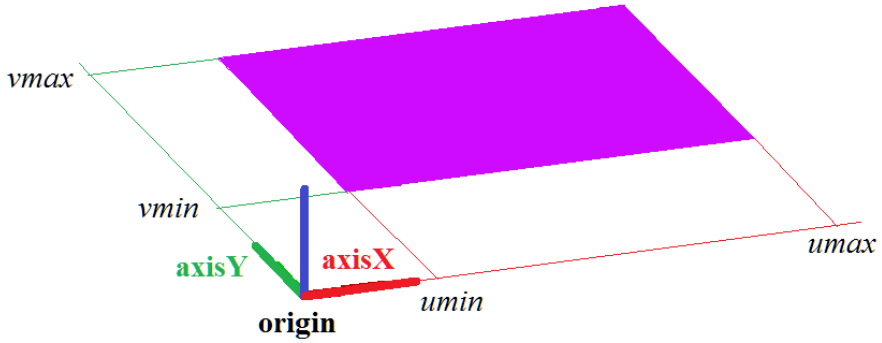


Fig. 1

In the method PointOn(double *u*, double *v*, MbCartPoint3D & **s**), the radius vector **s** of the plane is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + u \cdot \mathbf{position.axisX} + v \cdot \mathbf{position.axisY}.$$

The plane behaves as an infinite object, although its data contains the boundary parameter values *umin*, *umax* and *vmin*, *vmax*. Note that, unlike other surfaces, the plane does not adjust the parameters *u* and *v* when they go beyond the domain defined by the values *umin*, *umax* and *vmin*, *vmax* in the methods for computing the radius vector and its derivatives.

The local coordinate system **position** can be either right-handed or left-handed. If the local coordinate system is left-handed, the normal of the plane is directed opposite to the direction of the vector **position.axisZ**.

3.13.3 MbCylinderSurface Cylindrical Surface

The MbCylinderSurface class is declared in the *surf_cylinder_surface.h* file.

The MbCylinderSurface cylindrical surface belongs to the group of elementary surfaces MbElementarySurface. The cylindrical surface is described by the radius *radius* and the height *height*, defined in the local coordinate system *MbPlacement3D* **position**.

The first surface parameter is measured along the arc from the vector **position.axisX** in the direction of the vector **position.axisY**. The first surface parameter *u* takes values on the interval: *umin* ≤ *u* ≤ *umax*. The values *u* = 0 and *u* = 2π correspond to a point on the XZ plane. The surface can be periodic with respect to the first parameter. For a periodic surface, *umax* - *umin* = 2π; for a non-periodic surface, *umax* - *umin* < 2π.

The second surface parameter is measured along the straight line in the direction of the vector **position.axisZ**. The second surface parameter *v* takes values on the interval: *vmin* ≤ *v* ≤ *vmax*. The value *v* = 0 corresponds to the origin of the local coordinate system, and the value *v* = 1 corresponds to the point at a distance *height* from the XY plane of the local coordinate system of the surface.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + height \cdot v\,\mathbf{position.axisZ} \tag{3.6}$$

Fig. 1 shows the cylindrical surface:

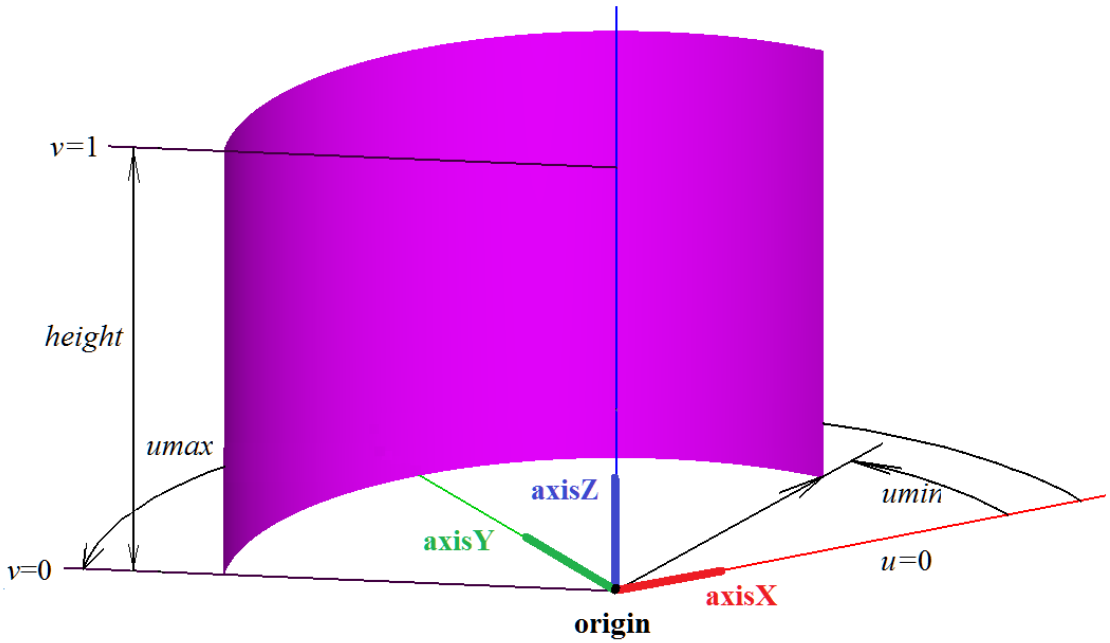


Fig. 1

The radius and height must be greater than zero: *radius* > 0, *height* > 0. The following inequalities must hold for the boundary parameters of the surface: *umin* < *umax*, *vmin* < *vmax*.

The local coordinate system **position** can be either right-handed or left-handed. If the local coordinate system is right-handed, the normal is directed towards the convex side of the surface (away from the surface axis); if the local coordinate system is left-handed, the normal is directed towards the concave side of the surface (towards the surface axis).

3.13.4 MbConeSurface Conical Surface

The MbConeSurface class is declared in the *surf_cone_surface.h* file.

The MbConeSurface conical surface belongs to the group of elementary surfaces MbElementarySurface. The conical surface is described by the radius *radius*, the height *height*, and the cone angle *angle*, defined in the local coordinate system *MbPlacement3D* **position**.

The first surface parameter is measured along the arc from the vector **position.axisX** in the direction of the vector **position.axisY**. The first surface parameter *u* takes values on the interval: *umin* ≤ *u* ≤ *umax*. The values *u* = 0 and *u* = 2π correspond to a point on the XZ plane. The surface can be periodic with respect to the first parameter. For a periodic surface, *umax* - *umin* = 2π; for a non-periodic surface, *umax* - *umin* < 2π.

The second surface parameter is measured along the straight line in the direction of the vector **position.axisZ**. The second surface parameter *v* takes values on the interval: *vmin* ≤ *v* ≤ *vmax*. The value *v* = 0 corresponds to the origin of the local coordinate system, and the value *v* = 1 corresponds to the point at a distance *height* from the XY plane of the local coordinate system of the surface.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (radius + height \cdot v \cdot \tan(angle))\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + hei \tag{3.7}$$

Fig. 1 shows the conical surface:

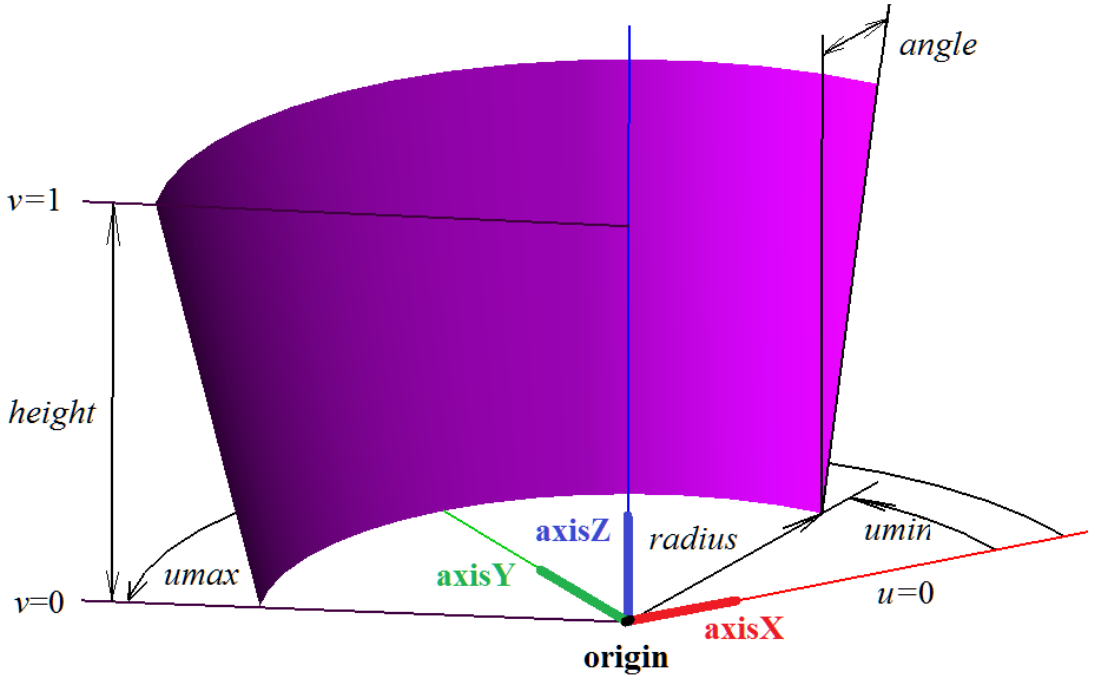


Fig. 1

The radius and height must be greater than zero, and the absolute value of the angle must not exceed π/2: *radius* > 0, *height* > 0, -π/2 < *angle* < π/2. When *angle* = 0, the conical surface is equivalent to a cylindrical surface. The following inequalities must hold for the boundary parameters of the surface: *umin* < *umax*, *vmin* < *vmax*. The pole of the surface corresponds to the value of the second parameter *v* = -*radius* / (*height* · tg(*angle*)). The boundary parameters *vmax* and *vmin* take values such that the surface is located on one side of the pole.

The local coordinate system **position** can be either right-handed or left-handed. If the local coordinate system is right-handed, the normal is directed towards the convex side of the surface (away from the surface axis); if the local coordinate system is left-handed, the normal is directed towards the concave side of the surface (towards the surface axis).

3.13.5 MbSphereSurface Spherical Surface

The MbSphereSurface class is declared in the *surf_sphere_surface.h* file.

The MbSphereSurface sphere belongs to the group of elementary surfaces MbElementary-Surface. The sphere is described by the radius *radius*, defined in the local coordinate system *MbPlacement3D* **position**.

The first surface parameter is measured along the arc from the vector **position.axisX** in the direction of the vector **position.axisY**. The first surface parameter *u* takes values on the interval: *umin* ≤ *u* ≤ *umax*. The values *u* = 0 and *u* = 2π correspond to a point on the XZ plane. The surface can be periodic with respect to the first parameter. For a periodic surface, *umax* - *umin* = 2π; for a non-periodic surface, *umax* - *umin* < 2π.

The second surface parameter is measured along the arc from the XY plane of the local coordinate system of the surface in the direction of the vector **position.axisZ**. The second surface

parameter *v* takes values on the interval: *vmin* ≤ *v* ≤ *vmax*. The value *v* = 0 corresponds to a point on the XY plane of the local coordinate system of the surface. The surface is non-periodic with respect to the second parameter.

In the PointOn(double *u*, double *v*, MbCartPoint3D & **s**) method, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + radius \cdot \sin(v)\,\mathbf{position.axisZ} \tag{3.8}$$

Fig. 1 shows the sphere:

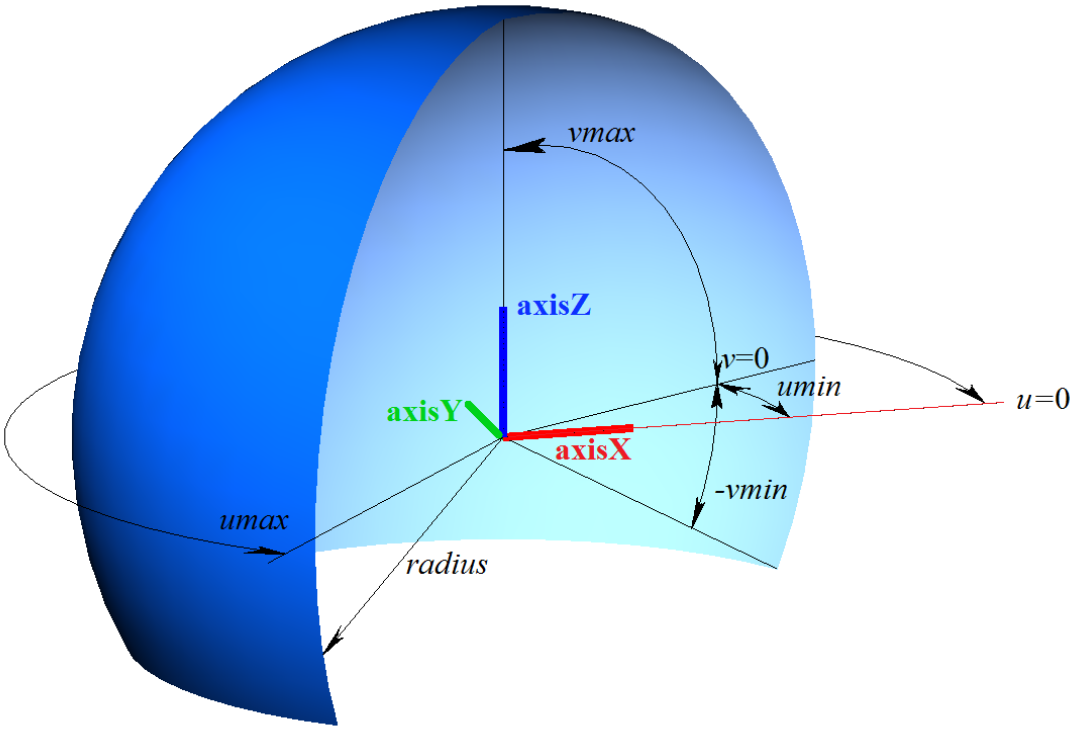


Fig. 1

The sphere radius must be greater than zero: *radius* > 0. The sphere has poles at parameter values *v* = π/2 and *v* = -π/2. The following inequalities must hold for the boundary parameters of the surface: *umin* < *umax*, *vmin* < *vmax*, *vmax* ≤ π/2, *vmin* ≥ -π/2.

The local coordinate system **position** can be either right-handed or left-handed. If the local coordinate system is right-handed, the normal is directed outward from the sphere; if the local coordinate system is left-handed, the normal is directed inward into the sphere.

3.13.6 MbTorusSurface Toroidal Surface

The MbTorusSurface class is declared in the *surf_torus_surface.h* file.

The MbTorusSurface toroidal surface belongs to the group of elementary surfaces MbElementarySurface. The toroidal surface is described by the major radius *majorRadius* and the tube radius *minorRadius*, defined in the local coordinate system *MbPlacement3D* **position**.

The first surface parameter is measured along the arc from the vector **position.axisX** in the direction of the vector **position.axisY**. The first surface parameter *u* takes values on the interval:

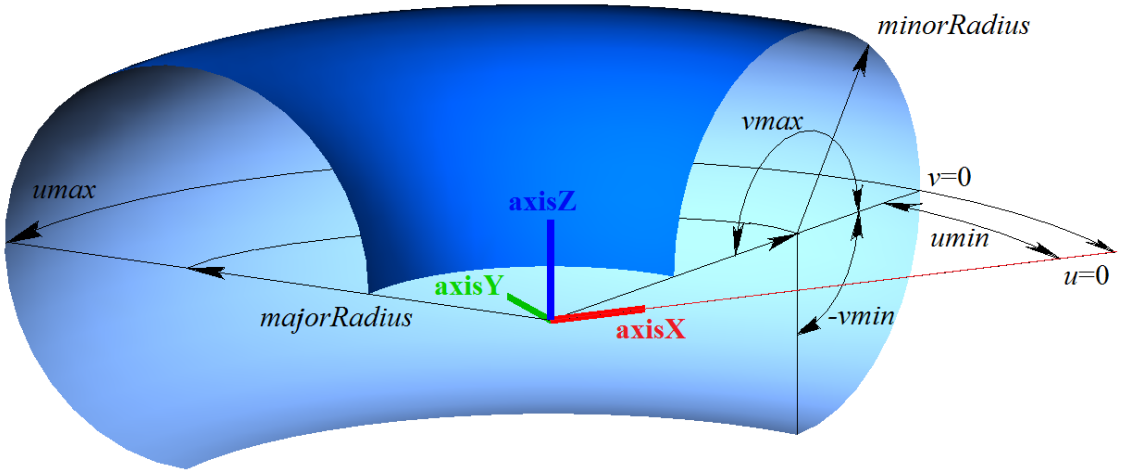
umin ≤ *u* ≤ *umax*. The values *u* = 0 and *u* = 2π correspond to a point on the XZ plane. The surface can be periodic with respect to the first parameter. For a periodic surface, *umax* - *umin* = 2π; for a non-periodic surface, *umax* - *umin* < 2π.

The second surface parameter is measured along the arc from the XY plane of the local coordinate system of the surface in the direction of the vector **position.axisZ**. The second surface parameter *v* takes values on the interval: *vmin* ≤ *v* ≤ *vmax*. The values *v* = 0 and *v* = 2π correspond to a point on the XY plane of the local coordinate system of the surface. The surface can be periodic with respect to the second parameter when *majorRadius* > *minorRadius*. For a periodic surface, *vmax* - *vmin* = 2π; for a non-periodic surface, *vmax* - *vmin* < 2π.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (majorRadius + minorRadius \cdot \cos(v))\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY} \quad (3.9)$$

Figure shows the toroidal surface:



The tube radius must be greater than zero: *minorRadius* > 0. The major radius must be no less than the tube radius taken with the opposite sign: *majorRadius* > -*minorRadius*. If *majorRadius* < *minorRadius*, the surface has a pole at the parameter values *v* = arccos(*majorRadius* / *minorRadius*) and *v* = 2π - arccos(*majorRadius* / *minorRadius*). The following inequalities must hold for the boundary parameters of the surface: *umin* < *umax*, *vmin* < *vmax*.

The local coordinate system **position** can be either right-handed or left-handed. If the local coordinate system is right-handed, the normal is directed away from the tube of the surface; if the local coordinate system is left-handed, the normal is directed inward into the tube of the surface.

3.13.7 MbExtrusionSurface Extrusion Surface

The MbExtrusionSurface class is declared in the *surf_extrusion_surface.h* file.

The MbExtrusionSurface extrusion surface belongs to the group of swept surfaces MbSweptSurface. The extrusion surface is a special case of a swept surface with a rectilinear spine curve. The extrusion surface is described by the generatrix curve *MbCurve3D* **curve**, the extrusion direction vector MbVector3D **direction**, and the extrusion length *distance*.

The first surface parameter *u* coincides with the parameter of the generatrix curve. The first surface parameter takes values on the interval *umin* ≤ *u* ≤ *umax*, which corresponds to the

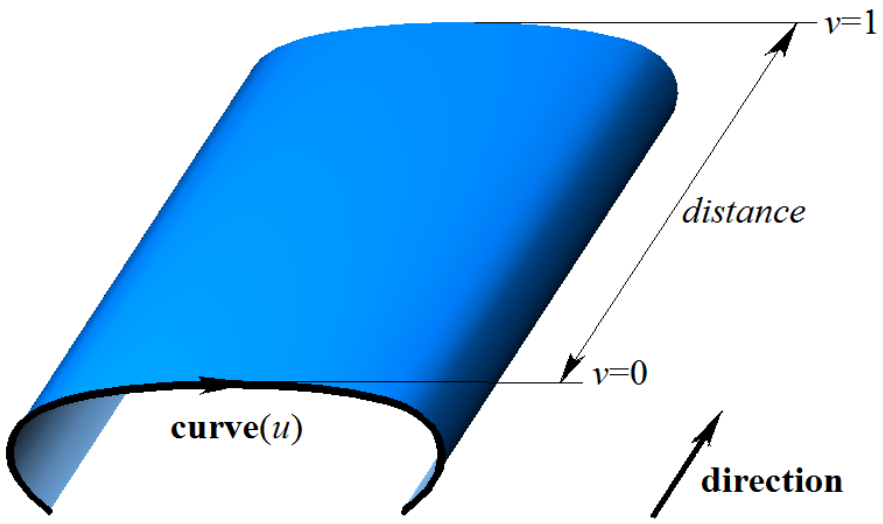
domain of the generatrix curve. The surface can be periodic with respect to the first parameter if the generatrix curve is periodic.

The second surface parameter *v* takes values on the interval: *vmin* ≤ *v* ≤ *vmax*. The value *v* = 0 corresponds to a point on the generatrix curve; the value *v* = 1 corresponds to a point of the generatrix curve displaced by the vector **direction** * *distance*. The surface cannot be periodic with respect to the second parameter.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{curve}(u) + \mathbf{direction} \cdot distance \cdot v \tag{3.10}$$

Figure shows the extrusion surface:



The following inequality must hold for the boundary values of the second surface parameter: *vmin* < *vmax*.

3.13.8 MbRevolutionSurface Revolution Surface

The MbRevolutionSurface class is declared in the *surf_revolution_surface.h* file.

The MbRevolutionSurface revolution surface belongs to the group of swept surfaces MbSweptSurface. The revolution surface is a special case of a swept surface with a spine curve in the form of a circle or its arc. The revolution surface is described by the generatrix curve *MbCurve3D* **curve**, the local coordinate system *MbPlacement3D* **position**, whose vector **position.axisZ** is the axis of revolution, the Boolean flag *planeData* indicating that the curve and the axis of revolution lie in one plane, the Boolean flag *poleMin* indicating that the surface has a pole at the initial value of the first parameter, the Boolean flag *poleMax* indicating that the surface has a pole at the terminal value of the first parameter, and the values of the first surface parameter at the poles of the surface *uPoleMin* and *uPoleMax*, if the corresponding pole is present. The surface also has some additional data that are optional and are used to speed up the execution of the surface methods.

The first surface parameter *u* coincides with the parameter of the generatrix curve. The first surface parameter takes values on the interval *umin* ≤ *u* ≤ *umax*, which corresponds to the

domain of the generatrix curve. The surface can be periodic with respect to the first parameter if the generatrix curve is periodic.

The second surface parameter *v* takes values on the interval: *vmin* ≤ *v* ≤ *vmax*. The values *v* = 0 and *v* = 2π correspond to a point on the generatrix curve. The surface can be periodic with respect to the second parameter. For a periodic surface, *vmax* - *vmin* = 2π; for a non-periodic surface, *vmax* - *vmin* < 2π.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (\mathbf{curve}(u) - \mathbf{position.origin})\,\mathbf{M}(v), \tag{3.11}$$

where **M**(*v*) is the rotation matrix. Note that the vector (**curve**(*u*) - **position.origin**) is multiplied by the matrix **M**(*v*) on the right. The rotation matrix has the form

$$M(v) = A^{-1}\cdot\begin{bmatrix}\cos v & -\sin v & 0\\ \sin v & \cos v & 0\\ 0 & 0 & 1\end{bmatrix}\cdot A = \begin{bmatrix}\mathbf{position.axisX}\\ \mathbf{position.axisY}\\ \mathbf{position.axisZ}\end{bmatrix}^{-1}\cdot\begin{bmatrix}\cos v & -\sin v & 0\\ \sin v & \cos v & 0\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}\mathbf{position.axisX}\\ \mathbf{position.axisY}\\ \mathbf{position.axisZ}\end{bmatrix}. \tag{3.12}$$

The matrix **A** is the transformation matrix of the coordinates of the position vector of a point from the local coordinate system **position** to the global coordinate system. The rows of the matrix **A** are composed of the components of the basis vectors of the local coordinate system. The matrix **M**(*v*) maps the vector **curve**(*u*) - **position.origin** to the local coordinate system, rotates it in this system by the angle *v* around the axis of revolution, and maps the rotated vector back to the global coordinate system. The revolution surface is shown in Fig. 1:

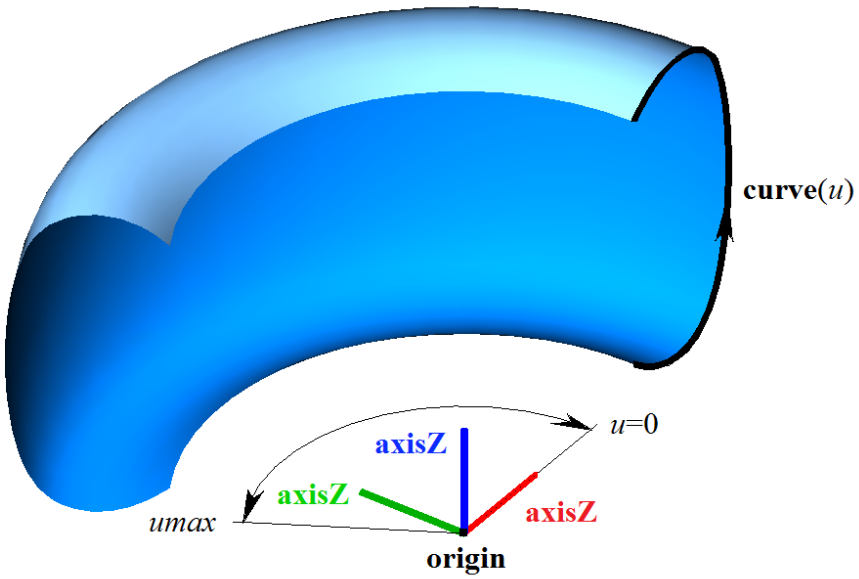


Fig. 1

If the initial or terminal edge of the generatrix curve passes through the axis of revolution, then the surface has a pole for the parameter *umin* or *umax*, respectively. The following inequality must hold for the boundary values of the second surface parameter: *vmin* < *vmax*.

The points of the generatrix curve are rotated along an arc around the vector **position.axisZ** from the vector **position.axisX** in the direction of the vector **position.axisY**. The local coordinate system **position** can be either right-handed or left-handed.

3.13.9 MbExpansionSurface Motion Surface

The MbExpansionSurface class is declared in the *surf_expansion_surface.h* file.

The MbExpansionSurface motion surface belongs to the group of swept surfaces MbSweptSurface. The motion surface is a special case of a swept surface with a curvilinear spine curve. The motion surface is described by the generatrix curve *MbCurve3D* * **curve**, the spine curve *MbCurve3D* * **spine**, and the point at the start of the spine MbCartPoint3D **origin**. The surface is formed by moving the generatrix curve along the spine curve. In a special case, the generatrix curve of the motion surface can change its shape. In the latter case, the curve data contain the second generatrix curve **brink**, the point at the end of the spine **ending**, the initial parameter *tmin* of the **brink** curve, and the derivative *dt* of the parameter of the **brink** curve with respect to the parameter of the generatrix curve **curve**. The derivative *dt* is defined by

$$dt = \frac{tmax - tmin}{umax - umin}, \tag{3.13}$$

where *tmax* is the terminal parameter of the **brink** curve. In the general case, the pointer to the second generatrix curve **brink** can be null, which means that the second generatrix curve is absent.

The first surface parameter *u* coincides with the parameter of the generatrix curve **curve**. The first surface parameter takes values on the interval *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of the generatrix curve. The surface can be periodic with respect to the first parameter if the generatrix curve is periodic.

The second surface parameter *v* coincides with the parameter of the spine curve and takes values on the interval *vmin* ≤ *v* ≤ *vmax*. The surface can be periodic with respect to the second parameter if the spine curve is periodic and the second generatrix curve is absent.

In the PointOn(double *u*, double *v*, MbCartPoint3D & **s**) method, in the general case, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u, v) = \mathbf{spine}(v) + \mathbf{curve}(u) - \mathbf{origin} \tag{3.14}$$

The motion surface in the general case is shown in Fig. 1.

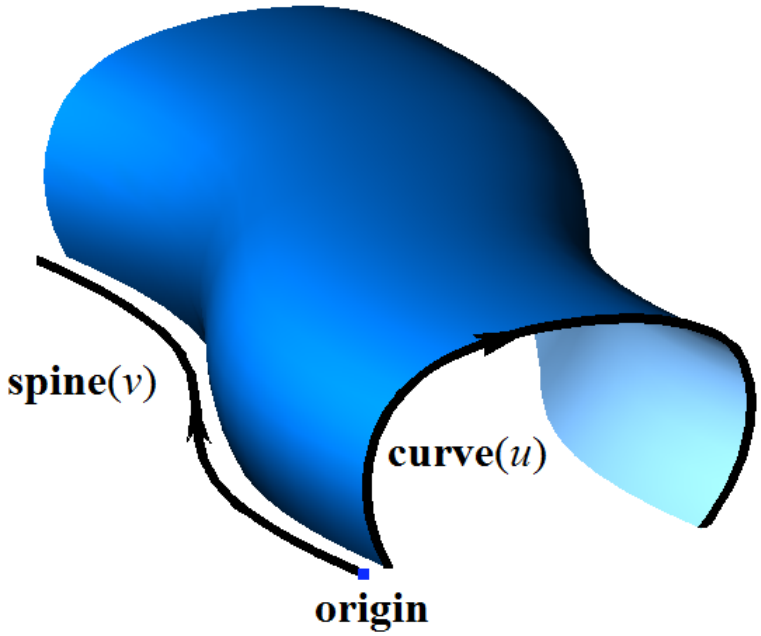


Fig. 1

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, in the special case, the position vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + (\mathbf{curve}(u) - \mathbf{origin})\,(1-w) + (\mathbf{brink}(t) - \mathbf{ending})\,w, \tag{3.15}$$

where

$$w = \frac{v - vmin}{vmax - vmin}, \quad t = tmin + (u - umin)\cdot dt. \tag{3.16}$$

The motion surface in the special case is shown in Fig. 2:

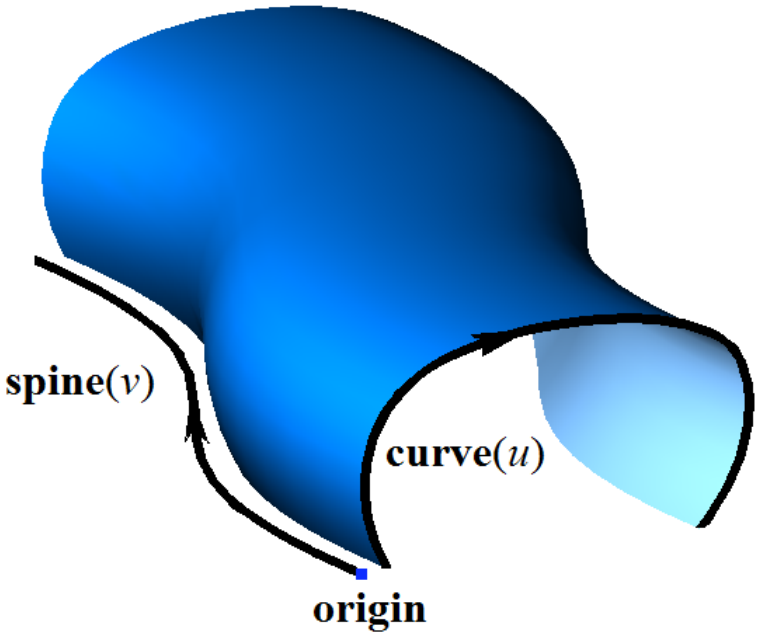


Fig. 2

To avoid self-intersections of the surface, the generatrix and spine curves must not have segments that are parallel to each other. In certain cases, the motion surface may have singular points.

3.13.10 MbSpiralSurface Spiral Surface

Class MbSpiralSurface is declared in file *surf_spiral_surface.h*.

Spiral surface MbSpiralSurface belongs to the group of swept surfaces MbSweptSurface. Spiral surface is a special case of a swept surface with a guiding curve in the form of a cylindrical spiral. The spiral surface is defined by a generatrix curve *MbCurve3D* * **curve**, a local coordinate system *MbPlacement3D* **position** whose vector **position.axisZ** is the spiral axis, the spiral radius *radius*, the spiral step *step*, the position of the spiral start **origin**, and the spiral boundary parameters *vmin* and *vmax*. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The spiral axis coincides with the coordinate axis **position.axisZ** of the local coordinate system. The first parameter of the surface *u* coincides with the generatrix curve parameter. The first parameter of the surface takes values on the segment $umin \le u \le umax$, which corresponds to the domain of the generatrix curve. The surface can be periodic in the first parameter if the generatrix curve is periodic.

The second parameter of the surface *v* takes values on the segment $vmin \le v \le vmax$. Value $v = 0$ corresponds to a point on the generatrix curve. Value of the second parameter *v* equal to 2π corresponds to points of the generatrix curve shifted by the vector **position.axisZ** multiplied by *step*. The surface cannot be periodic in the second parameter.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s**

is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(t)\,\mathbf{position.axisX} + \sin(t)\,\mathbf{position.axisY}) + \left(\frac{t \cdot step}{2\pi}\right)\mathbf{position.axisZ} + (\mathbf{cu} \tag{3.17}$$

where **M**(*v*) is a rotation matrix. Note that multiplication of the vector (**curve**(*u*) - **origin**) by matrix **M**(*v*) is performed on the right. The rotation matrix has the form

$$M(v) = A^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot A = \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}. \tag{3.18}$$

Matrix **A** is a coordinate transformation matrix of the radius vector of a point from the local coordinate system **position** to the global coordinate system. The rows of matrix **A** are composed of the components of the basis vectors of the local coordinate system. Matrix **M**(*v*) maps the vector **curve**(*u*) - **origin** into the local coordinate system, rotates it there by angle *v* around the rotation axis, and maps the rotated vector back into the global coordinate system. Spiral surface is shown in Fig. 1:

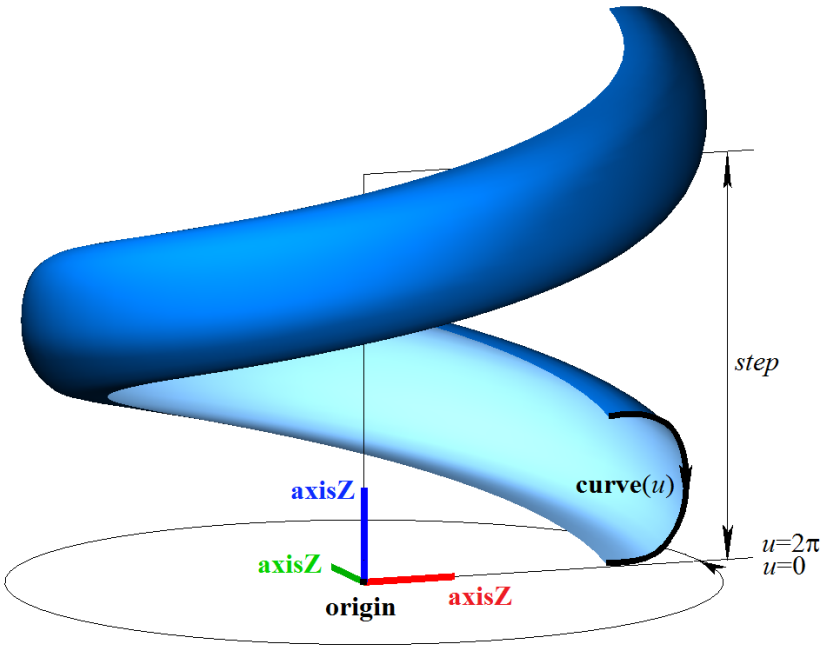


Fig. 1

For boundary values of the second parameter of the surface the inequality *vmin* < *vmax* must hold.

3.13.11 MbEvolutionSurface Swept Surface

Class MbEvolutionSurface is declared in file *surf_evolution_surface.h*.

Kinematic surface MbEvolutionSurface belongs to the group of swept surfaces MbSweptSurface.Kinematic surface is a general case of a swept surface with an arbitrary guiding curve. The swept surface is defined by a generatrix curve *MbCurve3D* * **curve**, a guiding object *MbCurve3D* * **spine**, and the start position of the guiding curve *MbCartPoint3D* **origin**. The surface also has

some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface *u* coincides with the parameter of the generatrix curve **curve**. The first parameter of the surface takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of the generatrix curve. The surface can be periodic in the first parameter if the generatrix curve is periodic.

The guiding object **spine** replaces the guiding curve, is constructed on the basis of the curve, and differs from it in that it can generate a local coordinate system associated with the curve. The second parameter of the surface *v* coincides with the curve parameter of the guiding object **spine**. The second parameter of the surface takes values on the segment *vmin* ≤ *v* ≤ *vmax*, which corresponds to the domain of the guiding curve. The surface can be periodic in the second parameter if the guiding curve is periodic.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + (\mathbf{curve}(u) - \mathbf{origin})\,\mathbf{M}(v), \tag{3.19}$$

where **M**(*v*) is a matrix associated with the guiding curve. Note that multiplication of the vector (**curve**(*u*) - **origin**) by matrix **M**(*v*) is performed on the right. Matrix **M**(*v*) has the form

$$\mathbf{M}(v) = \mathbf{A}^{-1}(vmin) \cdot \mathbf{A}(v), \tag{3.20}$$

where matrix **A**(*v*) is a coordinate transformation matrix of the radius vector of a point from the moving coordinate system associated with the guiding curve to the global coordinate system. Matrix **A**(*v*) depends on the second parameter of the surface. The rows of matrix **A**(*v*) are composed of the components of the basis vectors of the moving coordinate system:

$$A(v) = \begin{bmatrix} i_1(v) \\ i_2(v) \\ i_3(v) \end{bmatrix}, \tag{3.21}$$

where $\mathbf{i}_1(v)$ is the tangent vector of the guiding curve, $\mathbf{i}_2(v)$ is a vector orthogonal to $\mathbf{i}_1(v)$ and associated with vector **direction** of the guiding object **spine**, and $\mathbf{i}_3(v)$ is a vector orthogonal to $\mathbf{i}_1(v)$ and $\mathbf{i}_2(v)$. The swept surface is shown in Fig. 1:

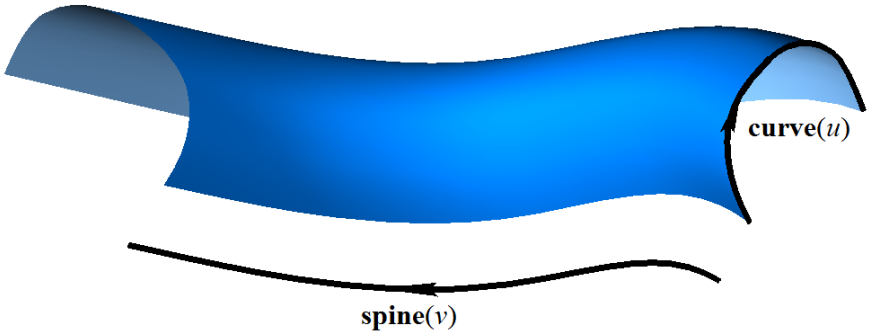


Fig. 1

The tangent vector $\mathbf{i}_1(v)$ is computed from the guiding curve. Vector $\mathbf{i}_2(v)$ is computed from the condition of smooth variation when moving from point to point along the guiding curve and orthogonality to $\mathbf{i}_1(v)$. Vector $\mathbf{i}_3(v)$ is computed as the cross product of vectors $\mathbf{i}_1(v)$ and $\mathbf{i}_2(v)$.

3.13.12 MbExactionSurface Swept Surface with Adaptation

Class MbExactionSurface is declared in file *surf_exaction_surface.h*.

Kinematic surface with adaptation MbExactionSurface is a derived kinematic surface MbEvolutionSurface. Kinematic surface with adaptation is used to construct solids by a kinematic operation with a guiding curve in the form of a composite curve whose segments are joined with a break, as shown in Fig. 1.

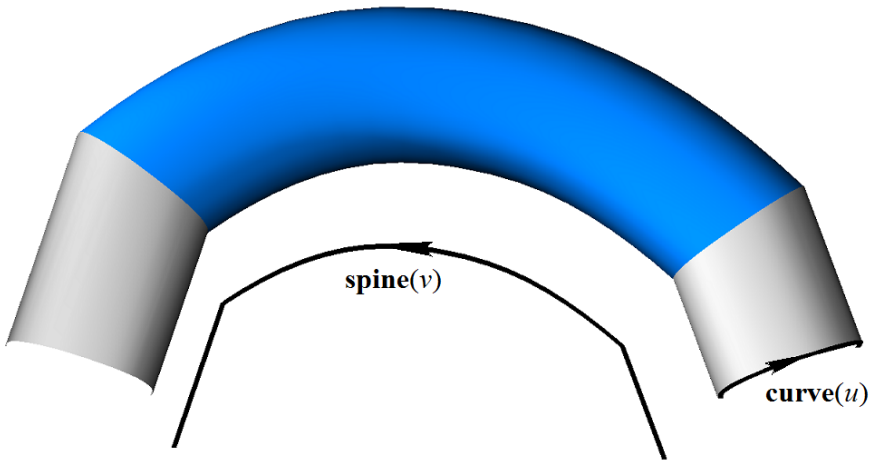


Fig. 1

Kinematic surface MbExactionSurface adjusts its end faces to ensure junction with another surface.

3.13.13 MbSectorSurface Sectorial Surface

Class MbSectorSurface is declared in file *surf_sector_surface.h*.

Sectorial surface MbSectorSurface belongs to the group of swept surfaces MbSweptSurface. The sectorial surface is defined by a curve *MbCurve3D* **curve** and a point *MbCartPoint3D* **origin**.

The first parameter of the surface *u* coincides with the parameter of curve **curve**. The first parameter of the surface takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of curve **curve**. The surface can be periodic in the first parameter if curve **curve** is periodic.

The second parameter of the surface *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*. Value *v* = *vmin* corresponds to a point on curve **curve**, and value *v* = *vmax* corresponds to point **origin**. The surface cannot be periodic in the second parameter.

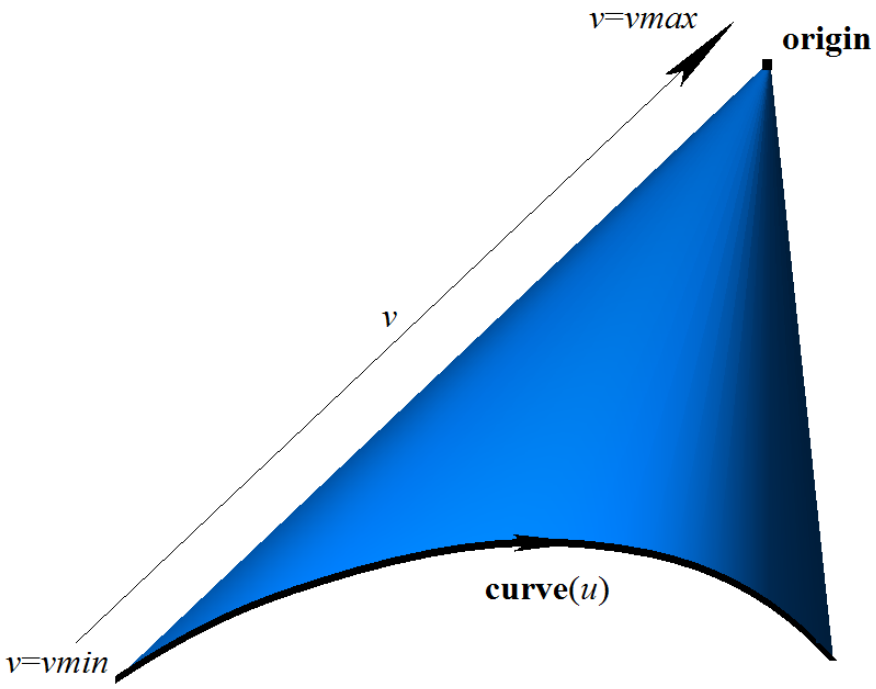
In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{curve}(u)(1-w) + \mathbf{origin}\,w, \tag{3.22}$$

where

$$w = \frac{v - vmin}{vmax - vmin}. \tag{3.23}$$

The sectorial surface is shown in the figure:



Curves of the surface **s**(*const*, *v*) are straight-line segments. The surface has a pole at point **origin** for *v* = *vmax*. Sectorial surface is a special case of a ruled surface.

3.13.14 MbRuledSurface Ruled Surface

Class MbRuledSurface is declared in file *surf_ruled_surface.h*.

Ruled surface MbRuledSurface belongs to the group of swept surfaces MbSweptSurface. The ruled surface is defined by a curve *MbCurve3D* * **curve**, a curve *MbCurve3D* * **sline**, a flag of the presence of a pole of the surface at the initial value of the first parameter *poleMin*, a flag of the presence of a pole of the surface at the final value of the first parameter *poleMax*, the initial parameter *tmin* of curve **sline**, the derivative *dt* of the parameter of curve **sline** with respect to the parameter of curve **curve**, and the surface shape type *type*. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface *u* coincides with the parameter of curve **curve**. The first parameter of the surface takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of curve **curve**. The derivative *dt* is defined by

$$dt = \frac{tmax - tmin}{umax - umin}, \tag{3.24}$$

where *tmax* is the final parameter of curve **sline**. The surface can be periodic in the first parameter if curves **curve** and **sline** are periodic.

The second parameter of the surface *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*. Value *v* = *vmin* corresponds to a point on curve **curve**, and value *v* = *vmax* corresponds to a point on curve **sline**. The surface cannot be periodic in the second parameter.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u, v) = \mathbf{curve}(u)(1 - w) + \mathbf{sline}(t)\, w, \tag{3.25}$$

where

$$w = \frac{v - vmin}{vmax - vmin}, \quad t = tmin + (u - umin) \cdot dt. \tag{3.26}$$

The ruled surface is shown in Fig. 1:

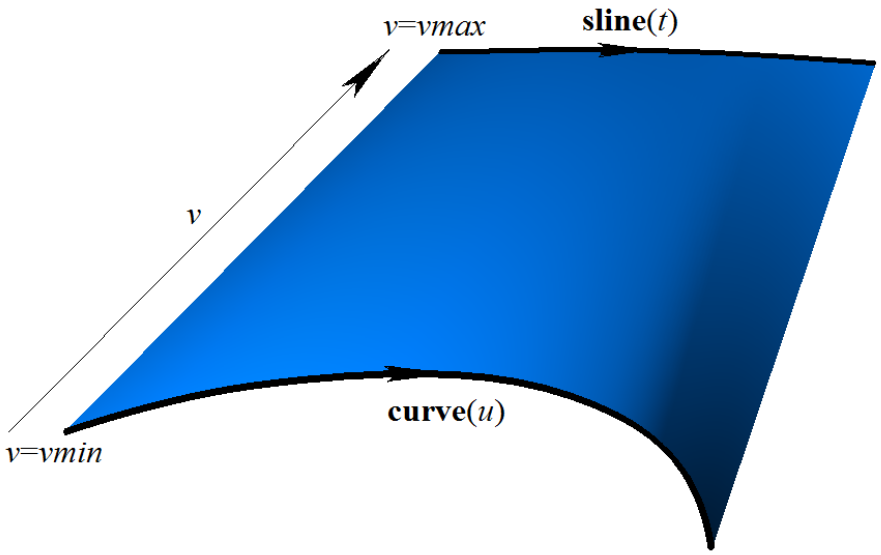


Fig. 1

Curves of the surface **s**(*const*, *v*) with parameters *u* = const are straight-line segments. The surface may have a pole if one of the curves **curve** or **sline** is contracted to a point, or if curves **curve** and **sline** coincide at one of the ends.

3.13.15 MbLoftedSurface Surface Based on a Family of Curves

Class MbLoftedSurface is declared in file *surf_lofted_surface.h*.

Surface MbLoftedSurface is defined by a set of curves RPArray< *MbCurve3D* > **uCurves**, a set of values of the second surface parameter for the curves *vParams*, a set of flags of identical curves *vLabels*, boundary values of parameters *umin*, *umax*, *vmin*, *vmax*, flags of closure of the surface in the first and second parameters *uClosed* and *vClosed*, a direction vector of a non-periodic surface at *vmin* *MbVector3D* **derive1**, a direction vector of a non-periodic surface at *vmax* *MbVector3D* **derive2**, and flags of the presence of poles of the surface at the boundary of the domain *poleUMin*, *poleUMax*, *poleVMin*, *poleVMax*. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface *u* coincides with the parameters of curves **curves**. All curves **curves** must have the same parameter domain; for this purpose, curves MbReperamCurve3D can be used. The first parameter of the surface takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of curves **curves**. The surface can be periodic in the first parameter if all curves of set **curves** are periodic.

The second parameter of the surface *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*. Value *v* = *vmin* corresponds to the initial value of set *vParams*[0], and value *v* = *vmax* corresponds to the final value of set *vParams*[*vParams*.MaxIndex()]. The surface can be periodic in the second parameter.

If all curves of set **curves** are different, then values of set *vLabels* equal the indices of the curves of set **curves**. If among adjacent curves of set **curves** there are identical curves (but shifted

relative to each other), then the corresponding values of set *vLabels* equal the minimum index of the replicated curve of set **curves**.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = (1-3w^2+2w^3)\,\mathbf{curves}[i](u)+(3w^2+2w^3)\,\mathbf{curves}[i+1](u)+(w-2w^2+w^3)\,\mathbf{derive}[i](u)+(-w^2+w^3)\,\mathbf{derive}[i+1](u) \tag{3.27}$$

where

$$w = \frac{v - vParams[i]}{vParams[i+1] - vParams[i]}, \tag{3.28}$$

derive[i] and **derive**[i+1] are derivatives of curves **curves**[i] and **curves**[i+1], respectively. Index *i* of the working segment for computing the radius vector of a point and its derivatives is determined from the condition *vParams*[i] ≤ *v* ≤ *vParams*[i+1]. If among neighboring elements of set *vLabels* there are identical values, then between the corresponding curves the surface is equivalent to an extrusion surface. Surface based on a family of curves is shown in Fig. 1:

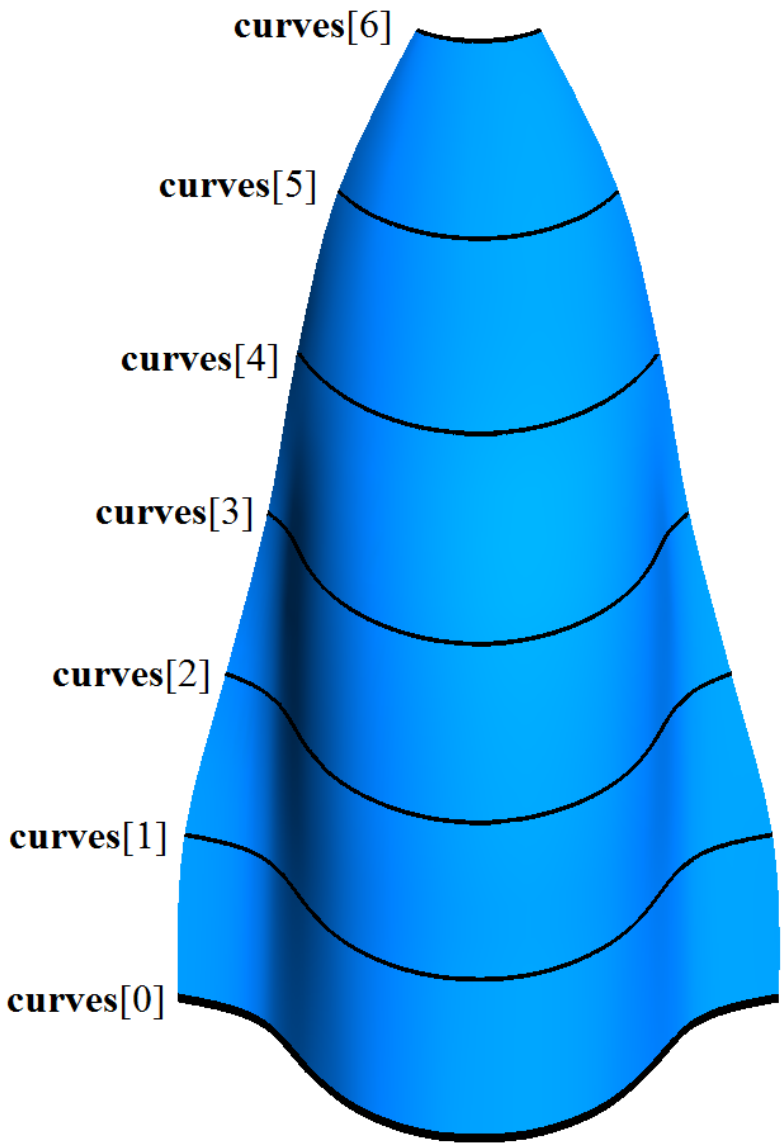


Fig. 1

The shape of the surface depends on the arrangement of the curves and on set *vParams* of parameter values at which the surface passes through the curves. To prevent self-intersections of the surface, values of set *vParams* must vary proportionally to the average distance between the curves.

Curves of the surface **s**(*const*, *v*) with parameters *u* = const are Hermite curves *MbHermit3D*. The surface may have poles if the first and/or last curve of set **curves** is contracted to a point, or if all curves **curves** coincide at one of the ends.

3.13.16 MbElevationSurface Surface Based on a Family of Curves And a Guiding Curve

Class MbElevationSurface is declared in file *surf_elevation_surface.h*.

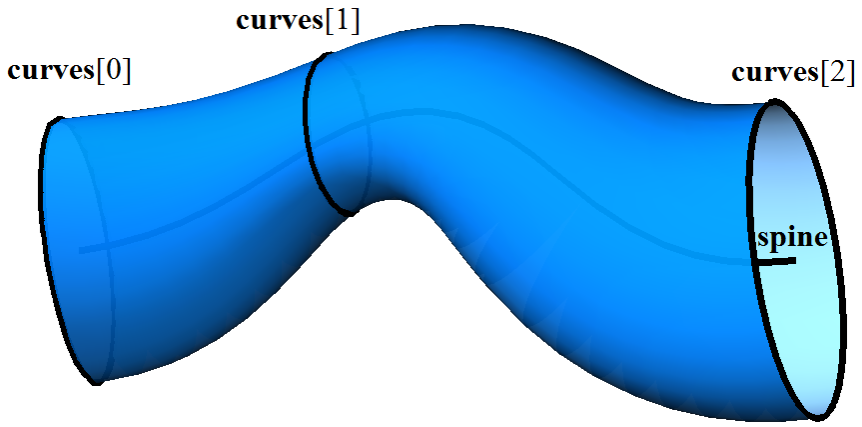
Surface based on a family of curves and a guiding curve is a derived class of *MbLoftedSurface*.

Just as surface *MbLoftedSurface*, surface MbElevationSurface is defined by a set of generatrix curves RPArray< *MbCurve3D* > **uCurves**, a set of values of the second surface parameter for the curves *vParams*, boundary values of parameters *umin*, *umax*, *vmin*, *vmax*, flags of closure of the surface in the first and second parameters *uClosed* and *vClosed*, and flags of the presence of poles of the surface at the boundary of the domain *poleUMin*, *poleUMax*, *poleVMin*, *poleVMax*. In addition to the listed data, surface MbElevationSurface is defined by a guiding curve MbCurve3D **spine**. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface *u* coincides with the parameters of curves **curves**. All curves **curves** must have the same parameter domain; for this purpose, curves MbReperamCurve3D can be used. The first parameter of the surface takes values on the segment $umin \le u \le umax$, which corresponds to the domain of curves **curves**. The surface can be periodic in the first parameter if all curves of set **curves** are periodic.

The second parameter of the surface *v* coincides with the parameter of the guiding curve **spine**. The second parameter of the surface takes values on the segment $vmin \le v \le vmax$, which corresponds to the domain of the guiding curve. The surface can be periodic in the second parameter if the guiding curve is periodic.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is computed similarly to the radius vector of surface MbLoftedSurface with a correction for the displacement of the guiding curve. Surface based on a family of curves and a guiding curve is shown in the figure:



The shape of the surface depends on the arrangement of the generatrix curves, the guiding curve, and on set *vParams* of parameter values at which the surface passes through the curves. Values of set *vParams* are determined by projecting the centers of mass of the generatrix curves onto the guiding curve.

3.13.17 MbCornerSurface Surface Based on Three Curves

Class MbCornerSurface is declared in file *surf_corner_surface.h*.

Surface based on three curves MbCornerSurface is defined by curves *MbCurve3D* * **curve0**, **curve1**, **curve2**, three points *MbCartPoint3D* **vertex**[3], and three pairs of boundary values of the parameters of the corresponding curves: *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface *u* takes values on the segment 0 ≤ *u* ≤ 1. The surface cannot be periodic in the first parameter. The surface has a singular point at the minimum value of the first parameter *u* = 0. At the singular point, the derivative of the radius vector of the surface with respect to the second parameter is equal to zero.

The second parameter of the surface *v* takes values on the segment 0 ≤ *v* ≤ 1. The surface cannot be periodic in the second parameter.

Curves **curve0**, **curve1**, **curve2** must have intersection points or crossing points with each other. Parameters of the curves *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max* are computed from the intersection or crossing points of the curves and define the working segments of the curves and points **vertex**[3]. The directions of the curves are not important for the surface.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = w_0\,(\mathbf{curve2}(t_2) + \mathbf{curve1}(s_1) - \mathbf{vertex}[0]) + w_1\,(\mathbf{curve0}(t_0) + \mathbf{curve2}(s_2) - \mathbf{vertex}[1]) + w_2\,(\mathbf{curve1}(t_1) + \mathbf{cur} \tag{3.29}$$

where w_0 = 1 - *u*, w_1 = 0.5(*u* - *uv*), w_2 = 0.5(*u* + *uv*) are the barycentric coordinates of the surface,

$$t_0 = w_2 \cdot t0min + (1 - w_2) \cdot t0max, \quad s_0 = (1 - w_1) \cdot t0min + w_1 \cdot t0max \tag{3.30}$$

are parameters of curve **curve0**,

$$t_1 = w_0 \cdot t1min + (1 - w_0) \cdot t1max, \quad s_1 = (1 - w_2) \cdot t1min + w_2 \cdot t1max \tag{3.31}$$

are parameters of curve **curve1**,

$$t_2 = w_1 \cdot t2min + (1 - w_1) \cdot t2max, \quad s_2 = (1 - w_0) \cdot t2min + w_0 \cdot t2max \tag{3.32}$$

are parameters of curve **curve2**.

Surface based on three crossing curves is shown in Fig. 1:

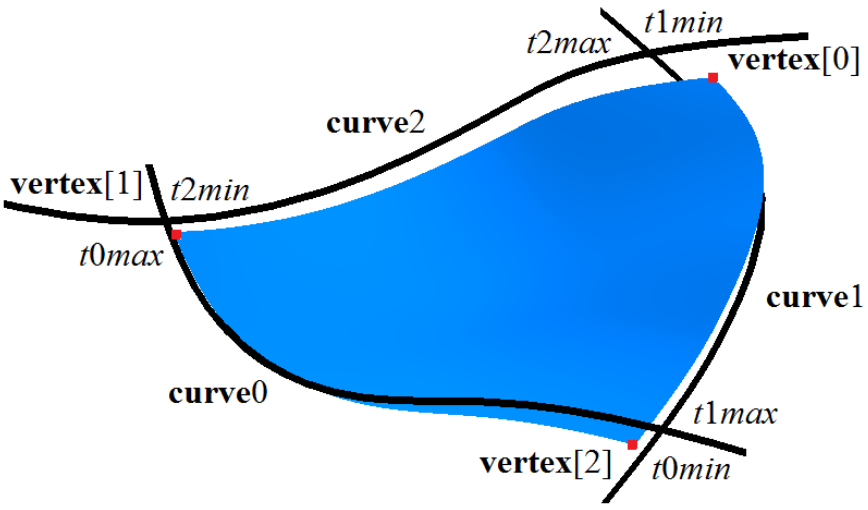


Fig. 1

Fig. 2 shows a surface coinciding with a part of a sphere surface constructed on three identical circular arcs: the planes of the circular arcs are orthogonal to each other, the arcs intersect at end points, and each arc contains a quarter of a circle.

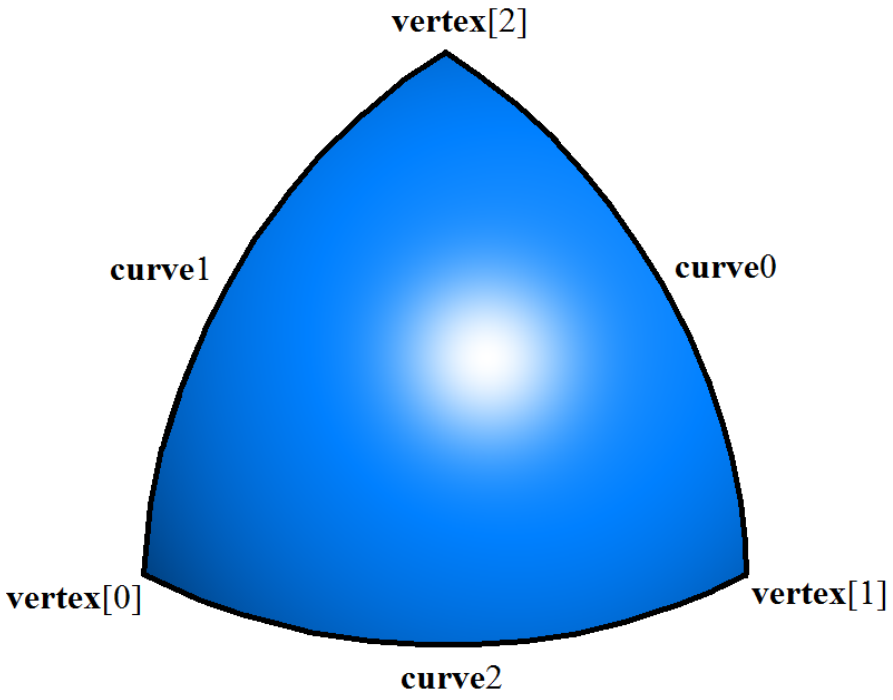


Fig. 2

The shape of the surface depends on the shape of the curves. If the curves do not intersect, the surface does not pass through the curves. If the curves intersect, points **vertex**[3] lie at the intersection points, and the surface passes along segments of the curves: for $w_0 = 0$ the surface passes along a segment of curve **curve0**, for $w_1 = 0$ the surface passes along a segment of curve **curve1**, and for $w_2 = 0$ the surface passes along a segment of curve **curve2**.

3.13.18 MbCoverSurface Coons Surface

Class MbCoverSurface is declared in file *surf_cover_surface.h*.

Coons surface MbCoverSurface is defined by curves *MbCurve3D* **curve0**, **curve1**, **curve2**, **curve3**, four points *MbCartPoint3D* **vertex**[4], four pairs of boundary values of the parameters of the corresponding curves: *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max*, a flag of periodicity of the surface in the first parameter *uclosed*, a flag of periodicity of the surface in the second parameter *vclosed*, a flag of the presence of a pole of the surface at the initial value of the first parameter *poleUMin*, a flag of the presence of a pole of the surface at the final value of the first parameter *poleUMax*, a flag of the presence of a pole of the surface at the initial value of the second parameter *poleVMin*, and a flag of the presence of a pole of the surface at the final value of the second parameter *poleVMax*. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface u takes values on the segment $0 \le u \le 1$. The surface can be periodic in the first parameter if curves **curve0** and **curve2** are periodic.

The second parameter of the surface v takes values on the segment $0 \le v \le 1$. The surface can be periodic in the second parameter if curves **curve1** and **curve3** are periodic.

Adjacent curves **curve0**, **curve1**, **curve2**, **curve3** must have intersection points or crossing points with each other. Parameters of the curves *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*,

t3min, *t3max* are computed from the intersection or crossing points of the curves and define the working segments of the curves and points **vertex**[4]. The directions of the curves are not important for the surface.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = (1-v)\,(\mathbf{curve0}(t_0) - (1-u)\mathbf{vertex}[0]) + u\,(\mathbf{curve1}(t_1) - (1-v)\mathbf{vertex}[1]) + v\,(\mathbf{curve2}(t_2) - u\mathbf{vertex}[2]) + (1-u \tag{3.33}$$

where w_0 = 1 - *u*, w_1 = 0.5(*u* - *uv*), w_2 = 0.5(*u* + *uv*) are the barycentric coordinates of the surface,

$$t_0 = (1-u)\cdot t0min + u\cdot t0max \tag{3.34}$$

is the parameter of curve **curve0**,

$$t_1 = (1-v)\cdot t1min + v\cdot t1max \tag{3.35}$$

is the parameter of curve **curve1**,

$$t_2 = (1-u)\cdot t2min + u\cdot t2max \tag{3.36}$$

is the parameter of curve **curve2**,

$$t_3 = (1-v)\cdot t3min + v\cdot t3max \tag{3.37}$$

is the parameter of curve **curve3**.

Coons surface on four crossing curves is shown in Fig. 1.

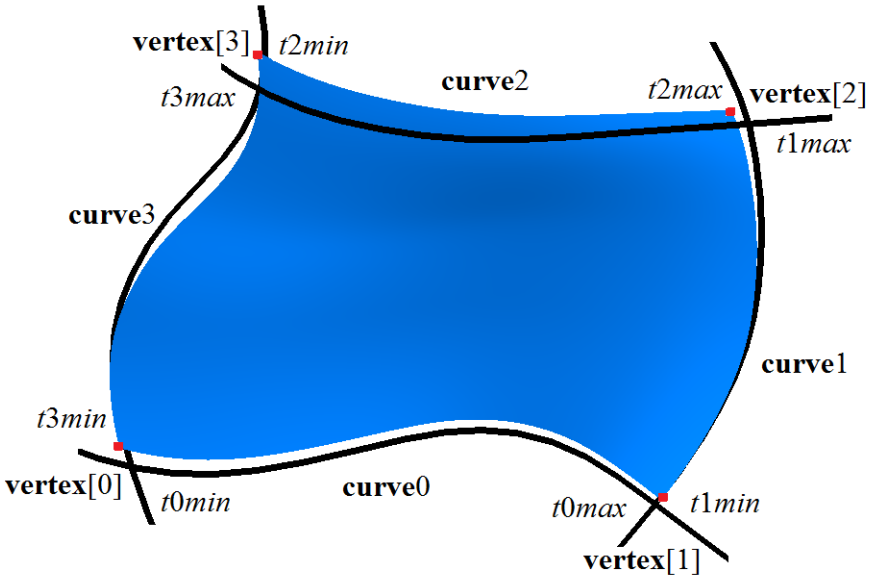


Fig. 1

The shape of the surface depends on the shape of the curves. If adjacent curves intersect at points **vertex**[4], the surface passes along segments of the curves: for *u* = 0 the surface passes along a segment of curve **curve3**, for *v* = 0 the surface passes along a segment of curve **curve0**, for *u* = 1 the surface passes along a segment of curve **curve1**, and for *v* = 1 the surface passes along a segment of curve **curve2**.

3.13.19 MbCoonsPatchSurface Coons Surface

Class MbCoonsPatchSurface is declared in file *surf_coons_surface.h*.

Bicubic Coons patch surface MbCoonsPatchSurface is constructed similarly to Coons surface *MbCoverSurface* and has additional conditions for the radius vector at the edges. Bicubic Coons patch surface MbCoonsPatchSurface is defined by four curves MbCurve3D * **curve0**, **curve1**, **curve2**, **curve3**, derivative **curve0V** of the surface radius vector along curve **curve0** with respect to the second surface parameter, derivative **curve1U** of the surface radius vector along curve **curve1** with respect to the first surface parameter, derivative **curve2V** of the surface radius vector along curve **curve2** with respect to the second surface parameter, derivative **curve3U** of the surface radius vector along curve **curve3** with respect to the first surface parameter, four corner points **vertex**[4], four derivatives with respect to the first surface parameter at the corner points **vertexU**[4], four derivatives with respect to the second surface parameter at the corner points **vertexV**[4], four mixed derivatives with respect to the first and second surface parameters at the corner points **vertexUV**[4], four pairs of boundary values of the parameters of the corresponding curves: *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max*, a flag of periodicity of the surface in the first parameter *uclosed*, and a flag of periodicity of the surface in the second parameter *vclosed*. The surface also has some additional data that are not mandatory and are used to speed up the work of the surface methods.

The first parameter of the surface u takes values on the segment $0 \le u \le 1$. The surface can be periodic in the first parameter if curves **curve0**, **curve2**, **curveV0**, **curveV2** are periodic and curves **curveU1** and **curveU3** coincide.

The second parameter of the surface v takes values on the segment $0 \le v \le 1$. The surface can be periodic in the second parameter if curves **curve1** and **curve3**, **curveU1**, **curveU3** are periodic and curves **curveV0** and **curveV2** coincide.

Adjacent curves **curve0**, **curve1**, **curve2**, **curve3** must have intersection points or crossing points with each other. Parameters of the curves *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max* are computed from the intersection or crossing points of the curves and define the working segments of the curves and points **vertex**[4], **vertexU**[4], **vertexV**[4], **vertexUV**[4]. The directions of the curves are not important for the surface, but the parameterization of pairs of curves **curve0** and **curveV0**, **curve2** and **curveV2**, **curve1** and **curveU1**, **curve3** and **curveU3** must coincide.

In method PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) the radius vector of the surface **s** is described by a vector function given in the book “Geometric Modelling” by N. N. Golovanov. Bicubic Coons patch surface is constructed from precomputed data and is used to construct blends and patches with edge continuity conditions. Bicubic Coons patch surface is shown in Fig. 1.

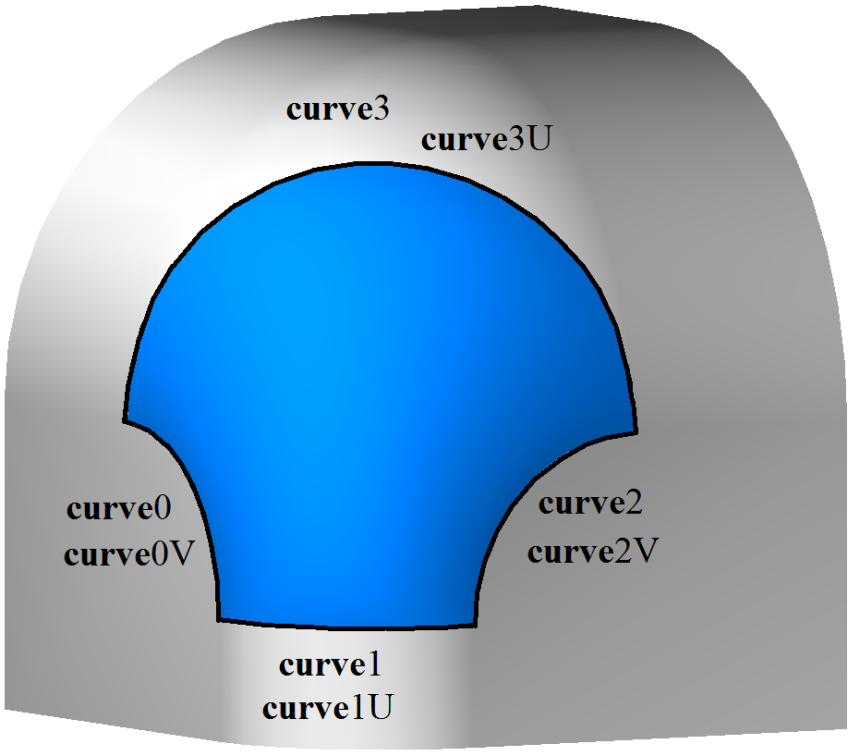


Fig. 1

The shape of the surface depends on the shape of the curves and derivatives. If adjacent curves intersect at points **vertex**[4], the surface passes along segments of the curves: for u = 0 the surface passes along a segment of curve **curve3**, for v = 0 the surface passes along a segment of curve **curve0**, for u = 1 the surface passes along a segment of curve **curve1**, and for v = 1 the surface passes along a segment of curve **curve2**.

3.13.20 MbMeshSurface Surface Based on a Network of Curves

Class MbMeshSurface is declared in the *surf_mesh_surface.h* file.

The mesh surface MbMeshSurface is defined by the set of curves RPArray< *MbCurve3D* > **uCurves**, the set of curves RPArray< *MbCurve3D* > **vCurves**, the set of values of the first surface parameter *uParams*, the set of values of the second surface parameter *vParams*, the boundary values of the first surface parameter *umin* and *umax*, the boundary values of the second surface parameter *vmin* and *vmax*, the flags that indicate the presence of the surface pole at the boundary values of the first parameter *poleUMin*, *poleUMax*, the flags that indicate the presence of the surface pole at the boundary values of the second parameter *poleVMin*, *poleVMax*, the flag of the surface periodicity in the first parameter *uclosed*, the flag of the surface periodicity in the second parameter *vclosed*, the type *type0* of the surface continuity at the edge that corresponds to the second parameter *vmin*, the type *type1* of the surface continuity at the edge that corresponds to the first parameter *umin*, the type *type2* of the surface continuity at the edge that corresponds to the second parameter *vmax*, the type *type3* of the surface continuity at the edge that corresponds to the first parameter *umax*. The surface also has some additional data that is optional and is used to speed up the surface methods.

The number of elements in the set of curves **vCurves** and in the set of values of the first surface

parameter *uParams* is coordinated so that parameter *uParams*[*j*] corresponds to the points of the curve **vCurves**[*j*]. The first surface parameter *u* takes values on the segment *uParams*[0] ≤ *u* ≤ *uParams*[*uParams*.MaxIndex()]. The surface can be periodic in the first parameter if all curves **uCurves** are periodic.

The number of elements in the set of curves **uCurves** and in the set of values of the second surface parameter *vParams* is coordinated so that parameter *vParams*[*i*] corresponds to the points of the curve **uCurves**[*i*]. The second surface parameter *v* takes values on the segment *vParams*[0] ≤ *v* ≤ *vParams*[*vParams*.MaxIndex()]. The surface can be periodic in the second parameter if all curves **vCurves** are periodic.

Each curve **uCurves**[*i*] must have intersection points or crossing points with each curve **vCurves**[*j*]. Neighboring curves of the **uCurves** set must not have opposite directions. Neighboring curves of the **vCurves** set also must not have opposite directions.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface **s** is described by a vector function given in the book «Geometric Modeling» by Golovanov N.N. A mesh surface is shown in the figure:

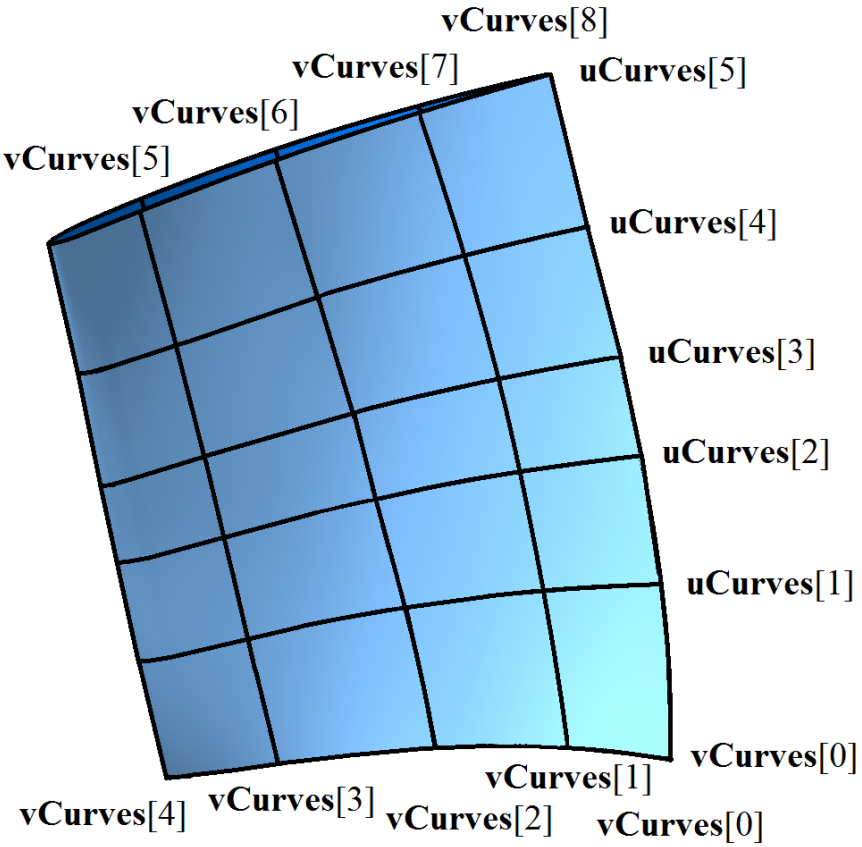


Fig. 1

The shape of the surface depends on the shape of the curves, their relative position, and the parameter values of the *uParams* and *vParams* sets. If every curve **uCurves**[*i*] intersects every curve **vCurves**[*j*], then the surface passes along the curves **vCurves**[*j*] at parameters *u* = *uParams*[*j*] and along the curves **uCurves**[*i*] at parameters *v* = *vParams*[*i*].

3.13.21 MbJoinSurface Joint Surface

Class MbJoinSurface is declared in the *surf_joint_surface.h* file.

The join surface MbJoinSurface is defined by the set of curves RPArray< *MbCurve3D* > **curves**, the knot vector *knots*, the spline order *degree*, the boundary values of the first surface parameter *umin* and *umax*, the flag of the first surface parameter periodicity *closedU*, the flag of the second surface parameter periodicity *closedV*, the flag that indicates the presence of the surface pole at the boundary values of the first parameter *isPoleUmin*, *isPoleUmax*, the flag that indicates the presence of the surface pole at the boundary values of the second parameter *isPoleUmin*, *isPoleUmax*.

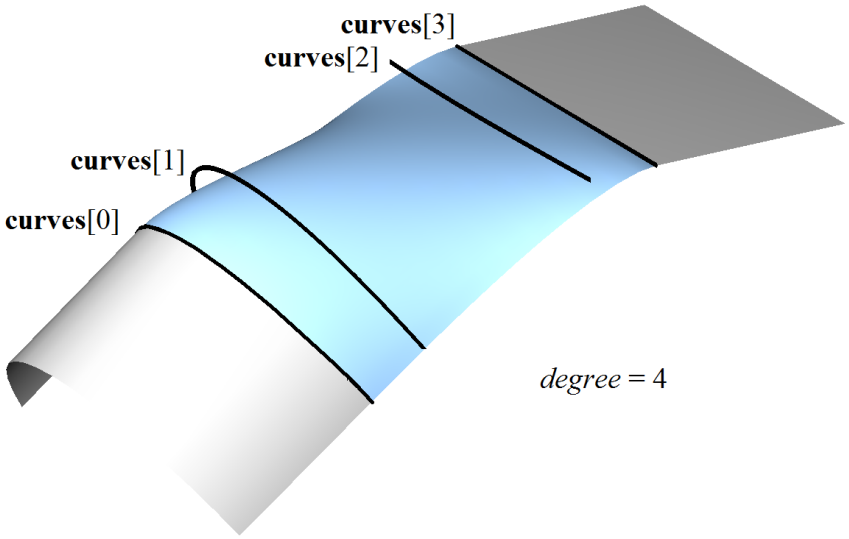
The curves **curves** are coordinated with each other: they have the same direction and the same parameter domain. The first surface parameter *u* coincides with the parameter of the **curves**, which is common for them. The first surface parameter takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of the **curves**. The surface can be periodic in the first parameter if all **curves** are periodic. Let the knot vector *knots* contain *knotsCount* elements, and the number of curves in the **curves** set be equal to *curvesCount*. The numbers of elements in the sets are related by the equality *curvesCount* + *degree* = *knotsCount*.

The second surface parameter *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*, where *vmin* = *knots*[*degree* − 1], *vmax* = *knots*[*knotsCount* − *degree*]. The surface cannot be periodic in the second parameter.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface **s** is described by the vector function

$$s(u,v) = \frac{\sum_{j=0}^{\text{curvesCount}-1} N_j^{degree}(v)\,\text{curves}[j](u)}{\sum_{j=0}^{\text{curvesCount}-1} N_j^{degree}(v)},$$

where $N_j^{degree}(v)$ are B-splines of order *degree* for the *j*-th curve **curves**[*j*]. The join surface is shown in the figure:



Each curve **s**(*const*, *v*) with a fixed first surface parameter *u* = const is a NURBS curve of order *degree* constructed by the points **curves**[*i*](*const*).

3.13.22 MbSplineSurface NURBS Surface

Class MbSplineSurface is declared in the *surf_spline_surface.h* file.

The NURBS surface (NonUniform Rational B-Spline surface) MbSplineSurface is defined by control points SArray<MbCartPoint3D> **points**[*i*][*j*], *i* = 0, 1, ..., *vcount* − 1, *j* = 0, 1, ..., *ucount* − 1, conventionally located at the nodes of a rectangular grid with *ucount* columns and *vcount* rows, by the weights of the control points given in the table *weight*[*i*][*j*], by the order of B-splines along the first surface parameter *udegree*, by the order of B-splines along the second surface parameter *vdegree*, by the knot vector along the first parameter *uknots*, by the knot vector along the second parameter *vknots*, and by the flags of the surface periodicity in the first and second parameters *uclosed* and *vclosed*. The surface also has some additional data that is optional and is used to speed up the surface methods.

The order of B-splines along the surface parameters coincides with the order of the divided difference used to compute the corresponding B-splines. The knot vectors *uknots* and *vknots* are nondecreasing sequences of real numbers and define the domain of the surface parameters and its shape. Let the knot vector *uknots* contain *uknotsCount* elements, and let the number of points in each row of the rectangular grid be equal to *ucount*. For a NURBS surface that is nonperiodic in the first parameter, the numbers of elements in the sets are related by the equality *ucount* + *degree* = *uknotsCount*. For a periodic NURBS surface, the numbers of elements in the sets are related by the equality *knotsCount* + 2 *degree* − 1 = *knotsCount*.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface **s** is described by the vector function

$$r(u,v) = \frac{\sum_{i=0}^{\text{vcount}-1} \sum_{j=0}^{\text{ucount}-1} N_i^{vdegree}(v) N_j^{udegree}(u) \text{weight}[i][j] \text{points}[i][j]}{\sum_{i=0}^{\text{vcount}-1} \sum_{j=0}^{\text{ucount}-1} N_i^{vdegree}(v) N_j^{udegree}(u) \text{weight}[i][j]},$$

where $N_i^{vdegree}(v)$ and $N_j^{udegree}(u)$ are B-splines.

Fig. 1 shows segments connecting neighboring control points **points**[*i*][*j*].

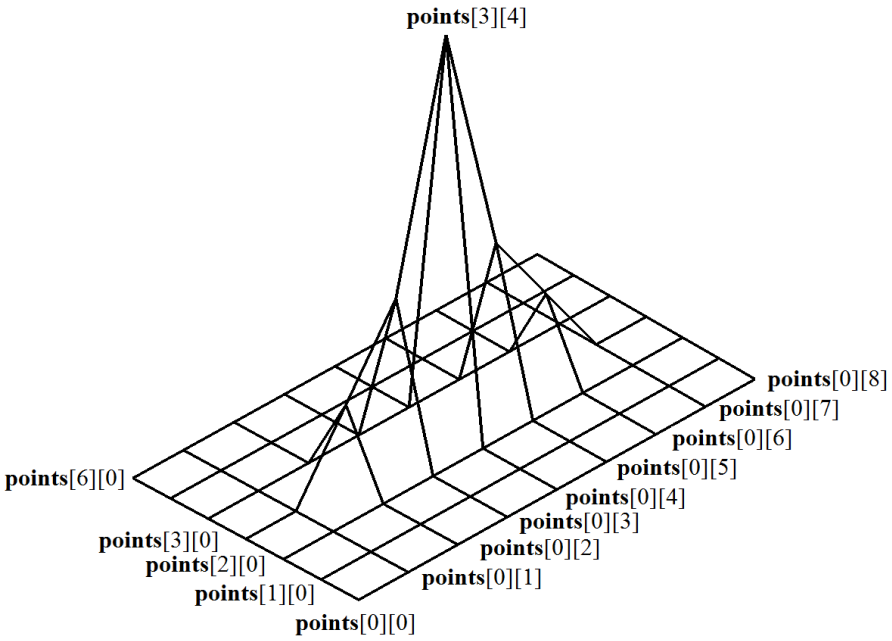


Fig. 1

Fig. 1 illustrates the conventional arrangement of the control points at the nodes of a rectangular grid. The NURBS surface constructed on the same control points is shown in Fig. 2:

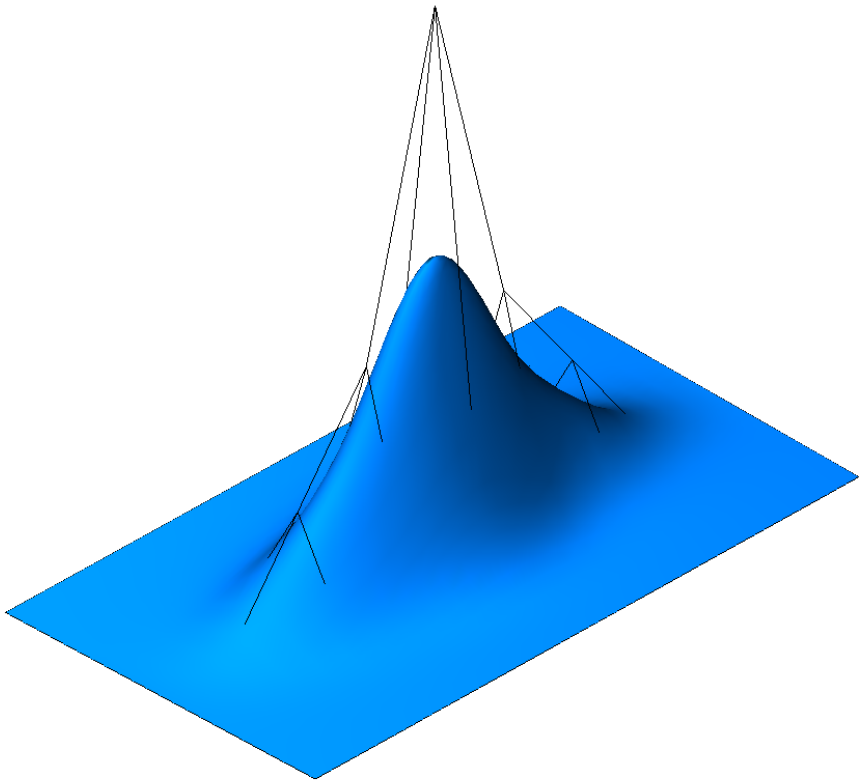


Fig. 2

The curves on the surface **s**(*const*, *v*) and **s**(*u*, *const*) with parameters *u* = const or *v* = const are B-curves of degree *udegree* and *vdegree* respectively. In the first parameter, the surface has order *udegree*, and in the second parameter, the surface has order *vdegree*. The domain of the parameters of a nonperiodic NURBS surface is a rectangle: *uknots*[*udegree* − 1] ≤ *u* ≤ *uknots*[*ucount*], *vknots*[*vdegree* − 1] ≤ *v* ≤ *vknots*[*vcount*].

The surface can be periodic both in the first parameter and in the second parameter. For a periodically closed surface, the knot vectors *uknots* and *vknots* have *udegree* − 1 and *vdegree* − 1 additional elements respectively. The domain of the parameters of a NURBS surface that is periodic in both parameters is a rectangle: *uknots*[*udegree* − 1] ≤ *u* ≤ *uknots*[*ucount* + *udegree* − 1], *vknots*[*vdegree* − 1] ≤ *v* ≤ *vknots*[*vcount* + *vdegree* − 1].

Each surface can construct its NURBS copy by the virtual method NurbsSurface(const MbNurbsParameters & *uParam*, const MbNurbsParameters & *vParam*).

3.13.23 MbOffsetSurface Equidistant Surface

Class MbOffsetSurface is declared in the *surf_offset_surface.h* file.

The equidistant surface MbOffsetSurface is defined by the base surface *MbSurface* *basisSurface*, by the offset along the normal of the base surface *distance*, by the boundary values of the first parameter of the base surface *u0min*, *u0max*, by the boundary values of the second parameter of the base surface *v0min*, *v0max*, by the flags of the base surface periodicity *u0closed*, *v0closed*, by the increments of the boundary values of the first parameter of the base surface *dumin*, *dumax*, and by the increments of the boundary values of the second parameter of the base surface *dvmin*, *dvmax*. The surface also has some additional data that is optional and is used to speed up the surface methods.

The radius vector of a point on the equidistant surface is computed as follows. For a given parameter, the point of the base surface and the normal at this point are computed.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface **s** is described by the vector function

$$\mathbf{r}(u, v) = \textbf{basisSurface}(u, v) + \textbf{normal}(u, v) \cdot \mathit{distance},$$

where **normal**(*u*, *v*) is the surface normal at the given point. The equidistant surface and its base surface are shown in Fig. 1:

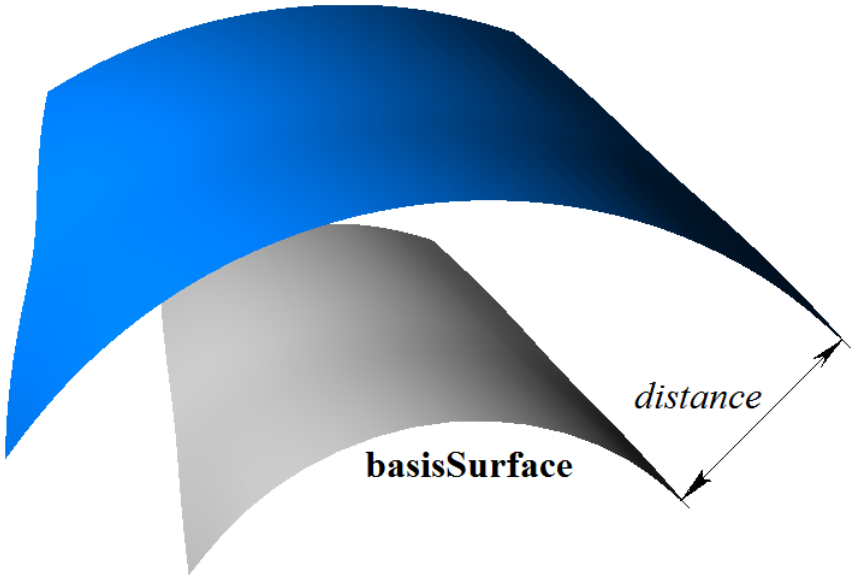


Fig. 1

The parameter domain of the equidistant surface may differ from the parameter domain of the base surface. The domain of the first parameter of the equidistant surface is defined by the inequalities *u0min* + *dumin* ≤ *u* ≤ *u0max* + *dumax*. The domain of the second parameter of the equidistant surface is defined by the inequalities *v0min* + *dvmin* ≤ *v* ≤ *v0max* + *dvmax*. The equidistant surface with a negative offset and an expanded parameter domain and its base surface are shown in Fig. 2:

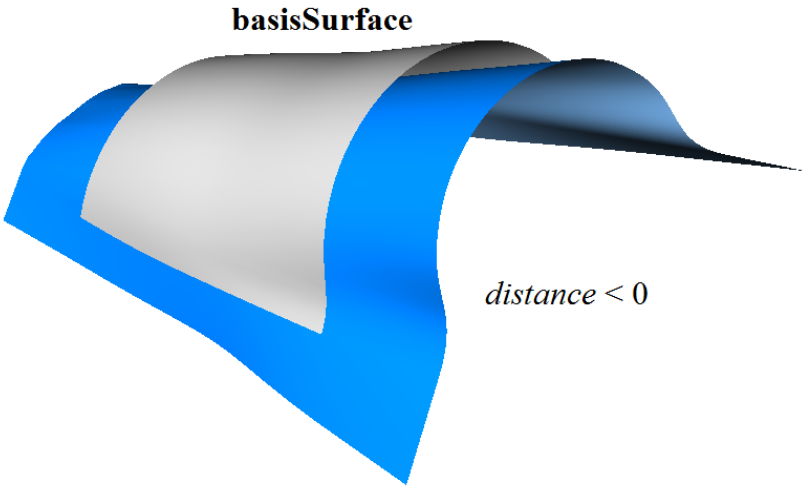


Fig. 2

Another equidistant surface must not be used as the base surface for an equidistant surface; instead, the base surface of the latter must be used with the corresponding recalculation of the offset value.

Each surface can construct an equidistant surface by the virtual method Offset(double *distance*, bool *sense*).

3.13.24 MbChamferSurface Chamfer Surface

Class MbChamferSurface is declared in the *surf_chamfer_surface.h* file.

The chamfer surface MbChamferSurface belongs to the group of blending surfaces MbSmoothSurface. The chamfer surface is defined by a curve on the first blended surface *MbSurfaceCurve* *curve1*, a curve on the second blended surface *MbSurfaceCurve* *curve2*, the chamfer construction method *form*, the chamfer legs *distance1* and *distance2*, the boundary values *umin* and *umax* of the parameter of the curves *curve1* and *curve2*, the boundary values of the second surface parameter *vmin* and *vmax*, the flag of the first surface parameter periodicity *uclosed*, the flag that indicates the presence of a surface pole at the initial value of the first parameter *poleMin*, and the flag that indicates the presence of a surface pole at the final value of the first parameter *poleMax*.

The curves *curve1* and *curve2* are coordinated with each other and have the same parameter domain. The first surface parameter *u* coincides with the parameter of the curves *curve1* and *curve2*, which is common for them. The first surface parameter takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of the curves *curve1* and *curve2*. The surface can be periodic in the first parameter if the curves *curve1* and *curve2* are periodic.

The second surface parameter *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*. The value *v* = *vmin* corresponds to a point on the curve *curve1*, and the value *v* = *vmax* corresponds to a point on the curve *curve2*. The surface cannot be periodic in the second parameter.

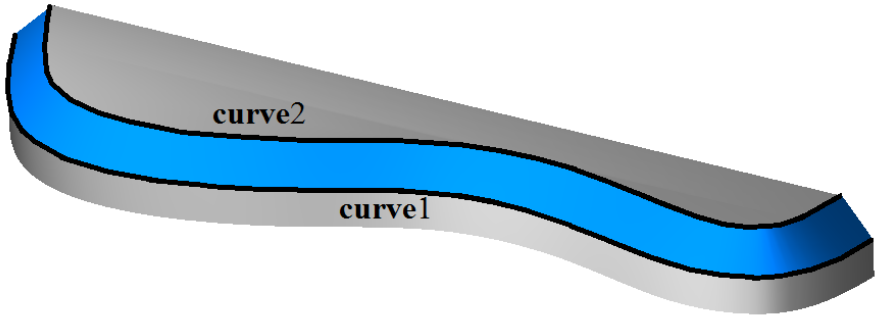
In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u,v) = \mathbf{curve1}(u)\,(1-w) + \mathbf{curve2}(u)\,w,$$

where

$$w = \frac{v - \text{vmin}}{\text{vmax} - \text{vmin}}.$$

The chamfer surface is shown in the figure:



The curves on the surface **s**(*const*, *v*) with *u* = const are straight-line segments. The surface can have a pole at *u* = *umin* and *u* = *umax* if the corresponding ends of the curves *curve1* and *curve2* coincide.

3.13.25 MbFilletSurface Fillet Surface

Class MbFilletSurface is declared in the *surf_fillet_surface.h* file.

The fillet surface MbFilletSurface belongs to the group of blending surfaces MbSmoothSurface. The fillet surface is defined by a curve on the first blended surface *MbSurfaceCurve* *curve1*, a curve on the second blended surface *MbSurfaceCurve* *curve2*, a curve MbCurve3D *curve0*, a function *MbFunction* *weights0* of the weights of the curve *curve0*, the fillet construction method *form*, the fillet radii *distance1* and *distance2*, the shape factor *conic*, the boundary values *umin* and *umax* of the parameter of the curves *curve1*, *curve2*, *curve0* and the function *weights0*, the boundary values of the second surface parameter *vmin* and *vmax*, the flag of the first surface parameter periodicity *uclosed*, the flag that indicates the presence of a surface pole at the initial value of the first parameter *poleMin*, the flag that indicates the presence of a surface pole at the final value of the first parameter *poleMax*, the flag of uniform parameterization of the surface in the second parameter *even*, the flag of smooth blending of the surface with the blended surfaces *equable*, and the flag *byCurve1* that indicates the presence of an edge along the curve *curve2* or *curve1* (*equable* = false). The surface also has some additional data that is optional and is used to speed up the surface methods.

The curves *curve1*, *curve2*, *curve0* and the function *weights0* are coordinated with each other and have the same parameter domain. The first surface parameter *u* coincides with the parameter of the curves *curve1*, *curve2*, *curve0* and the function *weights0*, which is common for them. The first surface parameter takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of the curves *curve1*, *curve2*, *curve0* and the function *weights0*. The surface can be periodic in the first parameter if the curves *curve1*, *curve2*, *curve0* and the function *weights0* are periodic.

The second surface parameter *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*. The value *v* = *vmin* corresponds to a point on the curve *curve1*, and the value *v* = *vmax* corresponds to a point on the curve *curve2*. The surface cannot be periodic in the second parameter.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface

s is described by the vector function

$$s(u,v) = \frac{(1-v)^2\,\mathbf{curve1}(u) + 2t(1-v)\,\mathrm{weight0}(u)\,\mathbf{curve0}(u) + v^2\,\mathbf{curve2}(u)}{(1-v)^2 + 2v(1-v)\,\mathrm{weight0}(u) + v^2}.$$

Each curve **s**(*const*, *v*) with a fixed first surface parameter *u* = const is a third-order NURBS curve constructed by the points *curve1*(*u*), *curve0*(*u*), *curve2*(*u*), where the end points have unit weights and the middle point *curve0*(*u*) has the weight *weights0*(*u*). If *conic* = _ARC_, the function *weights0*(*u*) is computed from the condition that all curves **s**(*const*, *v*) are circular arcs. If *conic* ≠ _ARC_, the function *weights0* is a constant and is equal to the shape factor *conic*. The fillet surface is shown in Fig. 1.

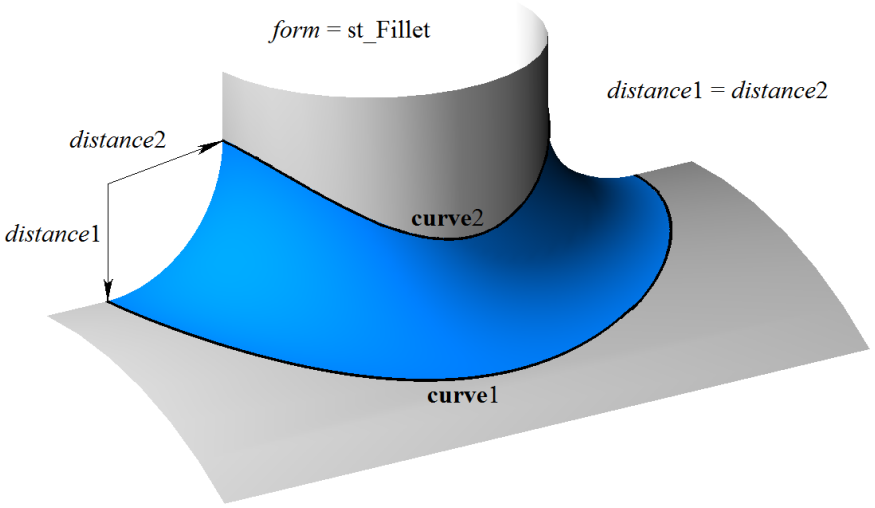


Fig. 1

In the general case, the surface blends smoothly with the surfaces on which the curves *curve1*(*u*) and *curve2*(*u*) lie. In this case, the parameter *equable* = true. If *equable* = false, then along one of the curves *curve1* or *curve2* the blending is smooth, and the other curve is an edge. If *byCurve1* = true, the blending is smooth along the curve *curve1*; otherwise, the blending is smooth along the curve *curve2*. A fillet surface with the preserved edge is shown in Fig. 2.

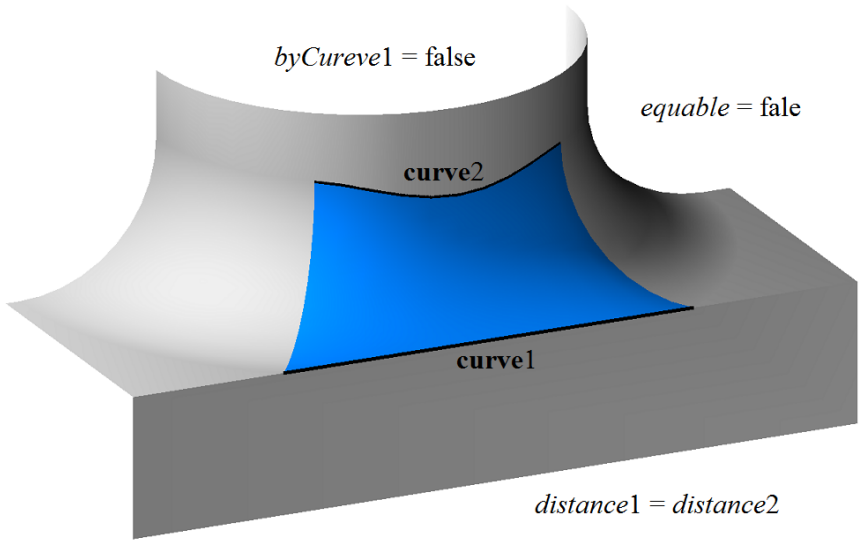


Fig. 2

If the fillet radii *distance1* and *distance2* are not equal, then the fillet surface in the section **s**(*const*, *v*) with *u* = const describes an ellipse. An elliptic fillet surface is shown in Fig. 3.

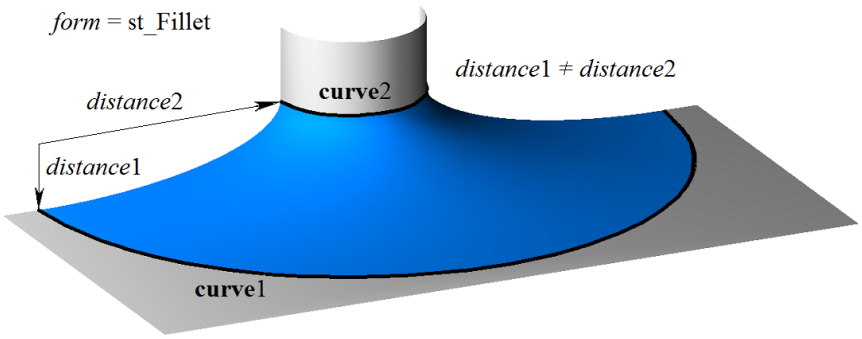


Fig. 3

In the general case, the fillet construction method is *form* = st_Fillet. If the fillet construction method is *form* = st_Span, then *distance1* = *distance2* and they are equal to the distance between the curves *curve1* and *curve2*, and the fillet surface radius is variable. A fillet surface with the preserved distance between the guide curves is shown in Fig. 4.

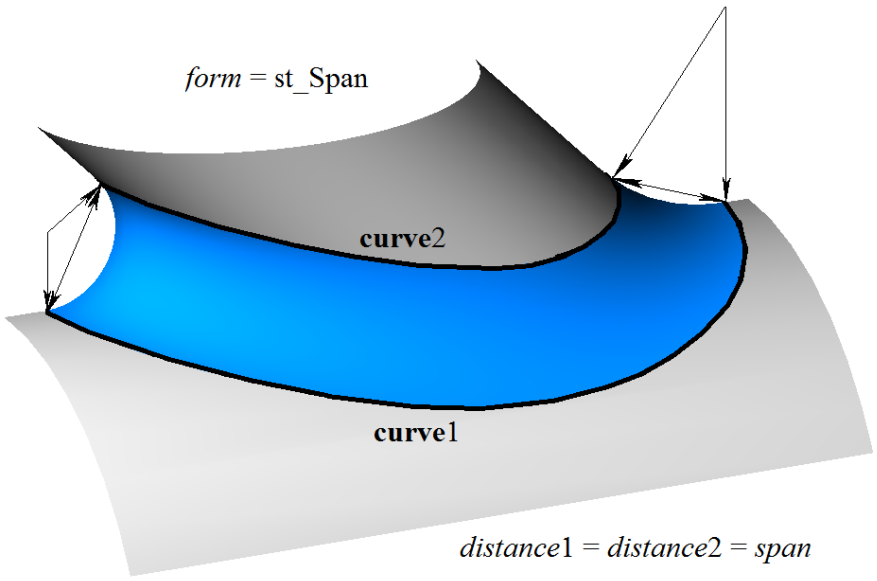


Fig. 4

The curves on the surface **s**(*const*, *v*) with *u* = const are conic sections whose shape depends on the parameter *conic*. If *conic* = _ARC_ = 0, then the curves on the surface **s**(*const*, *v*) are circular arcs. If *conic* = 0.5, then the curves on the surface **s**(*const*, *v*) are parabolic arcs. If 0.05 < *conic* < 0.5, then the curves on the surface **s**(*const*, *v*) are elliptic arcs. If 0.5 < *conic* < 0.95, then the curves on the surface **s**(*const*, *v*) are hyperbolic arcs. The surface can have a pole at *u* = *umin* and *u* = *umax* if the corresponding ends of the curves *curve0*, *curve1* and *curve2* coincide.

3.13.26 MbChannelSurface Fillet Surface

Class MbChannelSurface is declared in the *surf_channel_surface.h* file.

The variable-radius fillet surface MbChannelSurface is an inheritor of the fillet surface MbFilletSurface. The variable-radius fillet surface is defined by a curve on the first blended surface *MbFilletSurface* *curve1*, a curve on the second blended surface *MbSurfaceCurve* *curve2*, a curve MbCurve3D *curve0*, a function MbFunction *weights0* of the weights of the curve *curve0*, a radius variation function MbFunction *function*, the fillet construction method *form*, the fillet radii *distance1* and *distance2*, the shape factor *conic*, the boundary values *umin* and *umax* of the parameter of the curves *curve1*, *curve2*, *curve0* and the functions *weights0* and *function*, the boundary values of the second surface parameter *vmin* and *vmax*, the flag of the first surface parameter periodicity *uclosed*, the flag that indicates the presence of a surface pole at the initial value of the first parameter *poleMin*, and the flag that indicates the presence of a surface pole at the final value of the first parameter *poleMax*. The surface also has some additional data that is optional and is used to speed up the surface methods.

The curves *curve1*, *curve2*, *curve0* and the functions *weights0*, *function* are coordinated with each other and have the same parameter domain. The first surface parameter *u* coincides with the parameter of the curves *curve1*, *curve2*, *curve0* and the functions *weights0* and *function*, which is common for them. The first surface parameter takes values on the segment *umin* ≤ *u* ≤ *umax*, which corresponds to the domain of the curves *curve1*, *curve2*, *curve0* and the functions *weights0* and *function*. The surface can be periodic in the first parameter if the curves *curve1*, *curve2*, *curve0* and the functions *weights0* and *function* are periodic.

The second surface parameter *v* takes values on the segment *vmin* ≤ *v* ≤ *vmax*. The value *v* = *vmin* corresponds to a point on the curve *curve1*, and the value *v* = *vmax* corresponds to a point

on the curve *curve2*. The surface cannot be periodic in the second parameter.

In the PointOn(double *u*, double *v*, MbCartPoint3D & **s**) method, the radius vector of the surface **s** is described by the vector function

$$s(u,v) = \frac{(1-v)^2 \cdot \mathbf{curve1}(u) + 2v(1-v) \cdot \text{weight0}(u) \cdot \mathbf{curve0}(u) + v^2 \cdot \mathbf{curve2}(u)}{(1-v)^2 + 2v(1-v) \cdot \text{weight0}(u) + v^2}.$$

Each curve **s**(*const*, *v*) with a fixed first surface parameter *u* = const is a third-order NURBS curve constructed by the points *curve1*(*u*), *curve0*(*u*), *curve2*(*u*), where the end points have unit weights and the middle point *curve0*(*u*) has the weight *weights0*(*u*). As the first surface parameter varies, the fillet radii change according to the laws R1(*u*) = *distance1* · *function*(*u*) and R2(*u*) = *distance2* · *function*(*u*). If *conic* = _ARC_, the function *weights0*(*u*) is computed from the condition that all curves **s**(*const*, *v*) are circular arcs. If *conic* ≠ _ARC_, the function *weights0* is a constant and is equal to the shape factor *conic*. The variable-radius fillet surface is shown in Fig. 1.

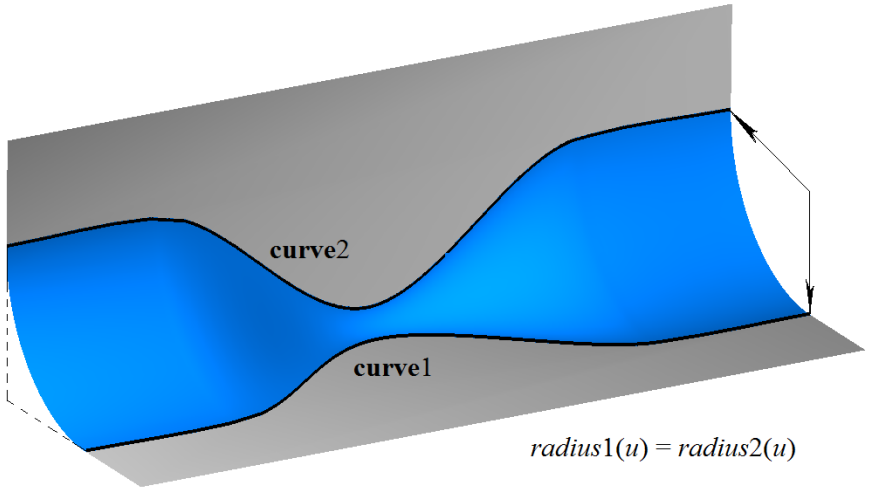


Fig. 1

The variable-radius fillet surface blends smoothly with the surfaces on which the curves *curve1*(*u*) and *curve2*(*u*) lie, with the parameter *equable* = true and *form* = st_Fillet.

The curves on the surface **s**(*const*, *v*) with *u* = const are conic sections whose shape depends on the parameter *conic*. If *conic* = _ARC_ = 0, then the curves on the surface **s**(*const*, *v*) are circular arcs. If *conic* = 0.5, then the curves on the surface **s**(*const*, *v*) are parabolic arcs. If 0.05 < *conic* < 0.5, then the curves on the surface **s**(*const*, *v*) are elliptic arcs. If 0.5 < *conic* < 0.95, then the curves on the surface **s**(*const*, *v*) are hyperbolic arcs. The surface can have a pole at *u* = *umin* and *u* = *umax* if the corresponding ends of the curves *curve0*, *curve1* and *curve2* coincide.

3.13.27 MbCurveBoundedSurface Surface with Arbitrary Borders

Class MbCurveBoundedSurface is declared in the *surf_curve_bounded_surface.h* file.

The surface MbCurveBoundedSurface is defined by the base surface MbSurface **basisSurface**, the set of boundary curves RPArray<*MbContourOnSurface*> **curves** (contours on the surface), and the boundary values of the parameters *umin*, *umax*, *vmin*, *vmax* that define the bounding rectangle of the parameter domain.

The surface MbCurveBoundedSurface has curvilinear edges and can have arbitrary cutouts inside. The boundaries of the surface are described by the contours on the surface of the container **curves**.

In the PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) method, the radius vector of the surface **s** is described by the vector function

$$\mathbf{s}(u, v) = \mathbf{basisSurface}(u, v), \quad u, v \in \Omega,$$

where Ω is the parameter domain, which is a two-dimensional connected region. The radius vector of the contour-bounded surface is described by the same law as the surface **basisSurface**, but has a different parameter domain. The base surface and the contours on it are shown in Fig. 1:

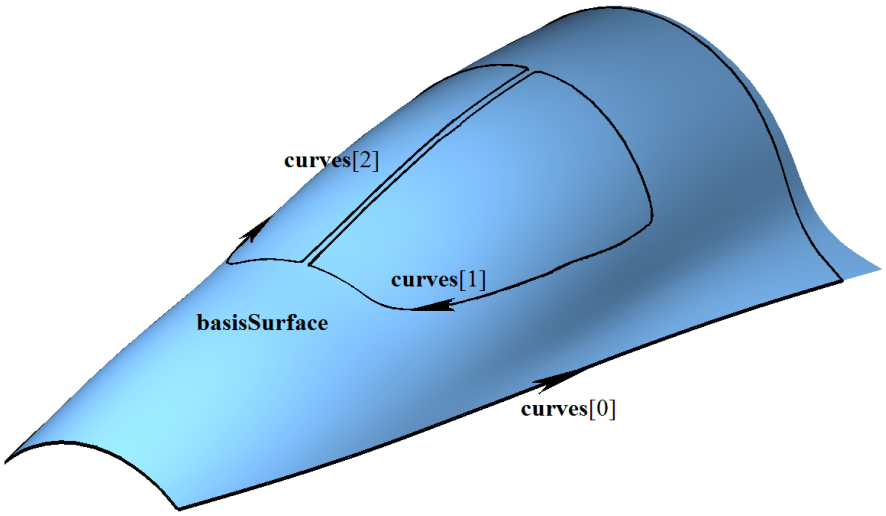


Fig. 1

In the general case, the parameter domain Ω of the contour-bounded surface may extend beyond the parameter domain of the surface **basisSurface**. Outside the parameter domain of the surface **basisSurface**, the radius vector **r**(*u*, *v*) is computed by the method _PointOn(*u*, *v*, **s**) in accordance with the extension rules of the surface **basisSurface**. The surface with an arbitrary boundary is shown in Fig. 2:

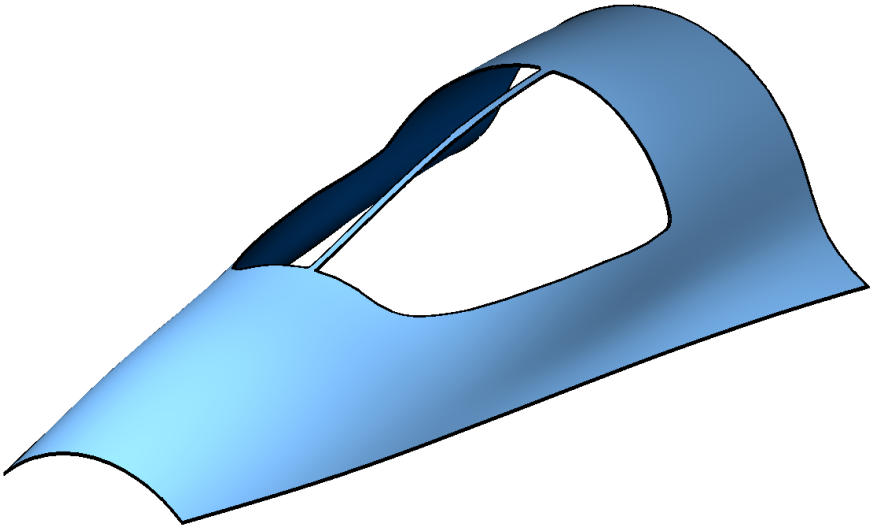


Fig. 2

Each curve of the container **curves** describes one closed boundary of the surface. Each curve of the container **curves** is a contour on the surface *MbContourOnSurface*. Each contour on the surface is described by a surface **surface** coinciding with **basisSurface**, and a two-dimensional contour **contour**. The contour **contour** is a composite closed two-dimensional curve. The segments of the contour **contour** can be any two-dimensional curves MbCurve except composite curves *MbContour*. In the general case, the derivatives of the contour undergo a discontinuity in length and direction at the junction points of the segments. The two-dimensional contours describe the boundaries of the parameter domain Ω of the surface MbCurveBoundedSurface. The boundary contours of the container **curves** satisfy the following conditions: they do not intersect themselves or each other; the first contour **curves**[0] of the container describes the outer boundary and contains all other contours inside, which describe the inner cutouts in the surface. The inner contours cannot be nested inside one another. To quickly determine the position of a given two-dimensional point relative to the parameter domain of the surface, the boundary contours are oriented so that when moving along the boundary the surface is always to the left when looking in the direction opposite to the surface normal. Thus, the outer contour is oriented so that it is traversed counterclockwise when looking at the surface in the direction opposite to its normal, and the inner contours are oriented in the opposite direction.

The base surface **basisSurface** cannot be a surface with arbitrary boundaries MbCurveBoundedSurface. If a surface with arbitrary boundaries needs to be constructed on the basis of another surface with arbitrary boundaries, the base surface of the latter must be used as the base surface.

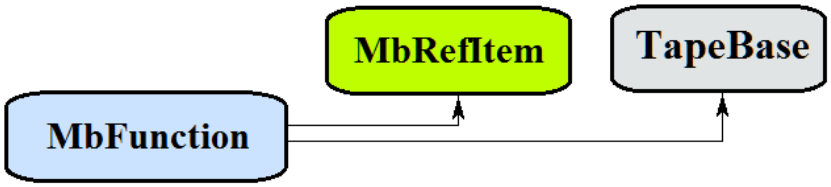
3.14 SPECIAL OBJECTS

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.

3.14.1 MbFunction Function

The MbFunction class is declared in the *function.h* file.

The MbFunction function is an abstract class derived from the *MbRefItem* and TapeBase classes, see the figure:



In the C3D geometric kernel, the following scalar functions are implemented, which are derived from the MbFunction class:

MbConctFunction – constant function,

MbLineFunction – linear function,

MbCubicFunction – cubic Hermite function,

MbCubicSplineFunction – cubic spline function,

MdCharacterFunction – analytical function.

MbFunction is a scalar function

$$function(t) = f(t)$$

of a scalar parameter t that takes values on the segment [t_{min}, t_{max}]. The range of variation of the function parameter is the segment [t_{min}, t_{max}] in a one-dimensional space. The function $f(t)$ must be single-valued and continuous.

The boundary values t_{min} and t_{max} of the parameter domain are returned by the methods double GetTMin() and double GetTMax(), respectively.

A function is called periodic if there exists $p > 0$ such that $f(t \pm kp) = f(t)$, where k is an integer. The IsClosed() method of a periodic function returns true. The double GetPeriod() method of a periodic function or a function that can be extended to a periodic one returns the period p. The domain of the parameter of a periodic function always lies within a single period.

The main method of the function is

double Value(double & *t*).

It returns the function value for the given parameter *t*.

The methods

double FirstDer(double & *t*),

double SecondDer(double & *t*),

double ThirdDer(double & *t*)

return the first, second, and third derivatives of the function, respectively, for the given parameter *t*. The listed methods adjust the function parameter when it goes beyond the domain. When the parameter *t* goes beyond the segment [t_{min}, t_{max}], non-periodic functions shift the

parameter *t* to the nearest boundary t_{min} or t_{max}, while periodic functions add or subtract the required number of periods.

The method

double _Value(double *t*)

returns the function value for the given parameter *t* both within the parameter domain and outside it. In the general case, a non-periodic function outside its parameter domain is continued along the tangent at the endpoint. Analytical functions are an exception. Periodic functions outside the parameter domain are continued cyclically.

The methods

double _FirstDer(double *t*),

double _SecondDer(double *t*),

double _ThirdDer(double *t*)

return the first, second, and third derivatives of the function, respectively, for the given parameter *t* both within the function domain and outside it.

Functions override the following methods:

methods that provide copying, checking for equality, checking the ability to make objects equal, and making objects equal,

MbFunction & Duplicate(),

Boolean IsSame(const MbFunction & **item**),

Boolean IsSimilar(const MbFunction & **item**),

Boolean SetEqual(const MbFunction & **item**),

a method that returns the type from the function enumeration,

MbeFunctionType IsA(),

methods that provide access to and editing of the internal data of the object,

MbProperty & CreateProperty(MbePrompt name),

void GetProperties(MbProperties & *properties*),

void SetProperties(MbProperties & *properties*).

All functions generally have no breaks. The MbAnaliticalFunction function defined by the user must also be continuous and have no singular points.

3.14.2 MbConstFunction Constant Function

The MbConctFunction class is declared in the *func_const_function.h* file.

The MbConctFunction constant function is defined by a single function value *value*.

In the double Value(double & *t*) method, the function

$$f(t) = value.$$

is evaluated.

The domain of the function parameter lies within $0 \le t \le 1$. The function cannot be periodic.

3.14.3 MbLineFunction Linear Function

The MbLineFunction class is declared in the *func_line_function.h* file.

The MbLineFunction linear function is defined by two boundary values of the function *value1*, *value2* and the boundary values of the parameter *tmin*, *tmax*.

In the double Value(double & *t*) method, the function

$$f(t) = value1(1 - t) + value2t.$$

is evaluated.

The domain of the function parameter lies within *tmin* ≤ *t* ≤ *tmax*. The function cannot be periodic.

3.14.4 MbCubicFunction Cubic Hermite Function

The MbCubicFunction class is declared in the *cur_cubic_function.h* file.

The Cubic Hermite function MbCubicFunction is defined by the set of control points *valueList*, the set of function derivatives at the control points *firctList*, the set of parameter values of the function at the control points *tList*, and the flag of periodicity of the function *closed*. The function also has some additional data that are optional and are used to speed up the work of the function methods.

For the parameter value *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, where *splinesCount* = *tList*.MaxIndex(), the Cubic Hermite function passes through the control point *valueList*[*i*] and has the derivative *firctList*[*i*] at this point. The function is constructed on the basis of *splinesCount* third-degree Hermite splines, which smoothly join each other. Each cubic Hermite spline describes a segment of the function between two neighboring control points. Each cubic Hermite spline is defined by two end points and two derivatives of the curve at these points.

When evaluating the function, first, by the value of the curve parameter *t*, the index *i* of the working segment (the index of the cubic Hermite spline) is determined from the condition *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1]. Then the function is evaluated for the found working segment using its local parameter *w*, which is defined by *tList*[*i*] and *tList*[*i* + 1].

In the double Value(double & *t*) method, the following function is used:

$$f(t) = (1 - 3w^2 + 2w^3)\, valueList[i] + (3w^2 + 2w^3)\, valueList[i + 1] +$$
$$+ \left((w - 2w^2 + w^3)\, valueList[i] + (-w^2 + w^3)\, valueList[i + 1]\right)(tList[i + 1] - tList[i]),$$

where

$$w = \frac{t - tList[i]}{tList[i + 1] - tList[i]}$$

is the local parameter of the working segment tList[i] ≤ t ≤ tList[i + 1].

The Cubic Hermite function is shown in Fig. 1:

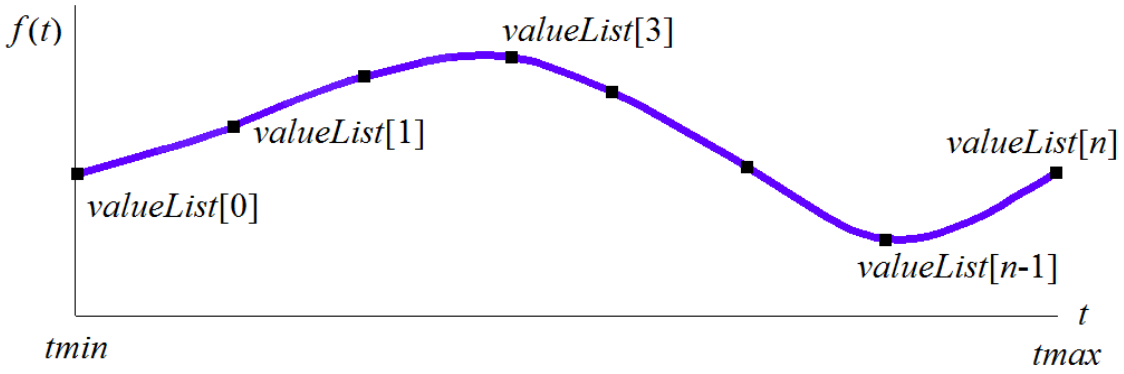


Fig. 1

The domain of the function parameter lies within *tmin* ≤ *t* ≤ *tmax*, where *tmin* = *tList*[0], *tmax* = *tList*[*splinesCount*]. The function can be periodic.

The shape of the function depends on the positions of the control points, on the function derivatives at the control points, and on the *tList* set of parameter values at the control points. When the function is constructed only by control points, the derivatives *firctList*[*i*] are computed by constructing a parabola that passes through three neighboring points *valueList*[*i* – 1], *valueList*[*i*], *valueList*[*i* + 1] at the corresponding parameter values *tList*[*i* – 1], *tList*[*i*], *tList*[*i* + 1], and by evaluating the derivative of this parabola at the middle point.

3.14.5 MbCubicSplineFunction Cubic Spline Function

The MbCubicSplineFunction class is declared in the *cur_cubic_spline_function.h* file.

The cubic spline function MbCubicSplineFunction is defined by the set of control points *valueList*, the set of second derivatives of the function at the control points *secondList*, the set of parameter values of the function at the control points *tList*, and the flag of periodicity of the function *closed*. The function also has some additional data that are optional and are used to speed up the work of the function methods.

For the parameter value *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, where *splinesCount* = *tList*.MaxIndex(), the cubic spline function passes through the control point *valueList*[*i*] and has the second derivative *secondList*[*i*] at this point.

When evaluating the function, first, by the value of the curve parameter *t*, the index *i* of the working segment is determined from the condition *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1]. Then the function is evaluated for the found working segment using its local parameter *w*, which is defined by *tList*[*i*] and *tList*[*i* + 1].

In the double Value(double & *t*) method, the following function is used

$$r(t) = (1-w)\cdot \text{valueList}[i] + w\cdot \text{valueList}[i+1] + \left((-2w+3w^2-w^3)\cdot \text{valueList}[i] + (-w+w^3)\cdot \text{valueList}[i+1]\right)\frac{(\text{tLis}}{}$$

where

$$w = \frac{t - \text{tList}[i]}{\text{tList}[i+1] - \text{tList}[i]}$$

is the local parameter of the working segment tList[i] ≤ t ≤ tList[i + 1].

The cubic spline function is shown in Fig. 1:

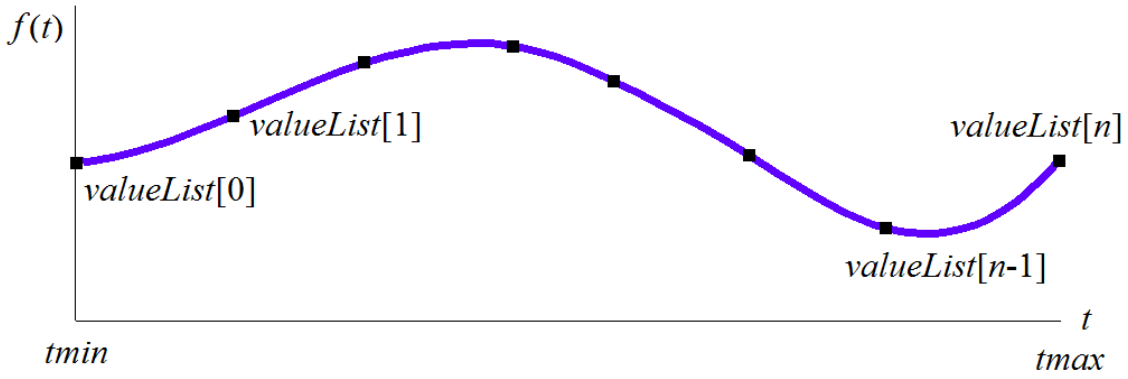


Fig. 1

The domain of the function parameter lies within *tmin* ≤ *t* ≤ *tmax*, where *tmin* = *tList*[0], *tmax* = *tList*[*splinesCount*]. The function can be periodic.

The shape of the function depends on the positions of the control points and on the *tList* set of parameter values at the control points. When the function is constructed by control points, the derivatives *secondList*[*i*] are computed under the assumption of linear variation of the second derivatives between the control points.

3.14.6 MdCharacterFunction Character Function

The MdCharacterFunction class is declared in the *func_analytical_function.h* file.

The character function MdCharacterFunction is defined by the string expression of the function, the action tree used to evaluate this expression, the boundary values of the parameter *tmin*, *tmax* and the direction *sense*. The function also has some additional data that are optional and are used to speed up the work of the function methods.

Using the character function, you can describe any function of the parameter *t* as a string expression containing analytical functions and arithmetic operations.

In the double Value(double & *t*) method, the action tree is used to evaluate the value of the character expression.

The domain of the function parameter lies within *tmin* ≤ *t* ≤ *tmax*. The function can be periodic.

3.14.7 Multiline MbMultiline

The MbMultiline class is declared in the *multiline.h* file.

The MbMultiline multiline is derived from the *MbPlaneItem* class, see Fig. 1:

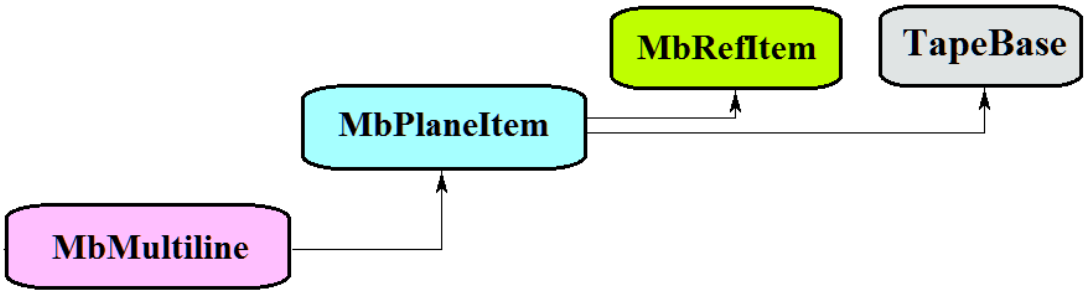


Fig. 1

The multiline is defined by the base contour **basisCurve**, the set of vertices **vertices**, the set of radii *equidRadii*, the parameters of the start edge of the multiline *begTipParams*, the parameters of the end edge of the multiline *endTipParams*, the flag of periodicity processing *processClosed*, the transparency flag *isTransparent*, the set of curves **curves**, the set of tips at the vertices of the multiline **tipCurves**, the tip at the start vertex of the multiline **begTipCurves**, and the tip at the end vertex of the multiline **endTipCurves**.

The multiline represents a contour that has thickness. The contour thickness is variable. The contour edges and the joints of the contour segments can have different shapes.

The multiline is shown in Fig. 2:

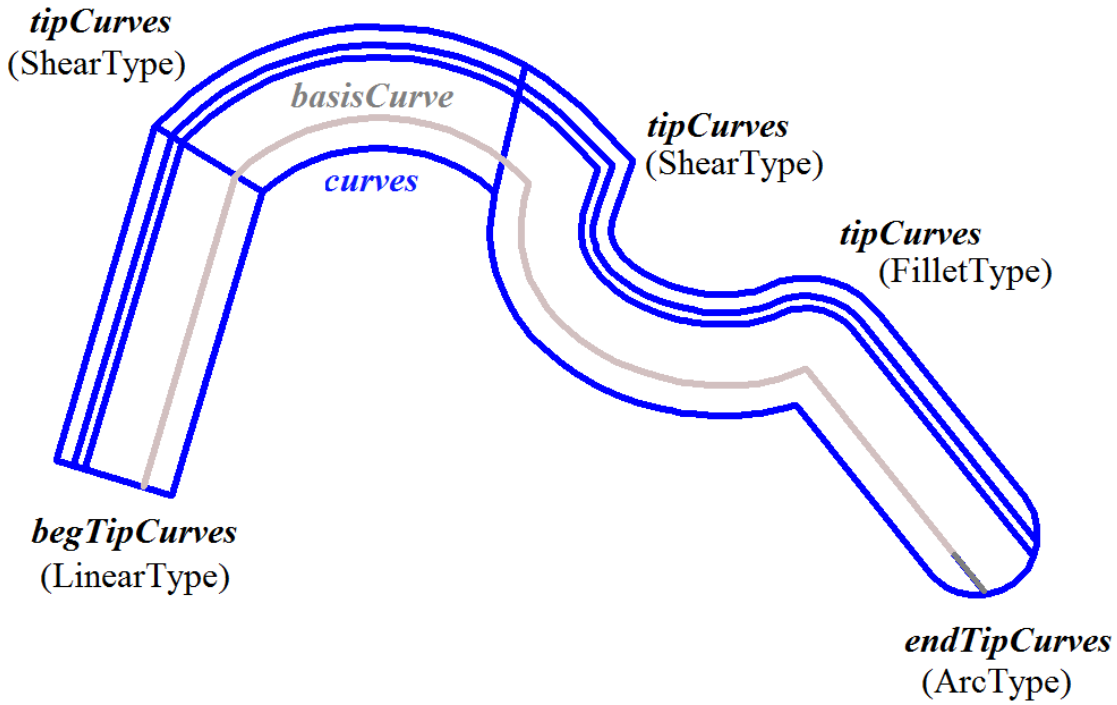


Fig. 2

The multiline is used for data exchange with other systems.

3.14.8 MbContourWithBreaks Two-Dimensional Contour with Breaks

Class MbContourWithBreaks is declared in file *cur_contour_with_breaks.h*.

A two-dimensional contour with breaks is a derived class of a two-dimensional contour *MbContour*, Fig. 1:

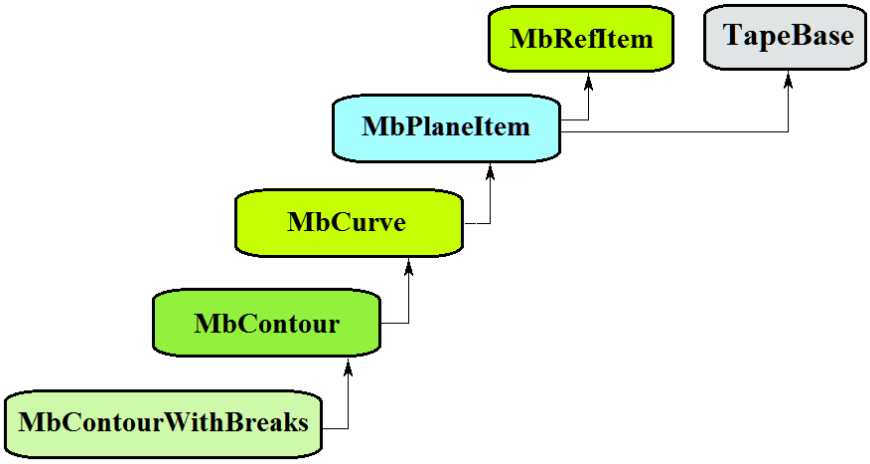


Fig. 1

A two-dimensional contour with breaks is defined by a set of **segments** of curves joined to each other, a periodicity flag *closed*, breaks **breaks**, a set of visible parts of the contour **visibleContours**, and a set of contour segment indices *baseSegNumbers* that define fixed break points.

A contour with breaks is shown in Fig. 2:

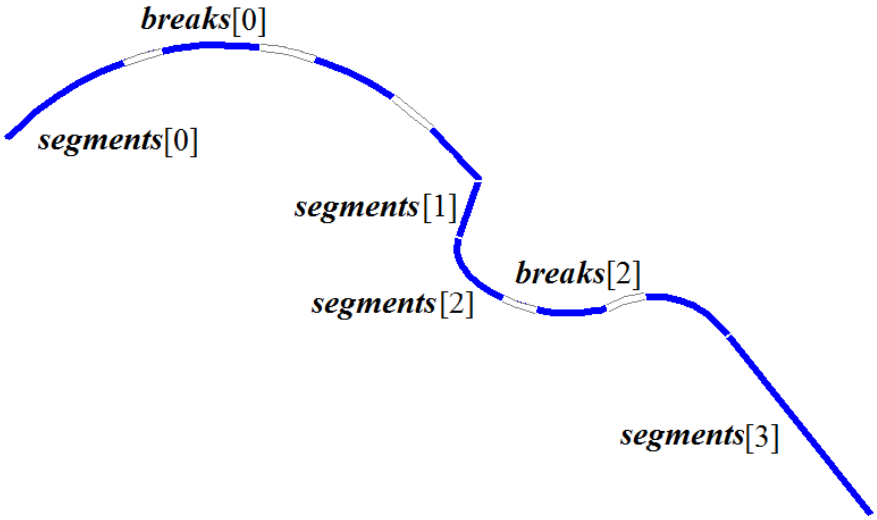


Fig. 2

Other two-dimensional contours must not be used as segments of a two-dimensional contour with breaks.

A two-dimensional contour with breaks is used only for building multilines.

3.14.9 MbRegion Region

Class MbRegion is declared in file *region.h*.

Region MbRegion is a derived class of *MbPlaneItem*, Fig. 1.

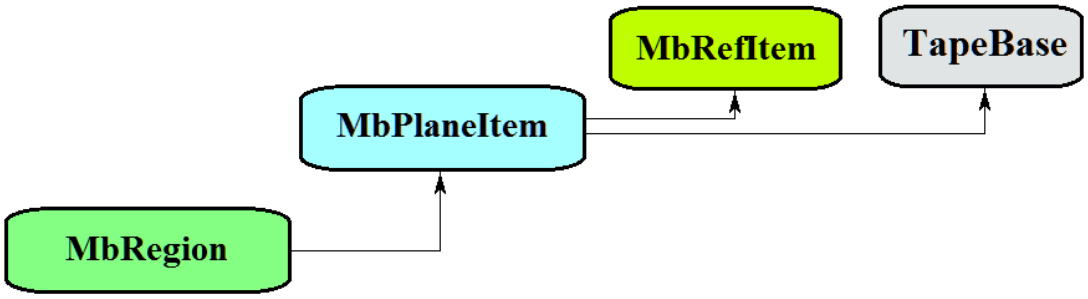


Fig. 1

A region is described by a set of contours **contours**. The contours of a region are closed and do not intersect themselves or each other. Among the contours of the set, one contour is external and the others are internal and lie inside the external contour. In the set **contours** the external contour is always the first one.

A region represents a connected set of points of a two-dimensional space whose boundaries are described by two-dimensional closed contours. The contours of a region are oriented so that when moving along any contour, the described point set lies to the left of the contour. That is, the external contour of a region is oriented counterclockwise, and the internal contours are oriented clockwise.

A region is shown in Fig. 2:

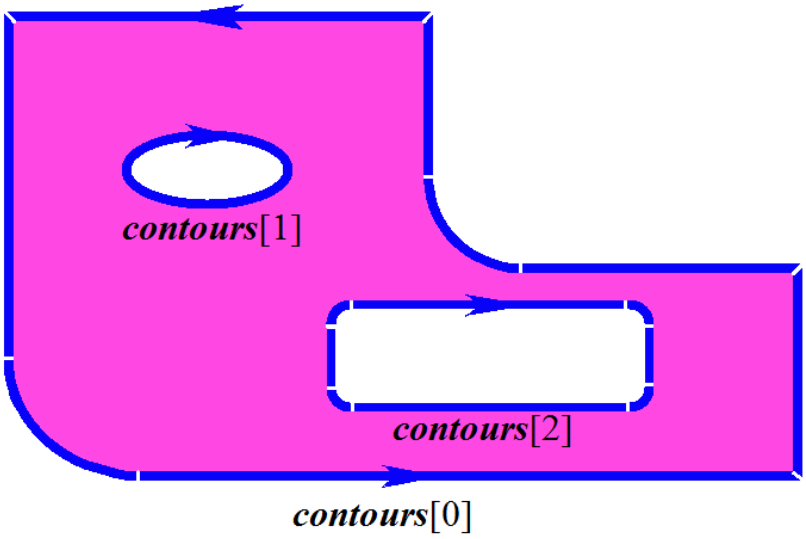


Fig. 2

A region is used to describe two-dimensional connected areas. Boolean operations can be performed on regions.

3.14.10 MbLegend Auxiliary Geometric Object

Class MbLegend is declared in file *legend.h*.

Auxiliary object MbLegend is a derived class of *MbSpaceItem*, Fig. 1.

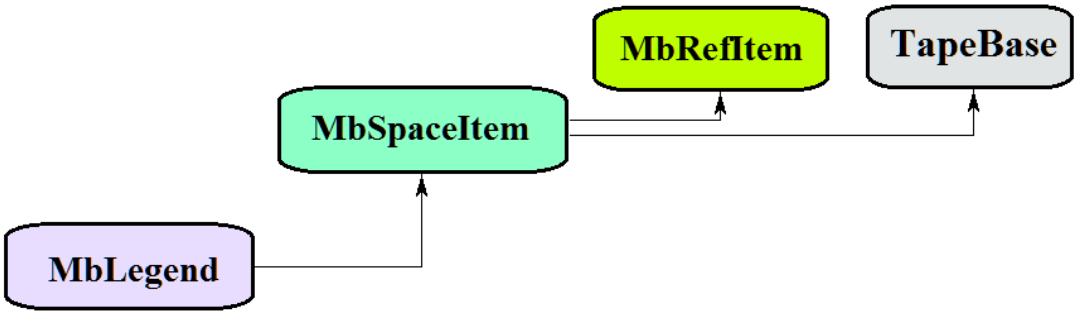


Fig. 1

The auxiliary object is an abstract class. The C3D geometric kernel implements the following auxiliary objects that are derived from class MbLegend:

MbMarker – a point and two orthonormal vectors,

MbThread – a thread,

MbPointsSymbol – a conventional symbol on base points,

MbRough – a surface finish symbol,

MbLeader – a leader line symbol.

Auxiliary objects are used for various purposes, but they all interact with curves, surfaces, and objects of a geometric model.

Auxiliary objects override the following methods of a three-dimensional geometric object:

methods that handle transformation of a geometric object,

Move(const MbVector3D & **v**, MbRegTransform * iReg = NULL),

Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),

Transform(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),

methods that provide copying, equality check, ability-to-match check, and make objects equal,

MbSpaceItem & Duplicate(MbRegDuplicate * iReg = NULL),

bool IsSame(const MbSpaceItem & **item**),

bool IsSimilar(const MbSpaceItem & **item**),

bool SetEqual(const MbSpaceItem & **item**),

methods that return a type from the enumeration of geometric objects,

MbeSpaceType IsA(),

MbeSpaceType Type(),

MbeSpaceType Family(),

methods that provide access to and editing of internal data of an object,

MbProperty & CreateProperty(MbePrompt name),

GetProperties(MbProperties & *properties*),

SetProperties(MbProperties & *properties*),

a method that fills a polygonal copy of a geometric object,

CalculateWire(double sag, MbMesh & **mesh**).

3.14.11 MbMarker Marker

Class MbMarker is declared in file *marker.h*.

Auxiliary object marker MbMarker is defined by a point **origin** and two orthogonal vectors **axisZ**, **axisX**.

A marker is used to apply geometric constraints to objects of a three-dimensional space. A marker is a representative of a geometric object and can replace a three-dimensional point, a line, a plane, a local coordinate system, and other objects when geometric constraints are applied.

3.14.12 MbThread Thread Graphic Symbol

The MbThread class is declared in the *mb_thread.h* file.

The MbThread thread graphic symbol is defined by the local thread coordinate system place, the initial thread radius on the surface *radObj*, the initial thread radius inside the body *radThr*, the thread length *length*, the thread taper angle *angle*, the thread name **name**, and the set of bodies **bodies**. The object also has some additional data that is not mandatory and is used to speed up the operation of the object methods.

The axis of the threaded connection is directed along the **place**.*axisZ* axis. The name is used to identify the thread graphic symbol among planar projections of the faces of bodies in the **bodies** set. The thread graphic symbol is shown in Fig. 1.

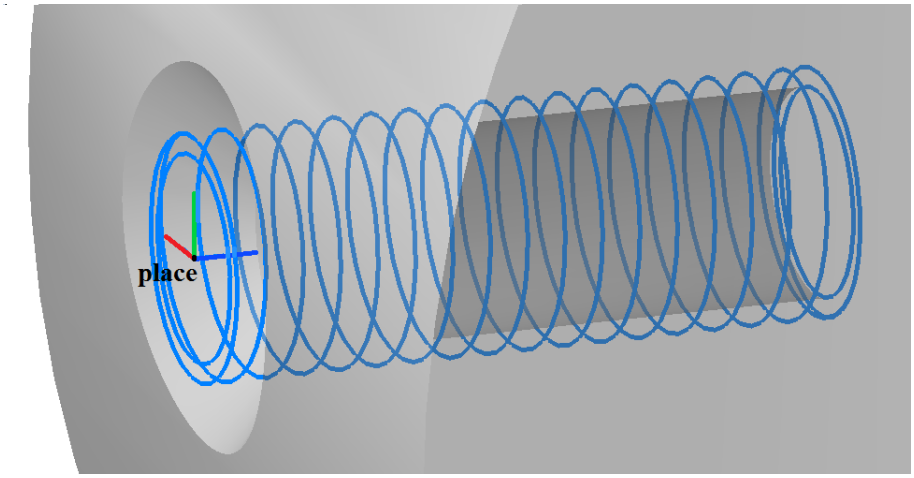


Fig. 1

The thread graphic symbol describes an element of the threaded connection of the geometric model and is used to construct planar projections of threaded connections.

3.14.13 MbPointsSymbol Symbol

The MbPointsSymbol class is declared in the *mb_symbol.h* file.

The MbPointsSymbol symbol is a derived class of MbSymbol. The MbPointsSymbol symbol is defined by a set of identifiers, a set of three-dimensional points **points**, and data about segments of complex sections *steps*.

The object carries information about the base points of symbols that are associated with elements of the geometric model.

The object is used when constructing planar projections of symbols for which it is sufficient to know the positions of base points. It defines the base points of symbols for elements of the geometric model.

3.14.14 MbRough Surface Finish Symbol

The MbRough class is declared in the *mb_rough.h* file.

The MbRough surface finish symbol is a derived class of MbPointsSymbol. The surface finish symbol is defined by a set of three-dimensional points **points**, data about segments of complex sections *steps*, and a topological attachment object **item**.

The object carries information about the base points of the surface finish symbol that are associated with the topological object **item**.

The object is used when constructing planar projections of the surface finish symbol of the geometric model element **item**.

3.14.15 MbLeader Leader Line Symbol

The MbLeader class is declared in the *mb_rough.h* file.

The MbLeader leader line symbol is a derived class of MbSymbol. The leader line symbol is defined by a set of identifiers and a set of surface finish symbols **branches**.

The object carries information about the leader line of the surface finish symbol for topological objects.

The object is used when constructing planar projections of the leader line surface finish symbol for elements of the geometric model.

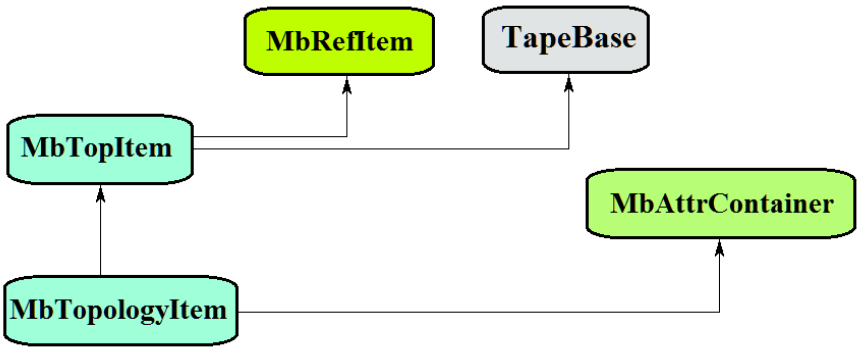
3.15 TOPOLOGICAL OBJECTS

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.

3.15.1 MbTopologyItem Topological Object

The MbTopologyItem class is declared in the *topology_item.h* file.

The MbTopologyItem topological object differs from other topological objects by having a **name**, a modification flag *changed*, and a marker *label*. The MbTopologyItem topological object is a derived class of the *MbTopItem* topological object.



The MbAttrContainer attribute container endows named topological objects with attributes.

The C3D geometric kernel implements the following topological objects that are derived from the MbTopologyItem class:

MbFace – face,

MbEdge – edge,

MbVertex – vertex.

The MbEdge edge has a derived class *MbCurveEdge* – a face-joining edge.

Named topological objects have the following methods:

methods that handle transformation of a 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 that handle the name of a topological object:

MbName & GetName(),

SimpleName & GetMainName(),

SimpleName & GetFirstName(),

a method that returns a type from the enumeration of topological objects:

MbeTopologyType IsA().

Named topological objects are used as elements for constructing geometric model objects.

3.15.2 MbFace Face

The MbFace class is declared in the *topology.h* file.

The MbFace face is a derived class of the *MbTopologyItem* topological object and represents a finite portion of a surface with an assigned normal direction and defined boundaries. The side of a surface and of a face that we see when looking along the normal direction is called the outer side, and the other side is called the inner side. The sides of the *MbSurface* surface are not equivalent with respect to the normal, because one side of the surface is always outer and the other side is always inner. In contrast to a surface, for a face we can assign the normal direction and thereby define outer and inner sides.

The face data structure contains a pointer to the surface *MbSurface* **surface**, a flag indicating coincidence of the face normal direction with the surface normal direction **sameSense**, and a set of face loops RPArray<*MbLoop*> **loops**. A face also has some additional data that is not mandatory and is used to speed up execution of face methods.

The surface pointer cannot be null. The flag indicating coincidence of the face normal direction with the surface normal direction is set to true if the face and surface normals coincide; otherwise, the flag is set to false. The side of the face that we see when looking along the normal direction is called the outer side, and the other side of the face is called the inner side.

Face loops define the boundaries of the face. Each face boundary is closed. Each loop is defined by a sequence of *MbOrientedEdge* edges following one another along the boundary. The number of face loops equals the number of boundaries of the face surface. One of the face boundaries is outer and contains the boundaries of inner holes. The loop that appears first in the loop container describes the outer boundary of the face and contains inside it inner loops that describe inner boundaries of the face. The outer loop of the face is oriented counterclockwise, and inner loops are oriented clockwise when viewed along the face normal. Thus, when moving along a face loop on its outer side, the face always lies to the left. In Fig. 1, arrows indicate the directions of the face loops and its normal.

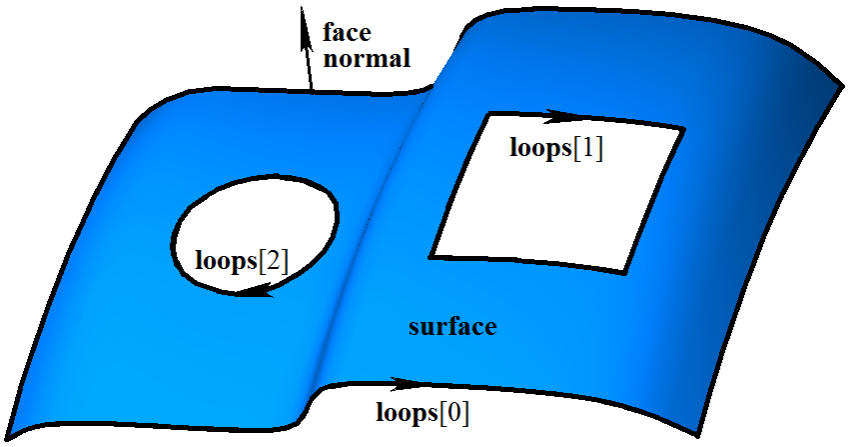


Fig. 1

Loops of a single face must not intersect one another or themselves.

A face can be named and can have attributes. The name can be used to identify the face, and attributes can carry additional information about the face, such as color, transparency, origin, and so on.

A face is used for solid and hybrid modeling.

3.15.3 MbEdge Edge

The MbEdge class is declared in the *topology.h* file.

The MbEdge edge is a derived class of the MbTopologyItem topological object and represents a curve with an assigned direction. The direction of the MbCurve3D curve is strictly linked to the direction of increase of its parameter. Unlike a curve, an edge can be directed both in the direction of increasing the parameter and in the direction of decreasing the curve parameter. An edge always starts and ends at some *MbVertex* vertex.

The edge data structure contains a pointer to the curve MbCurve3D **curve**, a flag indicating coincidence of the edge direction with the curve direction *sameSense*, a pointer to the start vertex MbVertex **begVertex**, and a pointer to the end vertex MbVertex **endVertex**. Fig. 1 shows an edge.

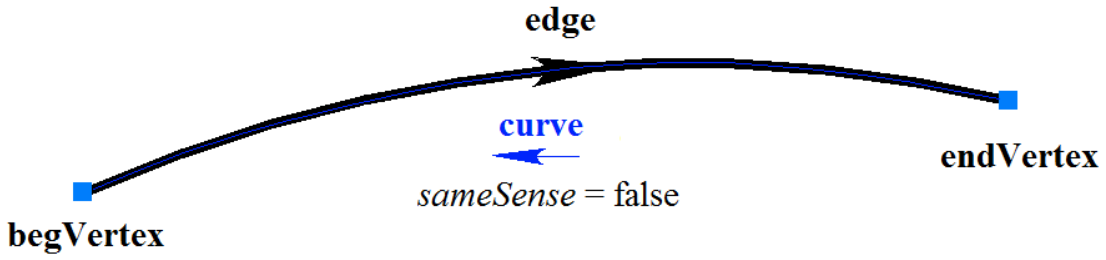


Fig. 1

Pointers to the curve and vertices cannot be null. The flag indicating coincidence of the edge

direction with the curve direction is set to true if the edge coincides with the curve, and to false if the edge is directed opposite to the curve. If an edge starts and ends at the same vertex, then such an edge is closed. Pointers to the start and end vertex of a closed edge are equal.

The edge can be named and can have attributes. The name can be used to identify the edge, and attributes can carry additional information about the edge, such as color, display style, origin, and so on.

The edge is used for wireframe modeling.

3.15.4 MbVertex Vertex

The MbVertex class is declared in the *topology.h* file.

The MbVertex vertex is a derived class of the MbTopologyItem topological object and represents a point with a known positioning tolerance. The vertex data structure contains a *MbCartPoint* point **point** and a positioning tolerance *tolerance* for this point.

A vertex can represent an individual point of a point frame or a joint of edges. Any finite number of edges can meet at a vertex. Joining edges refer to the same vertex shared by them. Fig. 1 shows a vertex where three edges meet.

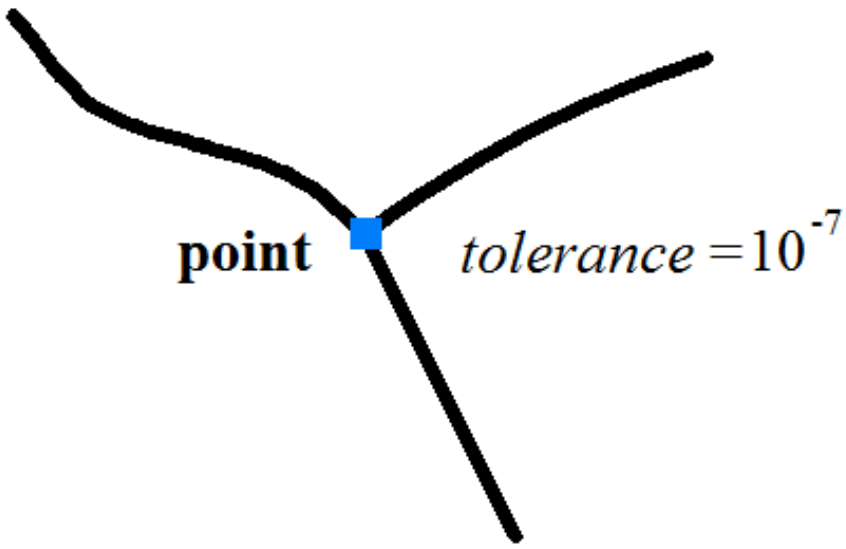


Fig. 1

In the case of an inaccurate junction of edges, the positioning tolerance of a vertex equals the distance from the vertex point to the end of the most distant edge. Fig. 2 shows a tolerant vertex where four edges meet.

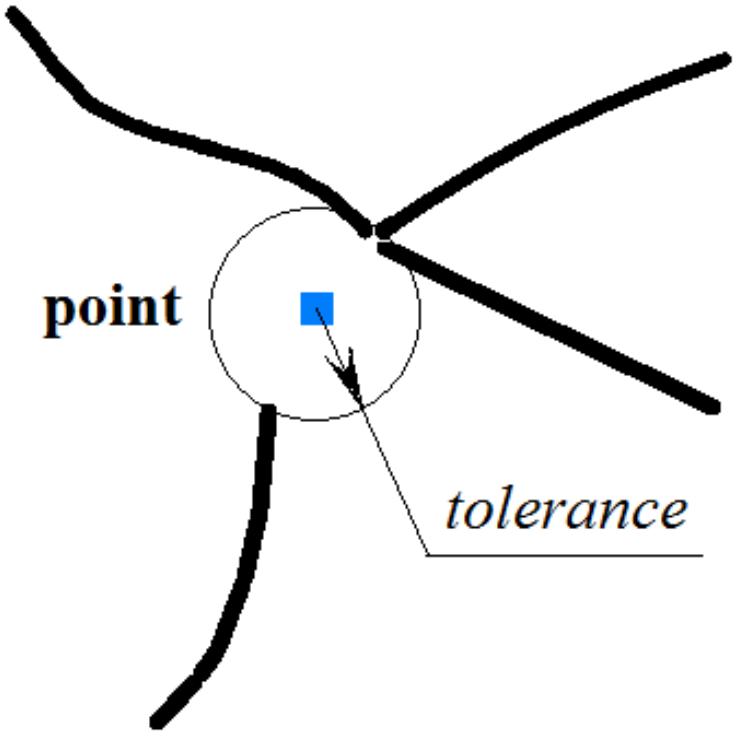


Fig. 2

A vertex can be named and can have attributes. The name can be used to identify the vertex, and attributes can carry additional information about the vertex, such as color, display style, origin, and so on. A vertex is used in all modeling methods.

3.15.5 MbCurveEdge Face Edge

Class MbCurveEdge is declared in file *topology.h*.

Edge MbCurveEdge is a derived edge of MbEdge and represents an edge constructed on a surface intersection curve *MbSurfaceIntersectionCurve*. Edge MbCurveEdge is intended to describe a portion of a face boundary. Unlike edge MbEdge, edge MbCurveEdge describes not just a curve, but a portion of the junction of two faces or a portion of a face border.

The data structure of the face edge contains a pointer to the surface intersection curve *MbSurfaceIntersectionCurve* **curve**, a flag indicating coincidence of the edge direction with the curve direction *sameSense*, a pointer to the start vertex MbVertex **begVertex**, a pointer to the end vertex MbVertex **endVertex**, a pointer to the face on the left MbFace **facePlus** and a pointer to the face on the right MbFace **faceMinus** of the edge. Fig. 1 shows an edge connecting two faces.

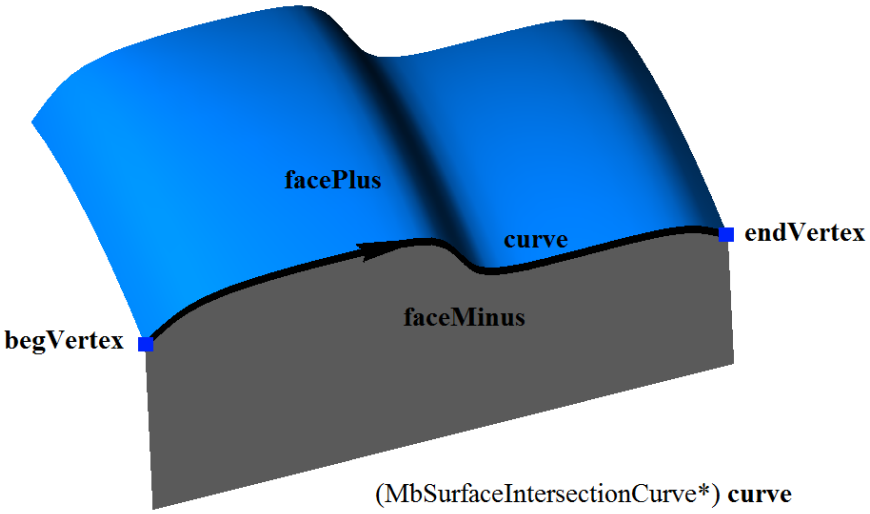


Fig. 1

A face edge can describe a portion of the junction of two different faces. In this case, the pointers to the face on the left and the face on the right of the edge are not null and not equal to each other. The surfaces stored in the data structures of the faces connected by the edge coincide with the surfaces stored in the data structure of the edge curve, namely:

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.

For a face of a surface that is cyclically closed in one or both parameters, there are boundary segments where the face joins to itself. Such an edge is a seam. In this case, the pointers to the face on the left and the face on the right of the edge are equal to each other, see Fig. 2:

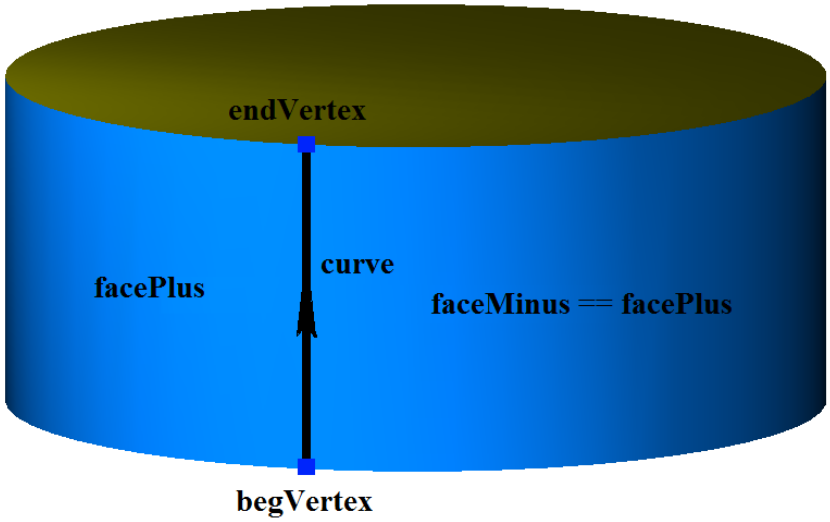


Fig. 2

A face edge can describe a portion of a face border. Such an edge is a border edge. In this case, the pointer to the face on the left or the face on the right of the edge is null, see Fig. 3.

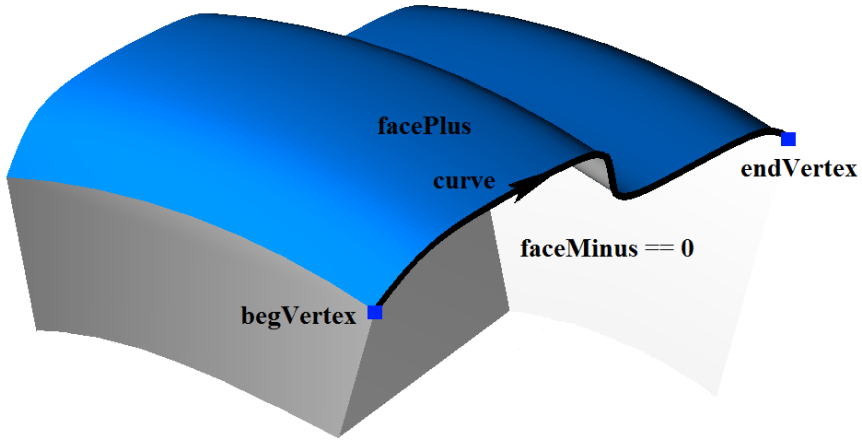


Fig. 3

An edge with zero length is a pole edge and describes a face pole. A pole edge is not a border edge, because the face has no border at the pole edge. As a rule, a pole edge is located at special points of the face surface. In the parametric domain of the face surface, a certain curve corresponds to the pole segment. The pointers to the start and end vertex of a pole edge are equal. A pole edge is shown in Fig. 4:

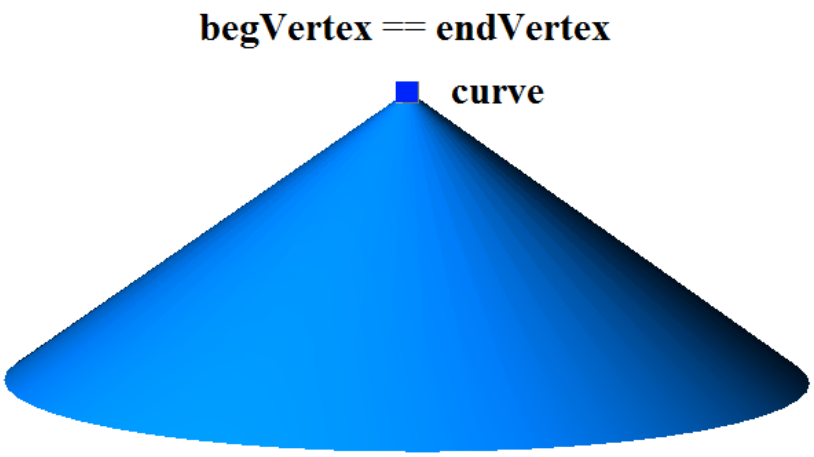


Fig. 4

For a pole edge, the pointer to the face on the left or the face on the right of the edge is null. The intersection curve of the pole edge has the following data:

curve→curveOne.suface == **curve**→curveTwo.suface, and segment

curve→curveOne.curve is a duplicate of segment **curve**→curveTwo.curve.

A face edge may be named and may have attributes. The name can be used to identify the edge; attributes can carry additional information about the face edge, for example, color, style, origin, and so on.

A face edge is used for solid and hybrid modeling.

3.15.6 MbLoop Face Cycle

Class MbLoop is declared in file *topology.h*.

Face loop MbLoop is a derived class of the topological object *MbTopItem* and describes a sequence of edges that exhausts a certain face boundary.

The data structure of the loop contains a set of oriented edges RPArray<*MbOrientedEdge*> **edgeList** in the order of their traversal along the face boundary. The loop also has some additional data that is optional and serves to speed up loop methods.

The directions of the oriented edges coincide with the loop direction. The end of each oriented loop edge is joined with the start of the next oriented loop edge, see Fig. 1:

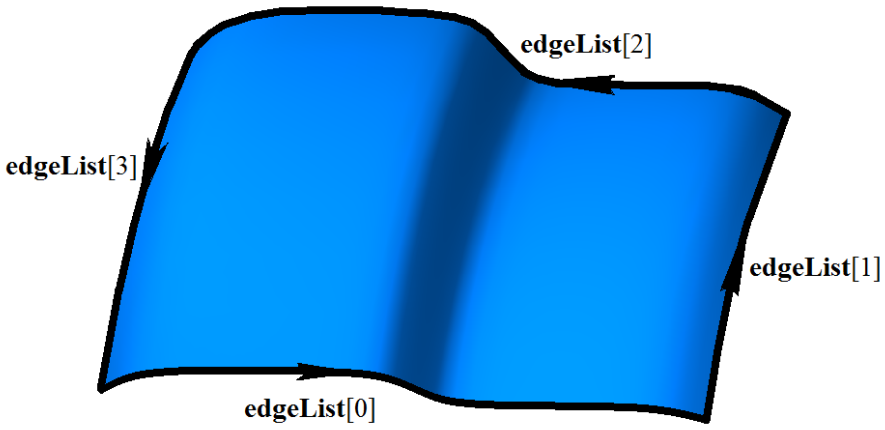


Fig. 1

The end of the last oriented loop edge is joined with the start of the first oriented loop edge.

Fig. 2 shows a loop of a spherical face consisting of four oriented edges, two of which are constructed on a seam edge and two of which are pole edges:

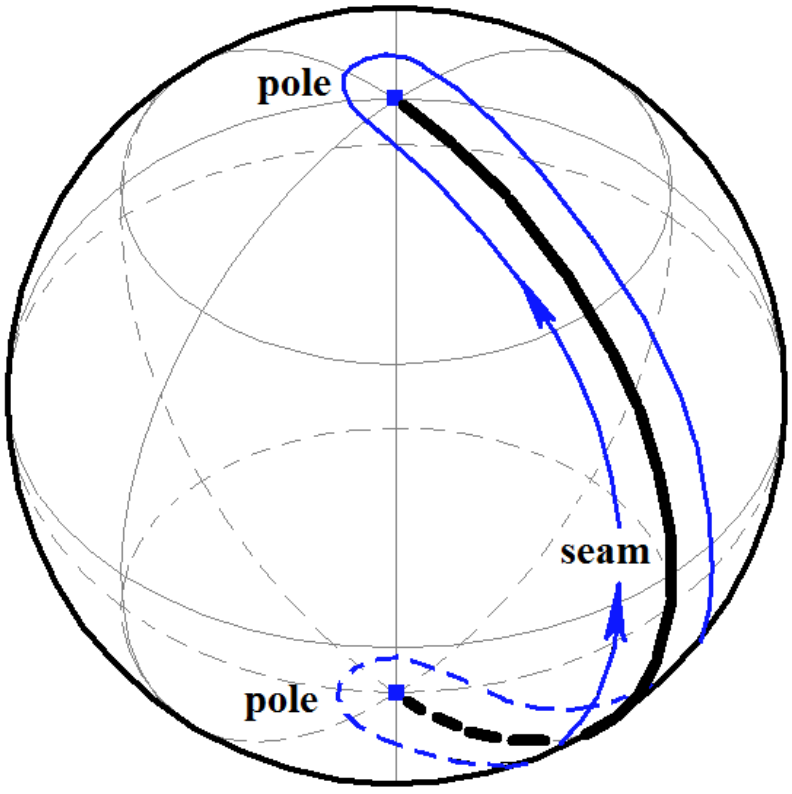


Fig. 2

If the loop of a spherical face is drawn in the parametric space of the sphere, it will be a rectangle. The loop, as well as the face boundary, is always closed. The loop is directed in such a way that the face is always on the left side when moving along the loop on the external side of the face.

3.15.7 MbOrientedEdge Oriented Face Edge

Class MbOrientedEdge is declared in file *topology.h*.

Oriented edge MbOrientedEdge is a derived class of the topological object *MbTopItem* and describes a portion of a face boundary. The data structure of an oriented edge contains a face edge MbCurveEdge *curveEdge* and a flag indicating coincidence of the face edge direction with the oriented edge direction and, accordingly, with the face loop direction *orientation*.

If the direction of the face edge *curveEdge* coincides with the loop direction, then in the loop of this face the corresponding oriented edge has flag *orientation* = true. If the direction of the face edge *curveEdge* does not coincide with the loop direction, then in the loop of this face the corresponding oriented edge has flag *orientation* = false. Fig. 1 shows two oriented edges constructed on the same face edge *curveEdge*.

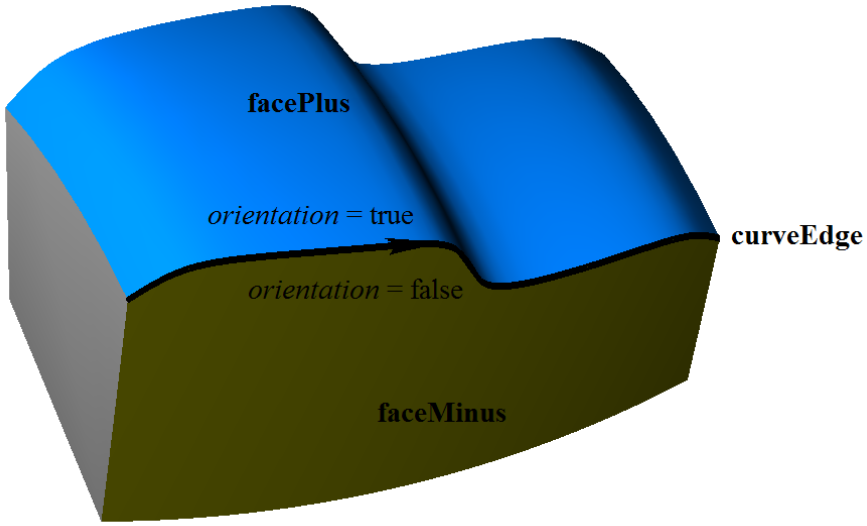


Fig. 1

In the general case, a face edge MbCurveEdge belongs to two loops; these loops belong to the faces joined by the edge. In one loop, the face edge enters with orientation flag *orientation* = true, this face is located on the left of the edge and the data field **facePlus** points to it. In the other loop, the face edge enters with orientation flag *orientation* = false, this face is located on the right of the edge and the data field **faceMinus** points to it. Thus adjacent faces and the edges connecting them are linked to each other.

3.15.8 MbFaceShell Set of Faces

Class MbFaceShell is declared in file *topology_faceset.h*.

Set of faces MbFaceShell is a derived class of the topological object *MbTopItem*. The data structure of MbFaceShell contains a set of faces RPArray<*MbFace*> **faceSet** and a closedness flag *closed* of the face set. The face set also has some additional data that are optional and serve to speed up methods of the face set.

The face set usually describes a connected portion of the surface of a modeled object. Faces of the set that are connected to each other satisfy certain conditions, namely: faces join each other along common edges; at each edge only two faces meet; moreover, faces join so that the outer side of one face passes into the outer side of another face, and the common surface formed by the adjoining faces must not have self-intersections.

Fig. 1 shows a set of faces connected to each other. All face edges belong to two loops; for all edges the pointers **facePlus** and **faceMinus** are not null, and the curves *MbSurfaceIntersectionCurve* on which the face edges are constructed have **curveOne.surface** != **curveTwo.surface**.

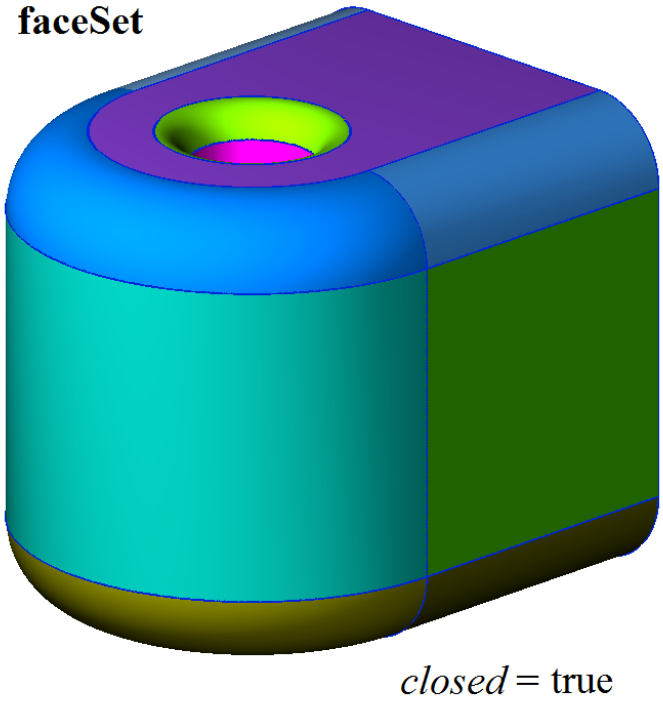


Fig. 1

The faces shown in the figure form a common closed surface; the outer side of each face passes into the outer side of the neighboring face. For a face set that has no boundary edges, the *closed* flag takes the value true.

If at least one face of the set has at least one boundary edge, then the *closed* flag of the face set takes the value false. Fig. 2 shows a set of faces, some of which have boundary edges. Boundary edges belong only to one loop, and for boundary edges of a face only one of the pointers **facePlus** or **faceMinus** is not null, while the curves *MbSurfaceIntersectionCurve* on which the edges are constructed have **curveOne.surface** == **curveTwo.surface** and parameter *buildType = cbt_Boundary*. A pole edge is not a boundary edge, because it does not form a border.

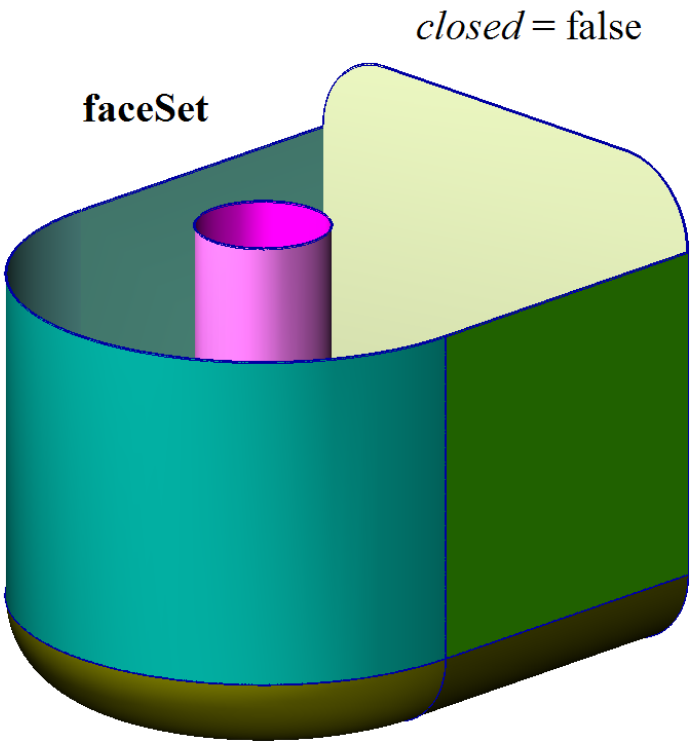


Fig. 2

A connected face set is called a *closed shell* if the faces of the set have no border. If faces adjoining each other have at least one boundary edge, then such a connected face set is called an *open shell*. Note that a closed shell and an open shell represent a connected set of faces adjoining each other.

A face set may consist of several parts that are not connected to each other. Fig. 3 shows a face set describing two open shells.

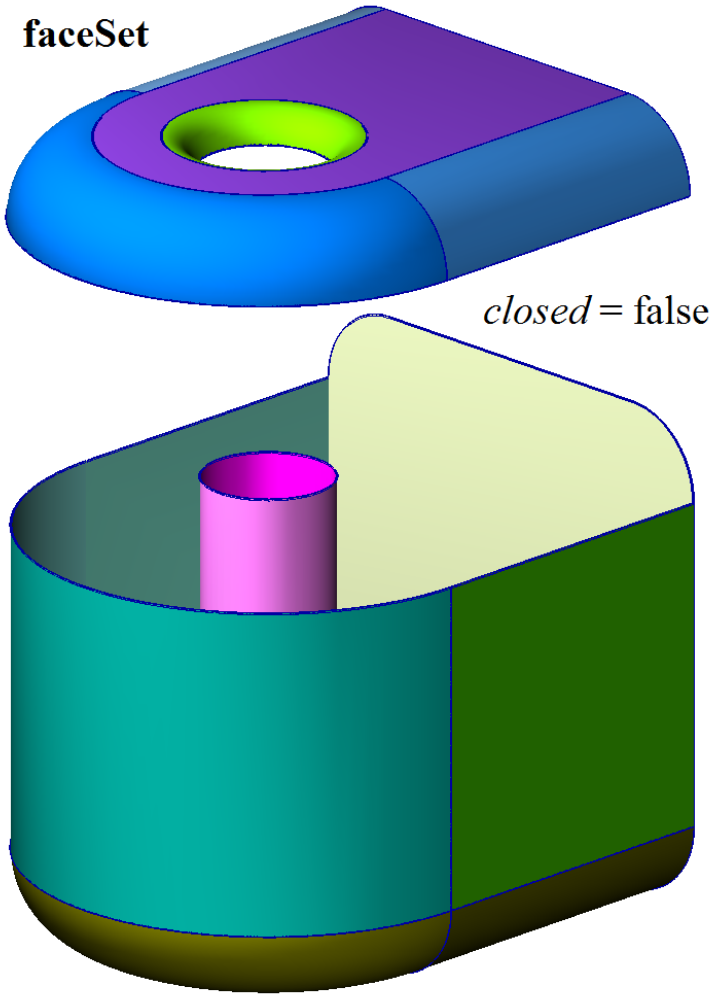


Fig. 3

Formally, no constraints are imposed on faces of the set. A face set may consist of separate faces. Fig. 4 shows a set of faces that are not connected to each other. Each face shown in the figure forms a separate shell; all edges of the faces are boundary edges.

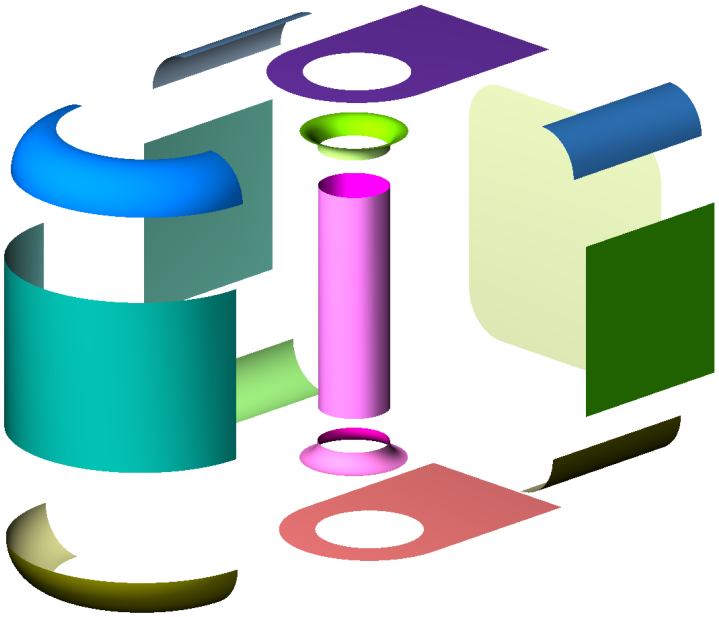


Fig. 4

The main methods of a shell are methods for transforming it 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),

methods for searching faces, edges, and vertices, as well as methods for determining the position of a point relative to a closed shell:

bool DistanceToBound(const *MbCartPoint3D* &, ...),

bool PointClassification(const *MbCartPoint3D* &, ...).

An open shell describes only a part of the boundary surface of a modeled object. Open shells are used for surface modeling. If a closed shell is supplemented with a set of points lying inside it, then we obtain a *solid body*. Closed shells are used for solid modeling. Trimming and joining of faces are performed simultaneously with shell construction. This is provided by methods of the C3D geometric kernel. Objects of a geometric model are constructed on the basis of closed and open shells.

3.15.9 Copying a Set of Faces MbFaceShell

Each construction method that uses a set of faces in its input parameters, represented as *MbFaceShell* or *MbSolid*, modifies some vertices, edges, and faces of the source objects. To speed up constructions and keep the source face set unchanged, full or partial copying of MbFaceShell data is applied. The C3D geometric kernel uses four ways of copying a MbFaceShell face set, defined by the MbeCopyMode enumeration. As a rule, construction methods receive, together with the modified face set, a parameter of type MbeCopyMode that controls the transfer of faces, edges, and vertices from the source object to the constructed object.

Enumeration MbeCopyMode is declared in file *mb_enum.h*. A parameter of type MbeCopyMode can take one of four values: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*.

With value *cm_Copy*, the source face set of the modified object is fully copied, so the source object and the constructed object do not share surfaces, curves, faces, edges, vertices, or other objects. This option is used when the constructed object must not be linked to the source object.

With value *cm_KeepSurface*, the source object and the constructed object share the same base face surfaces. This option is used when high construction speed is required.

With value *cm_KeepHistory*, the source object and the constructed object share the same vertices, base face surfaces, and faces that are not modified by the performed construction or other operation. This option is used when maximum memory saving is required.

With value *cm_Same*, all necessary data of the source object are transferred to the constructed object, therefore after construction the source object must be deleted. This option is used when the source object is no longer needed and was created specifically for the performed construction.

Enumeration MbeCopyMode is used in the method for copying a face set MbFaceShell MbFaceShell::Copy(MbeCopyMode, MbShellHistory). This method is used in solid construction operations whose input parameters include other solids.

With enumeration value *cm_Copy*, the source face set and its copy have no common data. The variant in which the source face set and its copy share common base surfaces corresponds to enumeration value *cm_KeepSurface*. The variant in which the source face set and its copy share common base surfaces, vertices, and faces not changed by the operation corresponds to enumeration value *cm_KeepHistory*. In this case, method Copy(...) uses a pointer to object MbShellHistory, which stores the correspondence between the source face set and its copy. After the operation has been performed, the copy of the face set is passed as a parameter to method MbShellHistory::SetOrigins(MbFaceShell &) to replace unchanged faces in the copy with their originals from the source face set. The variant in which the face set is not copied in method Copy(...) corresponds to enumeration value *cm_Same*. It should be noted that if the operation fails, the source face set will be modified.

To copy a face set MbFaceShell, method Duplicate(MbRegDuplicate * iReg) can be used. Object MbRegDuplicate is used to preserve in the copy the structure of mutual references that is present in the source face set. The copied face set and its copy will not be linked to each other.

3.15.10 Naming of Faces, Edges and Vertices

Faces, edges, and vertices in the data structure have a name MbName. Class MbName is declared in file *name_item.h*. Naming of faces, edges, and vertices is performed by the C3D geometric kernel when constructing a face set *MbFaceShell* in all shape-forming operations. In the parameters of each shape-forming operation there is an object MbSNameMaker. Object MbSNameMaker contains the main name of the operation and a container of simple names of type SimpleName that are used for naming faces. Object MbSNameMaker assigns names to the constructed faces using elements of the simple-name container. The simple-name container may contain either a set of unique integers, or a single integer, or no integers at all.

If the simple-name container contains a single integer, then MbSNameMaker will generate the remaining required simple names itself by adding integers of the natural sequence to the initial value. If the simple-name container is empty, then the starting integer is zero. Each integer from the simple-name container corresponds to one of the geometric input parameters (a segment of the operation's shape-forming contour, an edge processed by the operation, a face modified by the operation, and so on) and is used to name a new face created by the operation.

Face names are unique if the elements of the simple-name container are unique. Edges obtain names by hashing the names of the faces they connect. Vertices obtain names by hashing the names of the edges that meet at the vertex. As an identifier of a face, edge, or vertex one

can use the integer returned by method MbName::Hash() of the corresponding topological object. Faces, edges, and vertices in a solid can be searched by a known MbName using methods FindFaceByName(...), FindEdgeByName(...), FindVertexByName(...).

In addition, object MbSNameMaker contains the version of the construction method and ensures preservation of old construction methods when they are modified during evolution of the geometric kernel.

3.15.11 Operation Principles of Change Flags of Topological Objects

All topological objects (MbTopologyItem) have change flags. Before each operation, the flags are set to the *tct_Unchanged* state, which means that there are no changes.

There are the following change flags for topological objects:

tct_Unchanged — the object was not changed during an operation;

tct_Modified — any object modification sets this flag;

tct_Created — the object was created. The solid did not have this object, and then it was added (for example, when creating an elementary solid by the ElementarySolid() function);

tct_Transformed — the object was moved (for example, when applying the move MbSolid::Move() to the solid geometry, all objects also move);

tct_Reoriented — the face was created as a copy of another face, with the direction of the normal vector changed to the opposite, that is, the face "turned inside out". Example: during a Boolean operation, a new face in the original solid was created from a copy of the cutting solid face whose normal direction was changed;

tct_Deleted — when deleting child elements of an object or marking the object itself for deletion (for example, when deleting an edge cycle (when it is replaced in an operation) or a face surface);

tct_Truncated — an object (face, edge) was truncated in the operation;

tct_Merged — the flag is relevant for edges (MbEdge), an edge was merged or stitched;

tct_Replaced — a face surface or the face edge cycle was replaced during the operation;

tct_Added — a child element was added;

tct_Renamed — an object was renamed during an operation.

Let us consider the use of flags by the example of performing the SolidCutting operation (see Fig. 1). The large cube is cut by the small parallelepiped:

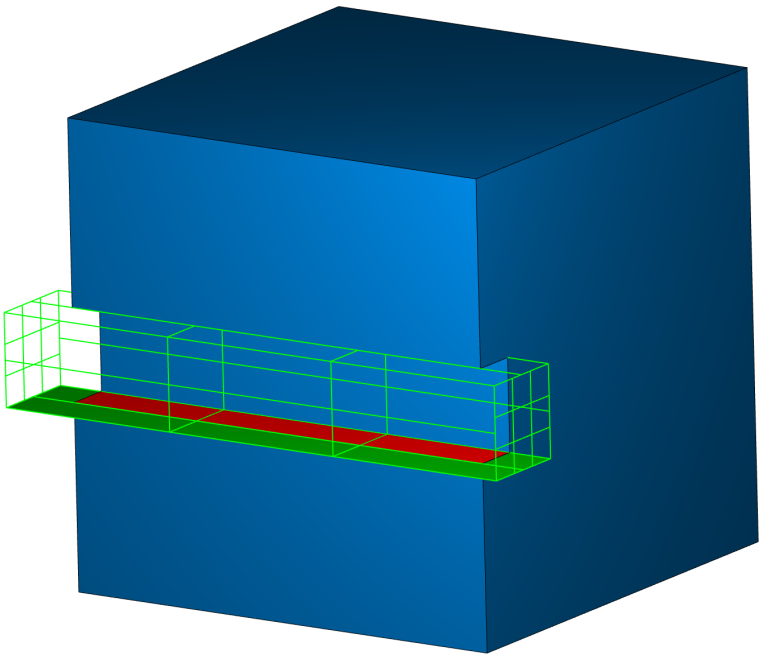


Fig. 1

Let us consider the lower face of the groove in the resulting solid (highlighted in red).

After the operation, it has the following flags:

tct_Modified, *tct_Created*, *tct_Reoriented*, *tct_Deleted*, *tct_Truncated*, *tct_Added*

This means the following:

1. The face was modified in some way in the performed operation (*tct_Modified*).
2. The solid did not have this face previously, and then the face was created (*tct_Created*).
3. In fact, the face was created from the cutting solid face (highlighted in green), but the normal vector was reversed (*tct_Reoriented*). The full name of this face completely coincides with the name of the cutting solid face.
4. The face was truncated (*tct_Truncated*) by removing (*tct_Deleted*) and adding (*tct_Added*) the face edge cycle.

Next, let us perform another cutting with the same cutting solid. Before the operation, we change the position of the cutting parallelepiped (Fig. 2):

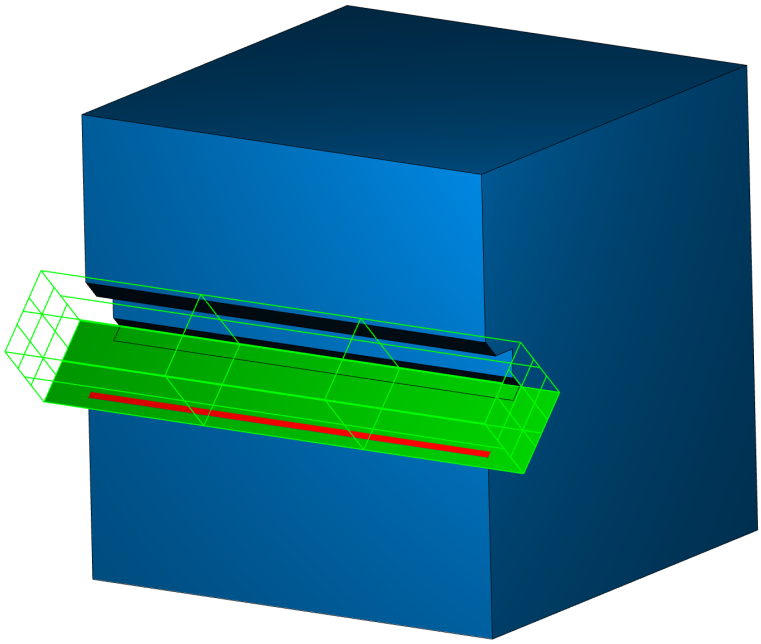


Fig. 2

Let us consider the next face resulting from this cutting (highlighted in green in Fig. 3):

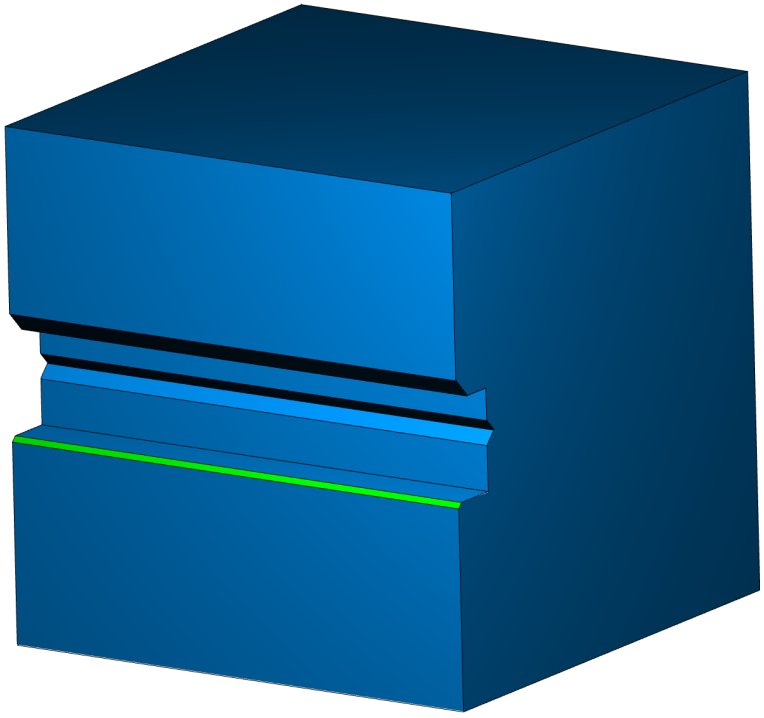


Fig. 3

After the operation, the new face has the following flags:

tct_Modified, *tct_Created*, *tct_Transformed*, *tct_Reoriented*, *tct_Deleted*, *tct_Truncated*, *tct_Added*, *tct_Replaced*

This means:

1. The face was modified in some way in the performed operation (*tct_Modified*).
2. The solid did not have this face previously, and then the face was created (*tct_Created*).
3. In fact, the face was created from the cutting solid face (Fig. 2), but the normal vector was reversed (*tct_Reoriented*). The cutting solid was moved before the operation (*tct_Transformed*).
4. The face was truncated (*tct_Truncated*) by removing (*tct_Deleted*) and adding (*tct_Added*) the face edge cycle.
5. The surface or edge cycle of the face was replaced during the operation (*tct_Replaced*).

The MbTopologyItem class defines (in file *topology_i_item.h*) the following methods for reading/writing flags:

Method

```
bool GetOwnChanged() const
```

returns the flag indicating that the object was (was not) modified.

Method

```
bool GetOwnChanged( MbeChangedType n ) const
```

returns the flag indicating that the object was (was not) modified.

Method

```
void SetOwnChanged( MbeChangedType )
```

sets the flag indicating that the object was (was not) modified.

Method

```
void CopyOwnChanged( const MbTopologyItem & ti )
```

copies the flag indicating that the object was (was not) modified.

Method

```
uint16 GetOwnChangedFlag() const
```

returns the flag indicating that the object was (was not) modified.

Method

```
template<class Uint> void SetOwnChangedFlag( Uint n )
```

sets the flag indicating that the object was (was not) modified.

Method

```
bool IsOwnRenamedOnly() const
```

returns the flag indicating that the object was only renamed.

Method

```
bool IsOwnTransformedOnly() const
```

returns the flag indicating that the object was only transformed.

Method

```
bool IsOwnReorientedOnly() const
```

returns the flag indicating that the object was only reoriented.

Method

```
bool IsOwnChangedWeakly() const
```

returns the flag indicating that the object was only created and/or only “weak” changes not modifying the object geometry structure (renaming, transformation, reorientation) were applied to it. If there are any other changes, the method returns *false*.

3.16 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.

3.16.1 MbItem Geometric Model Object

The MbItem class is declared in the *model_item.h* file.

The MbItem geometric model object is derived from the *MbSpaceItem*, MbTransactions, and MbAttributeContainer classes. The C3D geometric kernel operates on geometric model objects shown in Fig. 1.

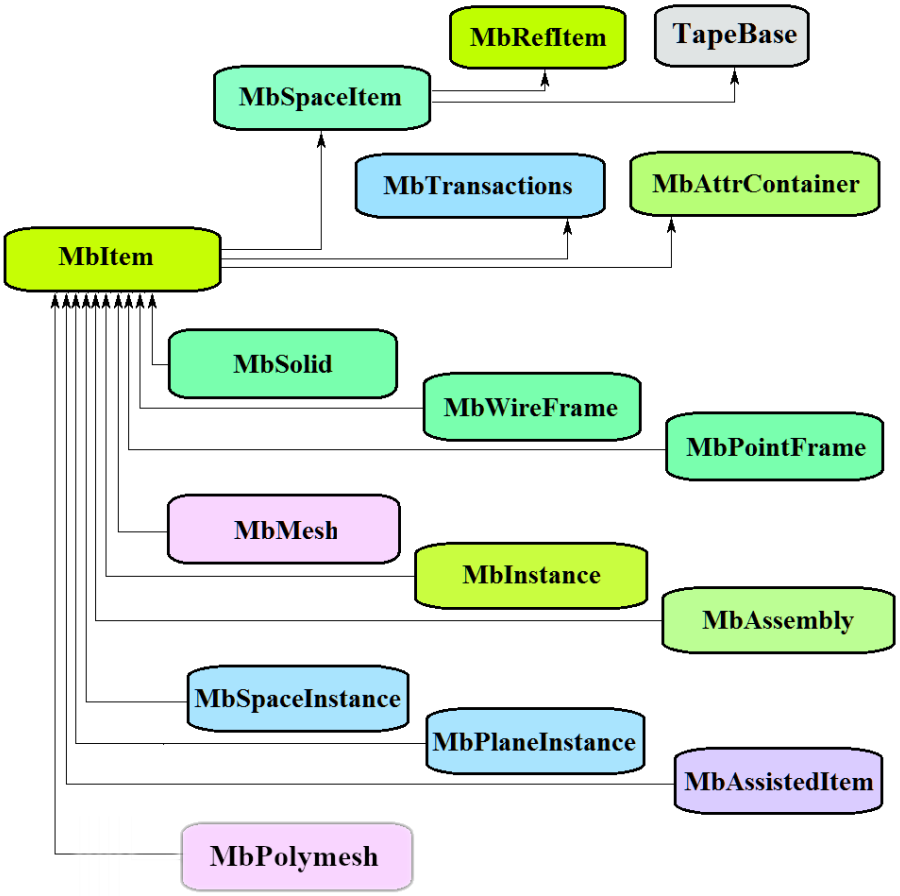


Fig. 1

The MbTransactions construction journal contains the data required to construct an object and allows you to repeat the construction of the object with edited parameters. The MbAttrContainer attribute container assigns attributes to geometric model objects. Thus, all geometric model objects, in addition to their specific data, contain the following data:

size_t *m_countRegistrable* – the number of object registrations during writing and reading,

ptrdiff_t *useCount* – the number of times the object is used by other objects,

std::vector< MbCreator * > **transactions** – an ordered set of object builders,

std::multimap< int, MbAttribute * > **attributes** – a set of object attributes.

The following geometric model objects are derived from the MbItem class:

MbSolid – solid body,

MbWireFrame – wireframe model,

MbPointFrame – point frame,

MbMesh – polygonal object,

MbInstance – insertion of a geometric model object,

MbAssembly – assembly unit of geometric model objects,

MbAssistingItem – auxiliary object,

MbSpaceInstance – insertion of a three-dimensional object,

MbPlaneInstance – insertion of a set of two-dimensional objects.

MbPolymesh – polygonal object with topology.

The main methods of geometric model objects are methods that provide editing and visualization of objects.

The method

bool RebuildItem(MbeCopyMode sameShell, RPArray<*MbSpaceItem*> **items**)

rebuilds the object according to the construction journal. This method is called after editing the internal data of the object.

The method

MbItem * CreateMesh(MbeStepData data, bool wire, bool grid, MbRegDuplicate * iReg)

creates a polygonal copy of the object. If the object is an assembly unit or insertion, then the copy of the object will also be an assembly unit or insertion containing polygonal objects.

The method

bool AddYourMesh(MbeStepData data, bool wire, bool grid, *MbItem* & **mesh**)

adds a polygonal copy of the object to the **mesh** object.

The method

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**)

searches for the nearest polygonal object **find**, its element **element**, their names findName and elementName, the path in the assembly unit structure path, and the transformation matrix **from** to the global coordinate system.

Geometric model objects overload the following methods of a three-dimensional object:

methods that handle the transformation of a geometric 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 that provide copying, equality checking, checking the possibility to make objects equal, and making objects equal,

MbSpaceItem & Duplicate(MbRegDuplicate * iReg = NULL),

bool IsSame(const *MbSpaceItem* & **item**),

bool IsSimilar(const *MbSpaceItem* & **item**),

bool SetEqual(const *MbSpaceItem* & **item**),

methods that return a type from the enumeration of geometric objects:

MbeSpaceType IsA(),

MbeSpaceType Type(),

MbeSpaceType Family(),

methods that provide output and editing of the internal data of the object:

MbProperty & CreateProperty(MbePrompt name),

GetProperties(MbProperties & *properties*),

SetProperties(MbProperties & *properties*).

3.16.2 MbSolid Solid Body

The MbSolid class is declared in the *solid.h* file.

A solid body, or simply a body, MbSolid is derived from the *MbItem* class and is described by a set of faces *MbFaceShell* *outer* and the connectivity type *multiState*.

A body is a set of faces joined to each other along edges and describing the surface of the modeled object. A body may describe one or several connected sets of points. The connectivity type *multiState* indicates whether the body describes a single connected set of points or whether the body describes several connected sets of points and can be split into several bodies.

The set of faces *outer* of a body, depending on the presence of boundary edges, may describe two fundamentally different sets of points. If the set of faces has no boundary, then the body describes the set of points lying on the surfaces of the faces and on the inner side of these faces. Such a body is called *closed* and is described by a closed shell. A closed body is shown in Fig. 1.

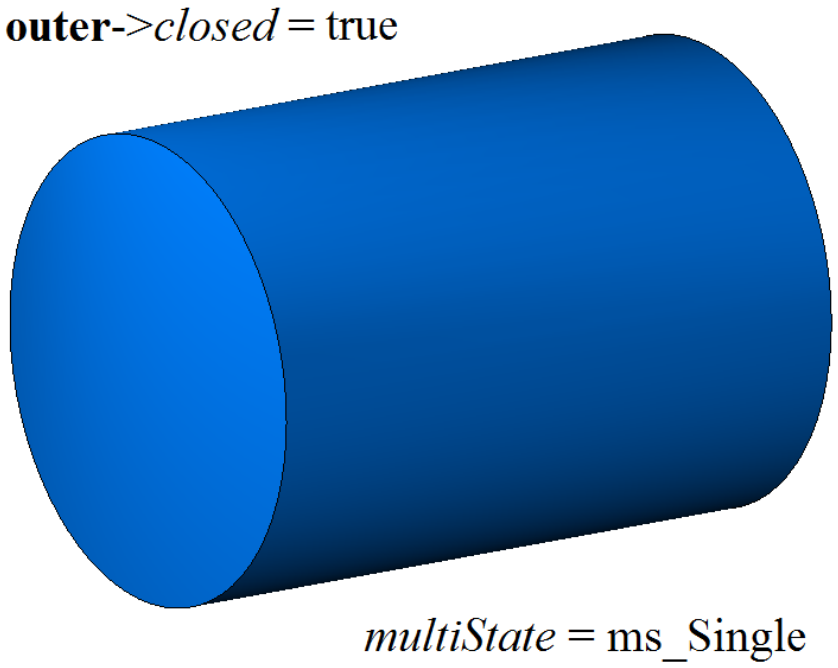


Fig. 1

If the set of faces of a body has a boundary or boundaries, then the body describes a set of points lying only on the surfaces of the faces. Such a body is called *open* and is described by an open shell. An open body is shown in Fig. 2.

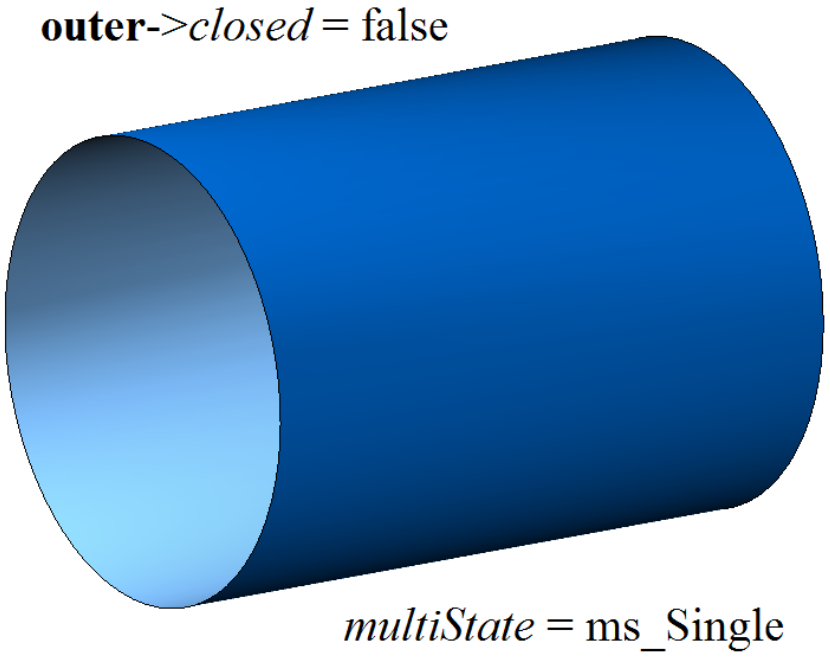


Fig. 2

In most cases, a closed body is described by a single connected set of faces, that is, by a single shell. If a closed body has internal cavities, the body is described by several connected sets of faces. Fig. 3 shows a closed body described by two closed shells, one outer and one inner, located inside the outer shell.

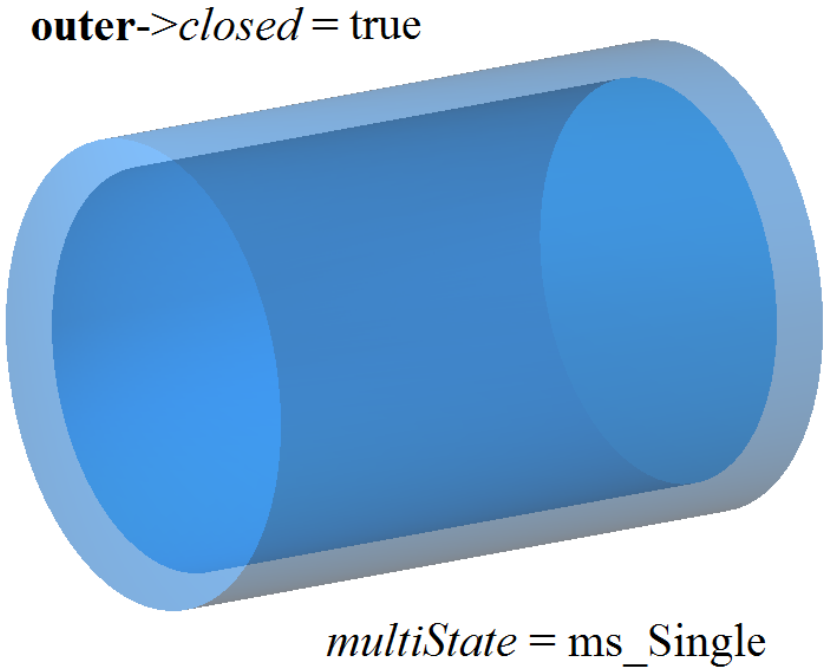


Fig. 3

To make the cavity of the body visible, the body is rendered semi-transparent.

Various operations can be performed on bodies – a set of actions resulting in a body of a different shape, for example, Boolean operations. The result of subtracting two closed bodies is shown in Fig. 4.

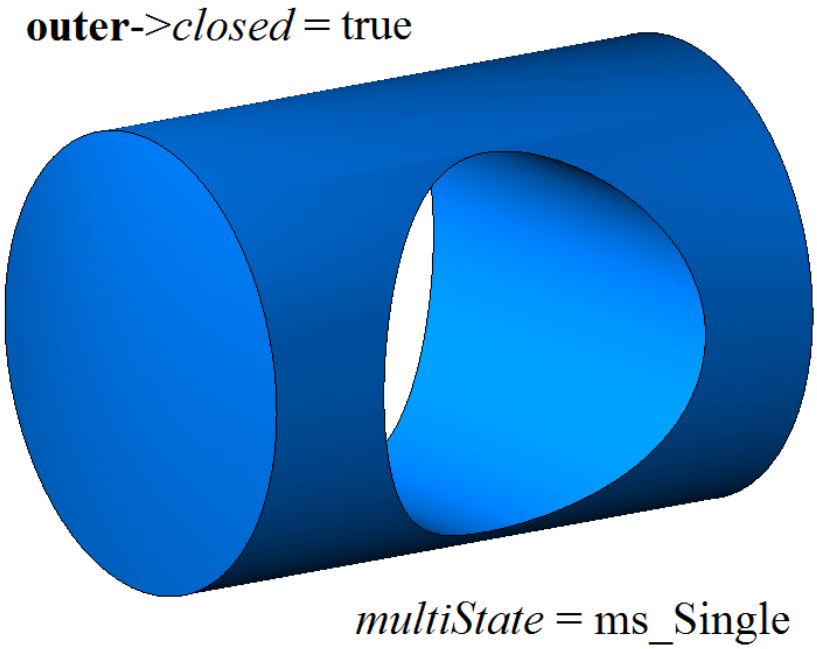


Fig. 4

The result of an operation on a closed and an open body is fundamentally different, because the actions are performed on different sets of points. The result of subtracting a body from an open body is shown in Fig. 5.

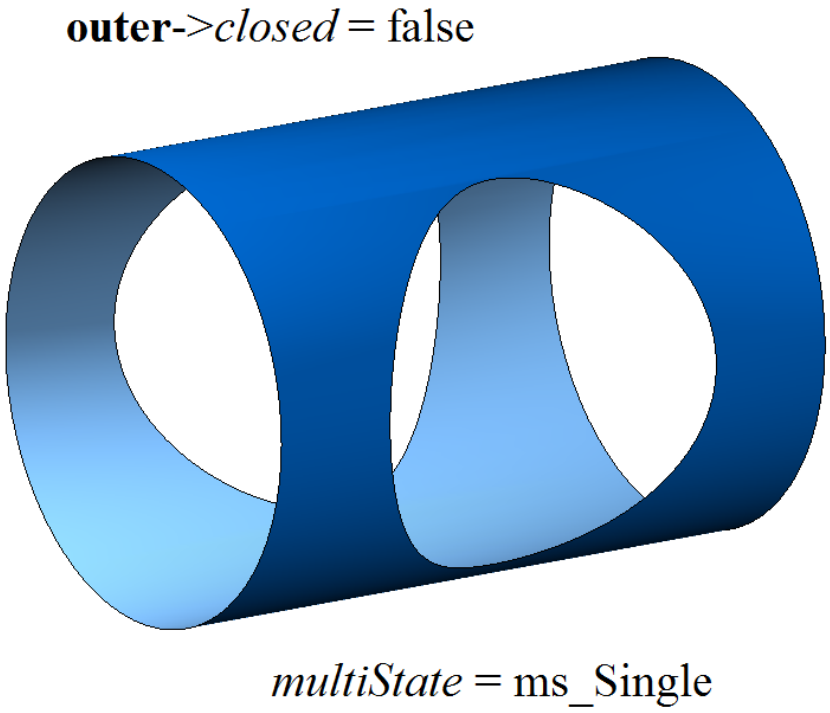


Fig. 5

A body may be multiply connected, that is, it may consist of several separate parts. In this case, *multiState* takes the value ms_Multiple. A doubly connected body described by two closed shells is shown in Fig. 6.

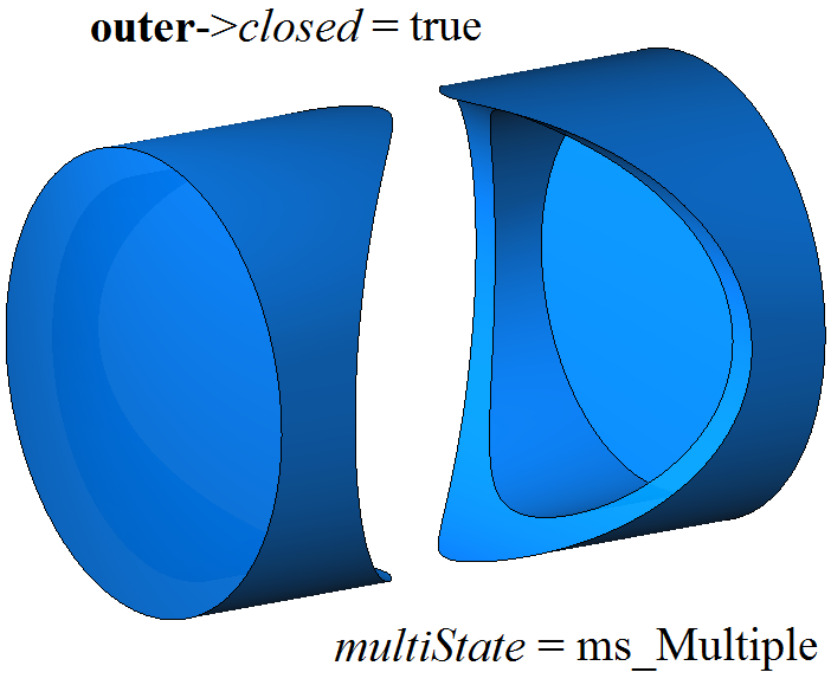


Fig. 6

Such a body can be divided into two simply connected closed bodies by the DetachParts() method. The body in Fig. 6 is rendered semi-transparent. A doubly connected body described by two open shells is shown in Fig. 7.

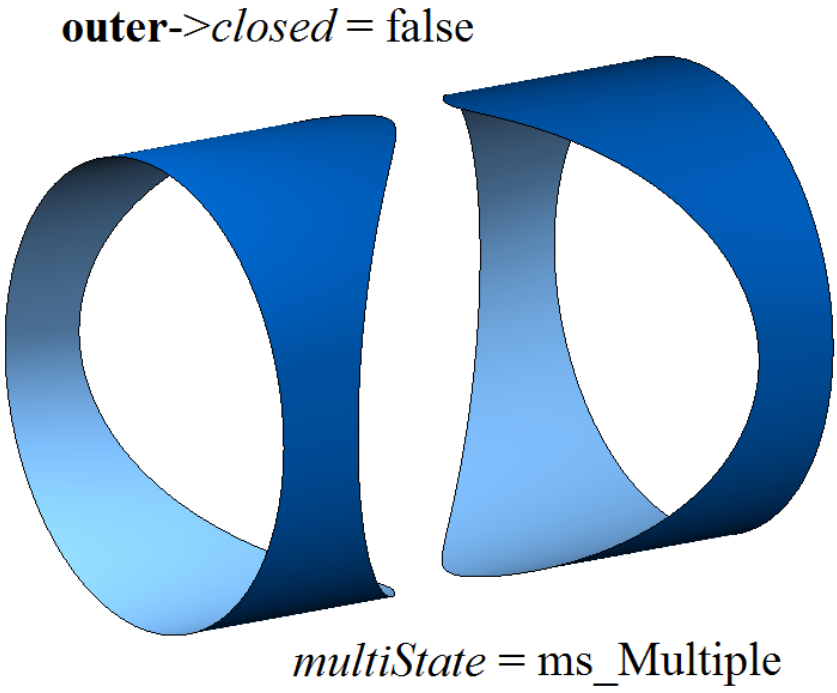


Fig. 7

Such a body can be divided into two simply connected open bodies.

Bodies are created by methods of the C3D geometric kernel. Simple-shaped bodies are created from points, curves, and surfaces. Using operations, more complex bodies can be obtained from simple-shaped bodies. The original bodies can be edited and similar bodies can be created by changing the parameters of the MbTransactions construction journal or by directly modifying the elements of already constructed bodies. Open bodies are used in surface modeling. An open body makes it possible to focus efforts on complex shapes of the modeled object.

3.16.3 MbWireFrame Wireframe

The MbWireFrame class is declared in the *wire_frame.h* file.

The MbWireFrame wireframe is derived from the *MbItem* class and is described by a set of edges std::vector< *MbEdge* * > **edges**, the number of connected parts *parts*, and the flag indicating the absence of boundary vertices *closed*.

A wireframe is a set of edges that are joined to each other at vertices and describe the frame structure of the modeled object. A wireframe may describe one or several connected sets of points. The number of connected parts *parts* indicates whether the frame describes a single connected set of points or whether the frame describes several connected sets of points and can be split into several wireframes.

Depending on whether boundary vertices are absent or present, a wireframe will be called *closed* or *open*. Fig. 1 shows a closed wireframe.

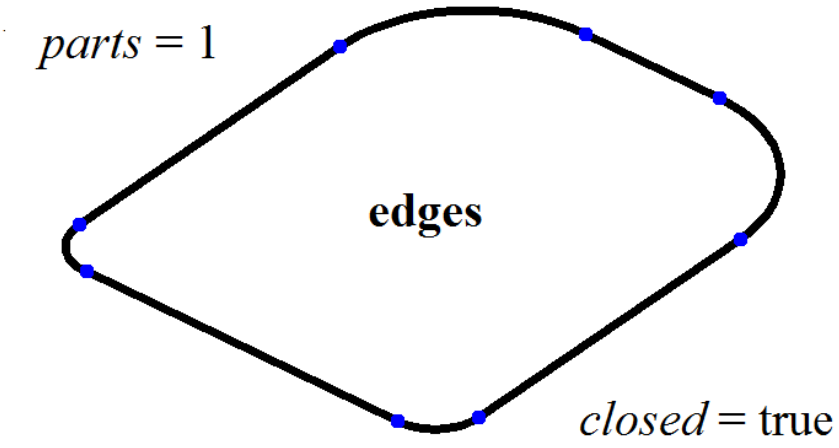


Fig. 1

An open wireframe is shown in Fig. 2.

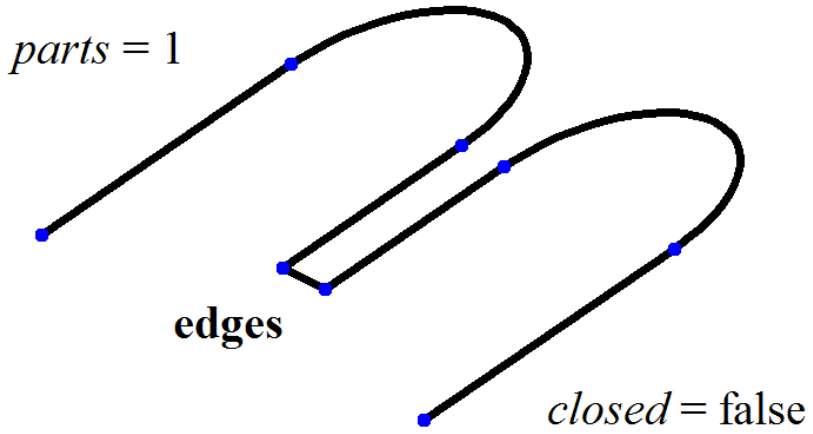


Fig. 2

Fig. 3 shows an open wireframe consisting of two connected sets of edges:

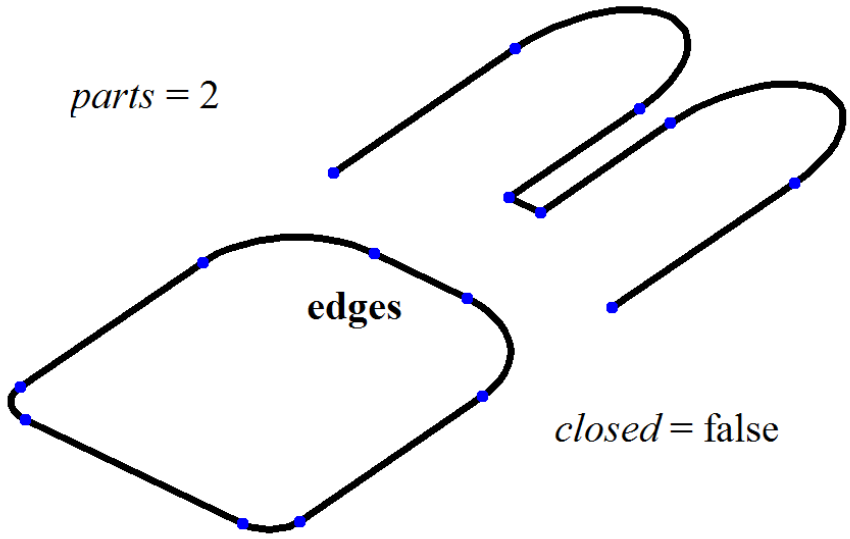


Fig. 3

Fig. 4 shows a closed wireframe consisting of two connected sets of edges:

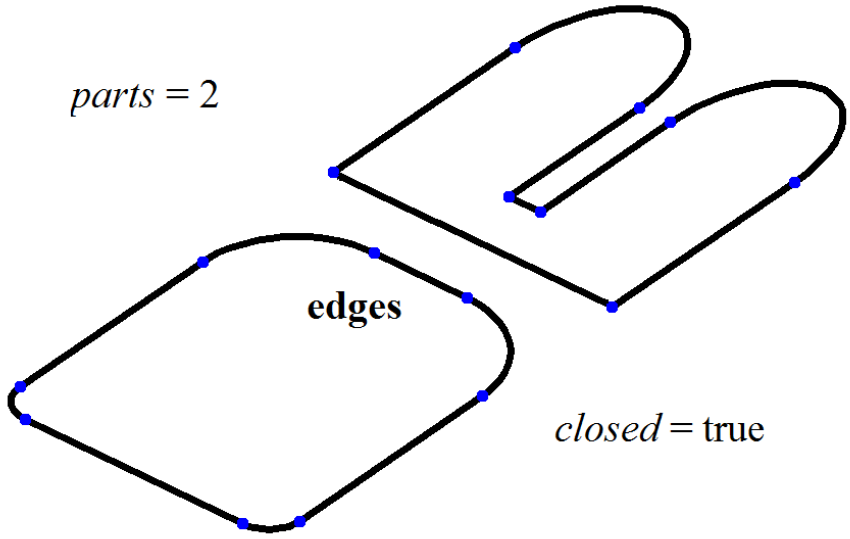


Fig. 4

A wireframe can be used for constructing trajectories, spatial sketches, and for auxiliary constructions.

3.16.4 MbPointFrame Point Frame

The MbPointFrame class is declared in the *point_frame.h* file.

The MbPointFrame point frame is derived from the *MbItem* class and is described by a set of points represented as vertices, std::vector< *MbVertex* * > **vertices**.

Fig. 1 shows a point frame.

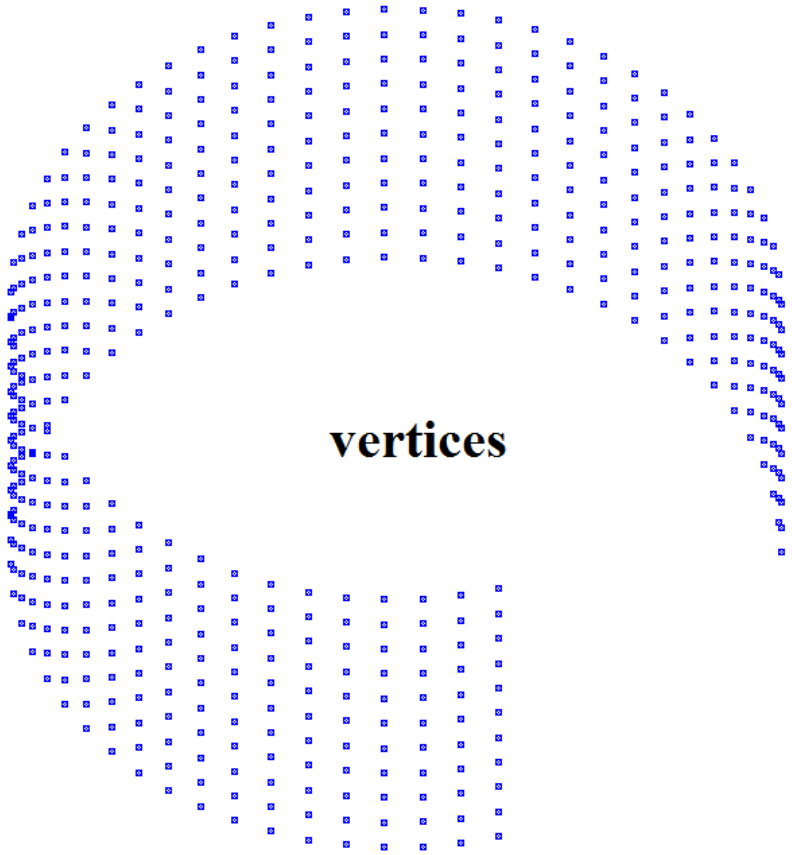


Fig. 1

A point frame can be used for positioning other objects and for auxiliary constructions.

3.16.5 MbMesh Polygonal Object

The MbMesh class is declared in the *mesh.h* file.

The MbMesh polygonal object is derived from the MbItem class and is described by a set of triangulations RPArray< MbGrid > **grids**, a set of polygons RPArray< MbPolygon3D > **wires**, a set of apexes RPArray< MbApex3D > **peaks**, a pointer to the original object *MbRefItem item*, the *type*, a bounding cube **cube**, and the closed flag *closed*.

A polygonal object is a set of triangular and quadrilateral plates, polyline segments, and individual points. One of the ways to construct a polygonal object is to approximate other geometric model objects, for example, a solid. Each *i* -th face of the solid is approximated by the triangulation **grids**[*i*], each *j*-th edge of the solid is approximated by the polygon **wires**[*j*], each *k*-th vertex of the solid corresponds to the apex **peaks**[*k*], and the closed flag *closed* corresponds to whether the solid is closed. Another way to obtain a polygonal object is to import data using a polygonal representation converter.

The MbGrid triangulation represents a set of points SArray< MbFloatPoint3D > **points**, the corre-

sponding set of normals SArray< MbFloatVector3D > **normals** (the number of points equals the number of normals), a set of two-dimensional points of the parametric domain of the surface SArray< MbFloatPoint > **params** (the number of two-dimensional points equals the number of three-dimensional points or is zero – the set may be empty), a set of triangular plates SArray< MbTriangle > *triangles* as three indices of the **points** set, and a set of quadrilateral plates SArray< MbQuadrangle > *quadrangles* as four indices of the **points** set. The plates of the triangulation approximate a certain surface.

The MbPolygon3D polygon represents an ordered set of points whose successive connection produces a polyline approximating a certain curve.

The MbApex3D apex represents a point endowed with additional data.

The pointer to the original object **item** may be null, and the *type* may be undefined.

A vector image of a polygonal object is shown in Fig. 1.

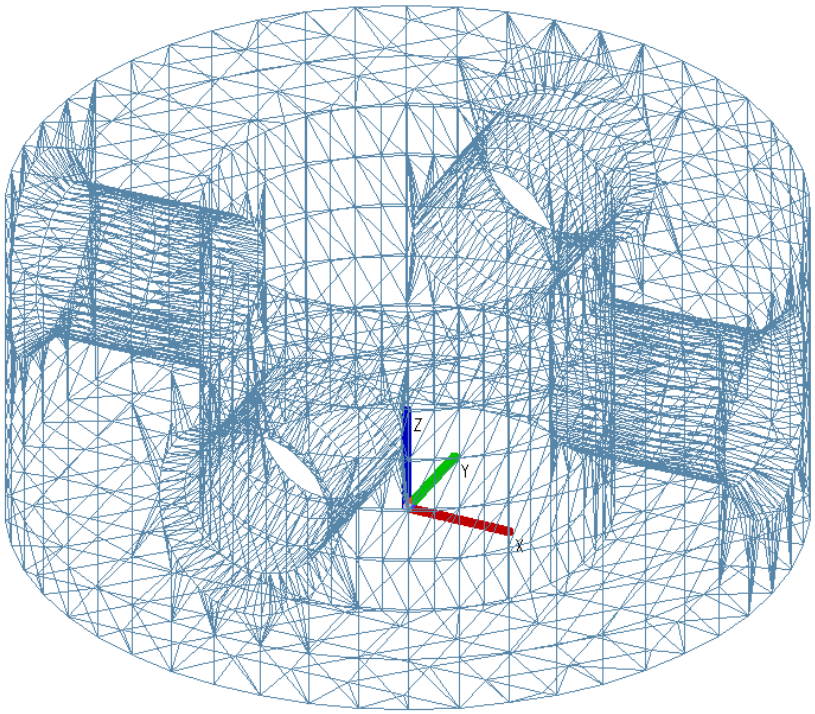


Fig. 1

A shaded image of a polygonal object is shown in Fig. 2.

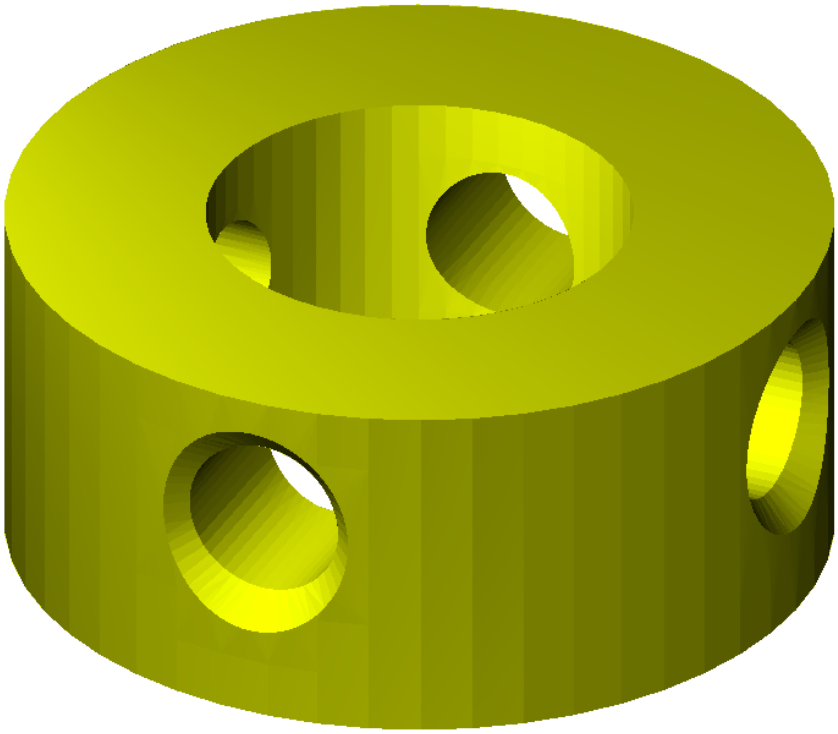


Fig. 2

Individual triangles of the object are visible in the figure because the normal direction is constant in each triangle. If the normal direction in the triangles of the triangulation changes continuously, individual triangles of the object become invisible, see Fig. 3.

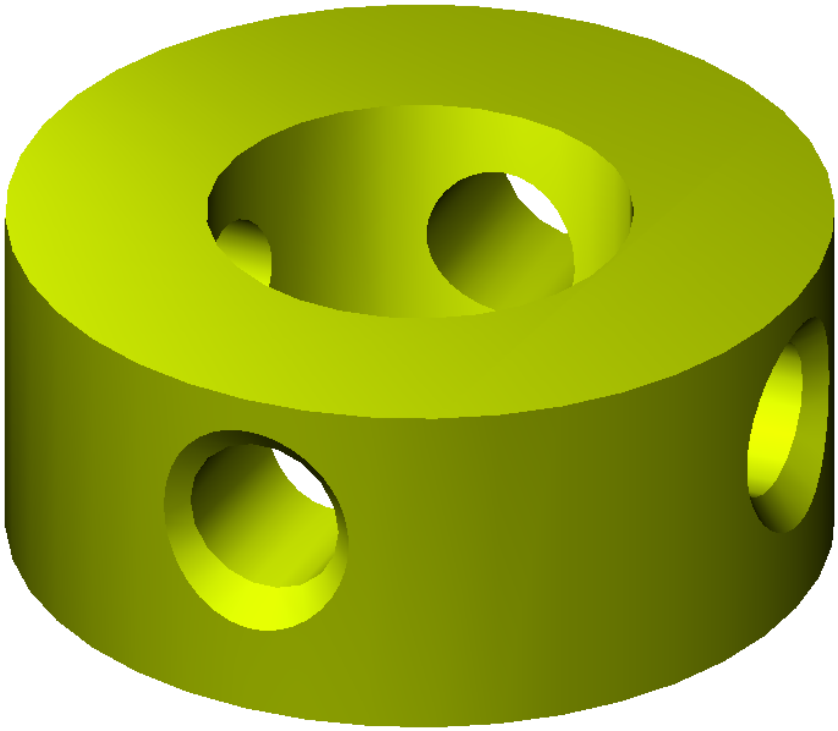


Fig. 3

A polygonal object can be created by the MbItem:: CreateMesh() method or by the MbItem:: AddYourMesh() method. A polygonal object is used for visualization, computations, and manufacturing of modeled objects. The main advantage of a polygonal object is its simplicity and high computational speed, for example, for intersection with a line.

3.16.6 MbInstance Insertion

The MbInstance class is declared in the *instance_item.h* file.

The **MbInstance** insertion is derived from the *MbItem* class and is described by the MbItem **item** geometric model object and the *MbPlacement3D* **place** local coordinate system. The insertion is the **item** object moved into the **place** local coordinate system.

The insertion has all properties of the **item** object. The difference is that the Move(), Rotate(), and Transform() methods modify the **place** coordinate system without changing the **item** geometric object, which can be shared by several insertions.

The insertion object may contain a solid, a wireframe, a point frame, or a polygonal object, but it cannot contain another insertion (that is, **MbInstance** cannot be used as **item** for another **MbInstance**).

3.16.7 MbAssembly Assembly Unit

The MbAssembly class is declared in the *assembly.h* file.

The MbAssembly assembly unit, or assembly, is derived from the MbItem class and is described by a set of geometric model objects std::vector< MbItem * > **assemblyItems**.

An assembly is a set of geometric model objects that can be manipulated as a single object.

An assembly unit is shown in Fig. 1.

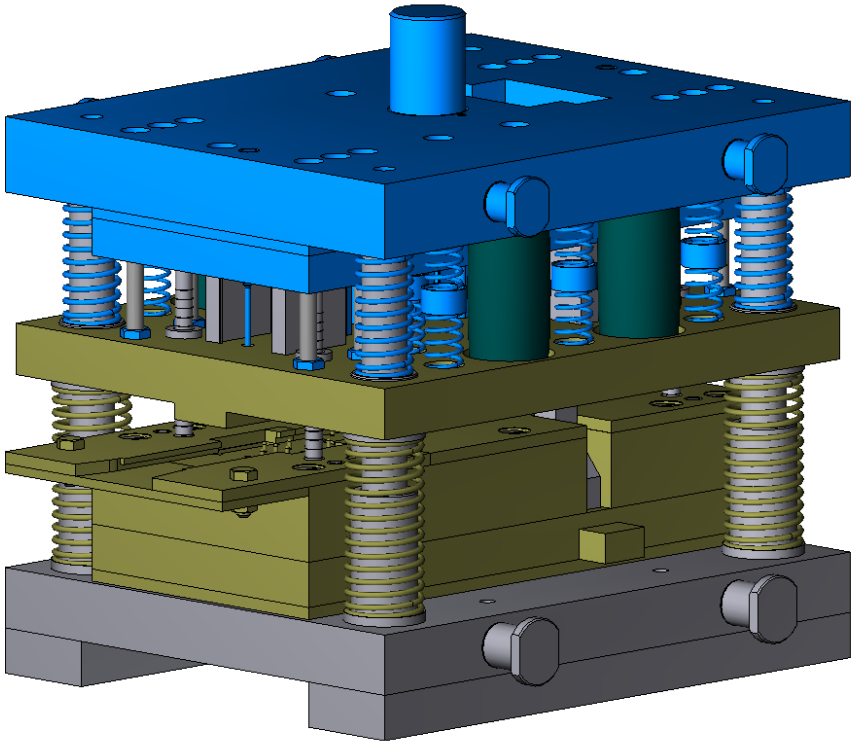


Fig. 1

An assembly unit may contain other assembly units as its objects, that is, it may have a tree-like structure.

3.16.8 MbSpaceInstance Three-Dimensional Object Insertion

The MbSpaceInstance class is declared in the *space_instanse.h* file.

The MbSpaceInstance three-dimensional object insertion is derived from the *MbItem* class and is described by the geometric object *MbSpaceItem spaceItem*.

The insertion acts as a wrapper for a geometric object, allowing it to be treated as a geometric model object. The insertion provides a regular geometric object with a construction journal, attributes, and the methods of a geometric model object *MbItem*. The insertion of a three-dimensional object is intended for auxiliary constructions. An insertion of a surface is shown in Fig. 1.

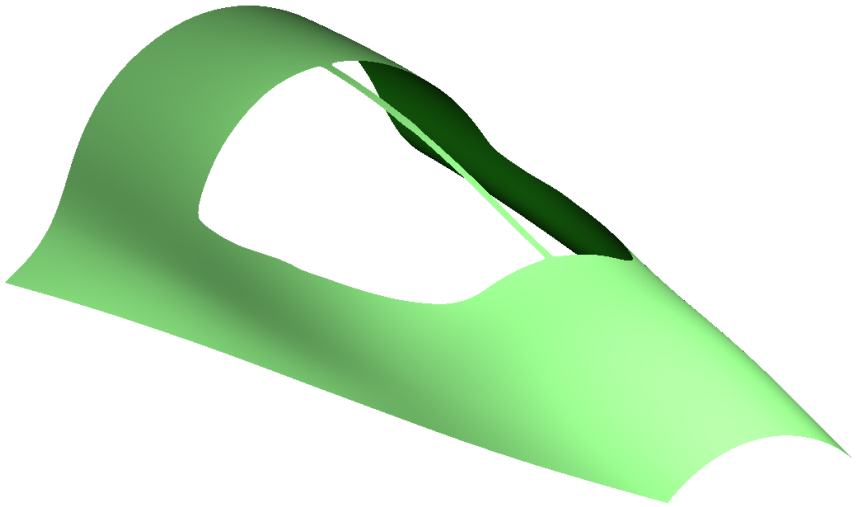


Fig. 1

An insertion object may contain a surface *MbSurface*, a curve *MbCurve3D*, a MbPoint3D point, or an auxiliary geometric object *MbLegend*. For objects derived from the MbItem geometric model object (solid, wireframe, point frame, polygonal object, assembly, insertion), insertion of a geometric object is not used.

3.16.9 MbPlaneInstance Two-Dimensional Object Insertion

The MbPlaneInstance class is declared in the *plane_instanse.h* file.

The MbPlaneInstance insertion of two-dimensional objects is derived from the *MbItem* class and is described by a set of two-dimensional geometric objects std::vector< *MbPlaneItem* > **planeItems** and the local coordinate system *MbPlacement3D* **place**. The two-dimensional objects are located in the XY plane of the local coordinate system.

The insertion acts as a wrapper for two-dimensional geometric objects, allowing them to be treated as a geometric model object. The insertion provides two-dimensional geometric objects with a construction journal and endows them with attributes and the methods of the geometric model object *MbItem*. The insertion of two-dimensional objects is intended for auxiliary constructions. An insertion of two-dimensional curves is shown in Fig. 1.

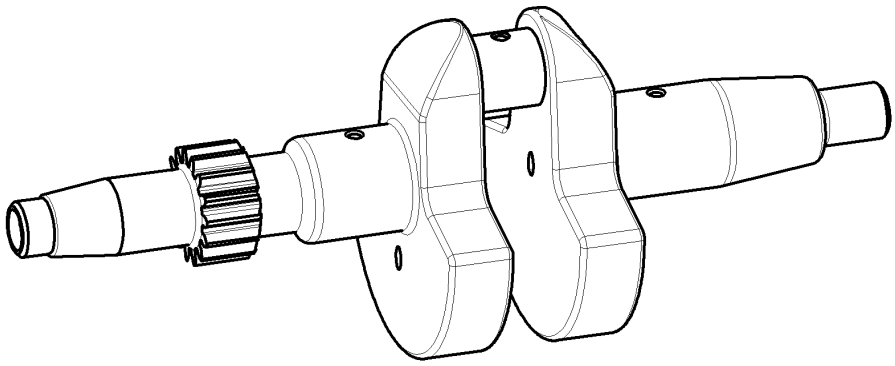


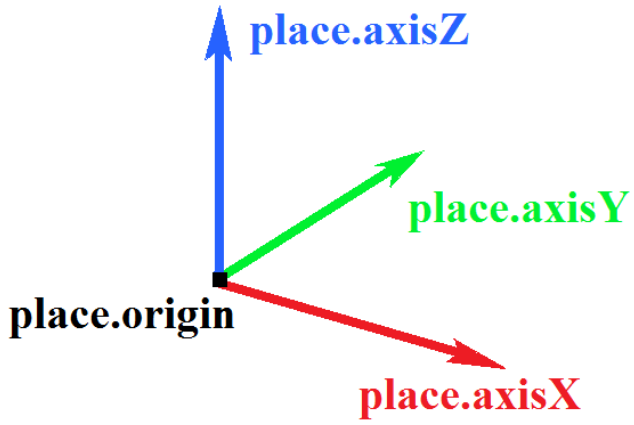
Fig. 1

The insertion of two-dimensional objects may contain a two-dimensional curve MbCurve, a multiline *MbMultiline*, or a region *MbRegion*.

3.16.10 MbAssistingItem Auxiliary Object

The MbAssistingItem class is declared in the *assisting_item.h* file.

The MbAssistingItem geometric model auxiliary object is derived from the MbItem class and is described by the local coordinate system MbPlacement3D **place**. The auxiliary object is intended for positioning other objects. The auxiliary object has a construction journal, attributes, and the methods of the geometric model object MbItem for auxiliary constructions. The auxiliary object is shown in the figure.



3.16.11 MbPolymesh Polygonal Object with Topology

The MbPolymesh class is declared in the *polymesh.h* file.

The MbPolymesh polygonal object is derived from the *MbItem* class and is described by the *_pTopo* topology, which represents a description of the polygonal object in the form of a Halfedge structure.

The Halfedge representation is a representation of a polygonal model as a graph, where the key element of the representation is a directed edge associated with the facet to which it belongs and the vertex from which the edge originates. Thus, each non-boundary edge of the surface is represented as a pair of oppositely directed edges (an edge pair). The edges are organized into closed cycles, that is, facets.

As an additional set of information, the object, in addition to the topology, stores segmentation data of the object. The object can be divided into regions. Each region contains a set of connected components called subregions. By analogy with the Halfedge organization, each subregion may have one or more segmentation edge cycles.

To provide flexibility of representation, each topology element can be assigned an attribute that stores additional information about this element. For example, a vertex may have an attribute that stores information about its 3D coordinates. Through attributes, the user can assign derived data of the types bool, float, double, unsigned int, int, and size_t.

An image of a polygonal object with edge visualization is shown in Fig. 1. Segmentation regions are shown in different colors.

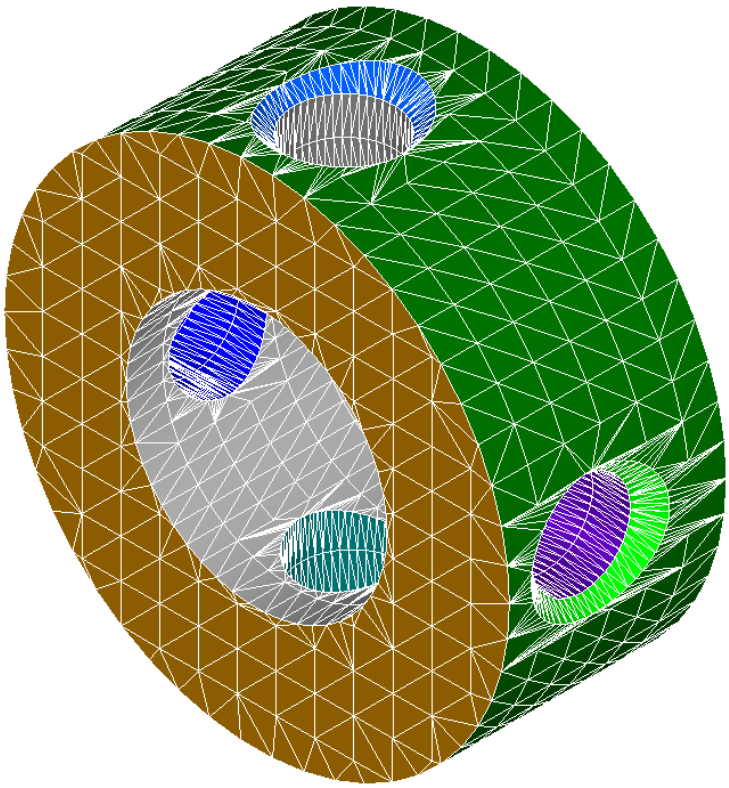


Fig. 1

A shaded image of the polygonal object is shown in Fig. 2. Subregion cycles are shown in white.

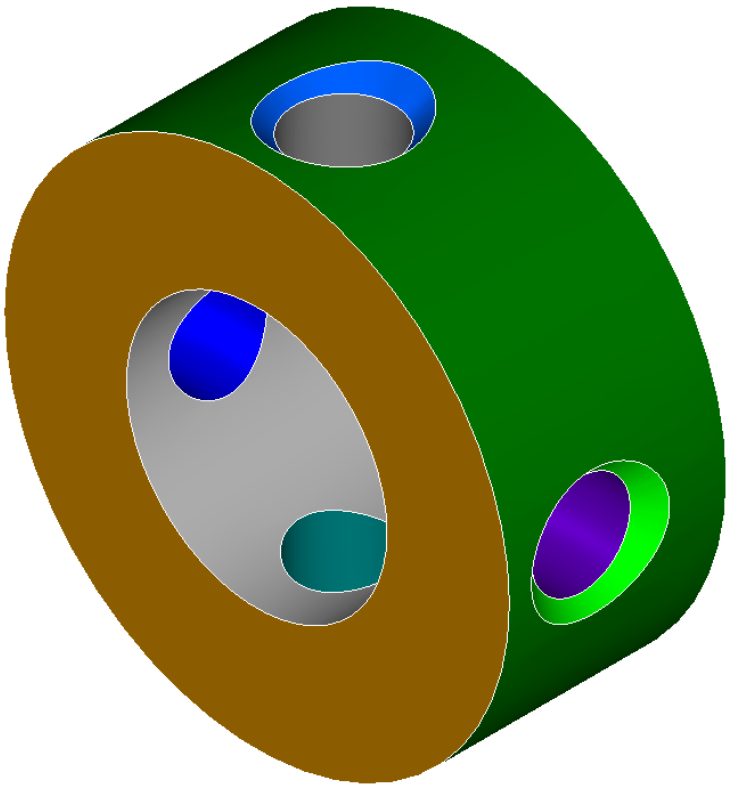


Fig. 2

Facets may have an arbitrary number of edges. For example, Fig. 3 shows pentagonal facets forming the faces of a dodecahedron.

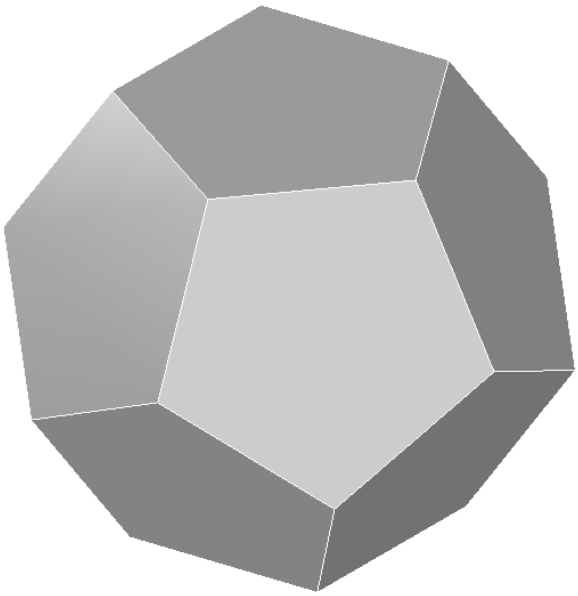


Fig. 3

A polygonal object with topology can be created from an MbMesh object, or it can be constructed by the user directly. An example of creating such an object can be found in the test application in the *test_sample_polymesh.cpp* file. Topology is required for analysis, repair, and editing of polygonal objects.

3.17 MULTITHREADING

Multithreading support in the mathematical kernel implies:

- Use of parallel calculations in the kernel.
- Support of multithreading in user applications.

The mathematical kernel implements 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 include (but are not limited to):

- Construction of planar projections
- Calculation of polygonal meshes
- Calculation of mass-inertia properties
- Converter 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.

3.17.1 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.

3.17.2 Multithreaded Caches Implementation

The base class for cached data and the cache manager class are defined in the *tool_multithreading.h* file.

Cache Manager CacheManager

The cache manager is defined as a template class **CacheManager** 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_Items 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 each cache as a parameter. After the function **Postprocess()** finishes 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, which allows cleaning caches on demand.

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 cache manager for the main cache with the data of 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 the 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.

3.17.3 Garbage Collection in Cache Manager

The cache manager CacheManager deletes unused caches at the first call from sequential code.

In addition, garbage collection runs automatically on exiting a parallel region.

The mathematical kernel provides an interface for clearing cached data – the MbGarbageCollection class.

The MbGarbageCollection class is the garbage collector which clears caches in registered objects of the CacheCleaner type. By request it deletes caches in the registered objects.

In order to enable garbage collection in an object of type CacheCleaner, the object needs to implement the ResetCacheData() method for caches cleaning and to subscribe to garbage collection in the MbGarbageCollection class.

The CacheManager class is derived from the CacheCleaner class, which makes it possible to run caches clearing in CacheManager automatically. For that, when creating a cache for a thread the cache manager registers itself in the MbGarbageCollection class.

Base Class for Objects that Require Garbage Collection

In order to subscribe to garbage collection, a class should be derived from the CacheCleaner class and should implement the **ResetCacheData** method 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 CacheCleaner class also defines the method:

- **bool ResetCacheData()** – reset cached data. Should return true if cached data was cleared and the object was unsubscribed from garbage collection.

Class MbGarbageCollection

The MbGarbageCollection class is the garbage collector for objects of type CacheCleaner, which keep cached data. By request MbGarbageCollection clears caches in registered objects by calling ResetCacheData for each object.

The MbGarbageCollection class implements the methods:

- **void Subscribe(CacheCleaner * obj)** – subscribe the object to garbage collection.
- **void Unsubscribe(CacheCleaner * obj)** – unsubscribe the object from garbage collection.
- **static bool Run(bool force = false)** – perform 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 force == true, then force garbage collection in all caches. Returns true if garbage collection is done.
- **static void Enable(bool allow = true)** – enable/disable collecting data for garbage collection. By default it is enabled.

3.17.4 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 the **MbeMultithreadedMode** enumeration in the **tool_multithreading.h** file.

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 multithreaded 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 the **mtm_Items** mode.

By default the kernel works in the **mtm_Max** mode.

The kernel provides interfaces for dynamic switching of the multithreading mode.

The interfaces for switching the kernel multithreading mode are declared in the **mb_variables.h** file.

The following static methods of the **Math** class can be 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 multithreading mode.

3.17.5 Synchronization Objects

The synchronization objects are defined in the *tool_mutex.h* file.

Locks

By default the kernel uses lock classes implemented on the basis 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 the basis of critical sections. On Linux system locks are implemented on the basis of pthread_mutex_t.

The lock classes implementation is controlled by the **C3D_NATIVE_LOCK** variable. Redefining the **C3D_NATIVE_LOCK** variable in user applications is prohibited.

The **CommonMutex** and **CommonRecursiveMutex** classes define lock and recursive lock. They provide the lock() and unlock() methods.

The **ScopedLock** and **ScopedRecursiveLock** classes 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.

Base Synchronization Objects

The **MbSyncItem** class is the base synchronization object with lazy initialization. It creates a lock (mutex) when needed and implements the **Lock()** and **Unlock()** methods.

The **MbNestSyncItem** class is the base synchronization object with lazy initialization which supports nested locks. It creates a lock (mutex) when needed and implements the **Lock()** and **Unlock()** methods.

The **MbPersistentSyncItem** class is the base synchronization object implementing the **Lock()** and **Unlock()** methods. It always creates a lock (mutex).

The **MbPersistentNestSyncItem** class is the base synchronization object which supports nested locks. It implements the **Lock()** and **Unlock()** methods and always creates a lock (mutex).

The kernel objects that need synchronization are inherited from one of these classes.

3.17.6 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 the 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 ParallelRegionGuard class (a scoped guard of a parallel region) or the pair of functions EnterParallelRegion and ExitParallelRegion can be used.

Protection of Parallel Code in User Application

The ParallelRegionGuard class notifies the kernel that code in the scope runs in parallel. It should be used to protect a parallel region.

Instead of the ParallelRegionGuard class, the following pair of functions also can be used to notify the kernel about parallel region boundaries:

- The **EnterParallelRegion** function notifies the kernel about entering a parallel region.
- The **ExitParallelRegion** function notifies the kernel about exiting a parallel region.

Interfaces ParallelRegionGuard, **EnterParallelRegion** and **ExitParallelRegion** must be used in cases when the kernel is used in several threads.

Note that before starting to use 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).

The Examples of Notifying the Kernel about its Use in Parallel Computing

ParallelRegionGuard should be called one time before starting work in several threads. For example:

```
// ... Sequential code ...

if( Math::CheckMultithreadedMode( mtm_Items ) ) {
  ParallelRegionGuard guard; // works in the scope

  std::thread t1( func1 );
```

(continues on next page)

(continued from previous page)

```
  std::thread t2( func2 );

  t1.join();
  t2.join();
}
```

```
// ... Sequential code ...
```

The same can be done using a pair 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 ParallelRegionGuard, EnterParallelRegion and ExitParallelRegion interfaces 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.

Quick Reference on Organizing Parallel cComputing 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.

   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 */
     }
   ```

(continues on next page)

(continued from previous page)

```
  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 starts using multithreaded caches, and after the last call of ExitParallelRegion it returns to the base mode of sequential execution.

3. It is not recommended to call ParallelRegionGuard(), EnterParallelRegion() and ExitParallelRegion() without need (in the absence of multithreading), because the use of multithreaded caches in sequential execution entails some time overhead of obtaining cached data.

3.18 FORMAT C3D

A geometric model can be saved in a compact form to a memory buffer or a disk file. There are two types of C3D format for storing a geometric model – compact and extended.

This article describes C3D formats for storing a geometric model and the interfaces the kernel provides for writing and reading geometric model data in the C3D format.

3.18.1 Format for Storing Geometric Model

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 are 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 cluster.

Geometric Model Serialization

Geometric model C3D can be serialized (represented as a sequence of bytes) to a set of file spaces.

All main objects of the geometric model are streaming, that is, they can 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 the 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) are sequentially copied.

The memory buffer created in this way contains geometric model data as a sequence of bytes and can be transferred to another location (another application) via memory for subsequent restoring of the model or can be saved to a disk file.

Compact Format C3D

The **compact format** provides a compact and fast storing of geometric model data to a memory buffer (file) and fast reading of 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 of the whole model.
- Information about the model structure and about model objects is available only after reading the whole model.
- It is 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 the given file space.

Fig. 1 shows the structure of serialized data of a geometric model before saving it in a sequential memory buffer when using the compact format.

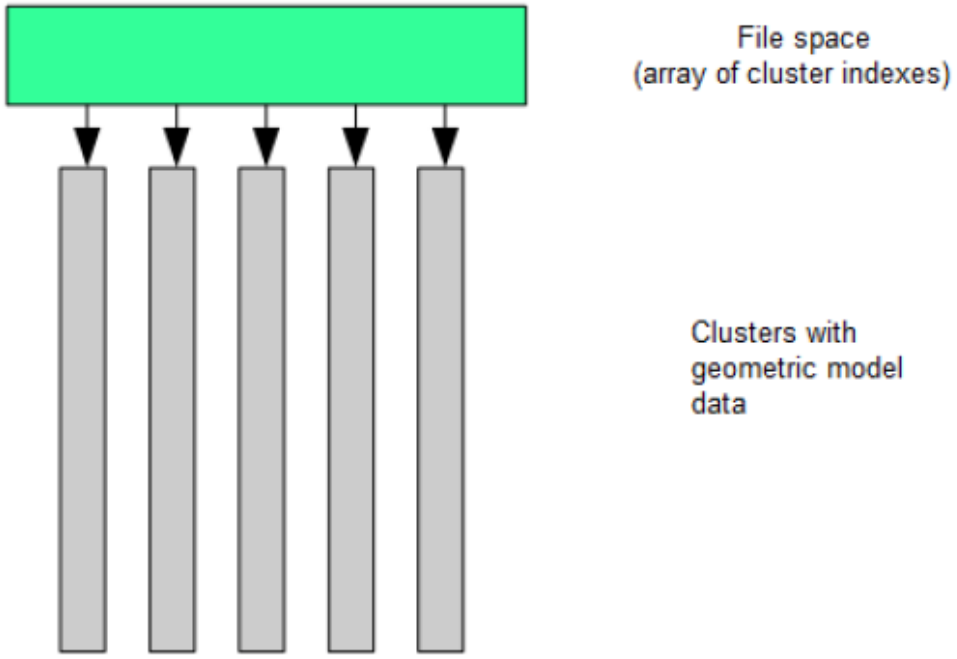


Fig. 1

While writing a model, if a geometric object contains another object or references another object, then the latter object is written inside the referring object.

An object referenced by multiple objects is declared **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.

Fig. 2 shows a structure of geometric model data in a sequential memory buffer for the compact format.

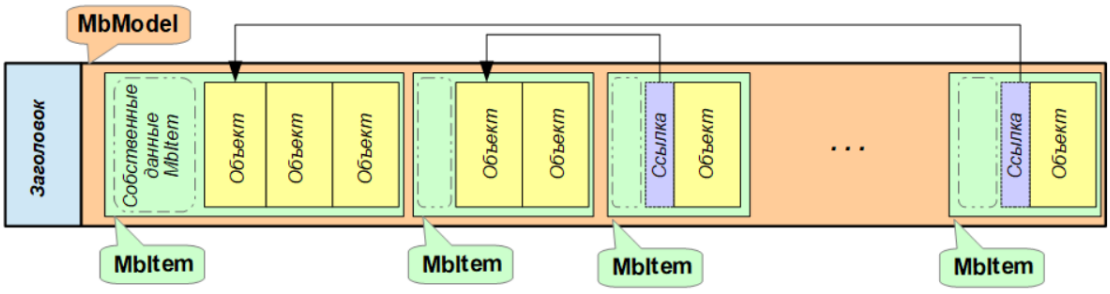


Fig. 2

Extended Format C3D

The **extended format** provides independent storing of geometric model objects. 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, which allows obtaining information about the model structure without fully 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.

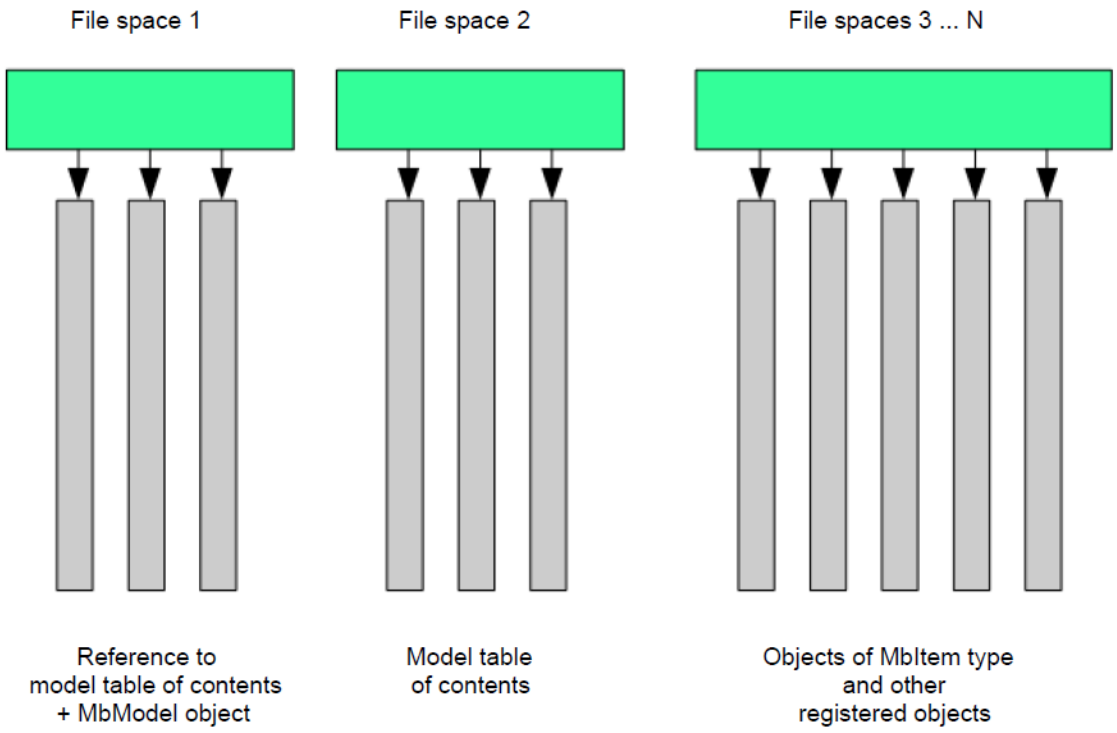


Fig. 3

Fig. 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 file space (a number of file spaces used is determined by the nesting depth of the model objects).

Fig. 4 shows a structure of geometric model data in a sequential memory buffer for the extended format.

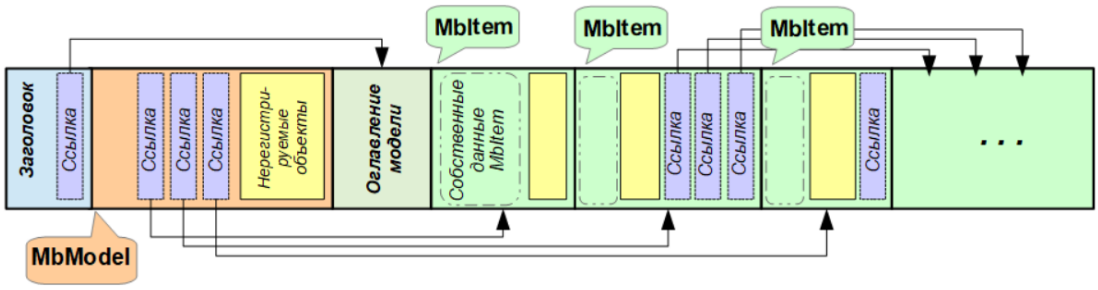


Fig. 4

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 can be read separately, which allows obtaining information about the model structure and its main objects without full 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 writer_ex class (in the extended format) and should be read using the reader_ex class.
- 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 the ReadModelItems() function), the first in order (default) embodiment will be read.

A model with embodiments is created as an MbModel object, which contains a set of assemblies (MbAssembly objects). Each assembly represents one embodiment (contains objects of one embodiment) and has an attribute of the **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 the **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 the 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 writer_ex class. After that the model with embodiments should be read using the reader_ex class.

When reading a C3D file with embodiments, only one embodiment can be read.

If a file with embodiments is read in a usual way, that is, without use of the model tree (for example, using direct call to the ReadModelItems() function), then the first (default) embodiment will be read.

In order to read any other embodiment, the 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.

3.18.2 Read and Write Streaming Objects

Classes for reading and writing streaming objects of the model (read and write streams) are defined in the **tape.h** file. They inherit from the base class tape.

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 the data buffer:

- uint8 mode() – get the operation mode of the buffer.
- void setMode(uint8) – set the 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 the 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 the 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 and set the parent progress indicator, if it exists.
- ProgressBarWrapper * GetProgress() – get the progress indicator.

Write Streams

Class writer

The writer class provides writing the model to a data buffer (to a file space) in the compact format. It inherits from the base class tape.

The **writer** class implements:

1. Static functions for creating the class instance:

- writer_ptr CreateWriter(std::unique_ptr< iobuf_Seq >, uint16) – create the class instance for a given sequential memory buffer of type iobuf_Seq.
- writer_ptr CreateMemWriter(membuf &, uint8) – create the class instance for a given memory buffer of type membuf.

2. 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 an unsigned 64-bit integer; returns the number of written bytes.
- void writeInt64(const int64 &) – write a 64-bit integer; returns the number of written bytes.
- writer & __writeChar(const char *) – write a CHAR string to the stream (ANSI encoding, Russian locale).
- writer & __writeWchar(const TCHAR *) – write a WCHAR string to the stream (stored in the stream as UTF-16).
- writer & __writeWcharT(const wchar_t *) – write a WCHAR string to the stream (stored in the stream as UTF-16).
- size_t __lenWchar(const TCHAR *) – get the length of a WCHAR string record in the stream (stored in the stream as UTF-16).

3. Methods for writing a model tree and access to it (these methods are not supported by this class, they 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 writer_ex class provides writing the model to a data buffer (to a set of file spaces) in the extended format. It inherits from the writer class.

The writer_ex class implements:

1. Static functions for creating the class instance:

- std::unique_ptr< writer_ex > CreateWriterEx(std::unique_ptr< iobuf_Seq >, uint16) – create the class instance for a given sequential memory buffer of type iobuf_Seq.
- std::unique_ptr< writer_ex > CreateMemWriterEx(membuf &, uint8) – create the class instance for a given memory buffer of type membuf.

2. 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.

Read Streams

Class reader

The reader class provides reading from a data buffer (a file space) written in the compact format. It inherits from the base class tape.

The **reader** class implements:

1. Static functions for creating the class instance:
   - reader_ptr CreateReader(std::unique_ptr< iobuf_Seq >, uint16) – create the class instance for a sequential memory buffer.
   - reader_ptr CreateMemReader(membuf &, uint8) – create the class instance for a memory buffer.
2. 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 an unsigned 64-bit integer from the buffer.
   - bool readInt64(int64 &) – read a 64-bit integer from the buffer.
   - int readByte() – read a byte from the buffer.
   - bool readBytes(void *, size_t) – read a bytes sequence from the buffer.
3. Methods for managing reading of the model tree and partial reading of the model objects (these methods are not supported by this class, they 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 the 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 full).
   - void SetFullRead(bool) – set an indicator of full reading of the current object (read the current object in full).
4. Method to get reading errors:
   - uint32 GetLastError() – get reading errors.
5. Methods for working with the progress indicator:
   - void InitProgress(IProgressIndicator *) – initialize the progress indicator.
   - void InitProgress(ProgressBarWrapper &) – initialize the progress indicator.

Class reader_ex

The reader_ex class 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 reader class.

The reader_ex class implements:

1. Static functions for creating the class instance:

- std::unique_ptr< reader_ex > CreateReaderEx(std::unique_ptr< iobuf_Seq >, uint16) – create the class instance for a sequential memory buffer.
- std::unique_ptr< reader_ex > CreateMemReaderEx(membuf &, uint8) – create the class instance for a memory buffer.

2. 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 the 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 full).
- void SetFullRead(bool) – set an indicator of full reading of the current object (read the current object in full).

3. Method to get reading errors:

- uint32 GetLastError() – get reading errors.

Scoped Progress Indicator for Reading Classes

The **ScopedReadProgress** class 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()().

Read-Write Stream

The rw class provides reading and writing the model in the data buffer (in a file space) in the compact format. It inherits from the reader and writer classes.

The class implements a static function for creating the class instance:

- rw_ptr CreateMemWriter(membuf &, uint8).

Model Tree

The classes for the model tree are defined in the **io_tree.h** file.

A tree node data is declared as the MbItemData structure.

The IModelTreeNode class 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 IModelTreeNode class contains ordered arrays of pointers to the immediate descendants and the immediate ancestors of the node.

The IModelTreeNode class 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 a 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 IModelTree class defines an interface for a generic model tree, which is used for creating, writing and reading 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 an object to the model tree and for filling the node data.

The IModelTree class defines the TreeType enumeration for tree types:

- mtt_Model – the tree contains a standard model (the MbModel object).
- mtt_Embodiment – the 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 of nodes, selected by filters, from the current tree. 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 a pointer to the embodiments tree; returns NULL if not applicable (in case of a 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 can 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 IEmbodimentNode class 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 of type mtt_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() const – access to the immediate node children.
- void AddChild(IEmbodimentNode * child) – add a child.

The IEmbodimentTree class defines an interface for an embodiment tree which presents a hierarchy of variants of model implementation (embodiments). Each node of the tree presents one 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.

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 **tape.h** file.

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. A not registered object is written inside the object which refers to it.

Types of streaming object registration

Types of streaming object registration are defined in the RegistrableRec enumeration:

- noRegistrable – not registered object.

- registrable – registered object.

Base class of streaming objects TapeBase

The ClassDescriptor class defines a packed name of a 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 the 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.

Modes of Streaming Operations

Modes of streaming operations are described in the io:: mode_flags enumeration in the **io_buffer.h** file:

- in – open stream for reading.
- out – open stream for writing.
- trunc – open an existing file and clear its content.
- speedOnClose – sort while closing.
- delIfEmpty – delete the file if it is empty.
- delOnClose – delete the file while closing.
- recovery – recovery mode.
- appSpecial – an auxiliary flag of the 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, it is rewritten.
- openExisting – open an existing file; an error is generated if the file does not exist.
- openAlways – open an existing file; if the file does not exist, a new file is created.
- truncExisting – open the file with clearing its content.

Stream States

Flags of stream state are defined in the state enumeration in the **io_buffer.h** file:

- 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 – cannot open the storage.
- cantCreateStore – cannot create a storage.
- badSig – there is no signature or the signature is wrong.
- cantReadCatalog – cannot read the storage catalog.
- cantWriteCatalog – cannot write the catalog of the storage.
- cantFind – cannot find a file in the catalog.
- cantRead – cannot read the file.
- cantWrite – cannot write the file.
- badClassId – the class identifier was not found in the database.
- doubledClassId – attempt of 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 for the file.
- cantWriteObject – cannot write the object (the stream version is newer than the new object class occurrence version).
- underflow64to32 – cannot read a file with 64-bit data in a 32-bit task (loss of upper word uint32 while reading uint64 in a 32-bit task).
- encrypted – the file is protected or encrypted.
- skippedUnknown – partial read of a file in the extended format (unknown objects skipped).
- readAborted – file reading aborted by user.
- allMask – all errors.

3.18.3 Working with Streaming Buffer

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.

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.

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 of the buffer iobuf_Seq and the offset in this cluster.

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).
  - VERSION 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.

Streaming Buffer with Arbitrary Access

The class VERSION 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.

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.

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).

3.18.4 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*.

3.19 ATTRIBUTES

Attributes are used to store information complementing an object's geometric shape. Different attribute types can store different data that can be used to identify objects, define data of an application subject area, and transfer data between different applications during import and export.

Attributes can be stored in any object that is a inheritor of the MbAttributeContainer class.

Attributes comprise the next families:

1. **Simple attributes**. These attributes include attribute types that contain object's ID data, visual properties and physical characteristics.
2. **Generalized attributes**. These attributes are used to store data in one of the standard types, for example, integer, real number, string, point, vector, pointer. Applications can use these attributes to exchange specific information.
3. **Linking attributes**. These attributes are used to link a geometric core object to an abstract data container, whose set of data, format and semantic value can not be described by other attribute types.
4. **Directive attributes**. These attributes determine the purpose of the object or the actions that need to be performed with it. For example, the «subtraction» attribute implies that some solid can be subtracted from another solid.
5. **Product attributes**. These attributes are used to store general information about the product.
6. **Custom attributes**.

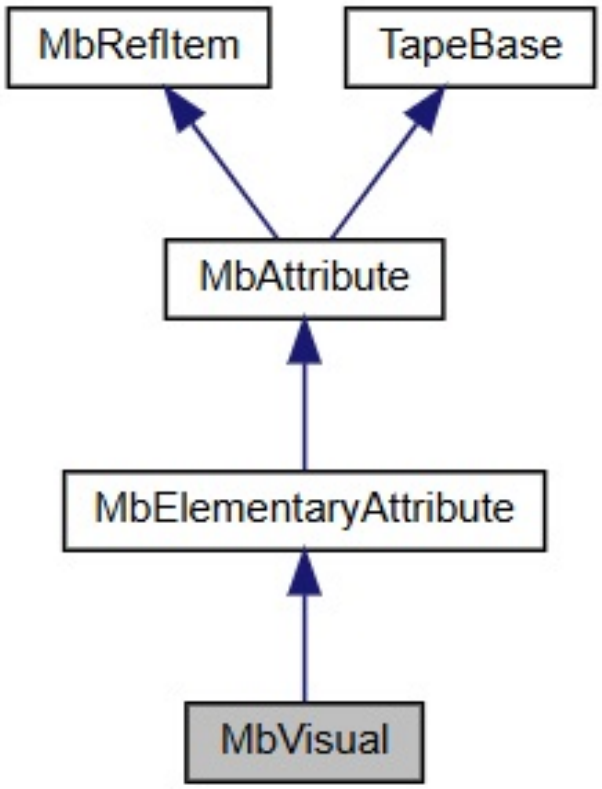
7. **External attributes**.

An example of using a custom attribute is in the *test_sample_attributes.cpp* file, supplied with the test application code in the C3D kernel distribution.

3.19.1 MbVisual Attribute Class for Storing Optical Properties of Geometric Objects for Display Properties

The MbVisual class is intended for specifying optical properties of geometric objects in C3D Modeler (it is recommended to use it instead of the deprecated MbColor class). Declarations and descriptions of the class members are located in the *attr_color.h* file.

The inheritance diagram of MbVisual is shown in the figure:



In previous versions of the C3D geometric kernel (before version 117978) information about the color of a solid and its child topological objects (faces, edges, vertices) was stored in the MbColor attribute. This attribute can store four color components in the RGBA model. However, more parameters are used to visualize an object in 3D applications, and in the absence of these parameters default values are used, which in some cases leads to differences in the resulting colors when opening the model in different programs.

Since version 117978 of the geometric kernel, only the MbVisual attribute is used to store all data required for visualization, and it contains detailed information about each visualization component.

The MbVisual attribute contains the following components:

1. The *_ambient* component stores RED, GREEN, BLUE colors and the ambient coefficient in the range 0.0–1.0 (0–255 in UInt8).

2. The *_diffuse* component stores RED, GREEN, BLUE colors and the diffuse reflection coefficient in the range 0.0–1.0 (0–255 in UInt8).
3. The *_specular* component stores RED, GREEN, BLUE colors and the specular light reflection coefficient in the range 0.0–1.0 (0–255 in UInt8).
4. The *_emission* component stores RED, GREEN, BLUE colors and the emission coefficient in the range 0.0–1.0 (0–255 in UInt8).
5. The *_shininess* component stores shininess (the exponent in the specular reflection law) in the range 0–128 UInt8.
6. The *_opacity* component stores the opacity coefficient (total reflection coefficient) in the range 0.0–1.0 (0–255 in UInt8).
7. The *_chrom* component stores the coefficient of specular reflection of objects in the range 0.0–1.0 (0–255 in UInt8).

Note – according to the OpenGL specification, the *_shininess* component can have values from 0 to 128 and the float type.

To store components 1–4, the MbRGBA type is used, where the RGB fields store color values and the A field stores the color usage coefficient.

Below are the methods used when only the diffuse color is used:

1. When setting color for a solid (face, edge, vertex) by the MbAttributeContainer:: SetColor(int iR, int iG, int iB) method, a MbVisual attribute is obtained; if it does not exist, it is created. The RGB color is written to the *_diffuse* component, and 255 (1.0, 100%) is written to the alpha channel (A).
2. When obtaining color by the uint32 MbAttributeContainer:: GetColor() const method, a MbVisual attribute is obtained; if it does not exist, the default color (MB_C3DCOLOR) is returned. If the attribute exists, the color is obtained from the *_diffuse* component (RGB).

When reading files of old versions containing the MbColor attribute, this attribute is converted to MbVisual as follows:

1. When reading a file that contains only MbVisual (new version), all color attributes and mixing coefficients are used according to the MbVisual specification.
2. When reading files of previous versions that contain only MbColor, a new MbVisual attribute is created; the color from MbColor is written to the *_diffuse* component, and 255 (1.0, 100%) is written to the alpha channel. The MbColor attribute is deleted.
3. When reading files of previous versions containing both MbColor and MbVisual, where each component (RGB) of MbVisual contains a mixing coefficient (the previous usage variant of the MbVisual attribute), and MbColor stores the base color, the following actions are performed:
   a) A new MbVisual attribute is created. The base color from the read MbColor is written to all components (*_ambient*, *_diffuse*, *_specularity* and *_emission*).
   b) The values from the read MbVisual file are written to the alpha channel of each component.
   c) The read MbColor and MbVisual attributes are deleted, and only the new MbVisual attribute remains.
4. When saving a C3D file in an old version format, the MbColor attribute is created, and the color from the *_diffuse* component of the new MbVisual attribute is written to it. Then another MbVisual attribute is created, and the coefficients from the alpha channels of the

components of the existing MbVisual attribute are written to it. The MbVisual attribute that was in the model (new variant) is deleted, and the file is written in the old format.

The MbColor attribute is deprecated and not recommended for use. Instead, the MbVisual attribute must be used.

To create an attribute, it is recommended to use the following static methods:

1. static SPtr<MbVisual> Create(uint32 **c**);
2. static SPtr<MbVisual> Create(int iR, int iG, int iB);
3. static SPtr<MbVisual> Create(const MbRGBA & **c**);

The input parameters of the methods are:

- **c** – color (32 bits). To construct it from components, inline functions uint32 RGB2uint32() can be used (located in the *attr_color.h* file).
- iR, iG, iB – color components.

As a result of these functions a created MbVisual object is returned.

The method

```
bool Init( const MbAttribute & ) override;
```

allows initializing data from the passed attribute.

The method

```
void Init( const MbRGBA & ambient,
           const MbRGBA & diffuse,
           const MbRGBA & specular,
           const MbRGBA & emission,
                 uint8    shininess,
                 uint8    opacity,
                 uint8    chrom );
```

allows setting a value for each visualization component used.

The method

```
void Get( MbRGBA & ambient,
          MbRGBA & diffuse,
          MbRGBA & specular,
          MbRGBA & emission,
          uint8 &  shininess,
          uint8 &  opacity,
          uint8 &  chrom ) const;
```

allows obtaining a value of each visualization component used.

The methods

```
float        Shininess()       const; // Get shininess (the exponent in the specular␣
↪reflection law).
float        Opacity()         const; // Get the opacity coefficient.
float        Chrom()           const; // Get the specular reflection coefficient of␣
↪objects.
```

(continues on next page)

(continued from previous page)

```
const MbRGBA & GetAmbient()    const; // Get the ambient coefficient.
const MbRGBA & GetDiffuse()    const; // Get the diffuse reflection coefficient.
const MbRGBA & GetSpecularity() const; // Get the specular light reflection␣
↪coefficient.
const MbRGBA & GetEmission()   const; // Get the emission coefficient.
const MbUuid & GetIdentifier() const; // Get the identifier.
uint8        GetShininess()    const; // Get shininess (the exponent in the specular␣
↪reflection law).
uint8        GetOpacity()      const; // Get the opacity coefficient.
uint8        GetChrom()        const; // Get the specular reflection coefficient of␣
↪objects.
```

provide individual properties for OpenGL.

The methods for setting individual properties for OpenGL are:

```
void SetShininess( float v );            // Set shininess (the exponent in the␣
↪specular reflection law).
void SetOpacity( float v );              // Set the opacity coefficient.
void SetChrom( float v );                // Set the specular reflection coefficient␣
↪of objects.
void SetAmbient( const MbRGBA & v );     // Set the ambient coefficient.
void SetDiffuse( const MbRGBA & v );     // Set the diffuse reflection coefficient.
void SetSpecularity( const MbRGBA & v ); // Set the specular light reflection␣
↪coefficient.
void SetEmission( const MbRGBA & v );    // Set the emission coefficient.
void SetIdentifier( const MbUuid & v );  // Set the identifier.
void SetShininessInt( uint8 v );         // Set shininess (the exponent in the␣
↪specular reflection law).
void SetOpacityInt( uint8 v );           // Set the opacity coefficient.
void SetChromInt( uint8 v );             // Set the specular reflection coefficient␣
↪of objects.
```

Methods for working with the color component (to be used when only color is used):

```
uint32 GetColor() const { return MbRGBA( _diffuse[0], _diffuse[1], _diffuse[2], 0␣
↪).Get(); } // Return the color component.
void   SetColor( uint32 c );                 // Set the color component.
void   SetColor( const MbRGBA & c );         // Set the color component.
void   SetColor( int iR, int iG, int iB );   // Set the color component.
```

Deprecated methods, not recommended for use (they are used for compatibility with previous versions of the C3D kernel)

```
MbVisual( float a = MB_AMBIENT,
          float d = MB_DIFFUSE,
          float s = MB_SPECULARITY,
          float h = MB_SHININESS,
          float t = MB_OPACITY,
          float e = MB_EMISSION );

MbVisual( uint8 a, uint8 d, uint8 s, uint8 h, uint8 t, uint8 e );
```

(continues on next page)

(continued from previous page)

```
void Init( float a = MB_AMBIENT,
           float d = MB_DIFFUSE,
           float s = MB_SPECULARITY,
           float h = MB_SHININESS,
           float t = MB_OPACITY,
           float e = MB_EMISSION,
           uint  rgb = 0 );

// Allows obtaining a value of each visualization component used.
void Get( float & a, float & d, float & s, float & h, float & t, float & e, uint rgb␣
↪= 0 ) const;

float Ambient( uint rgb = 0 )     const; // (deprecated) Get the ambient coefficient.
float Diffuse( uint rgb = 0 )     const; // (deprecated) Get the diffuse reflection␣
↪coefficient.
float Specularity( uint rgb = 0 ) const; // (deprecated) Get the specular light␣
↪reflection coefficient.
float Emission( uint rgb = 0 )    const; // (deprecated) Get the emission coefficient.

void SetAmbient( float v, uint rgb = 0 );     // (deprecated) Set the ambient␣
↪coefficient.
void SetDiffuse( float v, uint rgb = 0 );     // (deprecated) Set the diffuse␣
↪reflection coefficient.
void SetSpecularity( float v, uint rgb = 0 ); // (deprecated) Set the specular light␣
↪reflection coefficient.
void SetEmission( float v, uint rgb = 0 );    // (deprecated) Set the emission␣
↪coefficient.
```

3.20 CONSTRUCTING TRIANGULATION

C3D geometric kernel constructs a polygonal representation of a 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. The polygonal representation is used to visualize a geometric model, calculate inertial characteristics and detect collisions of model elements.

3.20.1 Triangulation Calculation Control

The MbStepData structure is used as input to the polygonal representation construction methods, and is shown in Fig. 1:

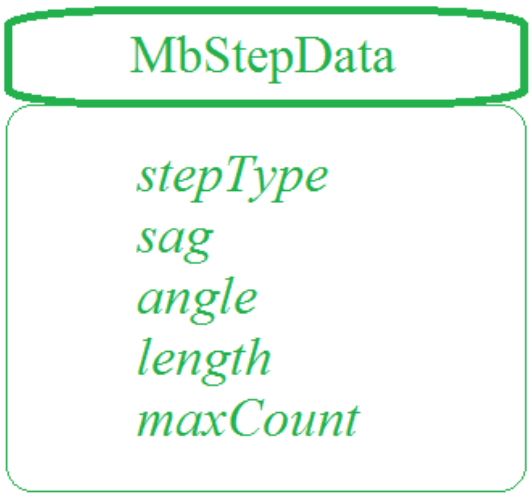


Fig. 1

The *MbStepData* structure is declared in the *mb_data.h* file. The *MbStepData* structure contains the following data:

- *unsigned char stepType* – field that defines the method for calculating the parameter increment;
- *double sag* – maximum allowable deviation by sag;
- *double angle* – maximum allowable deviation by the angle of tangents or normals;
- *double length* – maximum allowable distance between neighboring points;
- *unsigned int maxCount* – maximum number of cells in a row or column of the triangulation grid.

The *MbStepData* structure controls the density of the polygonal object mesh and contains data used to compute the parameter step when moving along curves and surfaces of the model. The *stepType* field defines the method for computing the parameter increment when moving along a curve or surface and can contain masks of the MbeStepType enumeration declared in the *mb_enum.h* file:

- *ist_SpaceStep* – for geometric shape visualization;
- *ist_DeviationStep* – for construction operations;
- *ist_MetricStep* – for 3D printers;
- *ist_ParamStep* – for visualizing the geometric shape of objects with texture binding to the surface parameters;
- *ist_CollisionStep* – for detecting collisions of model elements;
- *ist_MipStep* – for calculating inertial characteristics.

The *sag* parameter limits the parameter step of a curve or surface based on the maximum allowable deviation of the polygonal object from the original by sag. The *angle* parameter limits the parameter step of a curve or surface based on the maximum allowable deviation of the polygonal object from the original by the angular deviation of curve tangents or surface normals

at neighboring points at the step distance. The *length* parameter limits the parameter step of a curve or surface based on the maximum allowable size of a polygonal object element – the side of a triangle or a polygon segment. The *maxCount* parameter limits the parameter step of a curve or surface based on the maximum allowable number of divisions in a row or column of the triangulation grid.

The *MbFormNote* structure is also used as input to the polygonal representation construction methods, and is shown in Fig. 2:

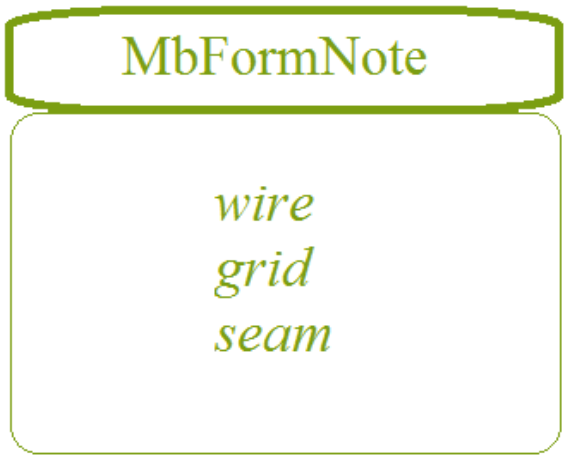


Fig. 2

The *MbFormNote* structure is declared in the *mb_data.h* file. The *MbFormNote* structure contains the following data:

- *bool wire* – polygonal object construction flag;
- *bool grid* – polygonal object construction flag;
- *bool seam* – seam edge usage flag.

The *MbFormNote* structure defines how the polygonal object is constructed: if *wire* == true, the polygonal object is filled with polyline segments, if *grid* == true, the polygonal object is filled with triangulations. The *seam* parameter defines how seams are represented in the polygonal object: if *seam* == true, triangulation is not closed at seams and seam edges are treated as ordinary edges – their points are considered boundary points in the triangulation and polygons are constructed for them; if *seam* == false, triangulation is closed at seams and seam edges are ignored – triangulations are constructed as closed, and polygons for seams are not constructed.

3.20.2 Constructing a Polygonal Object

The virtual method of geometric model objects

```
MbItem * MbItem::CalculateMesh( const MbStepData &       stepData,
                                const MbFormNote &       note,
                                      MbRegDuplicate *   iReg ) const;
```

constructs a polygonal object that approximates the given geometric model object.

The input parameters of the method are:

- *stepData* – data for calculating the approximation step;
- *note* – method of constructing the polygonal object;
- *iReg* – registrar of copied objects.

If the method succeeds, it returns a pointer to the newly constructed geometric model object *MbItem*; otherwise it returns null.

The method is declared in the *item.h* file and in the header files of MbItem inheritors.

This method approximates model objects by creating polygonal copies with a similar structure. For a solid, a wireframe and a point frame, the method creates a polygonal object *MbMesh* that approximates the original object and returns a pointer to the created object. Each triangulation *grids*[*i*] of the *MbMesh* object approximates the *i*-th face, each polygon *wires*[*i*] of the *MbMesh* object approximates the *i*-th edge, each apex *peaks*[*i*] of the *MbMesh* object approximates the *i*-th vertex. For a polygonal object *MbMesh* this method creates a polygonal copy. For an insertion this method creates an insertion with the object that this method creates for the insertion content. For an assembly unit this method creates an assembly unit with the objects that this method creates for the assembly unit objects.

The *stepData* parameter controls the density of the polygonal object mesh and contains data for calculating the step when moving along curves and surfaces of the model object. The *note* parameter defines the way to construct the polygonal object. The *stepData* and *note* parameters are described in the *Triangulation_Calculation_Control* paragraph. If *note.wire* == true, the method fills the set of polygon pointers *mesh.wires*; if *note.grid* == true, the method fills the set of triangulation pointers *mesh.grids* (for faces and surfaces), the set of polygon pointers *mesh.wires* (for edges), and the set of apex pointers *mesh.peakes* (for vertices).

The *iReg* parameter can be null. It is used to pass information about already processed objects to nested methods.

Fig. 1 shows a polygonal object of a solid constructed with *note.wire* == false, *note.grid* == true, containing triangulations, polygons and apexes. Fig. 2 shows a polygonal object of a solid constructed with *note.wire* == true, *note.grid* == false, containing only polygons.

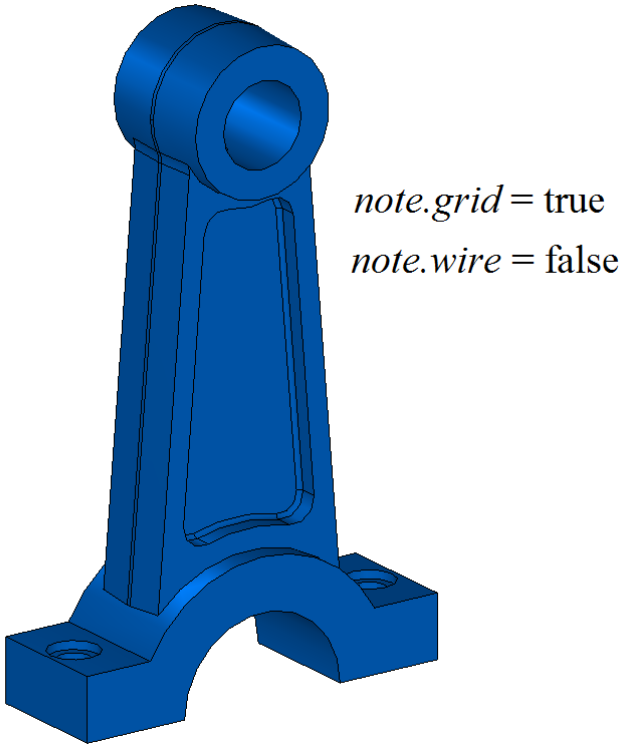


Fig. 1

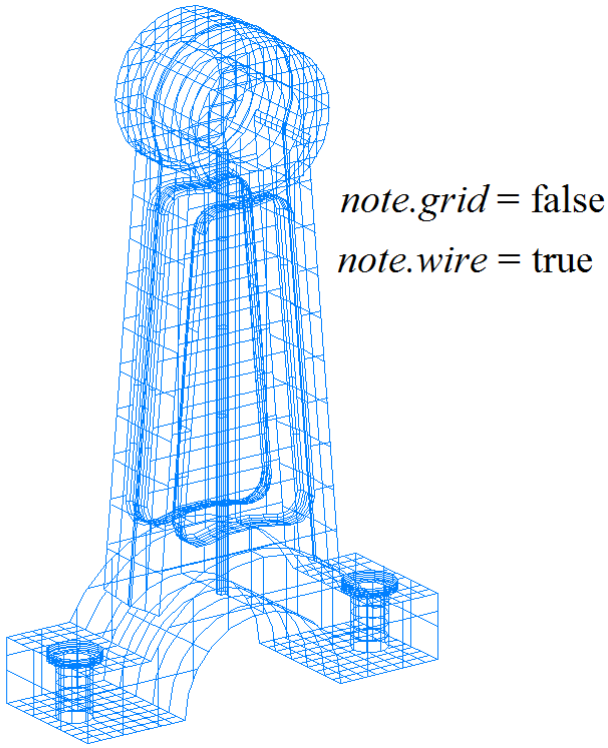


Fig. 2

Fig. 3 shows a polygonal object of an assembly unit constructed with *note.wire* == false, *note.grid* == true, consisting of an assembly unit of polygonal objects approximating the parts:

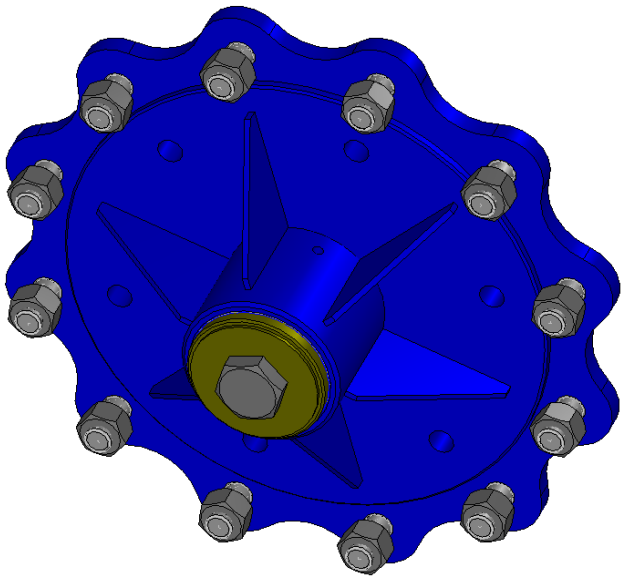


Fig. 3

This method is used to visualize geometric model objects. Polygonal objects are easy to transform, and intersections with a straight line can be found quickly for them. The screen displays exactly the polygonal copy of the geometric model object, while the object in its original representation remains behind the scenes.

3.20.3 Adding a Polygonal Object

The virtual method of geometric model objects

```
bool MbItem::AddYourMesh( const MbStepData & stepData,
                          const MbFormNote & note,
                                MbMesh     & mesh ) const;
```

constructs and adds its own polygonal copy to the supplied polygonal object *mesh*.

The input parameters of the method are:

- *stepData* – data for calculating the approximation step;
- *note* – method of constructing the polygonal object.

The output parameter of the method is the polygonal object *mesh*.

If the method succeeds, it returns true.

The method is declared in the *item.h* file and in the header files of MbItem inheritors.

This method approximates model objects by polygonal copies and adds them to the supplied *mesh* object. For an insertion, the method creates a polygonal copy of the insertion content, transforms it into the global coordinate system and adds it to the supplied *mesh* object. For an assembly unit, the method creates a polygonal copy of the assembly unit content, transforms it into the global coordinate system and adds it to the supplied *mesh* object.

As in the CalculateMesh() method, the *stepData* parameter controls the density of the polygonal object mesh, and the *note* parameter defines the method of constructing the polygonal object. The *stepData* and *note* parameters are described in the *Triangulation_Calculation_Control* paragraph. Unlike the above-mentioned method, this method creates a single polygonal object for complex objects. Fig. 1 shows a polygonal object of the assembly unit shown in *Fig. 3*, constructed by this method:

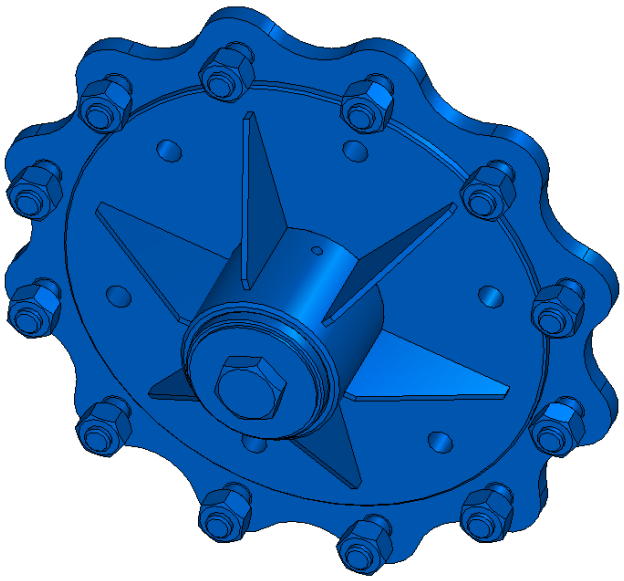


Fig. 1

3.20.4 Constructing Polygons for an Object

The virtual method of three-dimensional geometric objects

```
void MbSpaceItem::CalculateWire( double sag,
                                 MbMesh & mesh );
```

fills the supplied polygonal object *mesh* with a set of polygons that approximate the geometric object.

The input parameter of the method is *sag* – the maximum allowable deviation of a polygonal element from the original by sag.

The output parameter of the method is the polygonal object *mesh*.

The method is declared in the *space_item.h* file and in the header files of MbSpaceItem inheritors.

The polygonal object is described in the subsection *MbMesh Polygonal Object*. The *sag* parameter defines the maximum allowable distance between the object and the polyline passing through the polygon points. This method fills only the set of polygon pointers *mesh.wires* (polylines).

The method approximates a curve with a single polygon. It approximates a contour with several polygons, each passing along the corresponding contour segment. Fig. 1 shows a curve and its polygonal object consisting of a single polygon.

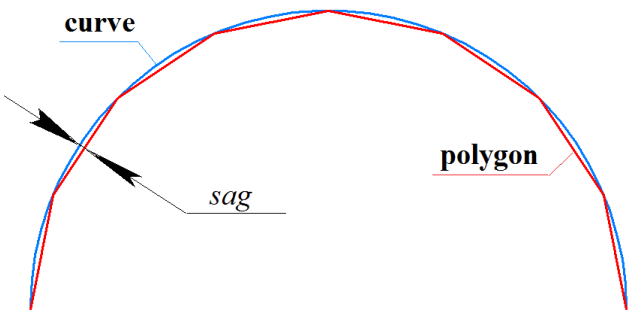


Fig. 1

The method approximates a surface with a set of polygons passing along *u*-lines, *v*-lines and along the surface boundary. Fig. 2 shows a polygonal object of a surface consisting of several polylines.

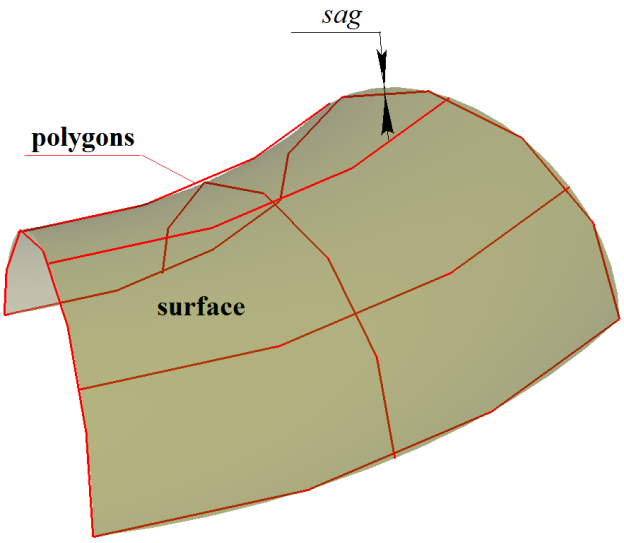


Fig. 2

The method approximates a solid with a set of polylines passing inside each face along the *u*-lines and *v*-lines of the face surfaces and with a set of polylines approximating the curves on which the solid edges are based. Fig. 3 (right) shows a polygonal object of a solid:

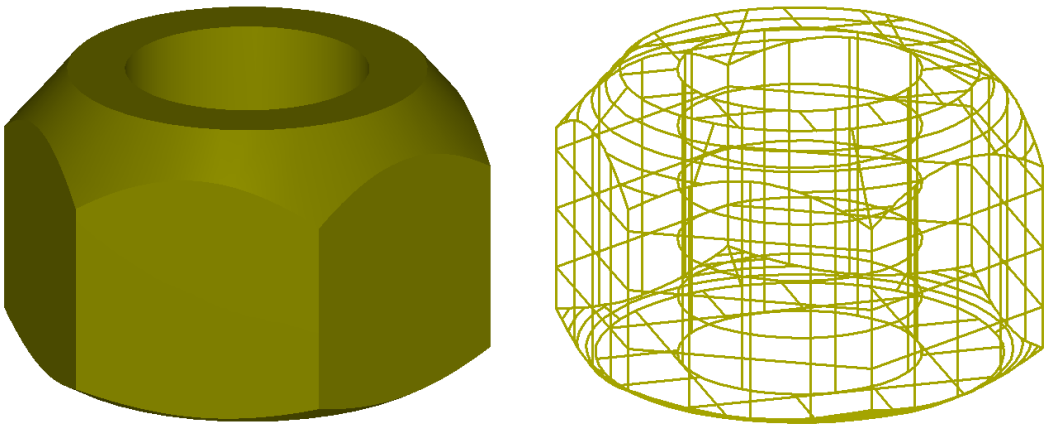


Fig. 3

For geometric model objects, this method works similarly to the CalculateMesh() method when *stepData.stepType* == *ist_SpaceStep*, *stepData.sag* == *sag*, *note.wire* == true, *note.seam* == true.

3.20.5 Constructing Triangulation for a Face

The method

```
void CalculateGrid( const MbFace &     face,
                    const MbStepData & stepData,
                          bool         edgePoints,
                          MbGrid &     grid,
                          bool         dualSeams = true );
```

approximates a face with triangular and quadrilateral plates.

The input parameters of the method are:

- *face* – face;
- *stepData* – data for calculating the approximation step;
- *edgePoints* – flag for using spatial points;
- *dualSeams* – flag for seam processing.

The output parameter of the method is the triangulation *grid*.

The method is declared in the *tri_face.h* file.

The *stepData* parameter controls the density of the triangulation and contains data for calculating the step when moving along the face surface. The *stepData* parameter is described in the subsection *Triangulation Calculation Control*. For different values of *stepData.stepType*, different triangulation data of *grid* are filled. If *stepData.stepType* contains the *ist_MipStep* mask, the *grid.params* set is filled. If *stepData.stepType* contains the *ist_CollisionStep* or *ist_ParamStep* mask, the *grid.params*, *grid.points* and *grid.normals* sets are filled. In all other cases, the *grid.params* and *grid.points* sets are filled.

Fig. 1 shows a triangulation of a planar face with the *ist_SpaceStep* mask:

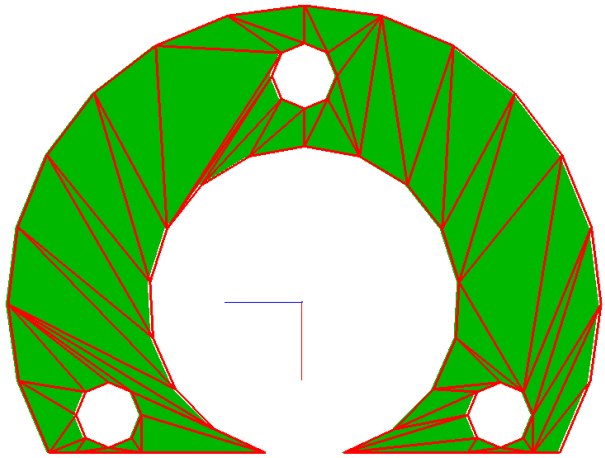


Fig. 1

Fig. 2 shows a triangulation of a curved face with the *ist_SpaceStep* mask.

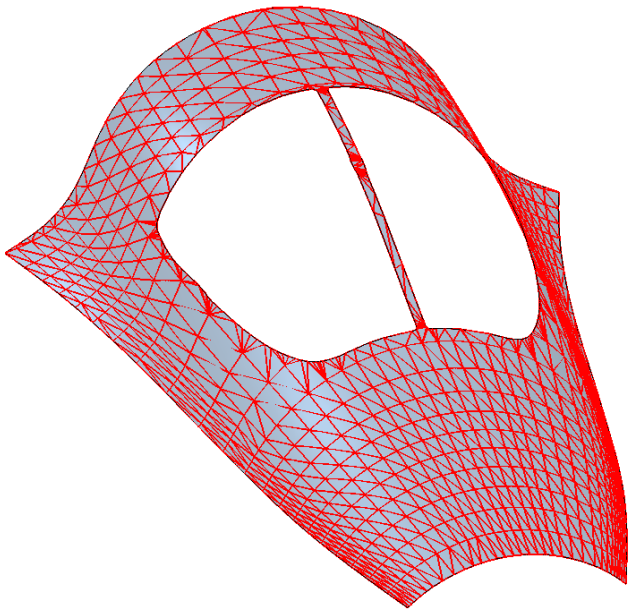


Fig. 2

This method is called when constructing polygonal objects by the CalculateMesh() and AddYourMesh() methods if *note.grid* == true.

3.20.6 Constructing Triangulation for a Body

The method

```
void CalculateGrid( const MbSolid &          solid,
                    const MbStepData &       stepData,
                          RPArray<MbGrid> & grids );
```

builds a set of triangulations for the faces of a solid.

The input parameters of the method are:

- *solid* – solid body;
- *stepData* – data for calculating the approximation step.

The output parameter of the method is the *grids* set.

The method does not return a value.

The method is declared in the *mip_solid_area_volume.h* file.

This method approximates the faces of the *solid* body with triangulations *grids*. When calling the method, the *grids* set must be empty. Each *i*-th face of the *solid* body corresponds to the constructed object *grids*[*i*].

The *stepData* parameter controls the density of polygonal object meshes and contains data for calculating the step when moving along curves and surfaces of solids. The *stepData* parameter is described in the subsection *Triangulation Calculation Control*.

3.20.7 Constructing Polygonal Objects for a Set of Bodies

The method

```
void CalculateGrid( const RPArray<MbSolid> & solids,
                    const MbStepData &       stepData,
                          RPArray<MbMesh> &  meshs );
```

builds a set of polygonal objects for a set of bodies.

The input parameters of the method are:

- *solids* – set of bodies;
- *stepData* – data for calculating the approximation step.

The output parameter of the method is the *meshs* set.

The method does not return a value.

The method is declared in the *mip_solid_area_volume.h* file.

This method approximates the *solids* bodies with polygonal objects *meshs*. When calling the method, the *meshs* set must be empty. Each body *solids*[*i*] corresponds to the constructed object *meshs*[*i*].

The *stepData* parameter controls the density of polygonal object meshes and contains data for calculating the step when moving along curves and surfaces of bodies. The *stepData* parameter is described in the subsection *Triangulation Calculation Control*.

3.21 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 the observation point. Flat projections are more informative if all edges and outlines invisible from the observation point are hidden in the wireframe model.

3.21.1 Data Required to Construct Flat Projections

At the input of the flat projection construction method, the **MbLump** structure is used to pass solids, which is shown in Fig. 1.

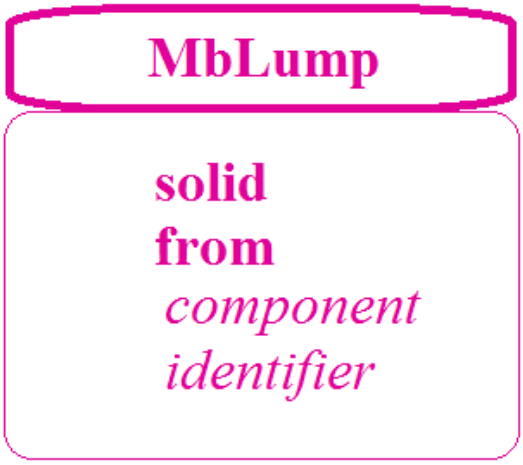


Fig. 1

The **MbLump** structure is declared in the *lump.h* file. The **MbLump** structure contains a pointer to the **solid** body, a transformation matrix **from** that converts the body from its local coordinate system, and identification parameters of the body, *component* and *identifier*.

To pass auxiliary objects to the flat projection construction method, the **MbProjectionsObjects** class is used, which is shown in Fig. 2.

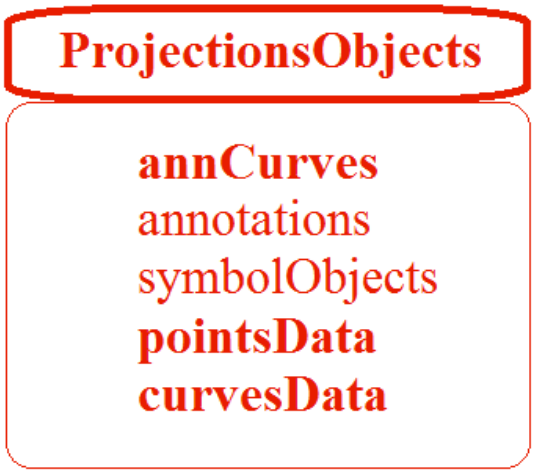


Fig. 2

The MbProjectionsObjects class is declared in the *map_create.h* file. The MbProjectionsObjects class contains the following data:

- TPointer< PArray< MbAnnCurves > > **annCurves** – annotation curves,
- TPointer< RPArray< MbSimbolthThreadView > > annotations – auxiliary objects,
- TPointer< RPArray< MbSymbol > > symbolObjects – conventional notations,
- TPointer< RPArray< MbSpacePoints > > **pointsData** – points,
- TPointer< RPArray< MbSpaceCurves > > **curvesData** – curves.

To pass the results of constructing a flat projection of an object, the **MbVEFVestiges** structure is used, which is shown in Fig. 3.

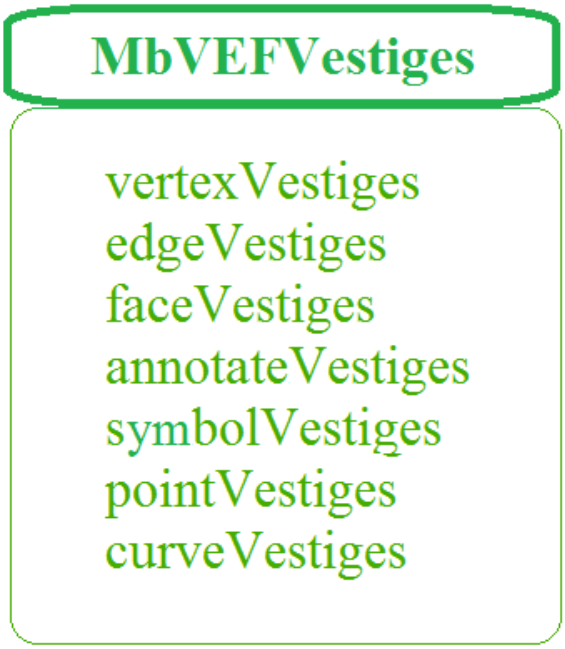


Fig. 3

The **MbVEFVestiges** structure is declared in the *map_vestiges.h* file. The **MbVEFVestiges** structure contains projections of object elements onto the projection plane, grouped together. Each element of a projection group consists of the constructed projection, a pointer to the object that produced the projection, information about the visibility of this projection, and other information about the projection. The **MbVEFVestiges** structure contains the following groups:

- PArray< MbVertexVestige > vertexVestiges – a set of vertex projections,
- PArray< MbEdgeVestige > edgeVestiges – a set of edge projections,
- PArray< MbFaceVestige > faceVestiges – a set of face projections,
- PArray< MbAnnotationVestige > annotateVestiges – a set of annotation object projections,
- PArray< MbSymbolVestige > symbolVestiges – a set of conventional notation projections,
- PArray< MbVertexVestige > pointVestiges – a set of point projections,
- PArray< MbEdgeVestige > curveVestiges – a set of curve projections.

3.21.2 Constructing Model Flat Projection

The method

```
void GetVestiges( const MbPlacement3D &        place,
                  double                       znear,
                  const RPArray< MbLump > &    lumps,
                  const MbProjectionsObjects & objects,
                        MbVEFVestiges &        result,
                        bool                   invisible,
                        VERSION                version );
```

constructs a flat projection of a set of solids and other objects.

The input parameters of the method are:

- **place** – projection plane,
- **znear** – observation point parameter,
- **lumps** – solids projected in local coordinate systems,
- **objects** – other projected objects,
- **invisible** – flag indicating construction of invisible lines,
- **version** – construction version, the latest version is Math::DefaultMathVersion().

The output parameter of the method is **result**, a structure that contains information about the projection.

The method does not return a value.

The method is declared in the *map_create.h* file.

The projection plane is the XY plane of the **place** local coordinate system.

The *znear* parameter determines the type of image. When *znear* = 0, a parallel projection of objects is constructed. The **lumps** parameter contains solids and matrices that transform them from the local coordinate system. The **objects** parameter contains auxiliary objects required to format the projection: auxiliary points, curves, conventional notations, and annotation objects. The *invisible* parameter indicates whether invisible lines must be constructed. In some cases, when invisible lines are not constructed, the method runs noticeably faster.

Fig. 1 shows a solid whose projection lines onto a plane parallel to the screen plane are shown in Fig. 2. Fig. 3 shows only the visible projection lines of the solid presented in Fig. 1.

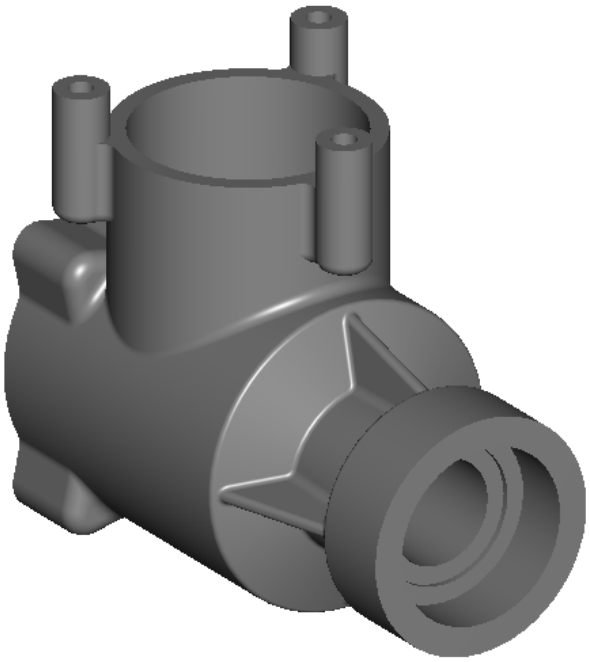


Fig. 1

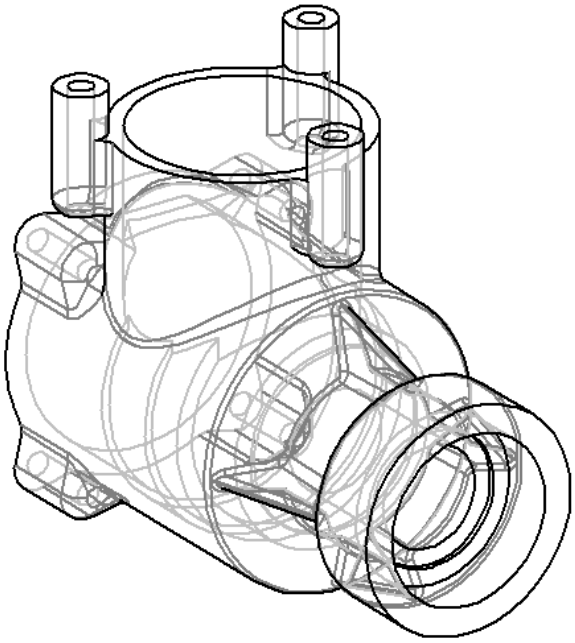


Fig. 2

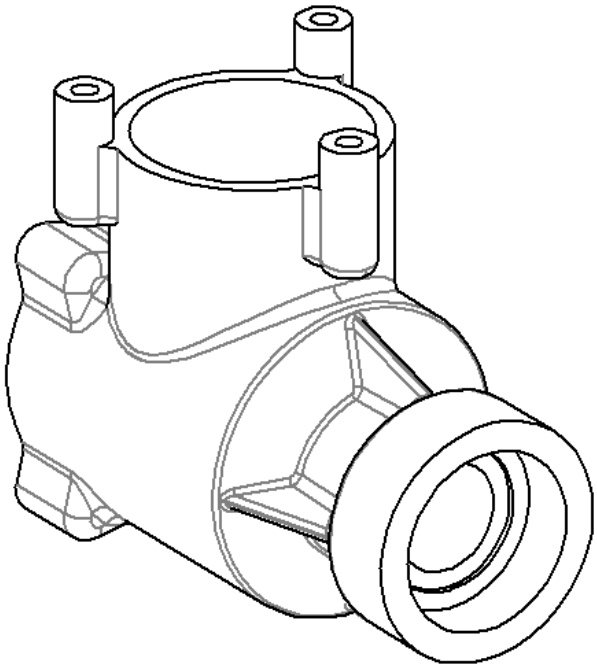


Fig. 3

The last parameter of the method is used to support previous construction versions.

The

```
void VisualLinesMapping( const MbPlacement3D &     place,
                               double              znear,
                         const RPArray< MbLump > & lumps,
                               MbVEFVestiges &     result,
                               bool                invisible = true );
```

method constructs a flat projection of a set of solids and differs from the GetVestiges() method by the absence of additional **objects**.

3.21.3 Constructing Polygonal Projections of Bodies

The method

```
void HiddenLinesMapping( const RPArray< MbLump > &            lumps,
                         const MbPlacement3D &                place,
                               double                         znear,
                               double                         sag,
                               RPArray< MbPolygon3DSolid > &  visibleEdges,
                               RPArray< MbPolygon3DSolid > &  hiddenEdges,
                               RPArray< MbPolygon3DSolid > &  visibleTangs,
                               RPArray< MbPolygon3DSolid > &  hiddenTangs );
```

constructs a polygonal projection of a set of bodies onto a given plane.

The input parameters of the method are:

- **lumps** – bodies projected in local coordinate systems,
- **place** – projection plane,
- **znear** – observation point parameter,
- **sag** – maximum allowable deflection.

The output parameters of the method are:

- visibleEdges – polygons of visible non-smooth edges,
- hiddenEdges – polygons of invisible non-smooth edges,
- visibleTangs – polygons of visible smooth edges,
- hiddenTangs – polygons of invisible smooth edges.

The method does not return a value.

The method is declared in the *map_create.h* file.

The projection plane is the XY plane of the **place** local coordinate system.

The **lumps** parameter contains bodies and matrices that transform them from the local coordinate system. The *znear* parameter determines the type of image. When *znear* = 0, a parallel projection of objects is constructed. The MbPolygon3DSolid class contains a component number and a pointer to a polygon consisting of a set of points whose sequential connection approximates the projection of an edge onto the plane. The *sag* parameter defines the approximation accuracy. The visibleEdges and hiddenEdges polygons represent visible and invisible polygonal projections of non-smooth edges onto the XY plane of the **place** local coordinate system, respectively. The **visibleTangs** and **hiddenTangs** polygons represent visible and invisible

polygonal projections of smooth edges onto the XY plane of the **place** local coordinate system, respectively.

The method

```
void VisualLinesMapping( const RPArray< MbLump > &            lumps,
                         const MbPlacement3D &                place,
                               double                         znear,
                               double                         sag,
                               RPArray< MbPolygon3DSolid > &  visibleEdges,
                               RPArray< MbPolygon3DSolid > &  visibleTangs );
```

constructs only the visible polygonal projection of a set of bodies onto a given plane. This method performs the same constructions as the HiddenLinesMapping() method, with the difference that it constructs only visible projections. In some cases, the VisualLinesMapping() method runs noticeably faster than the HiddenLinesMapping() method.

3.21.4 Constructing a Triangulation Outline

The method

```
void CalculateBoundsSltFast( const MbGrid &                    grid,
                             const MbMatrix3D &                matrix,
                                   bool                        perspective,
                                   RPArray< MbFloatPoint3D > & points );
```

constructs triangulation outline lines.

The input parameters of the method are:

- **grid** – triangulation of a body face,
- **matrix** – matrix that defines the viewing vector,
- **perspective** – flag of perspective display.

The output parameter of the method is:

points – a set of pointers to points from the **grid.points** triangulation.

The method does not return a value.

The method is declared in the *map_create.h* file.

The method is intended for constructing polygonal outline lines and is used to visualize the triangulation silhouette.

3.22 CALCULATION OF INERTIAL CHARACTERISTICS

C3D geometric kernel calculates the 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 the Gauss–Ostrogradsky theorem. Surface integration uses triangulation of a two-dimensional domain 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.

3.22.1 Inertial Characteristics of a Model

To pass inertial characteristics of a model, the InertiaProperties class is used, which is shown in Fig. 1.

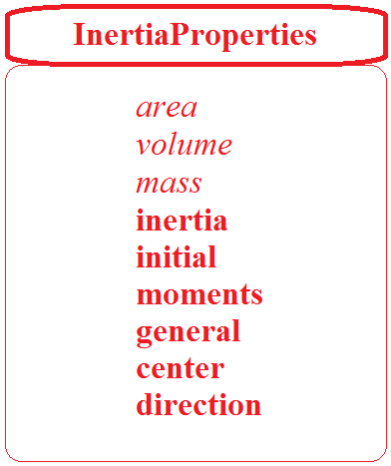


Fig. 1

The InertiaProperties class is declared in the *mip_solid_mass_inertia.h* file. The InertiaProperties class contains the following data about the model:

- double **area** – surface area,
- double **volume** – volume,
- double **mass** – mass,
- double **inertia**[3] – static moments in the source coordinate system,
- double **initial**[3][3] – moments of inertia in the source coordinate system,
- double **moments**[3][3] – moments of inertia in the central coordinate system,
- double **general**[3] – principal central moments of inertia,
- MbCartPoint3D **center** – center of gravity,
- MbVector3D **direction**[3] – direction vectors of principal axes of inertia.

When the InertiaProperties class is initialized, all data is set to zero, and the coordinates of the center of gravity are assumed to be NOT_INITIAL_DBL. The **initial** moments of inertia are calculated in the coordinate system in which the model is defined. The **moments** moments of inertia are calculated in a coordinate system whose origin is at the **center** point and whose coordinate axes coincide with the axes of the source coordinate system in which the model is defined. The **general** moments of inertia are calculated in the principal central coordinate system whose origin is at the **center** point and whose coordinate axes are obtained by computation and coincide with the principal axes of inertia of the model. Centrifugal moments of inertia are absent in the principal coordinate systems.

The **direction** vectors define the directions of the principal axes of inertia. If all principal moments of inertia **general**[*i*], *i* = 1, 2, 3, are different, then all vectors **direction**[*i*], *i* = 1, 2, 3, are nonzero. If all principal moments of inertia **general**[*i*], *i* = 1, 2, 3, are equal, then all vectors **direction**[*i*], *i* = 1, 2, 3, are zero, and any three mutually orthogonal vectors can be used as principal

directions. If two of the three principal moments of inertia are equal, for example **general**[*j*] == **general**[*k*], then two of the three vectors are zero, **direction**[*j*] = **direction**[*k*] = 0, and the nonzero **direction**[*i*] vector defines the direction of the principal axis of inertia whose moment differs from the others; any two mutually orthogonal vectors that are also orthogonal to the nonzero **direction**[*i*] vector can be used as the other two principal directions.

The possibility of using ready-to-use data for individual elements of the model is provided by the SolidMIAttire and AssemblyMIAttire classes, which are shown in Fig. 2 and Fig. 3. The classes are declared in the *mip_solid_mass_inertia.h* file.



Fig. 2



Fig. 3

The SolidMIAttire class contains the following data:

- const MbSolid & **solid** – body,
- double *density* – density or specific mass per unit area,
- MbMatrix3D **matrix** – matrix that transforms the body to the nearest assembly coordinate system,
- InertiaProperties * **properties** – assigned inertial characteristics of the body, which may be null or partially specified,
- bool *ready* – flag indicating that characteristics do not need to be calculated.

The AssemblyMIAttire class contains the following data:

- RPArray< **AssemblyMIAttire** > **assemblies** – set of next-level assembly units,
- RPArray< **SolidMIAttire** > **solids** – bodies of the assembly unit,
- MbMatrix3D **matrix** – matrix that transforms the assembly unit to the nearest assembly coordinate system,
- InertiaProperties * **properties** – assigned inertial characteristics of the assembly unit, which may be null or partially specified,
- bool *ready* – flag indicating that characteristics do not need to be calculated.

If instances of the **SolidMIAttire** and **AssemblyMIAttire** classes contain non-null **properties** and *ready* == true, then inertial characteristics of the corresponding object will not be calculated, and the result will contain data from **properties**.

If instances of the **SolidMIAttire** and **AssemblyMIAttire** classes contain non-null **properties** and *ready* == false, then the result will contain data from **properties** that are not equal to NULL_EPSILON or NOT_INITIAL_DBL. Data in **properties** that are equal to NULL_EPSILON or NOT_INITIAL_DBL will be calculated. These classes allow mixing computed data with assigned data.

3.22.2 Inertial Body Characteristics

The function

```
void MassInertiaProperties( const MbSolid *             solid,
                                  double                density,
                                  double                deviateAngle,
                                  InertiaProperties &   properties,
                                  IfProgressIndicator * progress = 0 );
```

calculates the surface area, volume, mass, center of gravity, and moments of inertia of a body.

The input parameters of the function are:

- **solid** – body,
- **density** – density or specific mass per unit area,
- **deviateAngle** – parameter that controls calculation accuracy.

The output parameters of the function are:

- **properties** – calculated inertial characteristics,
- progress – calculation progress indicator.

The function does not return a value.

The function is declared in the *mip_solid_mass_inertia.h* file.

For a closed **solid** body, the *density* parameter defines the body density. For an open **solid** body, the *density* parameter defines the specific mass per unit area of the body. In the general case, the calculation is performed numerically and uses triangulation of the domain of definition of face surface parameters. Triangulation of the parametric domain of faces is performed by the CalculateGrid() method described in the *Constructing Polygons for an Object* paragraph, with *stepData.stepType* = ist_MipStep and *stepData.angle* = *deviateAngle*. The *deviateAngle* parameter defines the maximum allowable angle between normals of neighboring triangles and quadrilaterals of the surface triangulation. The *deviateAngle* parameter controls calculation accuracy, and the calculation time depends on its value. The *deviateAngle* parameter must be

within the range 0.01 <= *deviateAngle* <= 0.35 radians. Note that for small *deviateAngle* values, complex models require a long calculation time.

The **properties** inertial characteristics are described in the *Constructing Polygons for an Object* paragraph. The progress parameter provides information about execution during the calculation and can be used to stop the computation.

3.22.3 Inertial Characteristics for a Set of Bodies

The function

```
void MassInertiaProperties( const RPArray<MbSolid> &            solids,
                            const SArray<double> &              densities,
                            const SArray<MbMatrix3D> &          matricies,
                            const RPArray<InertiaProperties> & mpSolids,
                                  double                        deviateAngle,
                                  InertiaProperties &           properties,
                                  IfProgressIndicator *         progress = 0 )
```

calculates the surface area, volume, mass, center of mass, and moments of inertia for a set of bodies.

The input parameters of the method are:

- **solids** – a set of bodies
- **densities** – a set of densities or specific masses per unit area
- **matricies** – a set of transformation matrices that bring bodies into a common coordinate system
- **mpSolids** – a set of existing body characteristics (may contain null values)
- **deviateAngle** – a parameter that controls the calculation accuracy

The output parameters of the method are:

- **properties** – the calculated inertial characteristics
- **progress** – a calculation progress indicator

The method does not return a value.

The method is declared in the *mip_solid_mass_inertia.h* file.

The number of elements in *densities*, **matricies**, and **mpSolids** must coincide with the number of bodies in **solids**. The second parameter defines the density of a body for closed bodies or the specific mass per unit area for open bodies. The **matricies** parameter contains transformation matrices that bring bodies into the coordinate system in which the calculation is to be performed. The **mpSolids** parameter contains assigned characteristics of the corresponding bodies that must replace the corresponding characteristics obtained as a result of the calculation. This method can be used to compute the inertial characteristics of an assembly unit whose elements' inertial characteristics were previously calculated in their local coordinate systems.

In the general case, the calculation is performed numerically and uses triangulation of the parameter domain of face surfaces. Triangulation of the parametric domain of faces is performed by the CalculateGrid() method described in subsection *Constructing polygons for an object*, with *stepData*.*stepType* = ist_MipStep and *stepData.angle* = *deviateAngle*. The *deviateAngle* parameter defines the maximum allowable angle between normals of adjacent triangles and quadrilaterals of the surface triangulation. The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter

must be within the range 0.01 <= *deviateAngle* <= 0.35 radians. Note that for complex models, small *deviateAngle* values require a long computation time.

The **properties** inertial characteristics are described in subsection *Inertial characteristics of a model*. The *progress* parameter provides information on execution during the calculation and can be used to stop the computation.

3.22.4 Inertial Characteristics of a Model

The function

```
void MassInertiaProperties( const AssemblyMIAttire &      assembly,
                                  double                  deviateAngle,
                                  InertiaProperties &     properties,
                                  IfProgressIndicator *   progress = 0 )
```

calculates the surface area, volume, mass, center of mass, and moments of inertia of a model represented as an assembly unit.

The input parameters of the method are:

- **assembly** – an assembly unit that may contain previously computed characteristics
- *deviateAngle* – a parameter that controls the calculation accuracy

The output parameters of the method are:

- **properties** – the calculated inertial characteristics
- progress – a calculation progress indicator

The method does not return a value.

The method is declared in the *mip_solid_mass_inertia.h* file.

The **assembly** parameter represents an analogue of an assembly unit whose elements are similar assembly units in local coordinate systems and bodies with specified density in local coordinate systems. The elements of the assembly unit may have inertial characteristics and control parameters that were computed earlier. If an element has previously computed characteristics, they will be used in the overall sum, which reduces the total computation time.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter defines the maximum allowable angle between normals of adjacent triangles and quadrilaterals of the surface triangulation. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians. Note that for complex models, small *deviateAngle* values require a long computation time.

The **properties** inertial characteristics are described in subsection *Inertial characteristics of a model*. The progress parameter provides information on execution during the calculation and can be used to stop the computation.

3.22.5 Surface Area Calculation

The method

```
double CalculateArea( const RPArray<MbFace> & faces,
                            double            deviateAngle )
```

calculates the surface area of a set of faces.

The input parameters of the method are:

- **faces** – a set of faces
- *deviateAngle* – a parameter that controls the calculation accuracy

The method returns the area of the set of faces.

The method is declared in the *mip_solid_mass_inertial.h* file.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians. In the general case, the calculation is performed numerically and uses triangulation of the parameter domain of face surfaces. Triangulation of the parametric domain of faces is performed by the CalculateGrid() method described in subsection *Constructing polygons for an object*, with *stepData.stepType* = ist_MipStep and *stepData.angle* = *deviateAngle*. The *deviateAngle* parameter defines the maximum allowable angle between normals of adjacent triangles and quadrilaterals of the surface triangulation.

The method

```
double CalculateArea( const MbFace & face,
                            double   deviateAngle )
```

calculates the surface area of a single face.

The input parameters of the method are:

- **face** – a face
- *deviateAngle* – a parameter that controls the calculation accuracy

The method returns the area of the **face** face.

The method is declared in the *tri_face.h* file.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians.

The method

```
double CalculateArea( const MbSurface & surface,
                            double      deviateAngle )
```

calculates the surface area.

The input parameters of the method are:

- **surface** – a surface
- *deviateAngle* – a parameter that controls the calculation accuracy

The method returns the area of the **surface** surface.

The method is declared in the *mip_solid_area_volume.h* file.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians.

The method

```
double CalculateAreaCentre( const MbFace &         face,
                                  double           deviateAngle,
                                  bool             byOuter,
                                  VERSION          version,
                                  MbCartPoint3D &  centre )
```

calculates the surface area and the center of mass of a face.

The input parameters of the method are:

- **face** – a face
- **deviateAngle** – a parameter that controls the calculation accuracy
- **byOuter** – a parameter that ignores inner cuts of the face
- **version** – a parameter that controls calculation versioning

The output parameter of the method is **centre**, the center of mass of the face.

The method returns the area of the face.

The method is declared in the *mip_solid_area_volume.h* file.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians.

The *byOuter* parameter allows excluding inner cuts of the face from the calculation. If *byOuter* = true, the calculation assumes that inner cuts of the face are absent. For a normal calculation of the surface area and the center of mass of a face, *byOuter* should be set to false. The version parameter is required to support old calculation versions.

The method

```
double CalculateAreaCentre( const MbFaceShell &     shell,
                                  double             deviateAngle,
                                  MbCartPoint3D &    centre )
```

calculates the surface area and the center of mass of a set of faces.

The input parameters of the method are:

- **shell** – a set of faces
- *deviateAngle* – a parameter that controls the calculation accuracy

The output parameter of the method is **centre**, the center of mass of the set of faces.

The method returns the area of the set of faces.

The method is declared in the *mip_solid_area_volume.h* file.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians.

3.22.6 Solid Volume Calculation

The method

```
double CalculateVolume( const MbSolid & solid,
                              double    deviateAngle )
```

calculates the volume of a solid.

The input parameters of the method are:

- **solid** – a solid body
- **deviateAngle** – a parameter that controls the calculation accuracy

The method returns the volume of the solid.

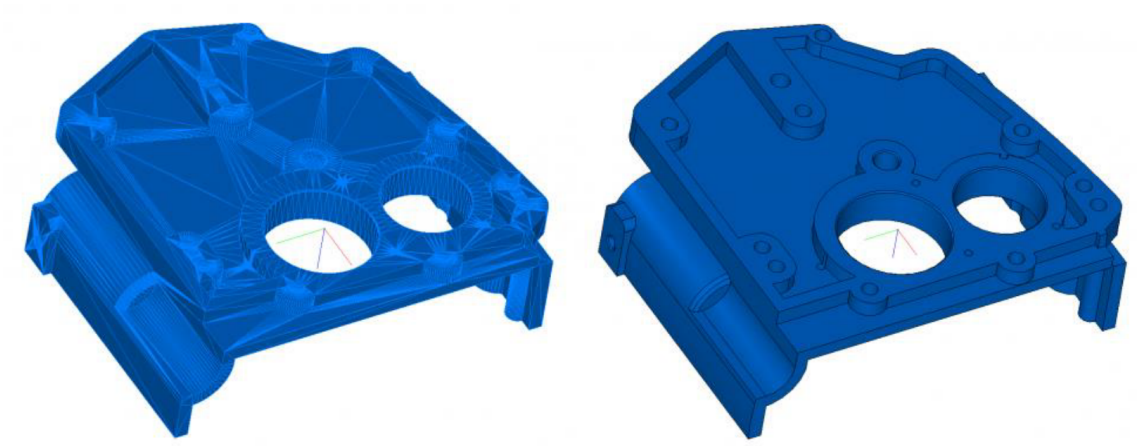
The method is declared in the *mip_solid_area_volume.h* file.

The *deviateAngle* parameter controls the calculation accuracy, and the computation time depends on its value. The *deviateAngle* parameter must be within the range 0.01 <= *deviateAngle* <= 0.35 radians.

CHAPTER 4

C3D B-Shaper

The C3D B-Shaper module converts polygonal models into solid models with boundary representation (B-Rep). As a result of the algorithms execution, a user can edit the resulting model with standard CAD system tools, significantly reducing the complexity and the amount of data.



Application areas of C3D B-Shaper:

- Working in CAD and BIM systems with polygonal models from catalogs of ready-made and standard models of parts, building and structure elements.
- Recognizing surfaces in polygonal models obtained as a result of 3D scanning or topology optimization in CAE systems.
- Mesh smoothing, decimation and compression algorithms in computer graphics.



The unique algorithm of the C3D B-Shaper module splits a polygonal mesh into segments — prototypes of the expected faces. After that, the selected regions can be recognized as elementary surfaces (plane, cylinder, cone, sphere, torus), a surface of revolution or a NURBS surface. Intersection curves are calculated between neighboring segments, and they are then used to construct the edges of the body faces.

Work with the module is carried out via API. Two usage modes are provided: fully automatic and interactive. It should be noted that the automatic shell recognition mode generally is not suitable for models obtained as a result of 3D scanning or topology optimization. This mode is intended for use on polygonal models that are based on some models created in CAD applications.

For license activation of the module, call the EnableMathModules() function with the appropriate parameters.

4.1 Automatic Shell Recognition Mode by Polygon Mesh

The C3D B-Shaper automatic conversion interface is represented by two functions: ConvertMeshToShell() and 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.

The 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 mesh is checked with respect to the size of the model. Also, the user has the option of switching recognition modes, which allows controlling 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 enumeration:
  - *srm_All* – build all surfaces;
  - *srm_NoGrids* – do not build surfaces based on triangulation;
  - *srm_CanonicOnly* – build only elementary surfaces;
  - *srm_Default* – default mode.

The method

```
MbResultType ConvertCollectionToShell(       MbCollection &           collection,
                                             MbFaceShel *&            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).

The 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».

4.2 MbMeshProcessor Class – Shell Creation

The MbMeshProcessor class is intended for creating a shell based on a polygonal mesh with user-defined settings. The class provides advanced capabilities for controlling segmentation and surface recognition processes.

```
class MbMeshProcessor {

public:
    // Create a processor instance by a collection of elements.
    MbMeshProcessor * Create( const MbCollection & collection );

    // Destructor
    ~MbMeshProcessor();

    // Recognition tolerance control
    void   SetRelativeTolerance( double tolerance );
    void   SetTolerance( double tolerance );
    double GetTolerance() const;

    // Mesh "healing".
    void SetUseMeshSmoothing( bool useSmoothing );

    // Mesh segmentation control.
    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 control.
    void             SetReconstructionMode( MbeSurfReconstructMode mode );
    void             FitSurfaceToSegment( size_t idxSegment );
    void             FitSurfaceToSegment( size_t idxSegment, MbeSpaceType surfaceType␣
↪);
    const MbSurface * GetSegmentSurface( size_t idxSegment ) const;

    // B-Rep model construction.
    MbResultType CreateBRepShell( MbFaceShell *& pShell );

    ..
};
```

The class description is located in the *action_b_shaper.h* file.

More detailed descriptions of the class methods are provided in next subsections.

4.3 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 mesh 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 mesh is checked relative to the dimensions of the current mesh. 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 bounding 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 mesh segments. The method returns the absolute tolerance as a *double* type value.

4.4 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 a face. The transformation of such meshes 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 used 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 GetSegmentedMesh() method.

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 mesh 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.

4.5 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 mesh 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 with surface type

```
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.

CHAPTER 5

C3D PolyShaper

The C3D PolyShaper module is designed for analyzing, healing, and editing polygonal models (meshes), and for converting them to smooth surfaces.

Polygonal representation is a description of geometry by a set of vertices, edges, and planar facets combined into a polyhedral three-dimensional object. Polygonal modeling is used in reverse engineering, topology optimization, engineering simulations, medicine, and preparation of models for 3D printing.

The module is based on a topological representation of a mesh in the form of a doubly linked list of edges, or a Halfedge structure. Logical division of the mesh into parts, or segmentation, is also supported in the form of Halfedge and is essentially an analog of the solid boundary representation. The module can be used via the API. The API includes the MbPolymesh and MbPolymeshTopo classes, which represent a polygonal object, and a set of functions and classes for working with them.

The MbPolymesh object can be constructed manually, obtained from the MbMesh object, or from the MbSolid object by linearization. The MbPolymesh object can also be converted back to the MbMesh object (note that this conversion results in a loss of topological data), which may be necessary if the rendering process uses the MbMesh object. Most of the module functionality is designed to work with MbPolymesh, but some features are also available for MbMesh.

For license activation of the module, call the EnableMathModules() function with the appropriate parameters.

5.1 Features of Implementing a Doubly Linked List of Edges

The Halfedge structure has a long history and is one of the most convenient ways to represent mesh topology. In the C3D PolyShaper module, this structure is implemented in a classic way with some special features. The implementation is the MbHalfedge class, which is located in the *halfedge.h* header file. This is simply an interface for the mesh topology that is provided to the user by the MbPolymeshTopo class.

The MbHalfedge container contains an array of edges MbHalfedgeEdge, an array of vertices

MbHalfedgeVertex, an array of facets MbHalfedgeFacet, and an array of mesh attributes MbHalfedgeAttrBase (not to be confused with MbAttribute).

An edge has references to the next edge, to the previous edge, to the twin edge, to the facet to which the edge belongs, and to the origin vertex (as shown in Fig. 1). If the edge is a boundary, then the reference to the twin edge is undefined.

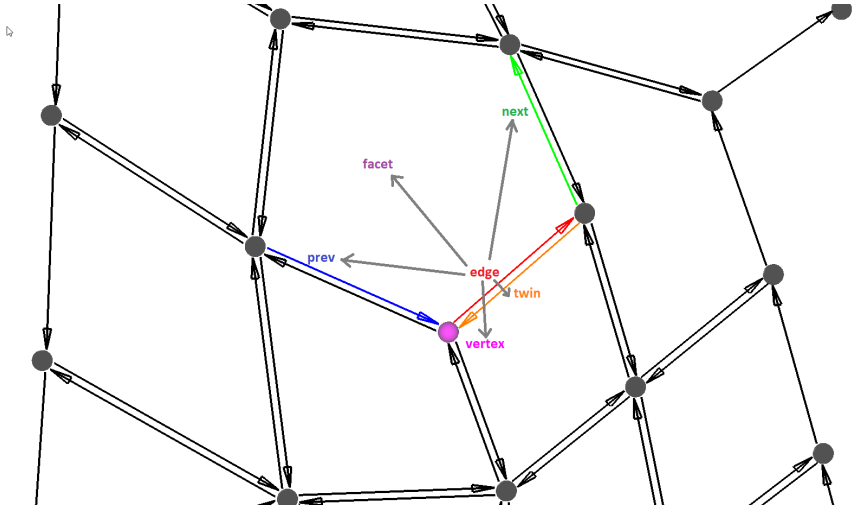


Fig. 1

A vertex has a reference to one of the edges outgoing from it (as shown in Fig. 2). For an internal vertex, this can be any edge from the fan. For a boundary vertex, this must necessarily be a boundary edge (such an edge is the only one in the fan).

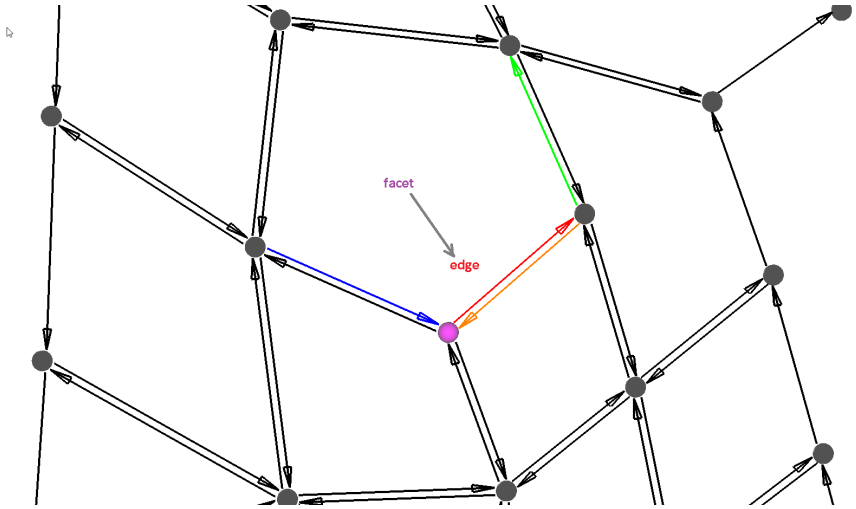


Fig. 2

A facet has a reference to any one of the edges that form it (as shown in Fig. 3). The edges follow each other in a counterclockwise direction. There are no internal loops for the facet. A facet can consist of any number of edges.

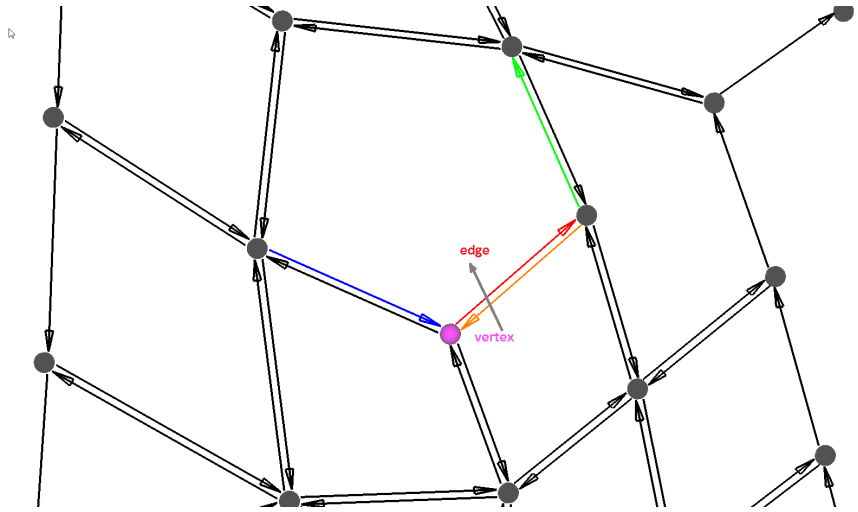


Fig. 3

Any mesh element can be in active and inactive states. For example, when a facet is removed from the mesh, it is not removed from the facet array, but it is set to an inactive state, and the array size does not change. When adding a facet, it first checks that there are inactive elements. If there are inactive elements, one of them is used for the new facet. If there are no inactive elements, the facet array is incremented by the corresponding number of elements.

A set of mesh elements forms the mesh topology only. Recording of geometric or other properties of the mesh is implemented through adding the corresponding attribute to the mesh inherited from the MbHalfedgeAttrBase class. Mesh elements use indexes for linking to each other instead of pointers. This allows you to associate integers, real numbers, 3D or 2D points, vectors, or more complex structures with mesh elements. The attribute types available to the user are located in the header files *mesh_halfedge_attr.h* and *mesh_segm_halfedge_attr.h*. Creating custom attributes is not supported.

For example, to work with 3D points an attribute of the MbHalfedgeAttrPoint3D type must exist in the mesh. Its API allows you to get the corresponding point of the vertex in the mesh, the normal or area of the triangle, and so on.

5.2 Mesh Iterators

There are several types of iterators available for working with the mesh topology. They are located in the header files *halfedge_iterators.h* and *halfedge_range_iterators.h*:

1. EdgeFanIterator is the iterator for traversing the edges of the vertex fan in a counterclockwise direction (see Fig. 1). Example of use:

```
for( EdgeFanIterator it( halfedge, iEd ); !it.Done(); ++it )
    ;

for( EdgeFanIterator it( halfedge, iVx, true ); !it.Done(); ++it ) {

    if( it.Edge() < SYS_MAX_T ) {
        ...
    }
    if( it.Twin() < SYS_MAX_T ) {
```

(continues on next page)

(continued from previous page)

```
        ...
    }
}
```

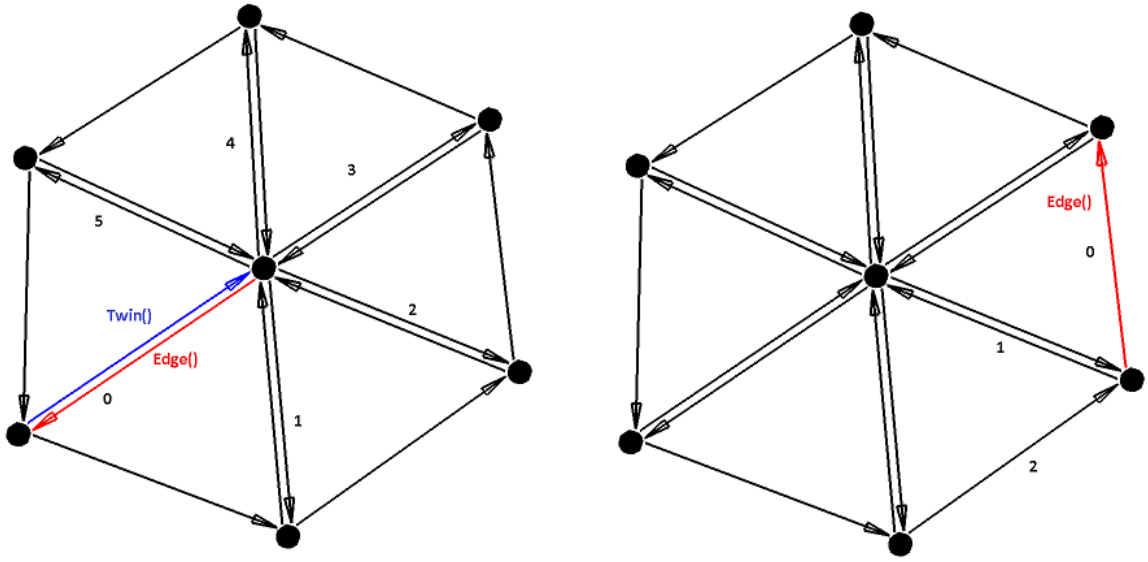


Fig. 1

2. EdgeFacetIterator is the iterator for traversing facet edges, starting from the facet edge in a counterclockwise direction (see Fig. 2).

Example of use:

```
for( EdgeFacetIterator it( halfedge, iFacet ); !it.Done(); ++it ) {

    const MbHalfedgeEdge & ed = halfedge.Edge( it.Edge() );
}
```

3. EdgesAroundFacetBegEndPair AllEdgesAroundFacet(*size_t* iFacet) const;

Range based iterator for traversing facet edges, starting from the facet edge in a counterclockwise direction.

Example of use:

```
for( const MbHalfedgeEdge & ed : halfedge.AllEdgesAroundFacet( iFacet ) ) {
    ...
}
```

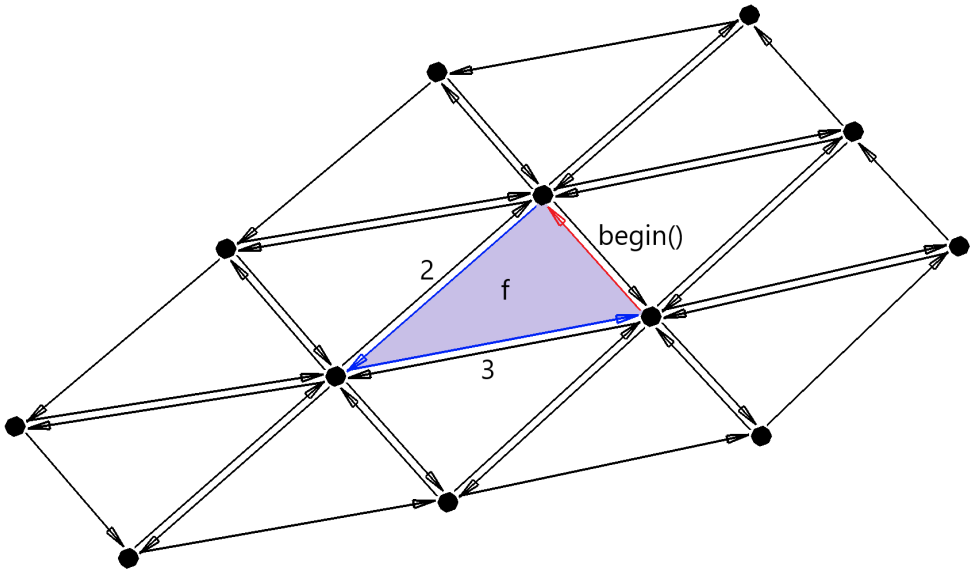


Fig. 2

4. VerticesAroundFacetBegEndPair AllVerticesAroundFacet(*size_t* iF) const;

It is the iterator for traversing all vertices around the facet, starting from the vertex belonging to the edge defining the facet in a counterclockwise order (see Fig. 3).

Example of use:

```
for( const MbHalfedgeVertex & v : halfedge.AllVerticesAroundFacet( iFacet ) ) {
    ...
}
```

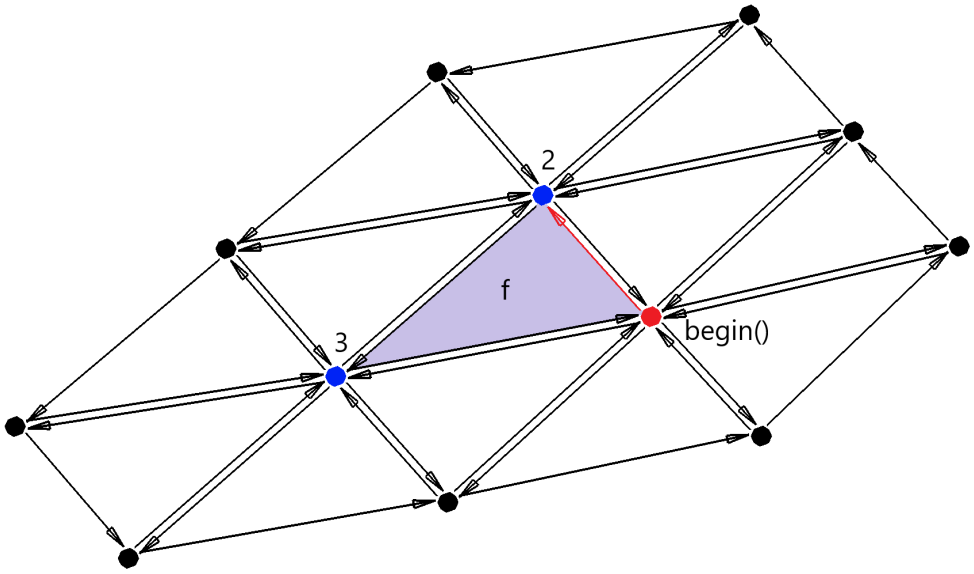


Fig. 3

5. EdgesAroundVertexBegEndPair AllEdgesAroundVertex(*size_t* iV) const;

It is the iterator for traversing all edges (incoming and outgoing) adjacent to a given vertex, starting with the edge that defines the vertex (see Fig. 4). The traversal order is counterclockwise, alternating incoming and outgoing edges.

Example of use:

```
for( const MbHalfedgeEdge & ed : halfedge.AllEdgesAroundVertex( iVx ) ) {
    ...
}
```

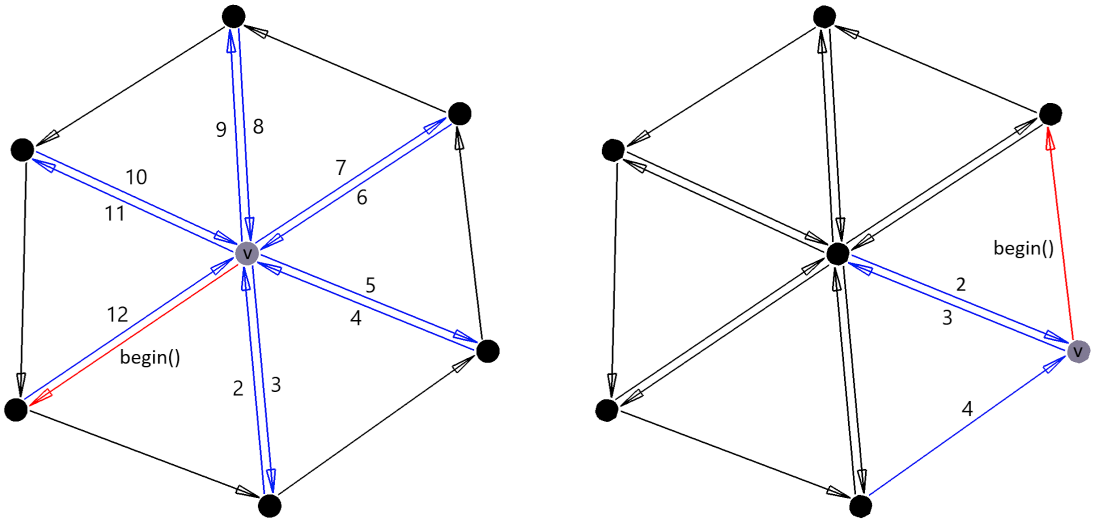


Fig. 4

6. VerticesAroundVertexBegEndPair AllVerticesAroundVertex(*size_t* iV) const;

It is the iterator for traversing all vertices adjacent to a given vertex (see Fig. 5). The order of traversal is counterclockwise, the first vertex is adjacent along the edge that defines the vertex.

Example of use:

```
for( const MbHalfedgeVertex & v : halfedge.AllVerticesAroundVertex( iVx ) ) {
    ...
}
```

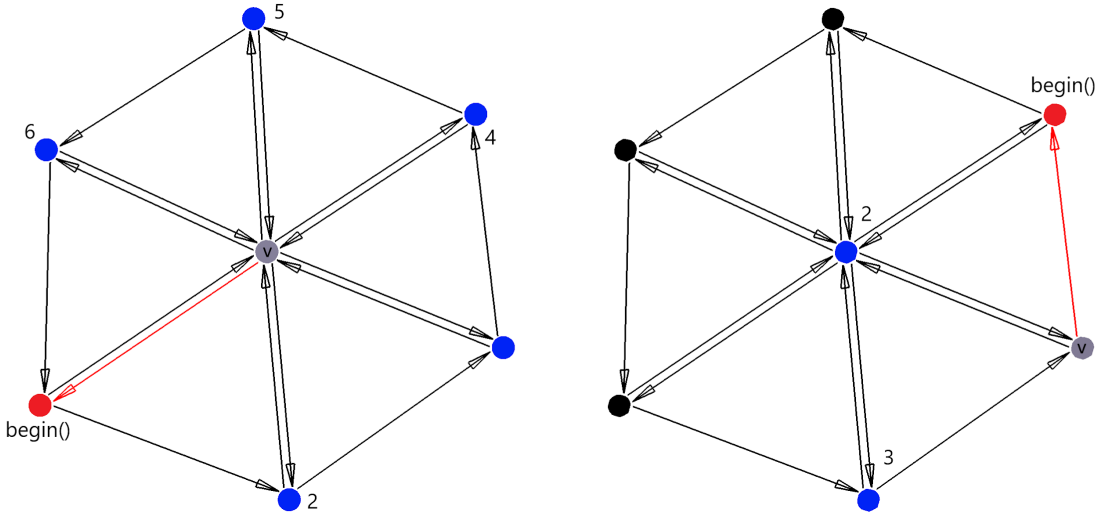


Fig. 5

7. FacetsAroundVertexBegEndPair AllFacetsAroundVertex(*size_t* iV) const;

It is the iterator for traversing all facets including the specified vertex (see Fig. 6). The traversal order is counterclockwise, the first facet includes the edge defining the vertex.

Example of use:

```
for( const MbHalfedgeFacet & f : halfedge.AllFacetsAroundVertex( iVx ) ) {
    ...
}
```

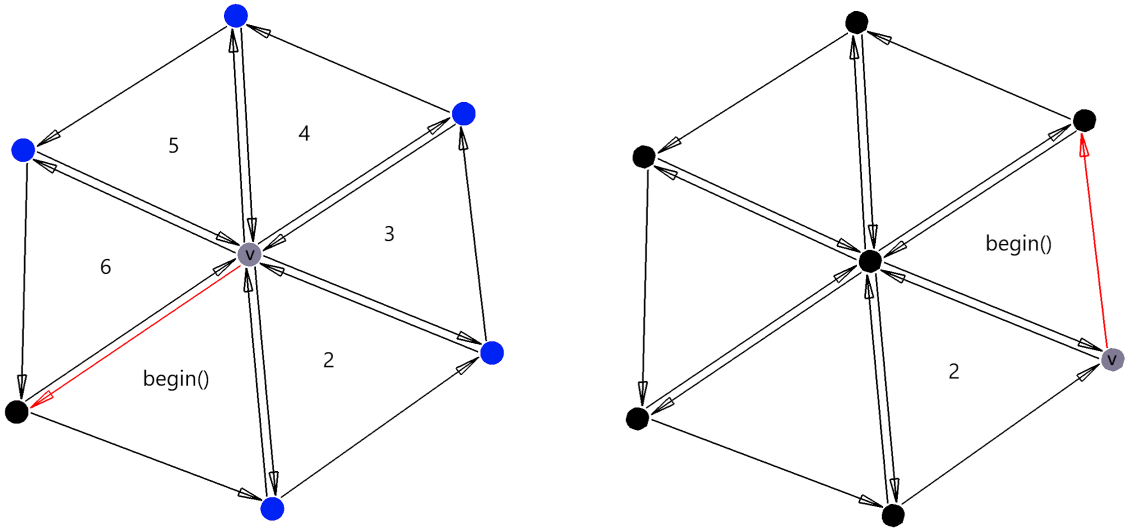


Fig. 6

5.3 Segmentation and Segmentation Iterators

Segmentation over a polygon mesh is also organized as a Halfedge structure. The following elements from the *halfedge_element.h* header file are used for segmentation representation. Each segmentation element can be either active or inactive.

A **subRegion** (MbSegmentationSubRegion) is a connected group of facets.

A **region** comprises one or more subregions, usually one. There is no separate class describing a region.

A **segmentation edge** is a sequence of mesh edges, where the subsequent edge starts from the vertex where the previous edge ends. It can be closed. Self-intersections are not allowed. Similar to a mesh edge, a segmentation edge contains links to its next and previous edges, its twin edge, and its associated vertex. These links are used to construct loops. If a segmentation edge belongs to an open boundary, there is no reference to the twin. Mesh edges that are part of a segmentation edge and its twin are also considered twins (see Fig. 1):

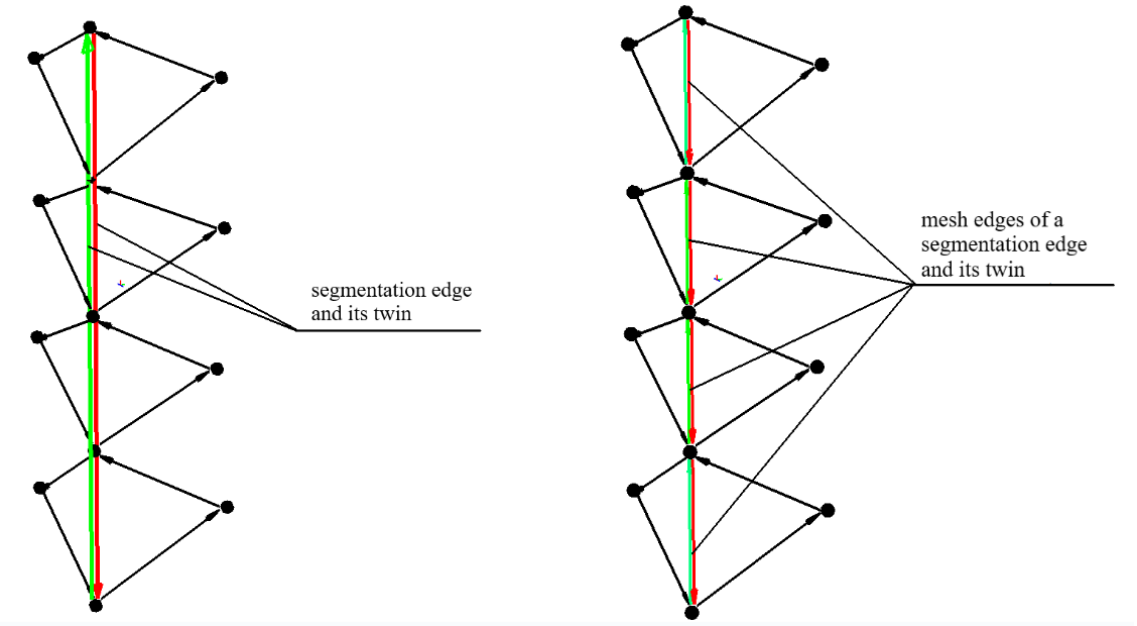


Fig. 1

A segmentation vertex corresponds to a mesh vertex where several segmentation edges are connected.

A **subRegion cycle** (MbSegmentationLoop) consists of a sequence of segmentation edges. A subregion can have any number of cycles, or none, if a subregion is a closed facet group. The subregion cycle has the same meaning as an edge cycle of a face in the boundary representation, with some differences. Usually, it is impossible to separate cycles into outer and inner ones on a mesh, so all cycles are treated equally. A cycle can also have “internal” edges.

An example of dividing the mesh into four regions, each consisting of a subregion, is shown in Fig. 2:



Fig. 2

An example for the same mesh, but now there are two regions in the segmentation, the first of which has one subregion and the second has three, is shown in Fig. 3:

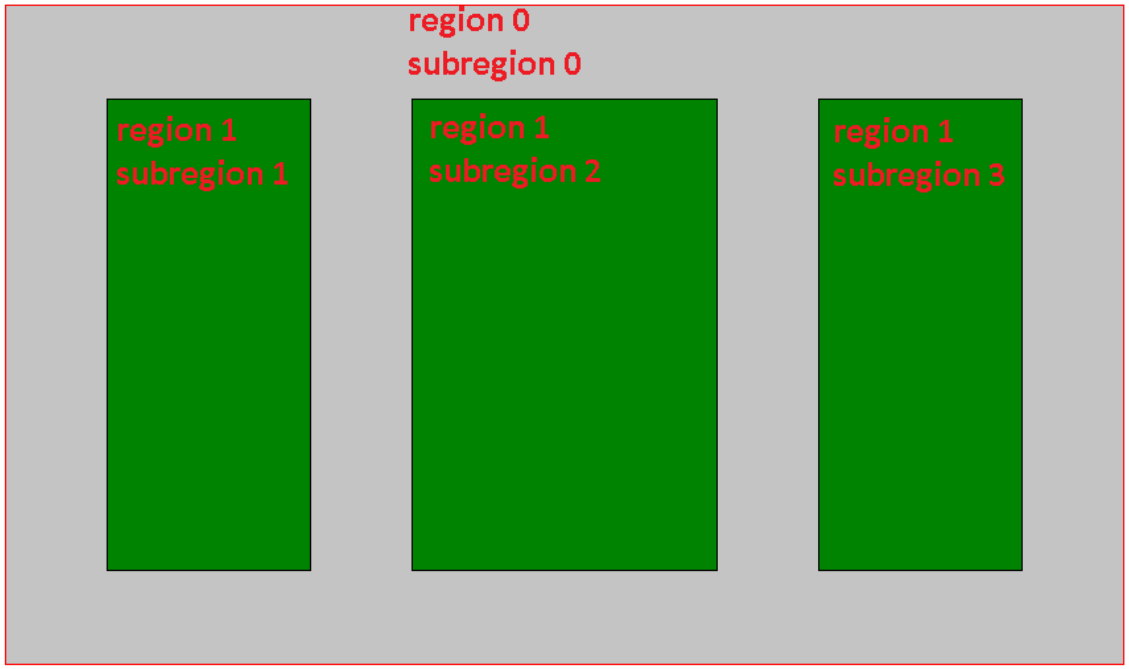


Fig. 3

An example of a two-cycle subregion is shown in Fig. 4. The green cycle consists entirely of internal edges, while the red cycle contains internal edges 1, 2, 4–9.

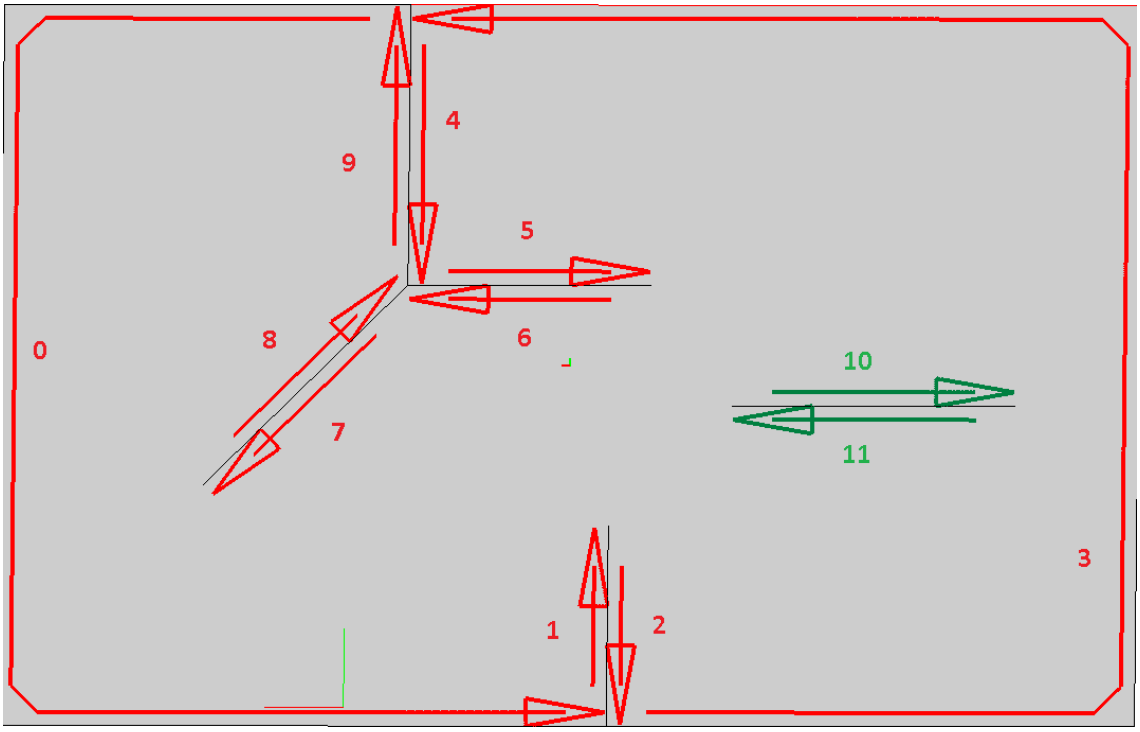


Fig. 4

The *halfedge_segm_iterators.h* header file provides segmentation iterators used to work with segmentation topology:

- EdgeLoopIterator is an iterator over edges of subregion cycle. The iterator returns a sequence of all cycle edges or only outer edges depending on the bExcludeInnerEdges flag.
- EdgeEdgesIterator is an iterator through mesh edges of a segmentation edge. The iterator returns a sequence of mesh edges in forward or reverse order depending on the bReverse flag.
- SubregionRegionIterator is an iterator through subregions of a region.
- LoopSubregionIterator is an iterator through subregion cycles.

To iterate through the segmentation vertex fan, use the EdgeFanIterator iterator, as shown in the example below:

```
MbPolymesh polymesh;

const MbPolymeshTopo & topo = *polymesh->GetTopo();

const MbHalfedge & halfedge = topo.GetMeshTopo();

for( size_t r = 0; r < topo.RegionCount(); ++r ) {

    if( !topo.IsRegionActive( r ) )
        continue;

    for( SubregionRegionIterator itSub( topo, r ); !itSub.Done(); ++itSub ) {
```

(continues on next page)

(continued from previous page)

```
        for( LoopSubregionIterator itLoop( topo, itSub.Subregion() ); !itLoop.Done();␣
↪++itLoop ) {

            for( EdgeLoopIterator itEdge( topo, itLoop.Loop() ); !itEdge.Done();␣
↪++itEdge ) {

                for( EdgeEdgesIterator it( topo, itEdge.Edge() ); !it.Done(); ++it ) {

                    const MbHalfedgeEdge & ed = halfedge.Edge( it.Edge() );
                    ...
                }
            }
        }
    }
}

for( EdgeFanIterator it( topo.GetSegmTopo(), iVxSegm, true ); !it.Done(); ++it ) {

    const MbHalfedgeEdge & edSegm = topo.GetSegmTopo().Edge( it.Edge() );
}
```

5.4 The Topology of the MbPolymeshTopo Polygon Object

The *MbPolymesh* polygon object inherits from *MbItem* and contains a topology implementation class MbPolymeshTopo (similarly to *MbSolid*, which contains *MbFaceShell* inside). The topology includes all methods for working with both the mesh and segmentation. More detailed descriptions of the methods of this class are provided below.

5.4.1 General Purpose Methods

The method

```
static MbPolymeshTopo * Create();
```

creates an empty instance of the class.

The method

```
static MbPolymeshTopo * Create( const MbMesh & mesh,
                                double   toleranceSew );
```

creates an instance of the class based on the MbMesh instance. The stitching tolerance *toleranceSew* is used to merge closely matching vertices into one.

The method

```
bool GetMesh( MbMesh & mesh ) const;
```

creates an MbMesh instance, where each subregion corresponds to a separate MbGrid object. The resulting object loses all topological information. The mesh and subregion indices are usually in different orders, and the method adds a mapping between their indices using the MapPolymeshToMbMesh class.

The method

```
MbPolymeshTopo * Clone() const;
```

creates a copy of the topology.

The method

```
void CalculateGabarit( const MbMatrix3D & matrInto,
                       MbCube &     bb ) const;
```

calculates the bounding box of an object.

The method

```
void Invert();
```

inverts the whole topology inside out, including the turning of cycles around facets.

Transformation methods

```
void Transform( const MbMatrix3D & matr );
void Move     ( const MbVector3D & to   );
void Rotate   ( const MbAxis3D   & axis, double angle );
```

transform all suitable components of the polygon object (3D points, normals, curvature directions, etc.).

The method

```
void DeleteInactive();
```

removes all inactive elements to reduce the size of the polygon object (edges, vertices, mesh facets, segmentation edges, segmentation vertices, segmentation regions, segmentation subregions and segmentation cycles).

The method

```
double FindNearest( const MbCartPoint3D & pnt,
                    MbCartPoint3D & pntProj,
                    size_t &        iFacet ) const;
```

finds the closest point to the given one and the index of the facet that it hit. Returns the distance from the given point to the nearest one on the polygon object.

The method

```
bool FindLineIntersection( const MbAxis3D  &     ray,
                           MbCartPoint3D & pntCross,
                           double &        tRes,
                           size_t &        iFacet ) const;
```

finds the nearest intersection with a straight line. The output parameters are the intersection point, the ray parameter and the index of the facet that this intersection occurred with.

The method

```
const MbFaceShell * GetSubstrateShell() const;
```

returns a const pointer to the boundary representation shell used for creating this polygon object, or nullptr if no such shell exists.

The method

```
void DeleteSubstrateShell();
```

deletes the corresponding Boundary Representation shell and all mapping information between regions and segmentation edges to faces and edges of the shell.

5.4.2 Methods for Working with the Mesh

The method

```
const MbHalfedge & GetMeshTopo() const;
```

returns a mesh topology as an MbHalfedge object. For example, in the module it is used to construct mesh iterators.

The method

```
void Reserve( size_t nVx,
              size_t nEd,
              size_t nF );
```

reserves memory for the mesh given size.

The method

```
size_t CloneVertex( size_t iVx );
```

clones a mesh vertex and returns the index of the cloned vertex.

Methods for adding mesh elements

```
size_t AddVertex();
size_t AddEdge();
size_t AddFacet();
```

add the active mesh element and return its index. If there are inactive elements in the mesh, one of them is switched to the active state and its index is returned.

Methods

```
void MakeVertexInactive( size_t iVx );
void MakeEdgeInactive   ( size_t iEd );
void MakeFacetInactive  ( size_t iF  );
```

set the mesh element in an inactive state. Caution! These methods may corrupt the mesh topology.

The method

```
void SewEdges( size_t iEd0,
               size_t iEd1 );
```

makes the two specified mesh edges twins.

The method

```
size_t AddFacet( const c3d::IndicesVector & vertices );
```

creates a facet by the specified vertices and returns its index. Vertices must already exist before the call. The method sets necessary links between the facet and all the corresponding edges (except for the twins).

The method

```
void SewByVertices();
```

sets links to twins for all mesh edges by the vertices where the edges begin. It is used for the final creation of the topology during manual construction, after adding all existing facets with the AddFacet() method.

The method

```
void DeleteFacets( const c3d::IndicesVector & facets );
```

deletes the specified facets and ensures the topology validity after the operation. It does not delete inactive elements. The example of the actions required to maintain the topology validity is shown in Fig. 1. In the initial state vertex A is the same for all edges of the fan. After deleting the selected facets, the vertex fan breaks, which is not allowed in the Halfedge structure. So, the vertex will be cloned, and one part of the remaining fan will correspond to vertex A, and the other part to its clone B. At this point, there will be a topological gap of the mesh, but these two vertices are geometrically identical.

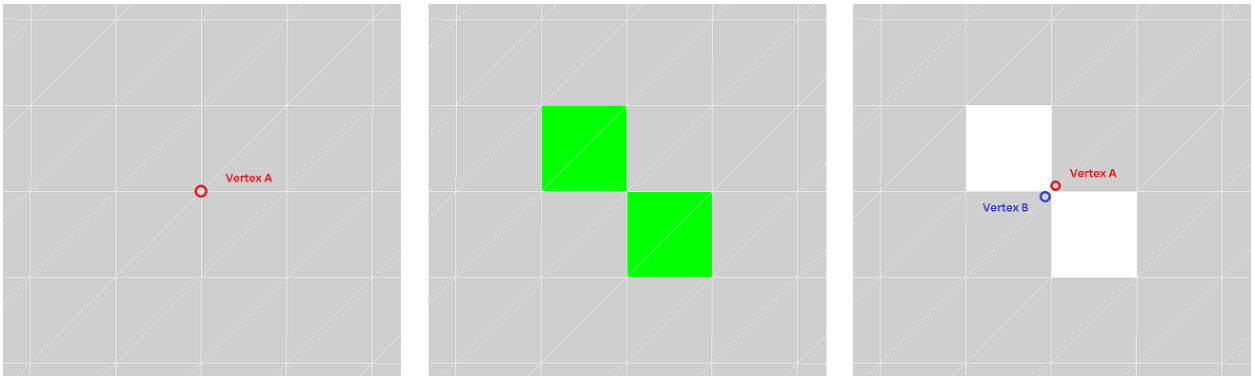


Fig. 1

The method

```
bool FlipEdge( size_t iEd );
```

flips an edge in the triangulation. You cannot flip open edges or edges that belong to segmentation edges. The example of working of the method is shown in Fig. 2:

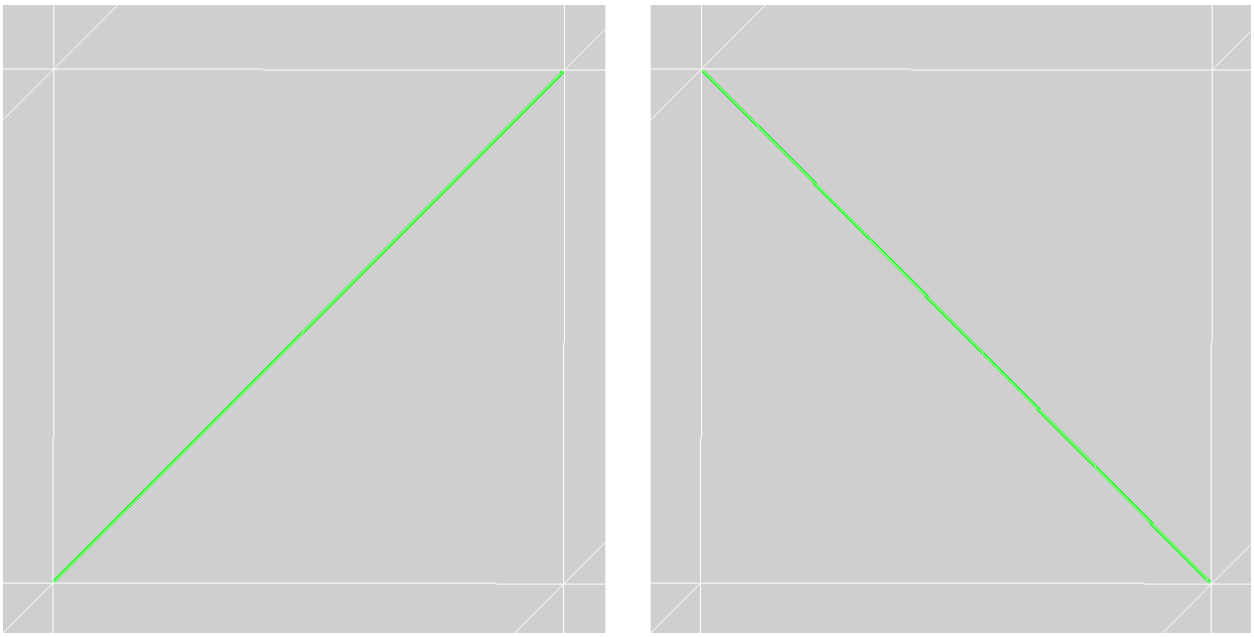


Fig. 2

The method

```
size_t SplitEdge( size_t iEd,
                  double prm );
```

splits an edge in the specified parameter that must be in the range from 0 to 1. It returns the index of the bivalent vertex created at this location. The number of edges for adjacent facets will increase by one. Fig. 3 shows the result after triangulating two quadrilateral adjacent facets:

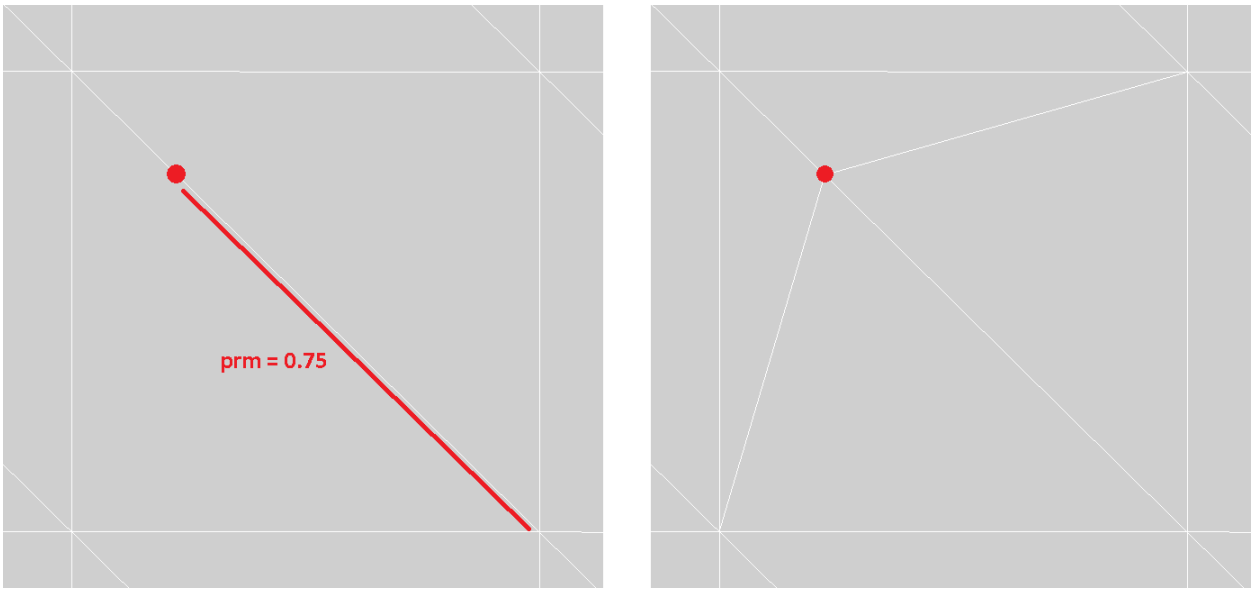


Fig. 3

The method

```
void SplitFacet( size_t              eVx0,
                 size_t              eVx1,
                 c3d::IndicesPair & eNew );
```

splits the facet in two by inserting a pair of edges between vertices *v0* and *v1* that the specified edges *eVx0* and *eVx1* begin. It returns the indices of these inserted edges. The given edges must belong to the same facet, which will be split. The example is shown in Fig. 4:

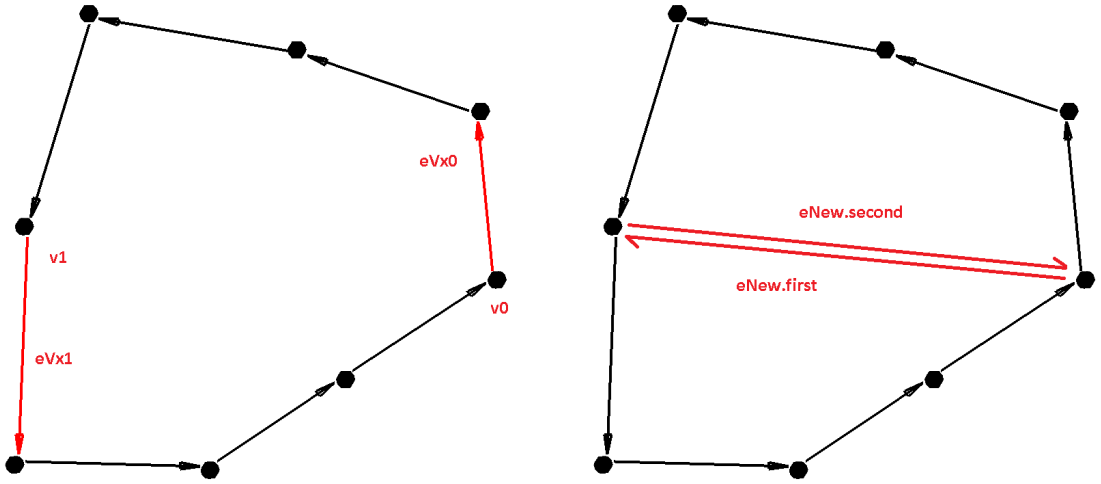


Fig. 4

The method

```
bool JoinEdgeFacets( size_t iEd );
```

combines two facets adjacent to an edge into one. This operation is the inverse of the SplitFacet() method. You cannot merge facets if the result is a facet that touches itself by an edge or vertex.

The method

```
bool RemoveVertexValence2( size_t iVx );
```

deletes a bivalent vertex of the mesh. This operation is the inverse of the SplitEdge() method.

The method

```
size_t InsertVertexTriangle( size_t iTr,
                             double u,
                             double v );
```

inserts a new vertex into the triangle and cuts it into three parts. It returns the index of the inserted vertex. Parameters *u* and *v* are barycentric coordinates that define a new vertex inside the triangle. The attribute values for the new vertex (3D point, 2D parameter, normal, curvature, etc.) are computed as a linear combination of the values at the triangle's vertices according to these barycentric coordinates.

The method

```
size_t InsertVertexFacet( const c3d::IndicesVector     & edgesSplit,
                          const std::array<size_t, 3> & edgesInterpolate,
                                double                  u,
                                double                  v );
```

inserts a new vertex into the facet and splits it into several new ones by connecting the edges of the inserted vertex and the vertices where the edges from the *edgesSplit* array begin. Edges of this array must belong to one facet. The *edgesInterpolate* array defines a “virtual” triangle of facet vertices, setting the barycentric coordinates *u* and *v*.

The example of using the method is shown in Fig. 5. Fig. a) shows the black facet to split by the blue vertex and the purple edges. Fig. b) shows green facet edges, whose indices should be specified in the *edgesSplit* array (they should be arranged in order within the facet). Fig. c) shows a “virtual” triangle in green setting barycentric coordinates, and in red — the indices of edges to be placed in the *edgesInterpolate* array. The calculation of attribute values for a new vertex is similar to the InsertVertexTriangle() method.

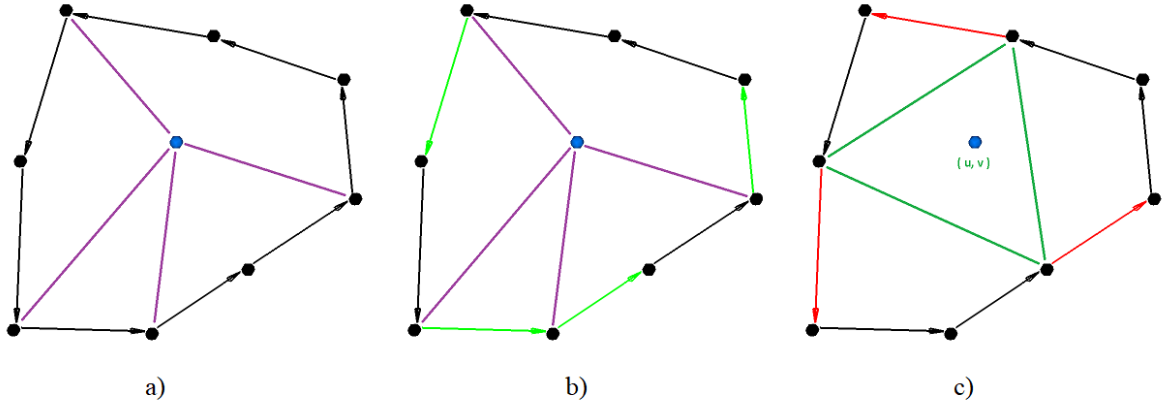


Fig. 5

5.4.3 Methods for Working with Segmentation

The method

```
bool IsSegmented() const;
```

checks a segmentation initiation. It is recommended to always initialize segmentation.

The method

```
void Initialize( const MbHalfedgeBoundaryCriterionBase & crit );
```

initializes segmentation according to the specified boundary criterion. The criterion should be implemented by a user as an inheritor of the MbHalfedgeBoundaryCriterionBase base class, which defines whether a mesh edge is a dividing line or not. For example, you can pre-calculate dihedral angles for mesh edges and create a criterion that checks if the angle exceeds a specified threshold or not. This produces a mesh segmentation by sharp boundaries.

The method

```
void Initialize();
```

initializes the segmentation with a single region. Each connected part of the mesh is assigned to its own subregion. Segmentation edges are formed from open mesh edges if any exist. It is recommended to always initialize segmentation in this way when manually creating a polygon object.

The method

```
const MbHalfedge & GetSegmTopo() const;
```

provides access to a constant segmentation topology as the Halfedge. It is used for constructing some segmentation iterators.

The method

```
size_t LoopsCount() const;
```

returns the number of cycles in the segmentation (including inactive ones).

The method

```
const MbSegmentationLoop & Loop( size_t iLoop ) const;
```

returns a segmentation cycle by its index.

The method

```
size_t RegionCount() const;
```

returns the number of regions in the segmentation (including inactive ones).

The method

```
bool IsRegionActive( size_t iRegion ) const;
```

checks if the region is active. The same operation can be done using the IsFacetActive() method with the same index for the segmentation topology returned by the GetSegmTopo() method.

The method

```
size_t SubRegionsCount() const;
```

returns the number of subregions in the segmentation (including inactive ones).

The method

```
size_t SubRegionsCount( size_t iRegion ) const;
```

returns the number of subregions included in the specified region.

The method

```
const MbSegmentationSubRegion & SubRegion( size_t iSub ) const;
```

returns a segmentation sub-region by its index.

The method

```
size_t EdgeSegmCount() const;
```

returns the number of edges in the segmentation (including inactive ones).

The method

```
const MbHalfedgeEdge & EdgeSegm( size_t iEdSegm ) const;
```

returns a segmentation edge by its index.

The method

```
bool IsInnerEdge( size_t iEdSegm ) const;
```

checks if the segmentation edge is internal.

The method

```
size_t GetSegmEdgeId( size_t iEdTopo ) const;
```

returns the index of the segmentation edge including the specified mesh edge.

The method

```
void GetFacetRegionId( size_t   iFacet,
                       size_t & iRegion,
                       size_t & iSubRegion ) const;
```

returns the indices of the region and subregion including the specified mesh facet.

The method

```
size_t VertexSegmCount() const;
```

returns the number of vertices in the segmentation (including inactive ones).

The method

```
const MbHalfedgeVertex & VertexSegm( size_t iVxSegm ) const;
```

returns a segmentation vertex by its index.

The method

```
void GetRegion( size_t              r,
                c3d::IndicesVector & facets ) const;
```

returns indices of all mesh facets included in the specified region.

The method

```
void GetSubRegion( size_t              iSub,
                   c3d::IndicesVector & facets,
                   bool                bAdd = false ) const;
```

returns indices of all mesh facets included in the specified sub-region.

The method

```
void DeleteRegion( size_t iRegion );
```

deletes the specified region. The topology validity of the mesh and segmentation is maintained similarly to deleting facets.

The method

```
void SplitByEdgesTopo( const c3d::IndicesVector & splitSet,
                       bool                       bKeepRegions );
```

splits the existing segmentation by a set of mesh edges. This operation may create new connected facet groups. A new subregion is created for each of them. If the bKeepRegions flag is true, the new subregions remain within their original region; otherwise, a new region is created for each subregion. This operation can also create internal segmentation edges.

The method

```
void MergeRegions( const c3d::IndicesVector & regions,
                   bool                       bRemoveValence2 );
```

merges the specified regions into one. Its index is equal to the first index of the *regions* array. In this case, sequences of segmentation edges separated by bivalent segmentation vertices can be formed. By the bRemoveValence2 flag these sequences are combined into segmentation edges.

The method

```
void SplitByFacetsTopo( const c3d::IndicesVector & splitSet,
                        bool                       bMakeOneRegion,
                        bool                       bKeepRegions );
```

splits the existing segmentation by a set of mesh facets. This set corresponds to one or more connected facet groups from old regions, each of which is assigned a new sub-region. Adding the groups to new regions and subregions depends on the two control flags bMakeOneRegion and bKeepRegions. The example is shown in Fig. 6 — the segmentation consists of four simple regions to be split by facets highlighted in bright green:

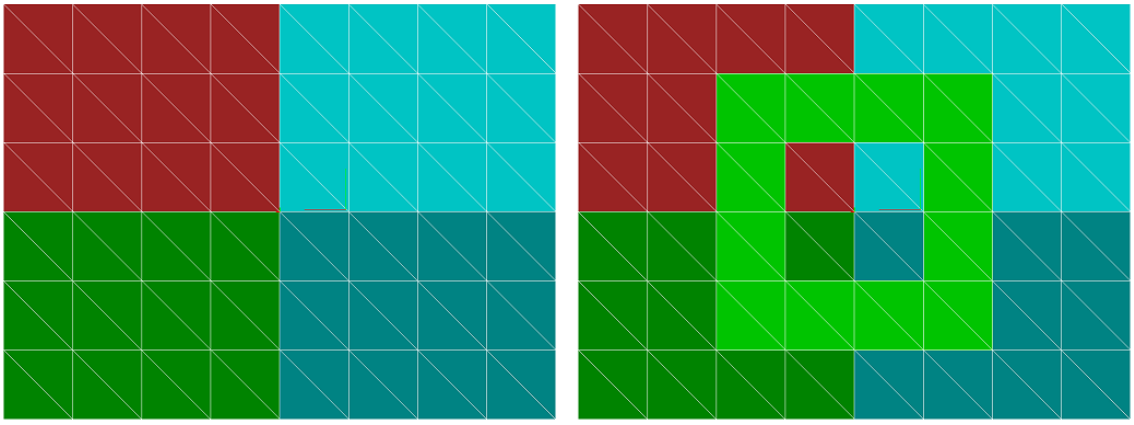


Fig. 6

One color shows facets that belong to one region.

1. bMakeOneRegion = true, bKeepRegions = true.

The selected facets formed one new region (highlighted in yellow). The old regions remained, but were divided into two subregions each. In total, there are now 5 regions in the segmentation (see Fig. 7).

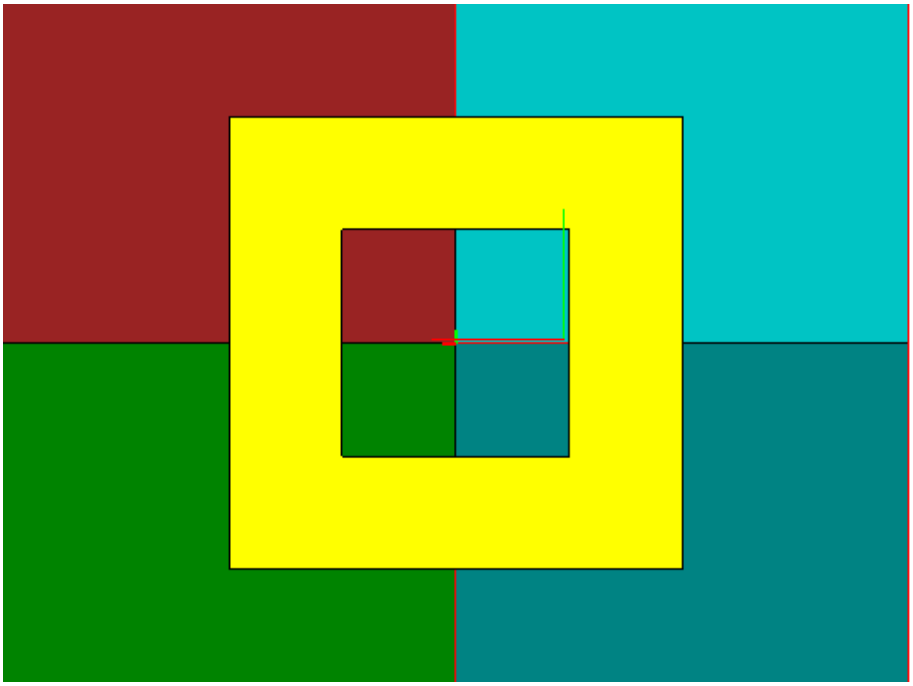


Fig. 7

2. bMakeOneRegion = true, bKeepRegions = false.

The selected facets formed one new region (yellow). Each of the old regions splits into two unrelated parts, each forming a new separate region. There are 9 regions in the segmentation now (see Fig. 8):

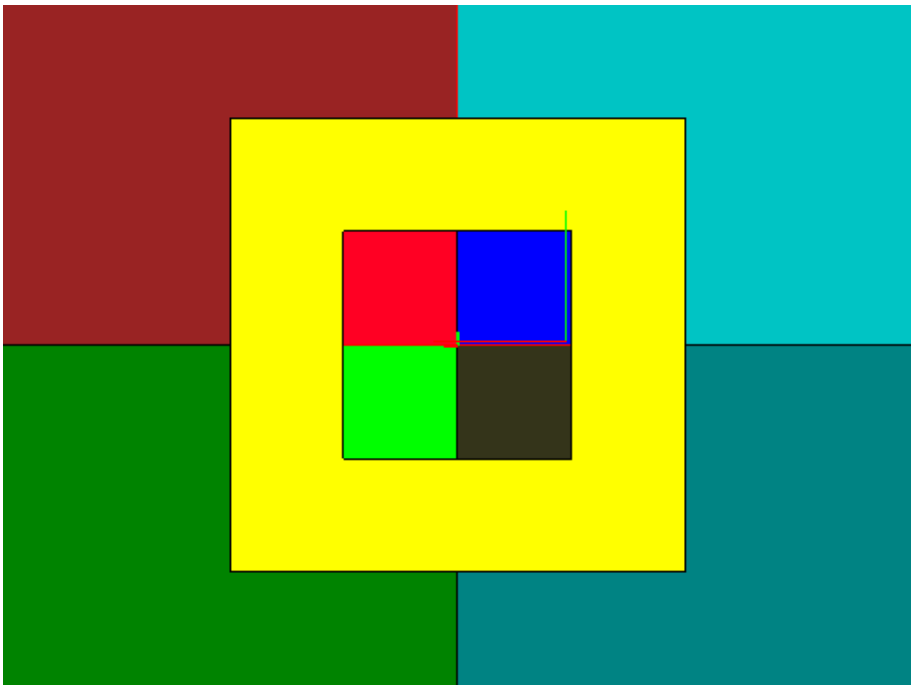


Fig. 8

3. bMakeOneRegion = false, bKeepRegions = true.

The selected facets split each of the old regions into 3 parts. There are still 4 regions in the segmentation, but each of them now consists of three subregions (see Fig. 9).

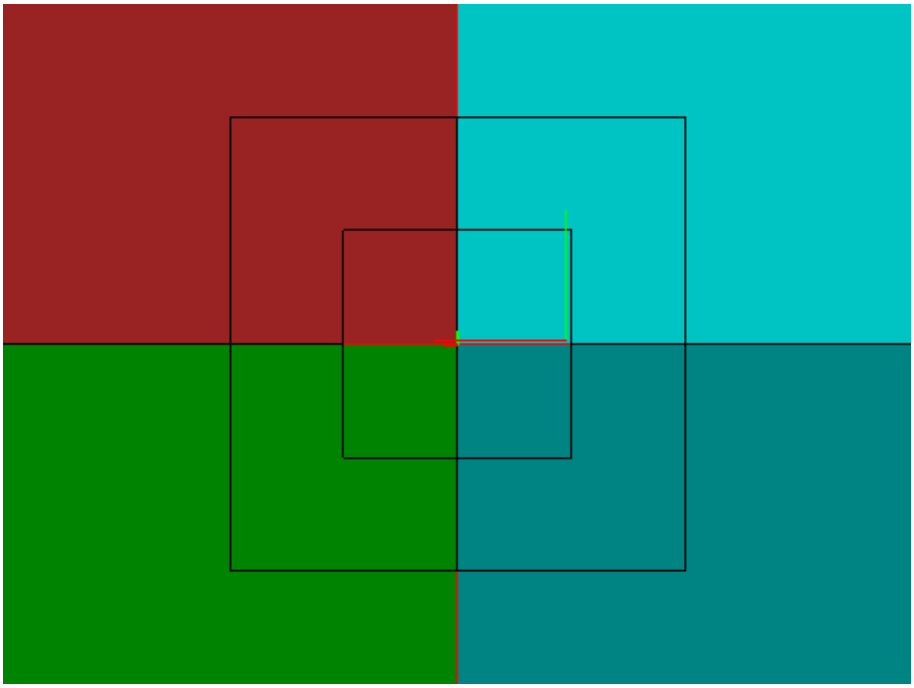


Fig. 9

4. bMakeOneRegion = false, bKeepRegions = false.

The selected facets split each of the old regions into 3 parts. Each such part is assigned a new region. In total, there are now 12 regions in the segmentation, each consisting of one subregion (see Fig. 10):

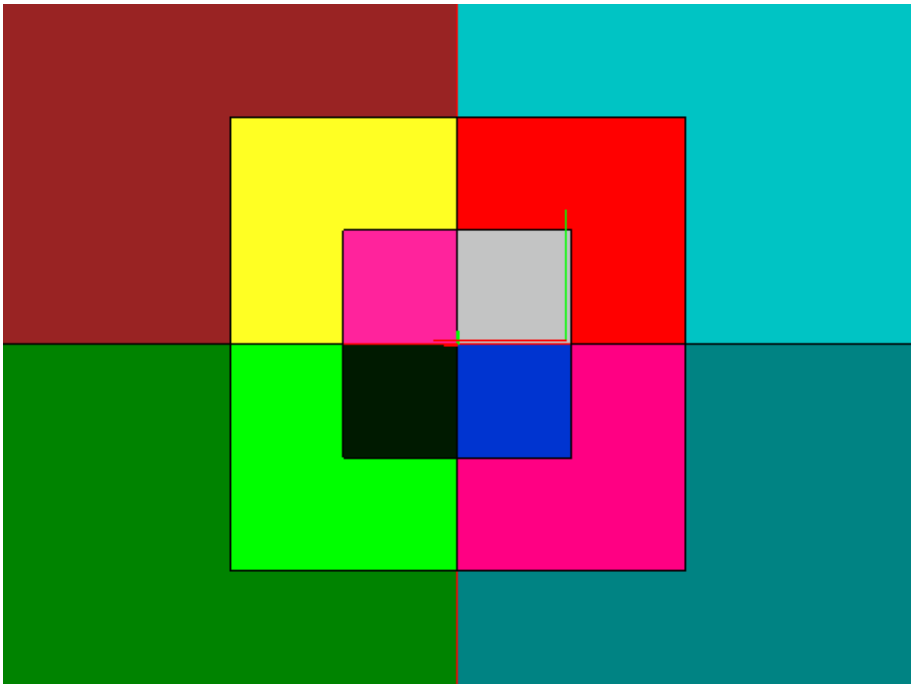


Fig. 10

The method

```
void SplitRegionBySubregions( size_t iReg );
```

separates subregions of the given region as new independent regions.

The method

```
c3d::IndicesPair SplitSegmentationEdge( size_t iEdSegm,
                                        size_t iEdTopo,
                                        size_t iVxSegmSplit = SYS_MAX_T );
```

cuts the segmentation edge with index *iEdSegm* (and its twin) at the origin vertex of the *iEdTopo* edge. The *iEdTopo* edge must belong to the segmentation edge and not be the first mesh edge for the segmentation edge. If the *iVxSegmSplit* index is defined, it will be assigned to the created bivalent segmentation vertex. It returns indices of new segmentation edges formed from the old segmentation edge and its twin (if it exists). The method does not change the mesh topology. New segmentation edges inherit attributes of old edges.

The method

```
bool ExtractSubRegionAsRegion( size_t iSub );
```

creates a new region containing the given subregion. If the old region consisted only of the specified subregion, no actions will be performed.

The method

```
void RemoveSegmVertexValence2( size_t iVx );
```

deletes a segmentation bivalent vertex. The inverse method to SplitSegmentationEdge().

The method

```
size_t GetTopoVertex( size_t iVxSegm ) const;
```

returns the index of the mesh vertex where the given segmentation vertex is located.

The method

```
size_t GetSegmEdgeRegionId( size_t iEdSegm ) const;
```

returns the index of the region including the segmentation edge.

The method

```
size_t GetSegmEdgeSubRegionId( size_t iEdSegm ) const;
```

returns the index of the subregion including the segmentation edge.

The method

```
size_t GetLoopRegionId( size_t iLoop ) const;
```

returns the index of the region including the cycle.

The method

```
size_t GetLoopSubRegionId( size_t iLoop ) const;
```

returns the index of the sub-region including the cycle.

The method

```
size_t GetNeighborRegionId( size_t iEdSegm ) const;
```

returns the index of the region opposite to the given segmentation edge.

The method

```
size_t GetNeighborSubRegionId( size_t iEdSegm ) const;
```

returns the index of the subregion opposite to the given segmentation edge.

5.4.4 Methods for Working With Mesh and Segmentation Attributes

The method

```
size_t MeshHalfedgeAttributeCount() const;
```

returns the total number of mesh attributes.

The method

```
size_t SegmHalfedgeAttributeCount() const;
```

returns the total number of segmentation attributes.

The method

```
template <class T>
T * AddMeshHalfedgeAttribute();
```

creates a mesh attribute of the specified type and returns a pointer to it.

The method

```
template <class T>
T * AddSegmHalfedgeAttribute();
```

creates a segmentation attribute of the specified type and returns a pointer to it.

The method

```
template <class T>
T * GetMeshHalfedgeAttribute();
template <class T>
const T * GetMeshHalfedgeAttribute() const;
```

returns a pointer to the first mesh attribute of the specified type. If multiple attributes exist, only the first one is returned.

The methods

```
template <class T>
T * GetSegmHalfedgeAttribute();

template <class T>
const T * GetSegmHalfedgeAttribute() const;
```

returns a pointer to the first segmentation attribute of the specified type. If multiple attributes exist, only the first one is returned.

The method

```
template <class T>
void DeleteMeshHalfedgeAttribute();
```

deletes a mesh attribute of the specified type. If the object has more than one attribute with this type, the method deletes the first one.

The method

```
template <class T>
void DeleteSegmHalfedgeAttribute();
```

deletes a segmentation attribute of the specified type. If the object has more than one attribute with this type, the method deletes the first one.

The methods

```
MbHalfedgeAttrBase * GetMeshHalfedgeAttributeByIndex( size_t ind );

const MbHalfedgeAttrBase * GetMeshHalfedgeAttributeByIndex( size_t ind ) const;
```

returns a pointer to a mesh attribute with the specified index in the attribute array.

The methods

```
MbHalfedgeAttrBase * GetSegmHalfedgeAttributeByIndex( size_t ind );

const MbHalfedgeAttrBase * GetSegmHalfedgeAttributeByIndex( size_t ind ) const;
```

returns a pointer to a segmentation attribute with the specified index in the attribute array.

The method

```
void DeleteMeshHalfedgeAttribute( MbHalfedgeAttrBase * pAttr );
```

deletes the given mesh attribute.

The method

```
void DeleteSegmHalfedgeAttribute( MbHalfedgeAttrBase * pAttr );
```

deletes the given segmentation attribute.

The method

```
MbAttributeContainer * GetSegmRegionModelAttributes( size_t iRegion,
                                                     bool   bCreate );
```

returns the attribute container of a geometric core object for the region. If it is missing, by the bCreate flag it is created, otherwise a nullptr is returned.

The method

```
const MbAttributeContainer * GetSegmRegionModelAttributes( size_t iRegion ) const;
```

returns the attribute container of a geometric core object for the region, or nullptr if it does not exist.

The method

```
MbAttributeContainer * GetSegmEdgeModelAttributes( size_t iEdSegm,
                                                   bool   bCreate );
```

returns the attribute container of a geometric core object for a segmentation edge. If it is missing, the bCreate flag is used to create it, otherwise a nullptr is returned.

The method

```
const MbAttributeContainer * GetSegmEdgeModelAttributes( size_t iEdSegm ) const;
```

returns the attribute container of a geometric core object for a segmentation edge, or nullptr if it does not exist.

The method

```
MbAttributeContainer * GetSegmVertexModelAttributes( size_t iVxSegm,
                                                     bool   bCreate );
```

returns the attribute container of a geometric core object for a segmentation vertex. If it is missing, the bCreate flag is used to create it, otherwise a null pointer is returned.

The method

```
const MbAttributeContainer * GetSegmVertexModelAttributes( size_t iVxSegm ) const;
```

returns the attribute container of a geometric core object for a segmentation vertex, or nullptr if it does not exist.

5.5 Mesh and Segmentation Attributes

As mentioned above, mesh and segmentation attributes are used to associate various types of values with topological elements of a polygon object – vertices, edges and facets of a mesh, regions, edges and vertices of segmentation.

Some attribute types are unique – in all algorithms a polygon object has only one attribute of this type, or the attribute does not exist. These attributes include:

- MbHalfedgeAttrPoint3D – 3D points for vertices;
- MbHalfedgeAttrParameters – 2D parameters for vertices;
- MbHalfedgeAttrParamByEdge – edge-dependent 2D parameters;
- MbHalfedgeAttrNormalsVertex – normals;
- MbHalfedgeAttrCurvaturesVertex – curvatures;
- MbHalfedgeAttrDensityVertex – target facet size for vertices;
- MbHalfedgeAttrDihedralAngles – dihedral angles for edges;
- MbHalfedgeAttrStrains – stresses in triangles;
- MbHalfedgeAttrSurfaces – surfaces for regions.

The remaining attributes are usually auxiliary and can be present in any number depending on the specific task. These include:

- MbHalfedgeAttrBoolFacet;
- MbHalfedgeAttrBoolEdge;
- MbHalfedgeAttrBoolVertex;
- MbHalfedgeAttrUintFacet;
- MbHalfedgeAttrUintEdge;
- MbHalfedgeAttrUintVertex;
- MbHalfedgeAttrIntFacet;
- MbHalfedgeAttrIntEdge;
- MbHalfedgeAttrIntVertex;
- MbHalfedgeAttrSizetFacet;
- MbHalfedgeAttrSizetEdge;
- MbHalfedgeAttrSizetVertex;
- MbHalfedgeAttrDoubleFacet;
- MbHalfedgeAttrDoubleEdge;
- MbHalfedgeAttrDoubleVertex;

- MbHalfedgeAttrFloatFacet;
- MbHalfedgeAttrFloatEdge;
- MbHalfedgeAttrFloatVertex;
- MbHalfedgeAttrDirectionEdge3D;
- MbHalfedgeAttrNormalsFacet;
- MbHalfedgeAttrDirectionVertex3D.

There is no possibility for a user to create their own attribute type.

5.6 Edge-Dependent Attributes

Suppose we want to obtain a polygonal object by linearization of the parametric shell and store in the vertices 2D parameters on the corresponding faces. Each face has a corresponding region in the segmentation. However, the numeration of vertices in a polygon object is continuous and uniform for the whole mesh. For the mesh vertices lying on the edges and in the segmentation vertices, we get an ambiguous situation. If a vertex belongs to one face, this vertex has a 2D parameter connected with this face, but at the same time this vertex also belongs to a different face with a different 2D parameter connected with that face. Thus, it is impossible to describe 2D parameters by a simple attribute that assigns only one parameter to each vertex. The same issue occurs with values such as normals and curvatures.

To solve this issue, the module provides edge-dependent attributes. They are inherited from the MbHalfedgeAttrEdgeDependentVertexBase base class. There are three such attributes:

- MbHalfedgeAttrParamByEdge;
- MbHalfedgeAttrNormalsVertex;
- MbHalfedgeAttrCurvaturesVertex.

Let's consider their structure based on the normals example. All normals are stored in a separate array, and their indices and number have no direct connection with the mesh vertices. Each edge of the mesh is assigned an index in the normals array. For an internal vertex, all edges of its fan point to the same normal. If the vertex belongs to a segmentation edge or a segmentation vertex, then different edges of the fan point to different normals.

This is the detailed example with a simple mesh with four triangle facets in Fig. 1. The normal in the vertex is calculated by averaging the normals of the facets included in the vertex fan. There are two cases:

- the first case when the whole mesh forms one smooth object (see Fig. a)),
- the second case when the mesh is segmented into two flat regions (see Fig. c)).

The direction of the normals is shown by the red lines. The vertex fan has three edges starting from this vertex. On the right image they all refer to the same normal. On the left image two red edges (see Fig. b)) refer to normal N1, and one green edge refers to normal N2.

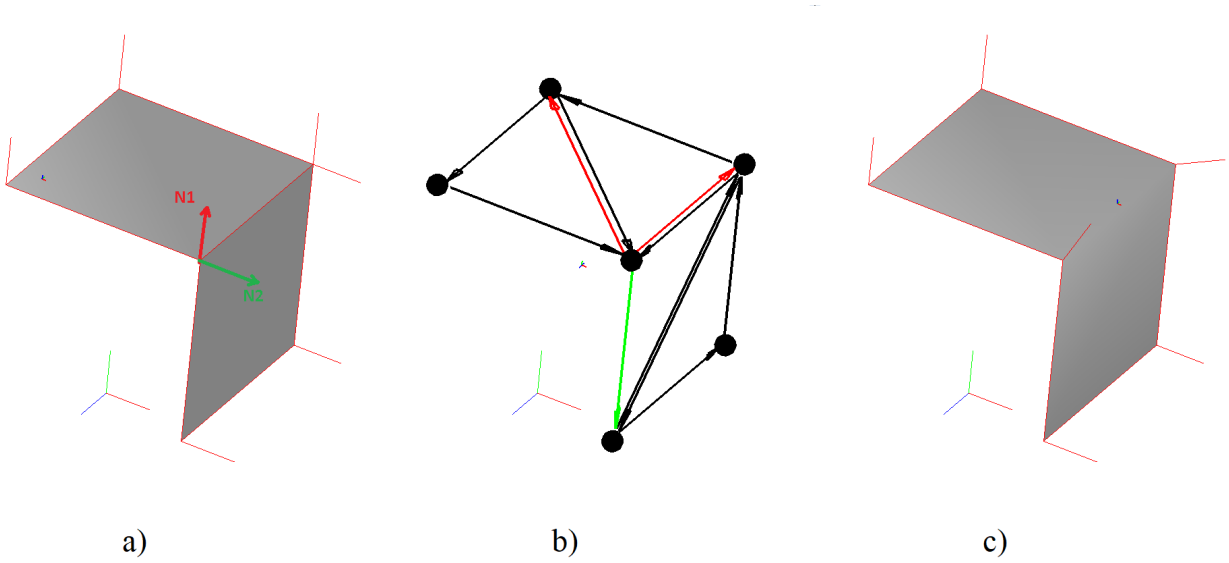


Fig. 1

So, if the normals exist, the visualization of segmented and unsegmented meshes is significantly different, as shown in Fig. 2:

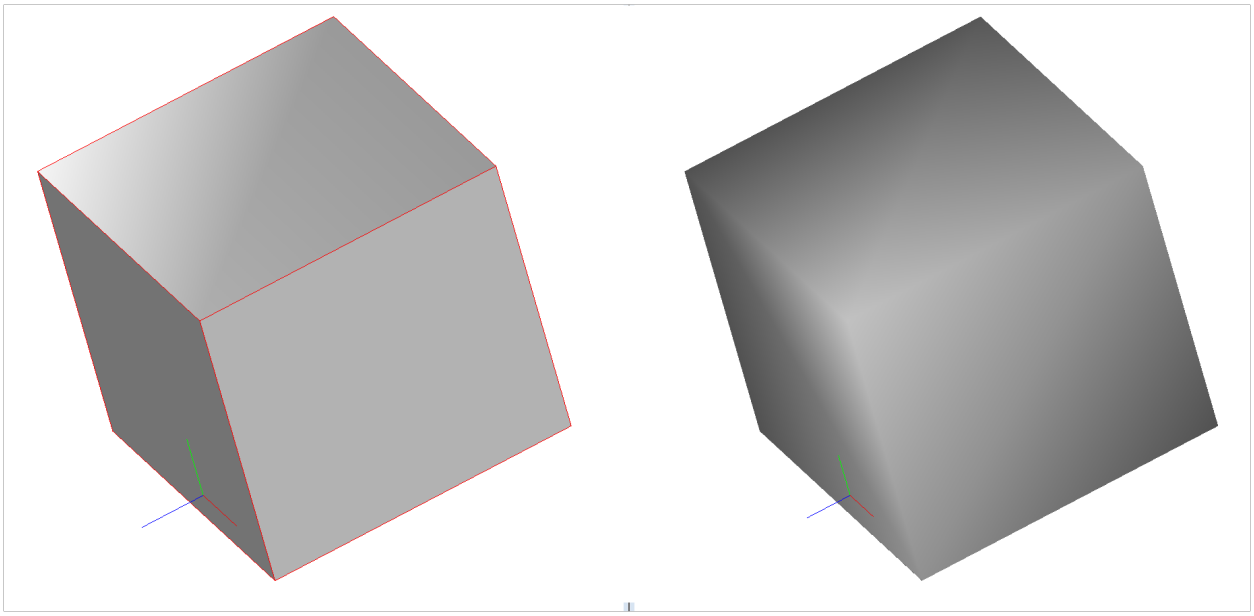


Fig. 2

Attention! Do not use the method

```
const MbVector3D & GetNormal( size_t iNm ) const;
```

with vertex indices as an input parameter or interpret it as a way to have one normal per vertex.

You must use the method

```
const MbVector3D & GetNormalEdge( size_t iEd ) const;
```

instead, receiving normals for each edge from the vertex fan separately.

There is no way for a user to explicitly create edge-dependent attribute types. They can be created only through functions providing these values calculation. These functions set the attribute structure corresponding to the current segmentation. In case of any changes in the segmentation topology (merging regions, splitting by mesh edges or facets and so on), the structure of the existing edge-dependent attributes is also changed according to the new segmentation.

CHAPTER 6

C3D Collision Detection

All assemblies made of parts in CAD models should be free of collisions, a form of unwanted design error.

With our Collision Detector, your CAD and BIM applications check assemblies for collisions, even at early stages, to ensure designs are free of errors. Collisions are identified as areas of contact and volumes of overlap. Early detection of collisions reduces costs during later design phases and especially when it comes to constructing physical products.

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.

For license activation of the module, call the EnableMathModules() function with the appropriate parameters.

6.1 Detecting the Intersection of Two Bodies

The method

```
MbResultType InterferenceSolids(MbSolid &                    solid1,
                                MbSolid &                    solid2,
                                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].

6.2 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. 1.2.1 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. 1.2.2 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:

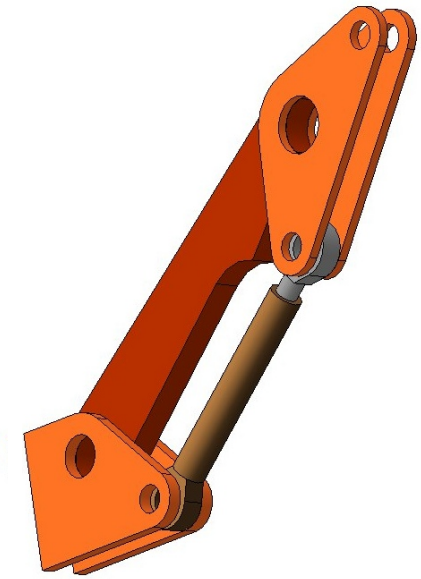


Fig. 1

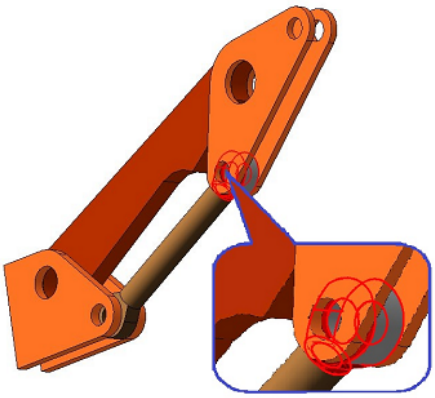


Fig. 2

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 MbCollisionDetectionUtilit 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* – 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** – a collision data query structure.

The method returns the collision search result code, which may contain the following codes:

- CDM_RESULT_None – result of undefined type;
- CDM_RESULT_NoCollisionDetected – no collision was detected: there are no intersections or the search was aborted;
- CDM_RESULT_CollisionDetected – volume intersection (collision) detected;
- CDM_RESULT_Error – unknown error or internal system error.

The **CheckCollisions**() method detects collisions between the solids or meshes added to the control set by calling the **AddItem**() method. The method returns CDM_RESULT_CollisionDetected, if a collision is detected in this body set. If no collisions were detected, then the method returns CDM_RESULT_NoCollisionDetected. 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.

6.3 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 *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* – 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.

6.4 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(CDM_RESULT_NoCollisionDetected) {}
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 started for all solids of␣
↪the set
        {
            q->result = CDM_RESULT_NoCollisionDetected;
            return CBACK_VOID_RESULT;
        }
        case CDET_INTERSECTED: // First intersection is founded.
        {
            q->result = CDM_RESULT_CollisionDetected;
            return CBACK_SUFFICIENT;
        }
        case CDET_FINISHED: // A pair of solids is finished.
        return (q->result == CDM_RESULT_CollisionDetected) ?␣
↪CBACK_BREAK:CBACK_VOID_RESULT;
        default:
        return CBACK_VOID;
```

(continues on next page)

(continued from previous page)

```
        }
    }
};
```

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
{
```

(continues on next page)

(continued from previous page)

```
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.

6.5 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.

CHAPTER 7

C3D Solver

Constraints represent all relationships among geometric objects in 2D/3D applications. These include dimensions that define angles and distances and logical constraints that define concepts like a coincidence, parallelism, perpendicularity, tangency, etc.

Using C3D Solver, developers incorporate dimensions and logical constraints in their applications to create connections between geometric objects in 2D and 3D models. C3D Solver maintains constraints as users make changes to the geometry. For example, when users modify dimensions, C3D Solver instantly recalculates all depending geometric objects and maintains existing relationships.

Our C3D Solver reduces 2D/3D modeler development costs significantly, as the constraint engine is ready-made for you. You can plug the C3D Solver in at any stage of your product development cycle.

For license activation of the module, call the EnableMathModules() function with the appropriate parameters.

7.1 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*.

7.1.1 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.

7.1.2 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. 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.

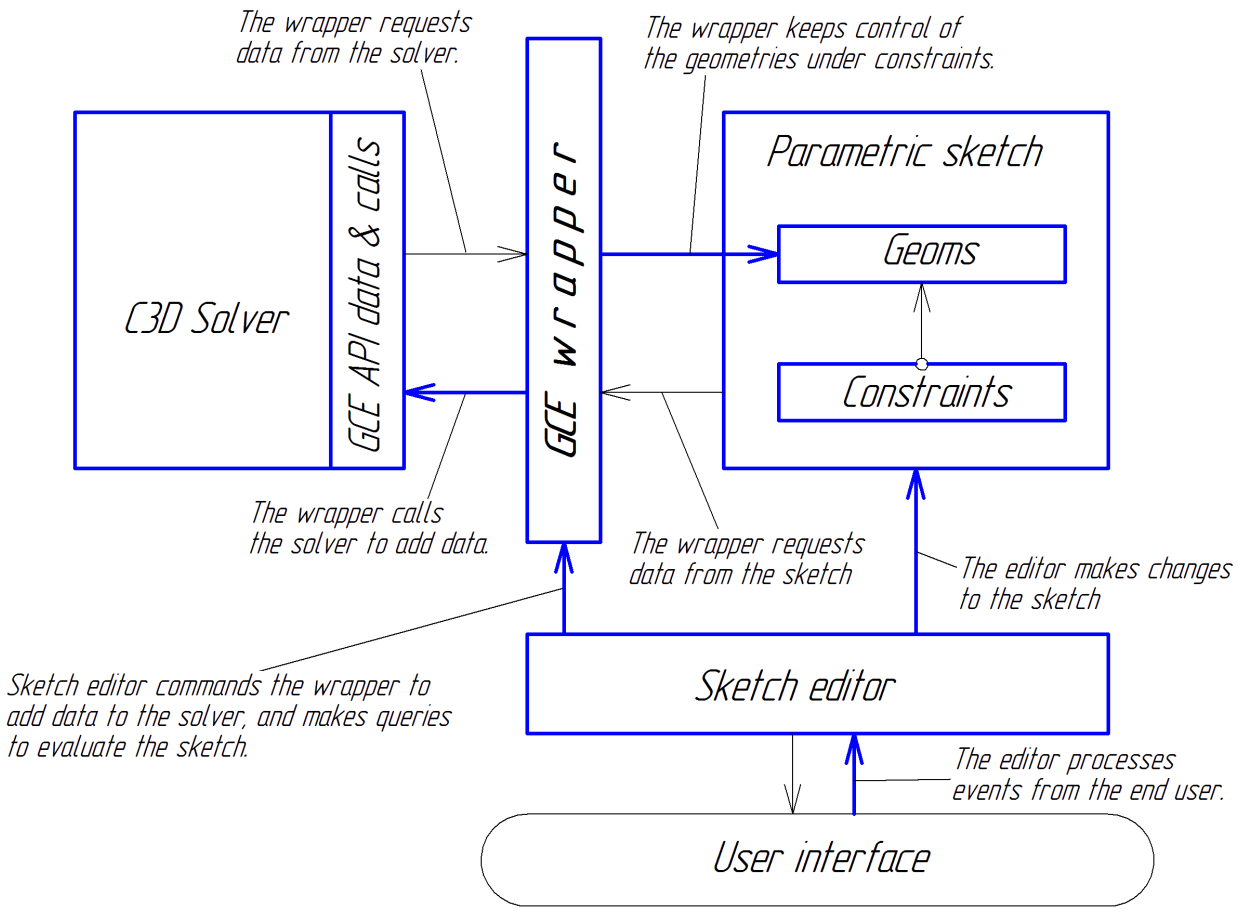


Fig. 1

The flow chart on Fig. 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.

7.1.3 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 direction 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.

7.1.4 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 *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 1.

Table 1 - Constraint types

Logical constraints:	**Dimensional constraints**:	**Special constraints**:
fixing	segment length	driving parameter
horizontality	radius	fix spline derivative
verticality	distance	point on curve with a position based on curve parameter
segment length	distance with a direction	Linear equation of the following form: $a1*v1+a2*v2+ ... + an * vn + c = 0$
coincidence	angle;	
point on curve		
alignment of points		
collinearity		
equal length		
equality of radii		
parallel		
perpendicularity		
tangency		
middle point		
mirror symmetry		
bisector		

Each of these constraints will be further described below. Logical constraints are described in Chapter *TWO-DIMENSIONAL LOGICAL CONSTRAINTS*. Dimensional constraints and the issues of adding constraints to the sketch are described in Chapter *TWO-DIMENSIONAL DIMENSIONS*.

7.1.5 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 1).

Table 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 2.

Table 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 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 direction vector.

Table 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

7.1.6 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 *Supported Geometry Types*, *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.

7.1.7 Representation of Geometric Objects

Geometric constraint solver works with a certain geometric object representation form shown in Fig. 1. All objects are expressed using point, vector and number coordinates (scalars).

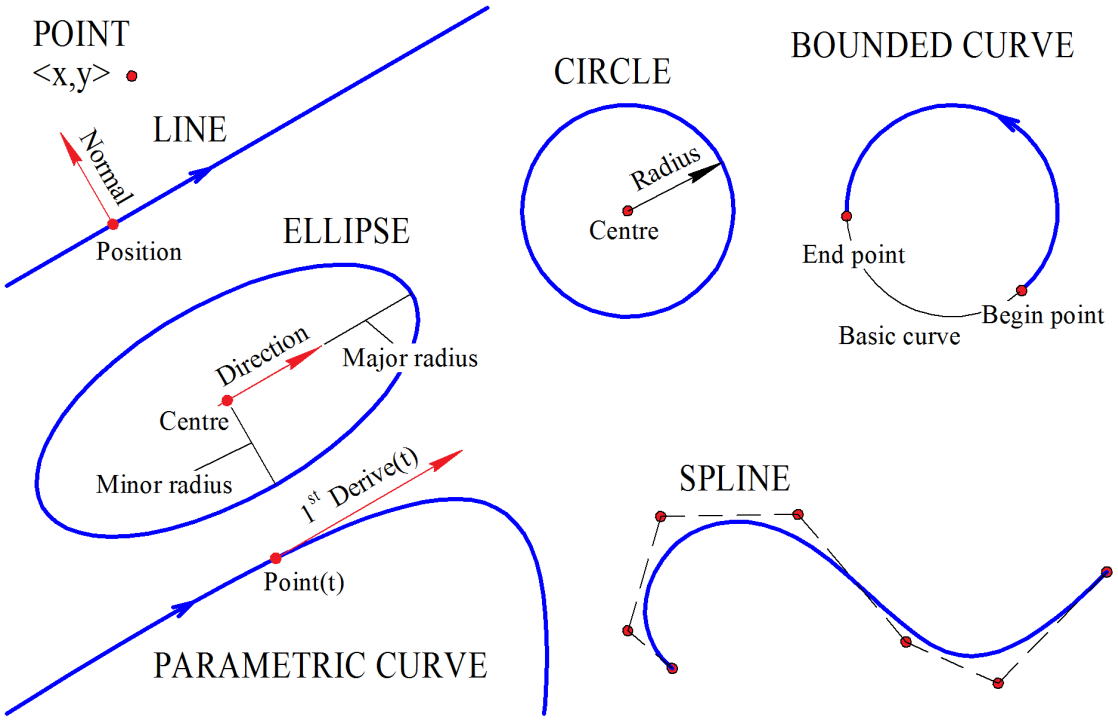


Fig. 1

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 direction vector equal to the normal vector rotated 90 degrees clockwise. In other words, the normal with <Y,-X> coordinates will correspond to the direction vector with <X,Y> coordinates.
- A **circle** is defined by its center point and a radius. C3D Solver supports circles with zero or positive radius. It is possible to specify to the C3D Solver that zero radius circles should be allowed on an individual basis using the **GCE_allow_zero_radius** callback function. Circles cannot have a negative radius. **Note**. The need for the zero radius option arises when solving the following problems:
  - constructing a parametric contour with disappearing and reappearing circular arcs;
  - construction of parametric 3D models with variable topology based on 2D contours;
  - construction of a surface along a trajectory of variable cross-section.

  One example of applying the zero-radius option is modeling the surface of an air duct with a variable cross-section, specifically a rectangular section with rounded corners transitioning into a rectangle without rounded corners.
- An **ellipse** is defined by its central point, radii along the major and minor semi-axes and the direction vector of the main semi-axis. Ellipse parameters are also defined by a periodic parameter ranging from ***0*** to ***2***π 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 *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 *General Parametric Curves*.
- A **bounded curve** is a curve portion bounded by end points on both sides. It is defined by three elements: a base curve, start and end points. C3D Solver supports a zero-length option for a bounded curve and line segment. In order to allow or prohibit the zero length of the curve on an individual basis, it is necessary to use the callback function **GCE_allow_zero_length**. By default, the zero-length option is disabled.

7.1.8 Degrees of Freedom

Each type of geometry has a number of degrees of freedom equal to the minimum number of coordinates required to specify the state of the geometric object. For example, a 2D point has two degrees of freedom. A circle has three degrees of freedom, as it is completely defined by three parameters <X, Y, R>, namely the 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 degrees of freedom are 2. For an ellipse, the degrees of freedom are 5. For a spline, the degrees of freedom are the sum of the degrees of freedom of its control points.

A parametric curve is fully determined by the application, i.e., it has zero degrees of freedom. Table 1 lists the degrees of freedom for all types supported by the solver.

Table 1 - Degrees of freedom for geometric objects

Geometry type	Degrees of freedom
Point	2
Line	2
Circle	3
Ellipse	5
Spline	2 · Number of control points
Parametric curve	0
Bounded curve	2 + Degrees of freedom of the base curve

Every geometric object included in the system adds a number of its degrees of freedom to the overall degrees 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, tan-

gency, 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.

7.1.9 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).
```

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).
```

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).
```

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).
```

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, direction 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).
```

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 *Spline Curves*.

A parametric curve of general form is added using

```
geom_item GCE_AddParametricCurve(GCE_system gSys, const MbCurve & crv).
```

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 MbCurve_Two-Dimensional_Curve_ 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 *General Parametric Curves*.

A **bounded curve** is added using

```
geom_item GCE_AddBoundedCurve(GCE_system gSys, geom_item curve, geom_item p[2]).
```

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.
// ...
// It is required to insert the code that assigns start values of the circle and its␣
↪endpoints.
// ...
geom_item p[2] = { GCE_AddPoint(gSys, endP1), GCE_AddPoint(gSys, endP2) };
geom_item circItem = GCE_AddCircle(gSys, circPars); // Create a circle. It's a base␣
↪curve of the arc.
geom_item arc = GCE_AddBoundedCurve(gSys, circItem, p); // Create 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).
```

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.

7.1.10 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 и **GCE_SetCoordValue**.

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).
```

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).
```

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 *Unary Constraints: Horizontality/Verticality and Fixing Variants*.

7.1.11 Geometric Object Control Points

Among the geometric types supported by the solver (see Item *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. 1).

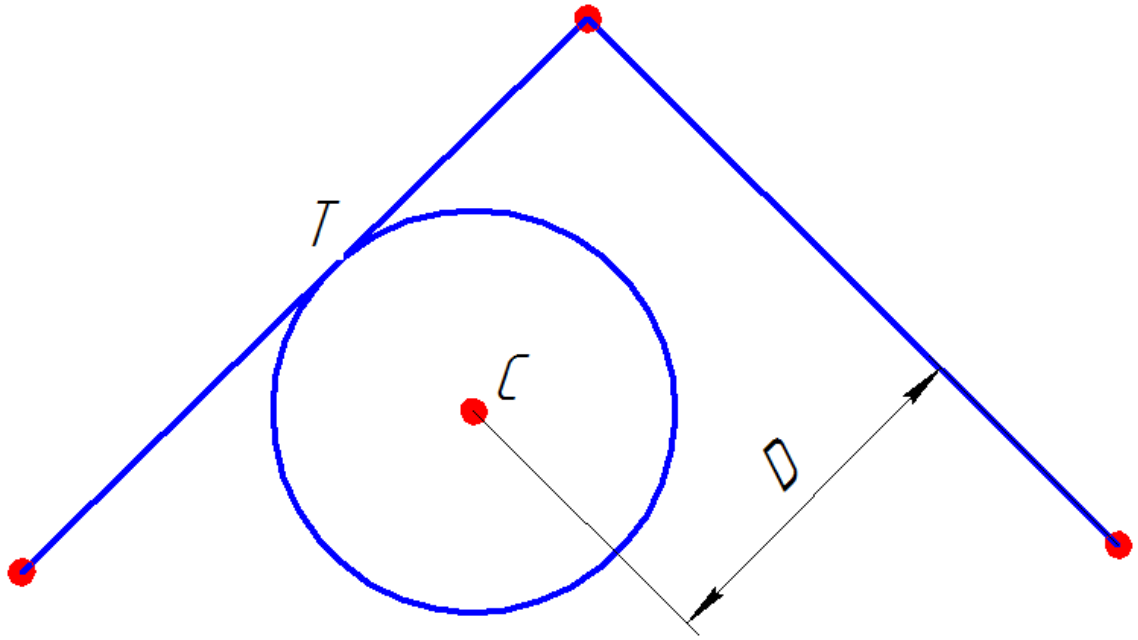


Fig. 1

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. 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).
```

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).
```

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. 2 shows a spline that has N = 7 control points (red circles) numbered starting from 0.

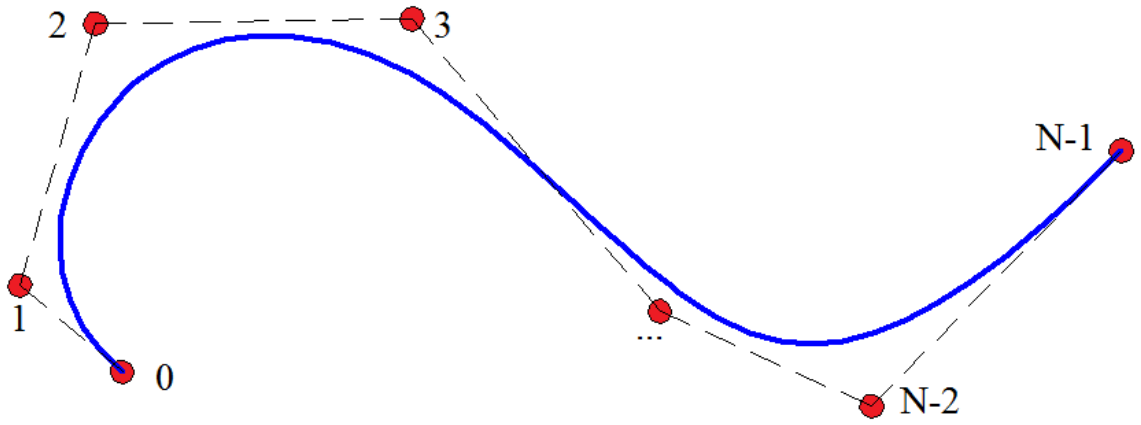


Fig. 2

7.1.12 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 *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 *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).
```

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),
```

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 *Driving and Variational Dimensions*.

7.1.13 Linear Equation

A linear equation is expressed by formula $a1 \cdot v1 + a2 \cdot v2 + a3 \cdot v3 + \ldots + an \cdot vn + c = 0$, where a1, a2, a3, ..., an, c are constant coefficients, and v1, v2, v3, ..., vn 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),
```

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.

7.1.14 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),
```

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.

7.1.15 Callback functions

Callbacks are a separate type of C3D Solver API function, the peculiarity of which is that they are called from within the C3D Solver and can be defined on the application side. The need for such functions arises when it is necessary to provide the solver with access to data stored on the application side, determine the properties of geometric objects, and also provide feedback to the solver with queries that affect the progress of calculations. Callbacks enhance the ability to integrate the solver with the application by providing another channel of interaction. All functions and data types for organizing callbacks are collected in a separate header file **gce_callback.h**.

In order for user-defined callback functions to work for a given constraint system, they must be registered using the `call`{#GCE_Register}

```
GCE_result GCE_Register(GCE_system gSys, const GCE_callback_table & cbTable)
```

Input data:

- **gSys** – constraint system that callbacks will serve;
- **cbTable** is a callback table that defines the following set of functions:
  - **gRegister** – a function of the `void (*GCE_geom_registered)(GCE_app_geom ag)` form called by the solver when a user geometry is registered with **gSys** using the GCM_Bind call.
  - **gUnregister** – a function of the `void (*GCE_geom_unregistered)(GCE_app_geom ag)` form called for the geometric object **ag** at the moment in time when the geometric solver releases the object **ag**, stopping all work with it. This event occurs on direct calls such as GCE_RemoveGeom, **GCE_RemoveSystem**, or **GCE_ClearSystem**.
  - **abortFunc** — function for requesting to abort calculations. This is a Boolean function of the form bool (***GCE_abort**)(), which is called from time to time in the solver while GCE_Evaluate is running and stops computing if the function returns true. A **GCE_Evaluate** that is aborted in this way will return the error code **GCE_RESULT_Aborted**.
  - **allowZeroRadius** – a callback function of the form `bool (*GCE_allow_zero_radius)(GCE_app_geom ag)` that tells the constraint system that circle **ag** allows a solution where its radius is zero.
  - **allowZeroLength** – a callback function of the form `bool (*GCE_allow_zero_length)(GCE_app_geom ag)`, that tells the constraint system that line segment or bounded curve **ag** allows a solution where their length is zero. For example, a CAD application can use the zero-length option to prohibit a line segment of zero length if it represents the axis of symmetry in a drawing. To enforce this prohibition, the application should return false when allowZeroLength is called.
  - **getLSegDirection** – a callback function of the form `GCE_vec2d(*GCE_get_lseg_direction)(GCE_app_geom ag)`, that allows to get the direction of the line segment **ag**. It is designed so that the application does not forget where the segment L = 0 is “looking” when it needs to return its length again. And also to avoid the effect of “turning out” the line segment when participating in such constraints as “tangency” and “at a distance”.

Return value:

- **GCE_RESULT_Ok** if the callback table was successfully registered for the constraint system;
- **GCE_RESULT_None** if callback registration fails.

Note

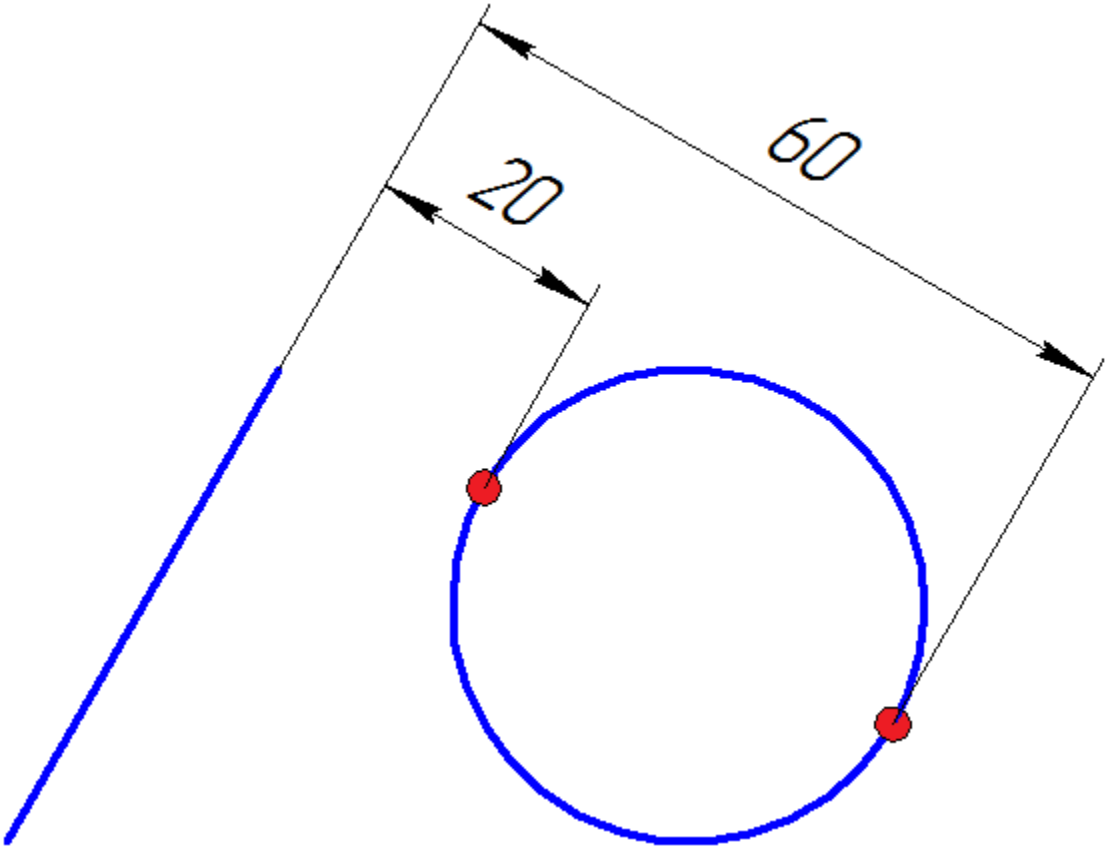
For the solver to work correctly, the ***GCE_Register*** call should be called once for the same constraint system, while it is still empty, for example, immediately after GCE_CreateSystem.

7.2 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 *Driving and Variational Dimensions*.

7.2.1 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.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.

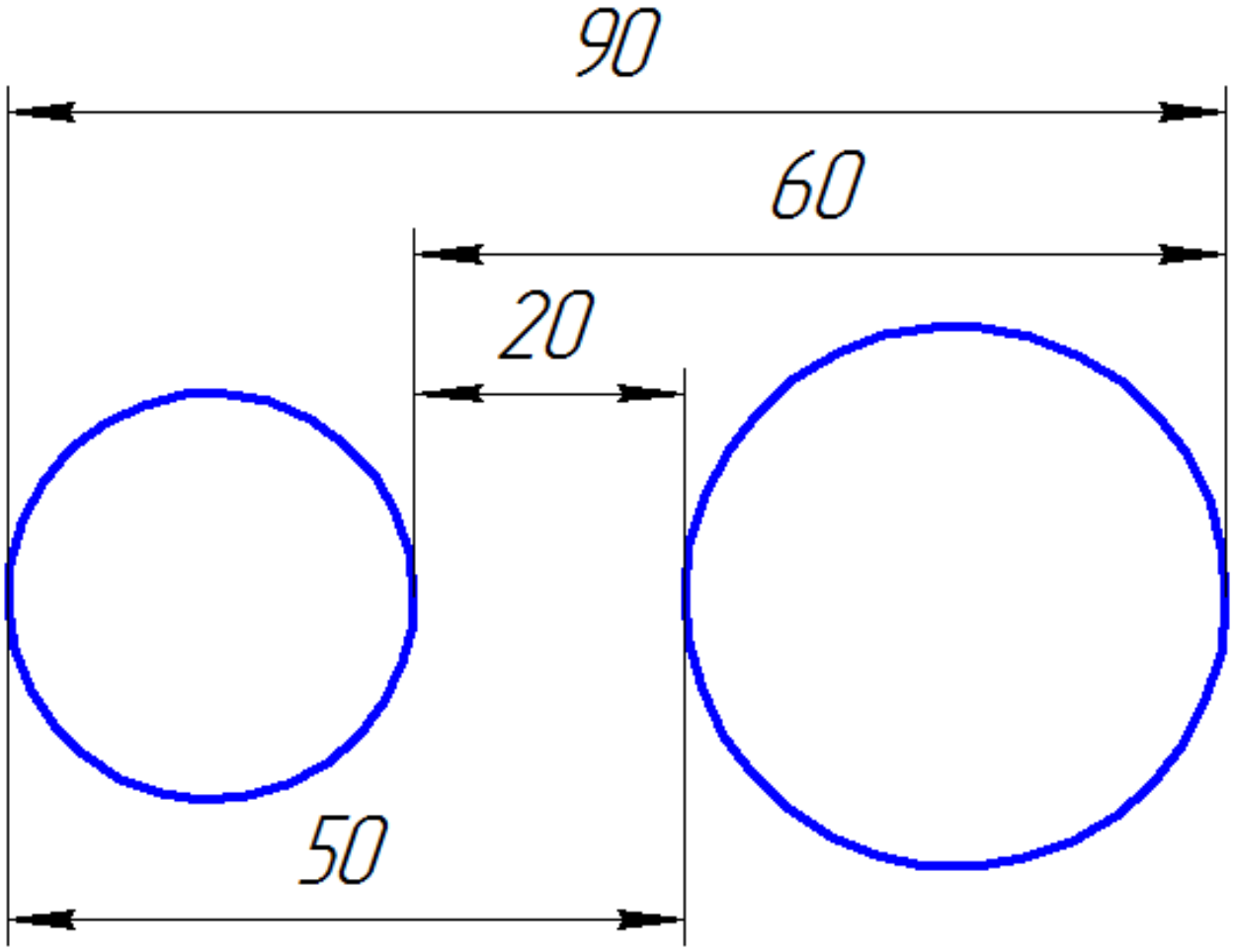


{width="50%"}

Fig. 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. 28.1.1, 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.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.



{width="50%"}

Fig. 2

7.2.2 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).
```

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.

7.2.3 Zero Dimensions and Signed Dimensions

All types of distance dimensions have a continuous domain of definition that includes zero, negative and positive values. Fig. 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.

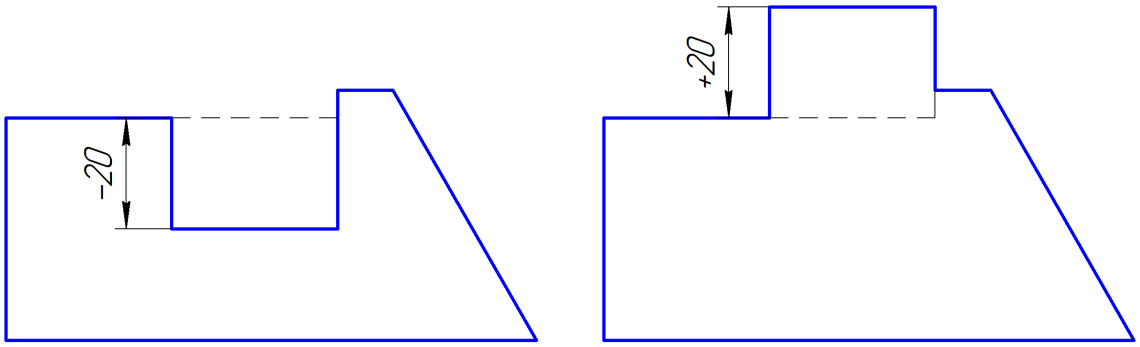


Fig. 1

Any type of distance and angular dimensions can have zero value. On the same Fig. 28.3.1, 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. 28.3.1, can be defined as “distance from point to line” or “point-to-point distance” constraint. In both cases, its behavior would be the same.

7.2.4 Distance Dimension

Distance dimension is set for a pair of geometric objects. Table 1 shows geometry type pairs supported for distance dimensions by the C3D Solver. Table 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. 28.1.2. in Item *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]{#GCE_AddDistance}

```
constraint_item GCE_AddDistance(GCE_system gSys, geom_item g[2]
                                , const GCE_ldim_pars & dPars);
```

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:

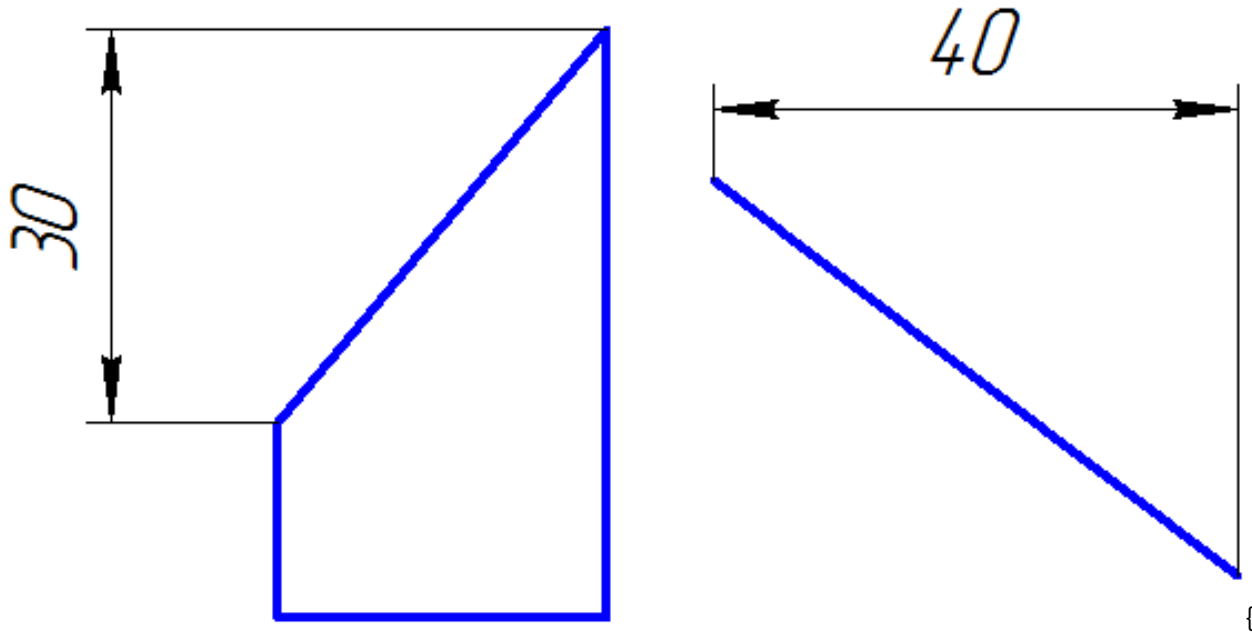
- **dPars.dPars** sets the parameter value, it can be either and independent parameter or a variable;
- **dPars.hp**[2] is a pair of auxiliary points that set the measurement result when it is required (learn more in Item *Auxiliary Points of Distance Dimension*).

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 *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 *Auxiliary Points of Distance Dimension*.

7.2.5 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. 1 shows the examples of vertical and horizontal dimensions for a point pair. Generally, the dimension direction can be arbitrary.



{width="50%"}

Fig. 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

```
{ll}
constraint_item GCE_AddDirectedDistance(GCE_system gSys, geom_item p[2]
                                        , const GCE_ldim_pars & dPars)
```

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** = 0; if it is required to define a *vertical* dimension, then **dPars.dirAngle** = $\pi/2$. Generally, **dPars.dirAngle** can range from 0 to 2π.

Note. In the current C3D Solver version, the directed dimension can be set only for points.

7.2.6 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).
```

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.

7.2.7 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 0 to 2π and it is defined by an angle between the direction vectors of the first and second objects measured counterclockwise. An angular dimension measured this way is shown in Fig.1 as two intersecting lines with given directions. The right part of Fig.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 $D = 2\pi - \alpha$, where α is the measured angle between the vectors, and 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.

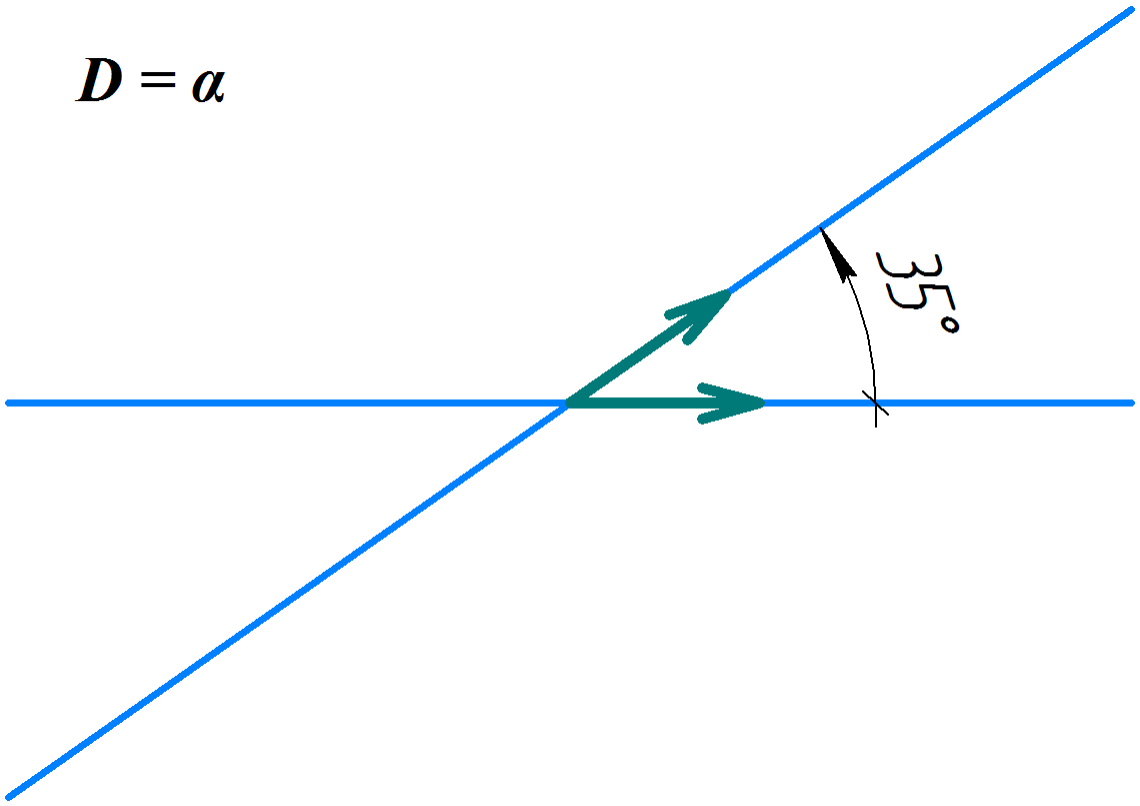


Fig. 1

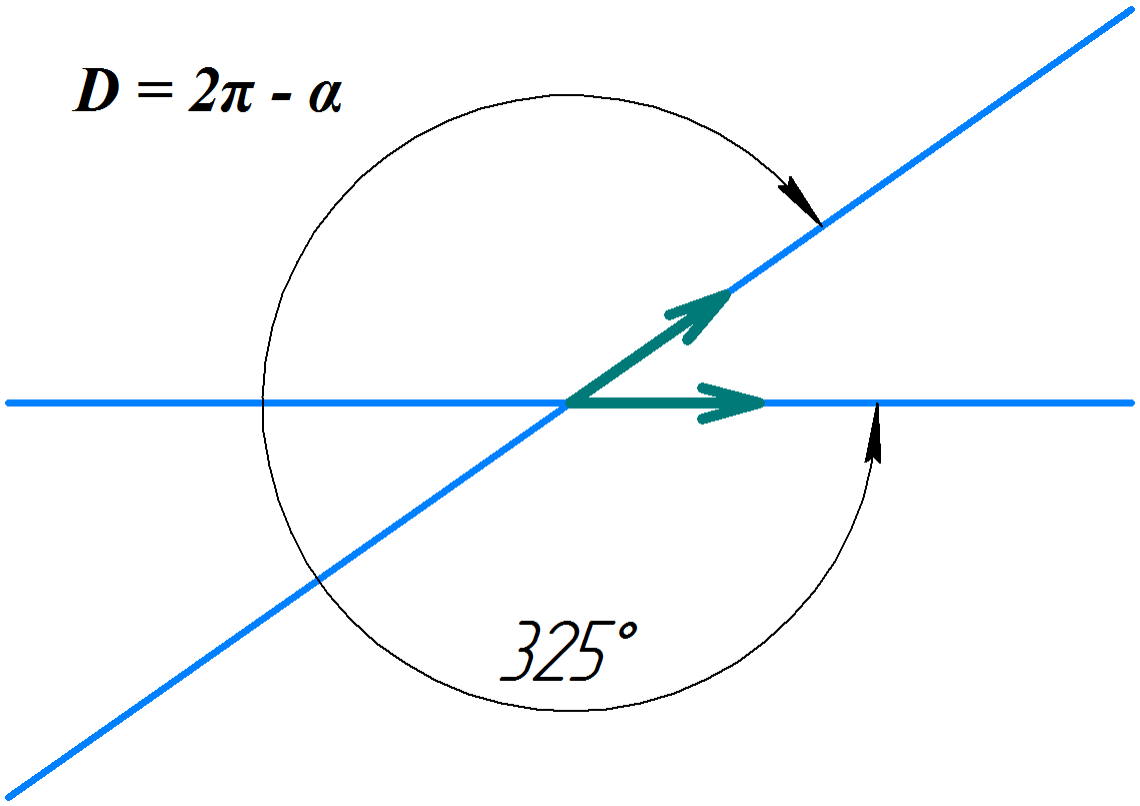


Fig. 2

There are four ways to define an angular dimension for one pair of directions, including the two ways described above. Fig.3 shows a dimension, its value is determined by adjacent angle formula: $D = \pi - \alpha$. Fig.4 (right part) shows an alternate adjacent angle version that is defined by the formula $D = \pi + \alpha$.

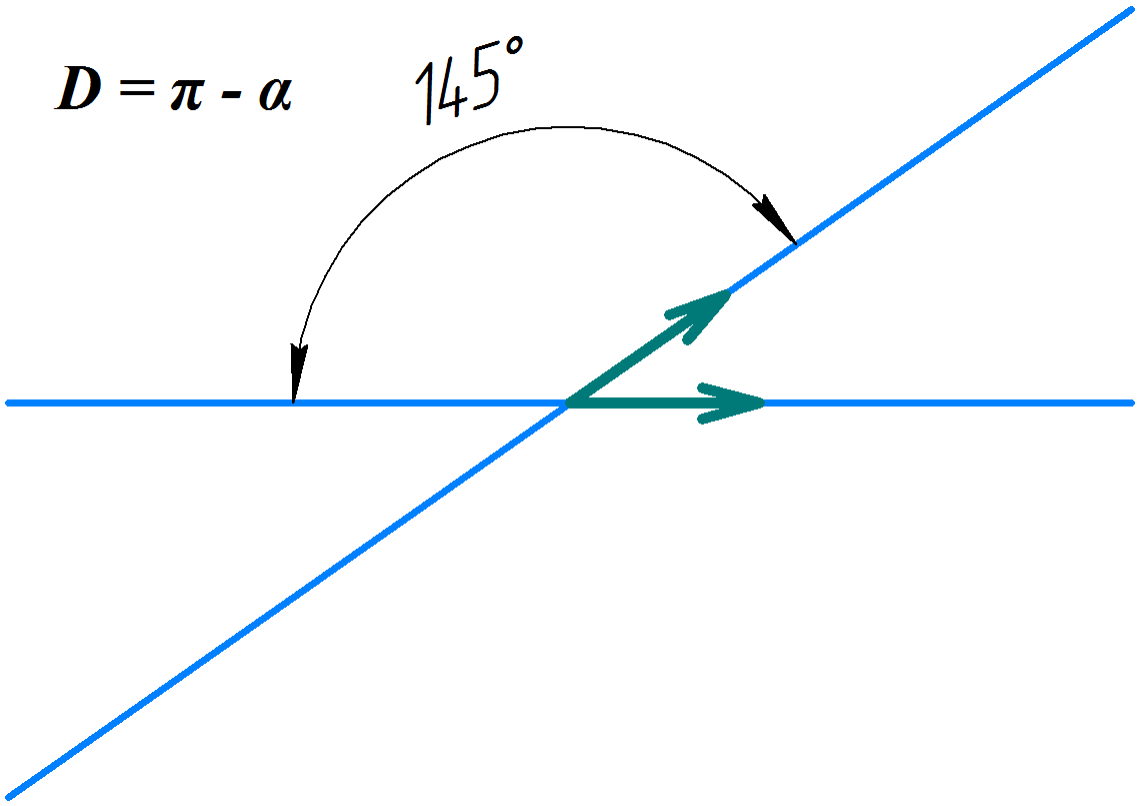


Fig. 3

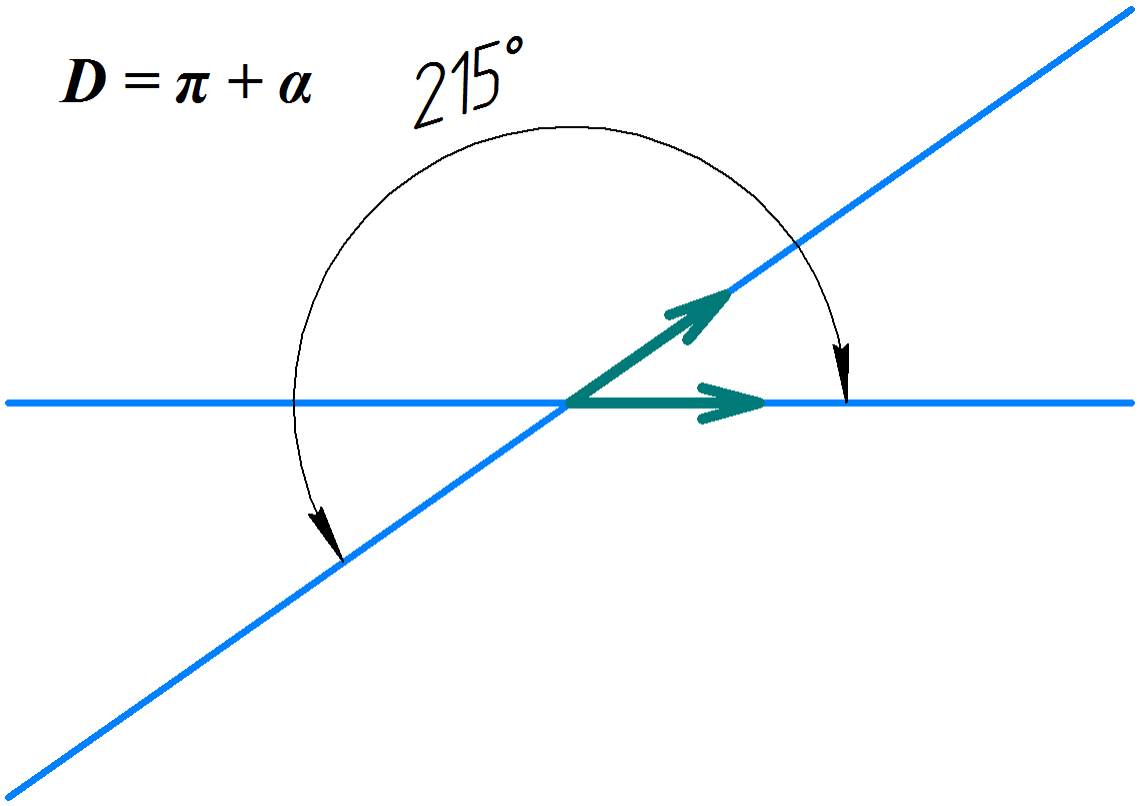


Fig. 4

One more pattern associated with four ways to set the angular dimension is that the dimension with the value $D = 2\pi - \alpha$ can be produced from the dimension with the value $D = \alpha$ after a simple swap of the first and second dimensional constraint objects. Similarly, the 4th variant ($D = \pi + \alpha$) can be received from the 3rd one ($D = \pi - \alpha$) 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).
```

The function creates a new angular dimension defined for the **l1** and **l**2 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 *Driving and Variational Dimensions*.

If **dPars.adjacent = false**, then the dimension value would be defined by the formula $D = f \cdot \alpha$. The ***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** = $180/\pi$, 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. 28.7.3 and Fig. 28.7.4, 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 ***f*** factor is also defined by the **dPars.factor** parameter.

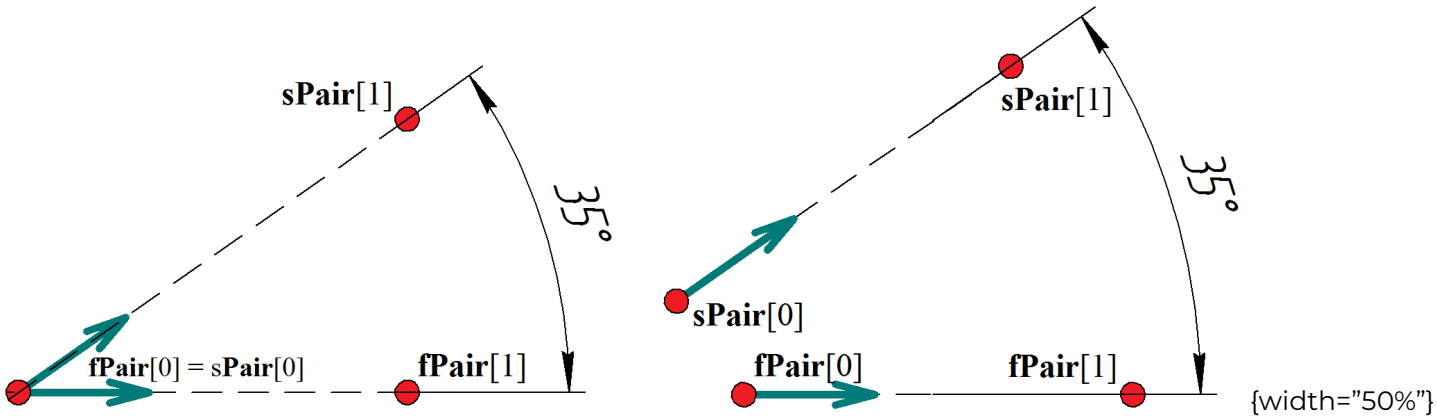
7.2.8 Angular Dimension Based on 3 or 4 Points

C3D Solver provides one more method to determine an angular dimension, in this case the direction 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).
```

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. 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].



{width="50%"}

Fig. 1

Other parameters of the **GCE_AddAngle4P** function are set in the same way as for the **GCE_AddAngle** function.

7.2.9 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 *Driving and Variational Dimensions*.

7.3 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.

7.3.1 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.

7.3.2 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 direction. Note that it does not guarantee that the order of aligned points along a given direction will be preserved.

Alignment of points with a constant alignment direction 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 $ang = \pi/2$, then the points would be aligned vertically.

Alignment of points with a variable alignment direction is set by the method

```
constraint_item GCE_AddAlignPoints(GCE_system gSys, geom_item p[2], geom_item dir)
```

Similar to the constant alignment direction method, this implementation takes a constraint system **gSys** and a pair of points **p[2]**.

The **dir** parameter is any object registered in the solver for which the call **GCE_GetVectorValue**(gSys, dir, GCE_DIRECTION) is valid.

Alignment will occur along the direction vector (**GCE_DIRECTION**) of the linear object or the major axis of the ellipse.

This constraint can be used in conjunction with GCE_AddLinearDimension to set a linear dimension as an alternative to the GCE_AddDistance method. This method is preferable as it allows the help points of the dimension to be better customized to suit user expectations.

7.3.3 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 GGCE_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.

7.3.4 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])
```

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])
```

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.

7.3.5 Equal Curve Properties

C3D Solver supports equality of four curve properties: lengths GCE_AddEqualLength, radii GCE_AddEqualRadius, offsets of equidistant curves GCE_AddEqualOffset, and curvatures at specified points GCE_AddEqualCurvature. Let's explore each of these types.

The lengths of two curves can be equalized using the GCE_AddEqualLength function.

```
constraint_item GCE_AddEqualLength(GCE_system gSys, geom_item curve1,
                                                    geom_item curve2);
```

It takes a constraint system **gSys** and two curves, **curve1** and **curve2**, and returns a descriptor of a new constraint.

The Equal Length constraint can be applied to curves that either have endpoints (i.e., queries for their GCE_FIRST_END and GCE_SECOND_END coordinates return a result other than GCE_NULL_G) or are closed. This constraint is supported for any combination of curves for which the *GCE_AddLength* constraint is available.

The radii of two curves can be equalized using the GCE_AddEqualRadius function.

```
constraint_item GCE_AddEqualRadius(GCE_system gSys, geom_item c1,
                                                    geom_item c2);
```

It takes a constraint system **gSys** and two curves, **curve1** and **curve2**, and returns a descriptor of a new constraint.

The Equal Radius constraint can be applied to curves for which querying the GCE_RADIUS value is valid.

The offsets of two equidistant curves can be equalized using the GCE_AddEqualOffset function.

```
geom_item GCE_AddEqualOffset(GCE_system gSys, geom_item curve1,
                                              geom_item curve2);
```

It takes a constraint system **gSys** and two equidistant curves, **curve1** and **curve2**, and returns a descriptor of a new constraint.

The Equal Offset constraint can be applied to curves for which querying GCE_OFFSET value is valid.

The curvatures at given points on two curves can be equalized using the GCE_AddEqualCurvature function.

```
constraint_item GCE_AddEqualCurvature(GCE_system gSys, geom_item curves[2],
                                                       var_item tPars[2]);
```

It takes a constraint system **gSys**, two curves **curves**, and parameters (if required), in which curvatures should be equal, and returns a descriptor of a new constraint

If given a GCE_CIRCLE or a GCE_BOUNDED_CURVE, based on a circle, the corresponding tPars[i] is expected to be set to GCE_NULL_V, since the curvature of a circle is same at every point. For a GCE_LINE or a GCE_LINE_SEGMENT the method is applied only in conjunction with GCE_SPLINE.

7.3.6 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 1.

Table 1 - Unary constraint types

Constraint type	Description
GCE_FIX_(	**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_HOR GCE_VER	**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 direction vector by 180 degrees.
GCE_ANG	This constraint fixes the angular position of a line or a segment. The GCE_ANGLE_OX constraint is described in greater detail in Item *Angular Position Constraint*.
GCE_LEN(	This constraint fixes the line segment length. It is described in more detail in Item *Length Fixing*.
GCE_RAD	This constraint is described in greater detail in Item *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.

7.3.7 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.

```
constraint_item GCE_AddLength(GCE_system gSys, geom_item curve,
                                               GCE_dim_pars dPar);
```

It takes a constraint system **gSys**, a curve **curve**, and parameters of linear dimension, and returns a new constraint descriptor.

This constraint is supported for the following curve types: line segment, circle, spline, and bounded curve (GCE_BOUNDED_CURVE) based on a line, circle, or spline.

A curve length is fixed using the method

```
constraint_item GCE_FixLength(GCE_system gSys, geom_item curve);
```

It takes a constraint system **gSys** and a curve **curve**, and returns a descriptor of a new constraint.

This constraint can be applied to all curves supported by GCE_AddLength method.

Length fixing is a dimensional constraint, so it's possible to change the length using the *GCE_ChangeDrivingDimension* method.

7.3.8 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 *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.

7.3.9 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 direction vector by 180 degrees. As this is a dimensional constraint, the line angular position can be changed by calling **GCE_ChangeDrivingDimension**.

7.3.10 Tangency

Tangency is a geometric constraint that positions a pair of curves so that they would be tangent in one point. Table 1 shows supported tangency curve pairs.

Table 1 - Curve pairs supported by tangency constraint

	Line	Circle	Ellipse	Spline	Parametric curve
Line		√	√	√	√
Circle	√	√	√	√	√
Ellipse	√	√		√	
Spline	√	√	√	√	
Parametric curve	√	√			

Selecting the tangency variant. Fig. 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. 29.10.1. 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.

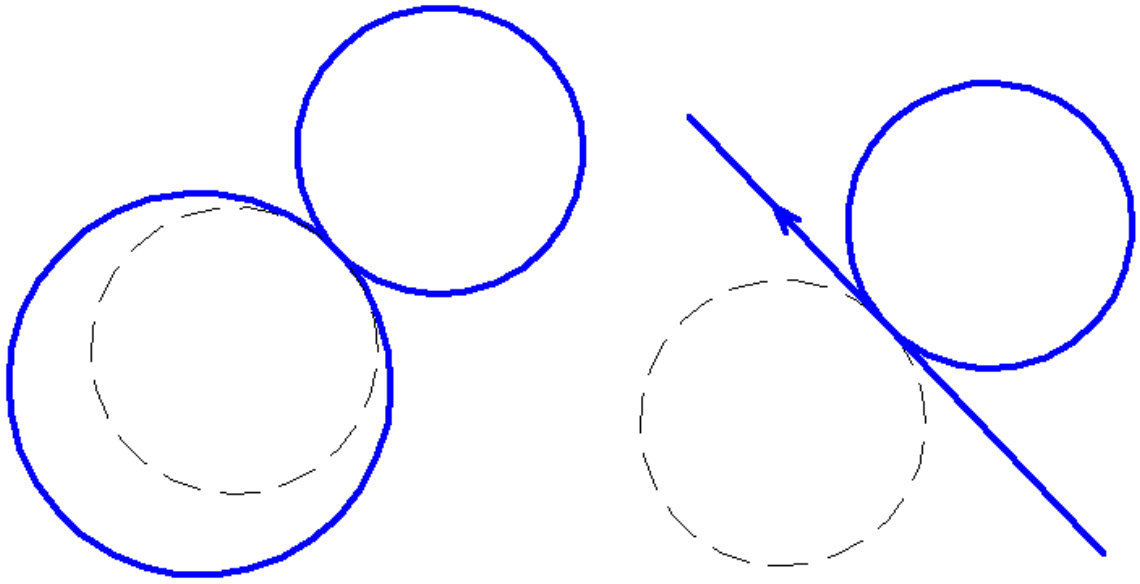


Fig. 1

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 *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. 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.

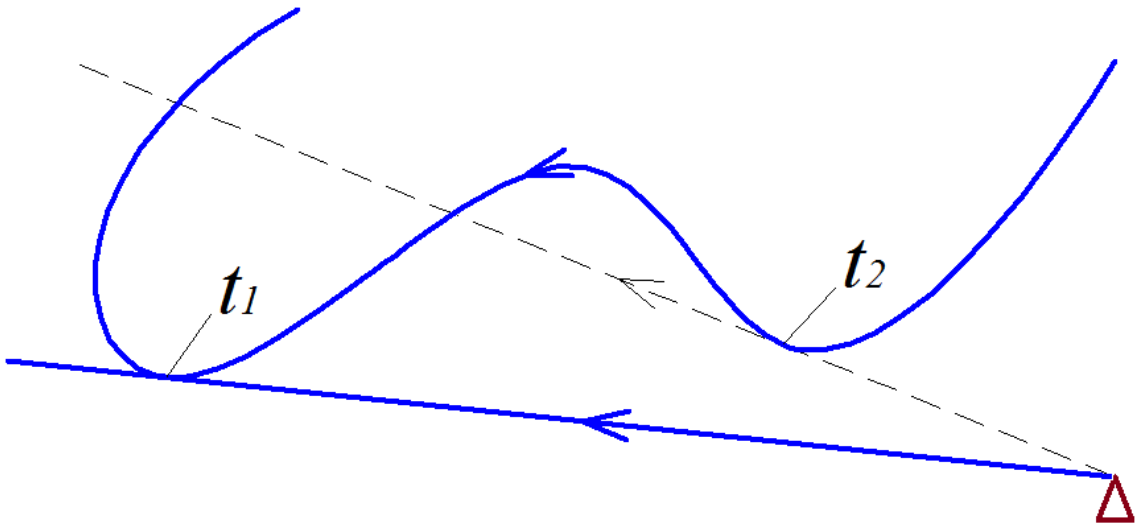


Fig. 2

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 *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.

7.3.11 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. 1 shows a horizontal line and a spline with two set tangency constraints that have various tangency points.

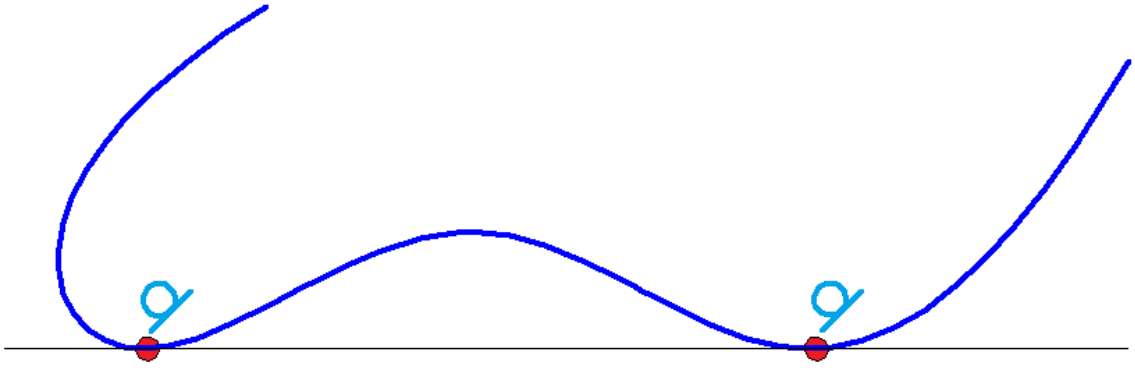


Fig. 1

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. 2 shows smooth joints of a spline with other curves.

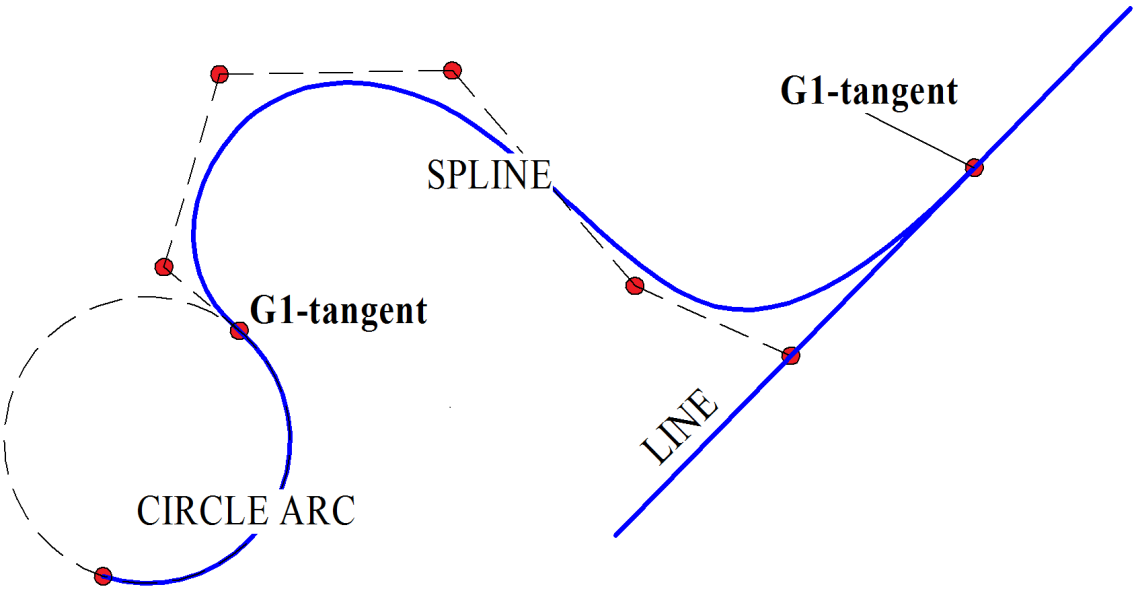


Fig. 2

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**.

7.3.12 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)
```

7.3.13 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)
```

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 *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.

7.3.14 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])
```

7.3.15 Mutual Object Positioning

The Item *Tangency* shows an example of tangency of a circle and a line that can have two mutual positioning variants (Fig. 29.10.1). 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.

7.4 CALCULATION OF TWO-DIMENSIONAL CONSTRAINTS

7.4.1 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)
```

The function returns the **gSys** constraint system calculation. 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 cant 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. Occurs if a geometric object (circle/line segment/bounded curve) is prohibited from degeneration by means of the **GCE_allow_zero_radius**/**GCE_allow_zero_length** callback functions. And also, in general, without using a structure with callback functions GCE_callback_table or with this structure set by default.
- **GCE_RESULT_InvalidGeometry**. The found solution leads to degeneration of geometric objects, for example, a segment or arc becomes zero length, and a circle or ellipse becomes zero radius. The geometry state remains unchanged. This situation occurs if the callback functions **GCE_allow_zero_radius** or **GCE_allow_zero_length** prohibit a geometric object (circle, segment, or bounded curve) from taking a zero radius or length. It can also occur generally if the callback function structure `GCE_callback_table` is used with default settings (without override via the GCE_Register call).

7.4.2 Changing or Requesting 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)
```

7.4.3 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 *Mutual Object Positioning*.

7.4.4 Underdefined Constraint Systems

The item above (see Item Overdefined Consistent And Inconsistent Constraint Systems) 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 degrees 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 *Initial Approximation*).

7.4.5 Degrees 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 *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. Degrees of freedom analysis functions show (visualize) underdefined geometric objects to the user.

One of them calculates the degrees of freedom for a point:

```
GCE_point_dof GCE_PointDOF(GCE_system gSys, geom_item pnt)
```

This method allows determining the current degrees 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.

Another one calculates the degree of freedom for a geometric coordinate:

```
GCE_point_dof GCE_CoordDOF(GCE_system gSys, geom_item g, coord_name cName)
```

This method allows determining the current degree of freedom of a geometric coordinate, such as the coordinate of a point or vector, the angle of a line's normal, the angle of an ellipse, the radius of a circle, and so on.

7.4.6 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)
```

Connectivity with constraints is evaluated by the

```
bool GCE_IsConstrainedGeom(GCE_system gSys, geom_item g )
```

Constraint satisfaction is evaluated by the

```
bool GCE_IsSatisfied(GCE_system gSys, constraint_item cItem)
```

Degenerated geometric objects are returned by the

```
std::vector<geom_item> GCE_DiagnoseGeometry(GCE_system gcSys)
```

Compliance of the control point with the given coordinates is assessed by the

```
double GCE_DimensionParameter(GCE_system gSys, constraint_item dItem)
```

The current constraint solution state is returned by the

```
bool GCE_CheckPointSatisfaction(GCE_system gSys, geom_item pnt
                                , point_type cp
                                , double px
                                , double py)
```

The current constraint solution state is returned by the

```
GcConState GCE_GetConstraintState(GCE_system, constraint_item gc_item).
```

The state of the constraint system according to the latest solution results is returned by the

```
GcConstraintStatus GCE_GetConstraintStatus(GCE_system gSys).
```

7.4.7 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)
```

7.4.8 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.

7.4.9 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.

7.5 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.

7.5.1 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).
```

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 )
```

7.5.2 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.

7.5.3 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)
```

7.6 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 *Collisions detection of bodies*) the geometric solver helps CAD users assess the motion boundaries of mechanism parts.

7.6.1 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 *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 *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 *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 *Driving dimensions*).

7.6.2 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.

7.6.3 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**. 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 *Adding and deleting geometric objects*). Descriptors (see *Table 32.6.1. 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 *Adding and deleting geometric objects*. Application constraints refer to C3D Solver constraints using special descriptors (see *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 *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 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.

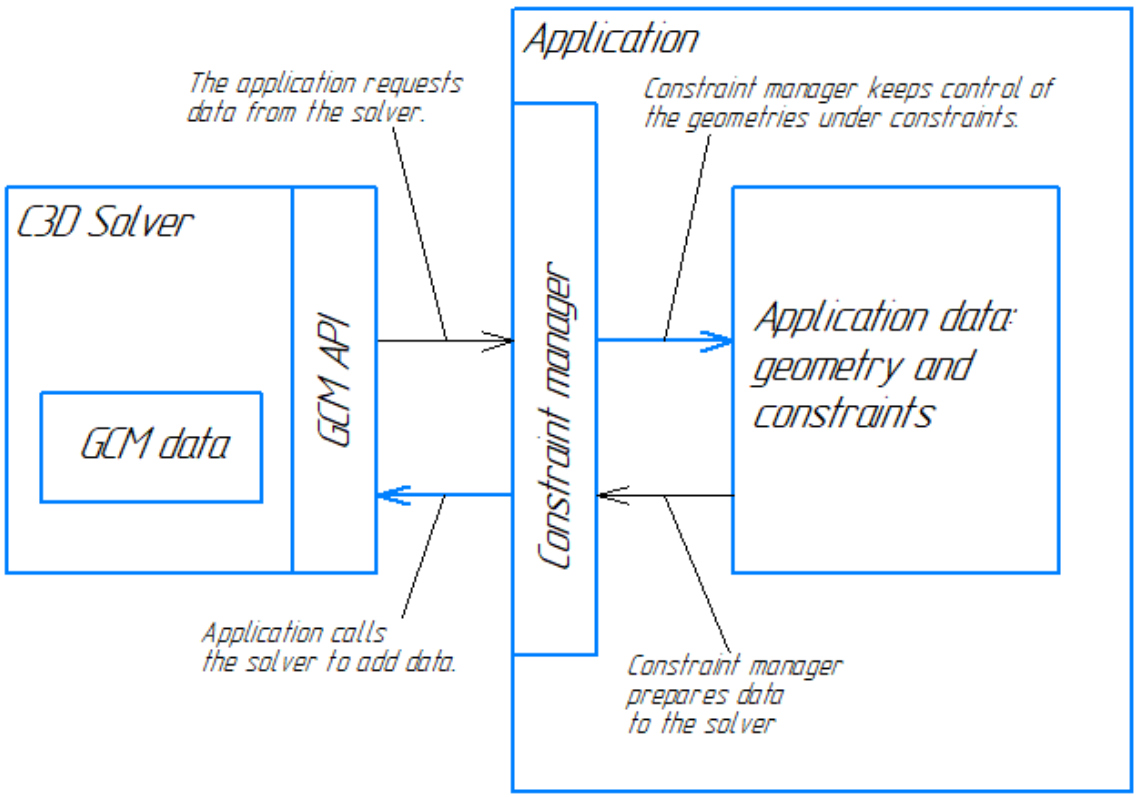


Fig. 1

The diagram shown in Fig. 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.

7.6.4 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(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).

7.6.5 Supported constraint types

Any geometric constraint links geometric objects that are called constraint *arguments*. Any object listed as a geometry type in section *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 1:

Table 1 - Geometric constraint types (enum GCM_c_type).

Constraint	Arity (number of arguments)	Designation in API
Logical constraints		
Coincidence		**GCM_COINCIDENT**
Concentricity		**GCM_CONCENTRIC**
Parallelism		**GCM_PARALLEL**
Perpendicularity		**GCM_PERPENDICULAR**
Tangency		**GCM_TANGENT**
Mirror symmetry		**GCM_SYMMETRIC**
User-defined dependency	N	**GCM_DEPENDENT**
Dimensional constraints		
Distance	+1	**GCM_DISTANCE**
Angle	+1	**GCM_ANGLE**
Radius	+1	**GCM_RADIUS**
Pattern constraints		
Dependency on a pattern	+1	**GCM_PATTERNED**
Linear pattern (translational)		**GCM_LINEAR_PATTERN**
Angular pattern (rotational)		**GCM_ANGULAR_PATTERN**
Mechanical transmission		
Mechanical transmission	+2	**GCM_TRANSMITTION**
Cam mechanism	+2	**GCM_CAM_MECHANISM**

7.6.6 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 1 Descriptor data types).

Table 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 descripto
GCM_pattern	GCM_object	pattern descriptor

Data types with finite enumerations of possible values are listed in Table 2.

Table 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 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 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

7.6.7 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 *Supported geometry types*, *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 *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).

7.6.8 Representation of geometric objects

GCM three-dimensional geometric solver uses a certain representation of data of geometric objects which is illustrated in Table 1. All objects are expressed in coordinates of points, vectors and numbers (scalars) written in a form specific for each type.

Records of all these geometric objects can be unified in the form of the tuple of values:

<$OZXYR_AR_B$>

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 <$OZXYR_AR_B$> is used in the GCM_g_record structure, whose data fields can be found in Table 2:

Table 2 - Description of data fields of the GCM_g_record structure.

Data type	Data field	Interpretation (geometrical meaning is specified in table 32.8.1.)
GCM_g_ty	type	Geometry type.
GCM_POI	origin	Point of the geometric object positioning (origin of LCS, center of a circle or sphere, etc.).
GCM_vec	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 *Adding and deleting geometric objects*.

7.6.9 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 objects 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.

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 32.8.1.).

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 emptyGCM_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));
GCM_geom cyl = GCM_AddGeom(gSys, GCM_Cylinder(cylPos, MbVector3D::zAxis, 10.0));
...
...
/*
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 *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.

7.6.10 Adding and deleting geometric restrictions

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 does't matter in 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. 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 1 - Data argument types for various constraints (GCM_c_record completion)

Constraint type	type				
type	args[0]	args[1]	args[2]	args[3]	args[4]
GCM_COINCIDENT	GCM_geo	GCM_geo	GCM_alignment		
GCM_CONCENTRIC	GCM_geo	GCM_geo	GCM_alignment		
GCM_PARALLEL	GCM_geo	GCM_geo	GCM_alignment		
GCM_PERPENDICULAI	GCM_geo	GCM_geo	GCM_alignment		
GCM_TANGENT	GCM_geo	GCM_geo	GCM_alignment	GCM_tan_ch	
GCM_SYMMETRIC	GCM_geo	GCM_geo	CM_geom	GCM_alignm	
GCM_DISTANCE	GCM_geo	GCM_geo	double	GCM_alignm	
GCM_ANGLE (планарный угол)	GCM_geo	GCM_geo	GCM_geom (ось вращения)	double	GCM_alignment
GCM_ANGLE (3D угол)	GCM_geo	GCM_geo	GCM_NULL (без оси)	double	GCM_alignment
GCM_RADIUS	GCM_geo	GCM_geo			

GCM_alignment is an option value considered in *GCM_alignment option*. GCM_tan_choice values will be described in sections dedicated to the tangency constraint (GCM_TANGENT). Table 32.10.1. 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.

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 *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:

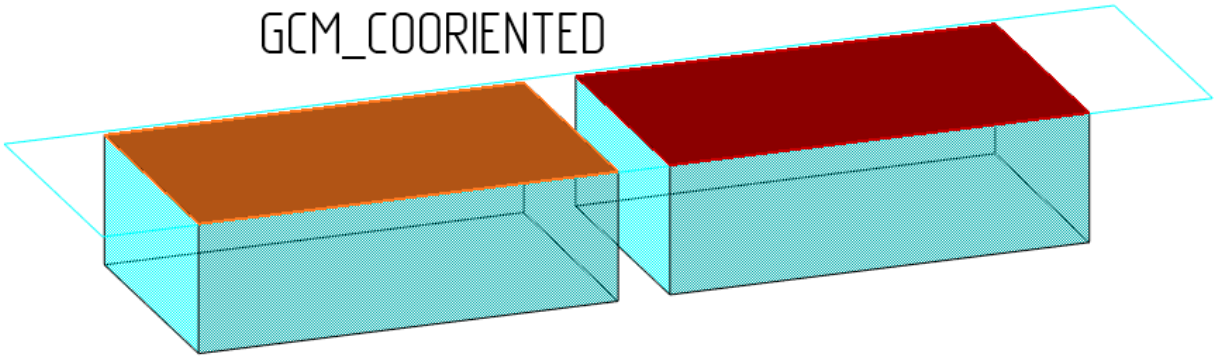
```
GCM_constraint GCM_AddDistance(GCM_system gSys, GCM_geom g1, GCM_geom g2, double␣
↪dVal, GCM_alignment aVal)
```

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.

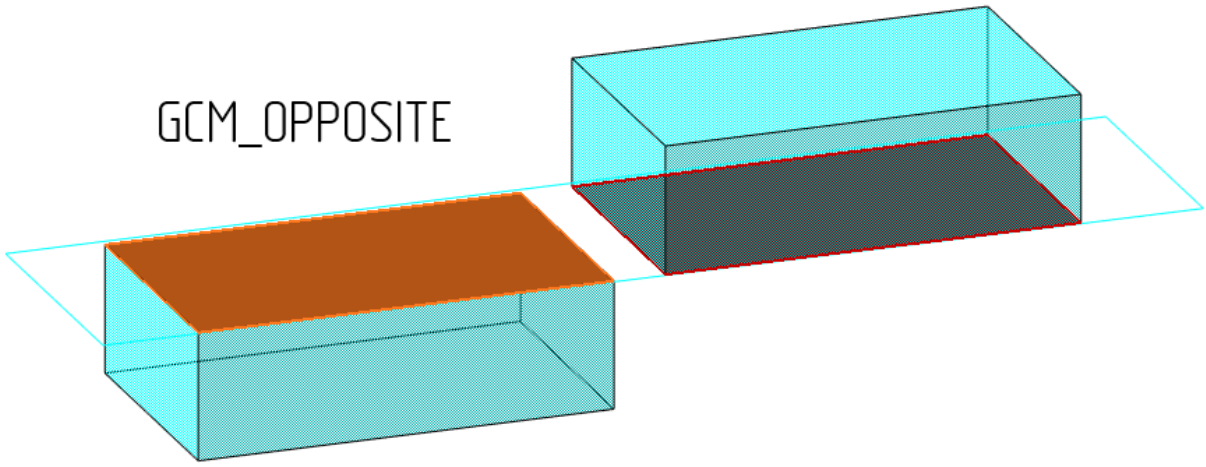
7.6.11 GCM_alignment option

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 1, 2 demonstrate alternative alignment variants for two bodies using the GCM_COINCIDENT constraint that binds a pair of colored faces.



{width="50%"}

Fig. 1



{width="50%"}

Fig. 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_AddDistance, GCM_AddConstraint 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 (1) or the opposite direction (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 1 - GCM_alignment option values.

Name	Value
GCM_NO_ALIGN	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_COORIEN	Sets cooriented Z vectors of constraint arguments.
GCM_OPPOSITE	Sets opposite Z vectors of constraint arguments.
GCM_ALIGNED_ GCM_ALIGNED_ GCM_ALIGNED_ GCM_ALIGNED_	Solution variants for constraints with cooriented Z vectors of constraint arguments (GCM_COORIENTED subvariants).
GCM_ALIGNED_ GCM_ALIGNED_ GCM_ALIGNED_ GCM_ALIGNED_	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_W	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.

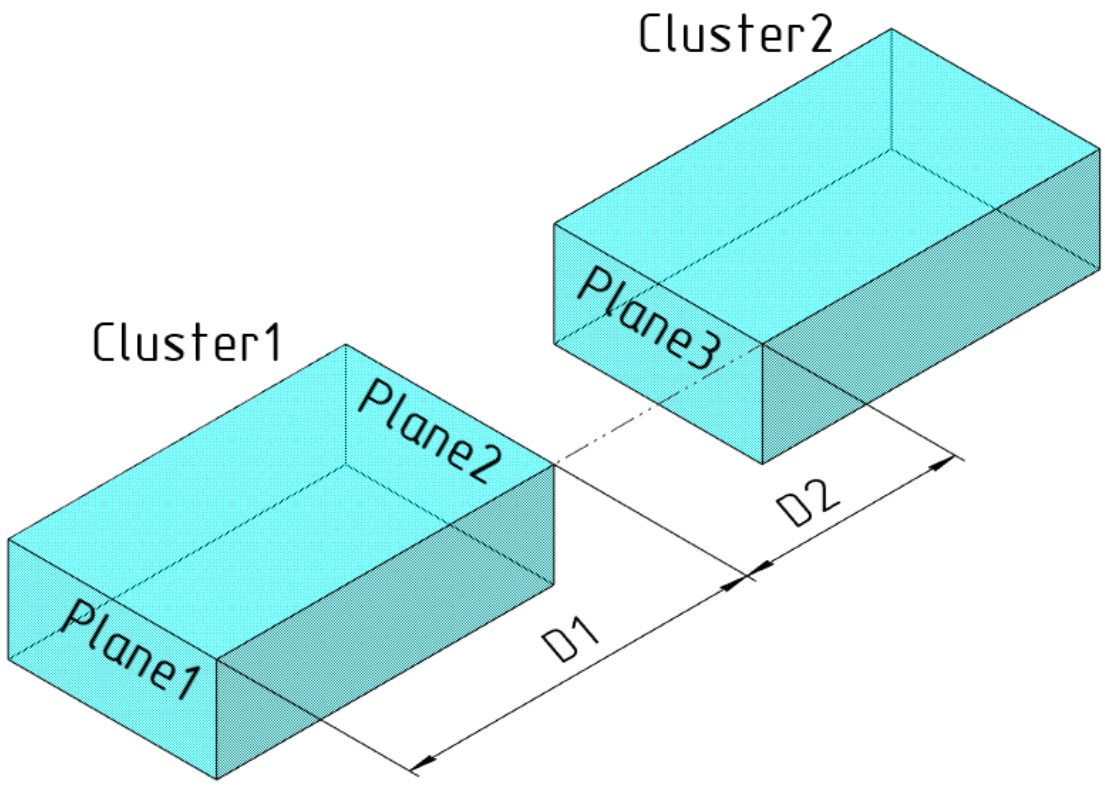
7.6.12 Geometric scene clustering, assembly modeling

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 *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. 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.

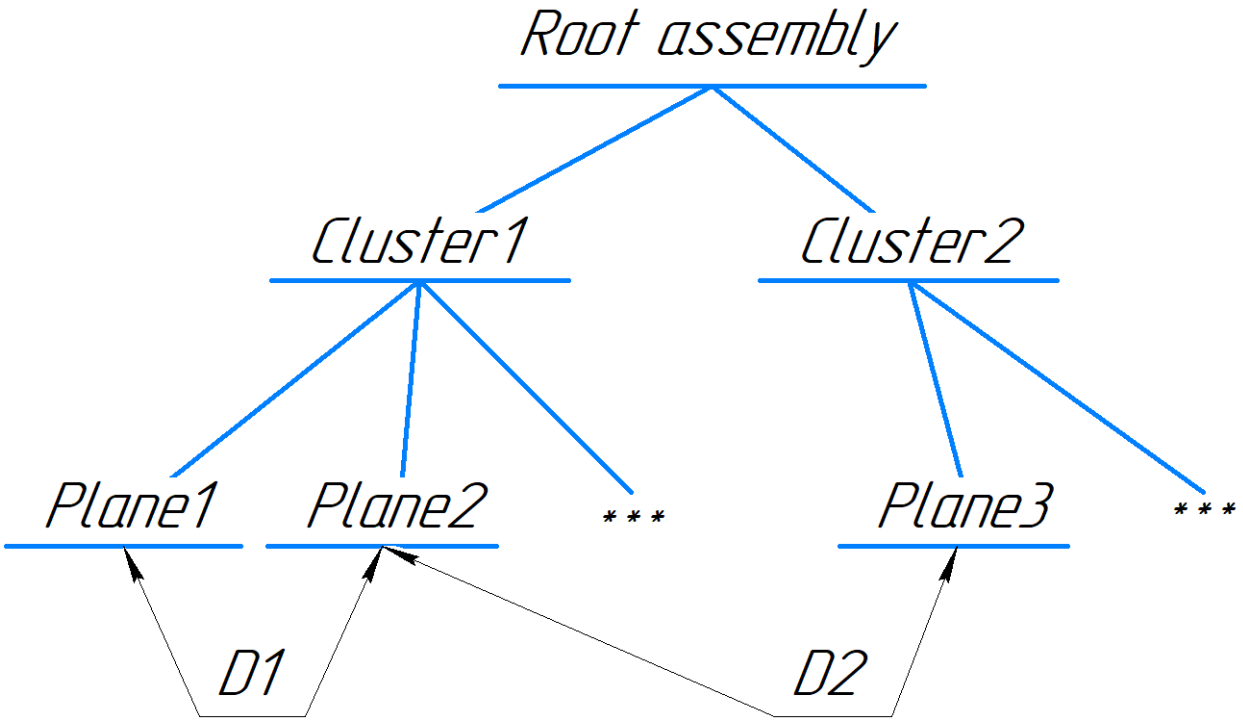


{width="50%"}

Fig. 1

The entire hierarchy consists of three constraint subsystems, calculated independently from

the bottom up (see Fig. 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.



{width="50%"}

Fig. 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. 1. This code is an example demonstrating how the solver creates the cluster hierarchy shown in Fig. 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));
```

(continues on next page)

(continued from previous page)

```
/*
Set a distance dimension D1 between 'Plane1' and 'Plane2'.
(opposite sides of the parallelepiped, see figure 32.12.1)
*/
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 32.12.1)
*/
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.

7.6.13 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 *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 *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 *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.

7.6.14 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 1 - Enumeration type values **GCM_result**

Resulting codes of executing the API solver functions	
GCM_RESULT_None	**No result.** For the call **GCM_EvaluationResult** 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_Overconstrained	**The constraint redefines the model.** The error occurs in the event of conflicting constraints (Inconsistent constraint system) within an over-defined constraint system (Overdefined constraint system).

continues on next page

Table 1 – continued from previous page

Resulting codes of executing the API solver functions	
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 a dimension is 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 objects 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.

continues on next page

Table 1 – continued from previous page

Resulting codes of executing the API solver functions	
GCM_RESULT_Duplicated	**Duplicate constraint.** The code indicates that the constraint system contains two identical constraints. In general, this situation does not make it impossible to calculate constraints.
Diagnostic situations for the GCM_DEPENDED constraint	
GCM_RESULT_DraggingFailed	
GCM_RESULT_InappropriateAlignment	
GCM_RESULT_InconsistentAngleType	
GCM_RESULT_InconsistentAlignment	
mtResCode_UnsupportedTangencyChoice	
mtResCode_IsNoPossibleForCircTanChoice	
GCM_RESULT_InconsistentPlanarAngle	
Diagnostic codes for mechanical transmissions	
GCM_RESULT_DependentConstraintUnsolv	
GCM_RESULT_CyclicDependence	
GCM_RESULT_MultiDependedGeom	
GCM_RESULT_OverconstrainingDependedC	
mtResCode_InvalidDependenceForFixGeor	
Diagnostic codes for mechanical transmissions	
mtResCode_CoaxialMtGearTransmissionIsN	
mtResCode _NoSeparatedSolutionForCamGear	
mtResCode_CyclicDependenceForTwoOrM	
mtResCode_InconsistentFollowerAxis	

7.6.15 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.

7.6.16 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 (*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);
```

(continues on next page)

(continued from previous page)

```
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.

CHAPTER 8

C3D Converter

The C3D Converter module performs conversion, reading, and writing of 3D models in various exchange formats and is responsible for data exchange with other systems. Its first key task is import: reading a file, an in-memory buffer, or a stream containing an external geometric model in one of the supported formats and constructing a C3D geometric model based on the imported data. Its second task is export: representing a C3D geometric model in one of the exchange formats and writing this representation to the specified file, memory buffer, or stream.

Along with information about the geometric shape of the modeled object, the converter can transfer a comprehensive set of data:

- Information about shape, which can be described using solids, surfaces, wireframes, and point sets.
- Information about model structure, where the model's components are identified so they can be reused multiple times.
- Information about products: identifiers, comments, data about authors, and similar meta-data.
- Attributes: visual properties and basic attributes.
- Annotation elements: dimensions, technical requirements, and designations.

The C3D geometric kernel supports the text formats ACIS, IGES, STEP, X_T, STL, VRML, OBJ, and SAT, and the binary formats X_B, STL, JT, 3MF, and SAB. The ACIS, IGES, STEP, X_T, SAT, X_B, and SAB formats transfer information about the geometric shape of the modeled object using boundary representation (B-Rep), while the STL, VRML, OBJ, JT, and 3MF formats use polygonal representation. The JT format can contain a model both in boundary and polygonal form.

The 3D model converter provides a flexible set of options: it can control automatic stitching of surfaces into shells, restrict transfer to objects of certain types, or enable special modes for analysis and debugging. For applications where file input/output may become a bottleneck, data exchange through memory buffers and multithreaded operation of STEP and Parasolid converters is available, which significantly improves performance.

For license activation of the module, call the EnableMathModules() function with the appropriate parameters.

8.1 Converter Operation Principles

The converter operation involves the use of interfaces that can be divided into two groups. The first group includes interfaces that have an implementation on the C3D side, the second group consists of those interfaces that must be implemented by the user. In addition, the C3D geometric kernel provides a standard implementation of all interfaces required for data exchange.

The interfaces correspond to the following concepts:

- Converter (interface IConvertor3D) — a special object that is used to transfer data. Implemented on the C3D side.
- Converter properties (interface IConvertorProperty3D) — serve to transfer exchange settings to the converter. The interface has a standard implementation.
- Document (interface ItModelDocument) — an object that corresponds to the transferred model as a whole. The interface has a standard implementation.
- Part (interface ItModelPart) and assembly (interface ItModelAssembly) — parts of the model that can be represented as parts or as assembly units.
- Instance (interface ItModelInstance) — an object responsible for positioning and/or reuse of a component in the model.

The generalizing concept of a part and an assembly is a **component**. It is assumed that a **component** can be a minimal unit to which a separate file that the end application works with can correspond. The use of components and instances allows forming a model tree in which components can be used multiple times.

Inside the converter, the internal model is transformed based on the format specifics. For example, a part may have a mirror version, and this corresponds to a left-handed coordinate system of the component, while in the STEP format left-handed coordinate systems are not provided. Also, in the STEP format visual properties cannot be assigned to a part instance, and duplicates are created for recolored parts. There are two main transformation criteria — the left-handed coordinate system and visual properties (in some cases it is acceptable to assign visual properties to an instance, for example color, and in some cases it is not).

The transformation scheme between data models is based on two entities:

- element types (*MbAssembly* or shape-containing *MbSolid*, *MbMesh*, *MbWireFrame*, *MbSpaceInstance*, *MbPlaneInstance*);
- attributes (primarily *MbProductInfo*).

8.2 How to Work with the Converter

Converters provide tools for exporting and importing MbModel, MbItem models and custom implementations of ItModelDocument documents.

A group of functions designed for data exchange through files is located in the *conv_model_exchange.h* file and declared in the c3d namespace:

The function

```
MbeConvResType c3d::ImportFromFile(       MbModel &               model,
                                  const path_string &             fileName,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

allows reading an exchange format file into a model.

The function

```
MbeConvResType c3d::ImportFromFile(       c3d::ItemSPtr &         item,
                                  const path_string &             filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

allows reading an exchange format file into an item.

The function

```
MbeConvResType c3d::ImportFromFile(       ItModelDocument &       mDoc,
                                  const path_string &             filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

allows reading an exchange format file into a model.

The function

```
MbeConvResType c3d::ExportIntoFile(       MbModel &               model,
                                  const path_string &             filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

allows writing a model to an exchange format file.

The function

```
MbeConvResType c3d::ExportIntoFile(       MbItem &                item,
                                  const path_string &             filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

allows writing an item to an exchange format file.

The function

```
MbeConvResType c3d::ExportIntoFile(       ItModelDocument &       mDoc,
                                  const path_string &             filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

allows writing a model document to an exchange format file.

The function arguments are:

- **model** – the exported or imported geometric model;
- **item** – the model item;
- **mDoc** – the model document;
- **fileName (filePath)** – the full path to the file;
- *prop* – the implementation of the converter properties interface;
- *indicator* – the progress indicator.

During import, the exchange format is determined based on the file extension.

A group of functions implementing the transfer of models to exchange formats through memory:

```
MbeConvResType c3d::ImportFromBuffer(       MbModel &              model,
                                      const C3DExchangeBuffer &    buffer,
                                            MbeModelExchangeFormat modelFormat,
                                            IConvertorProperty3D * prop = nullptr,
                                            IProgressIndicator *   indicator =␣
↪nullptr );
```

```
MbeConvResType c3d::ImportFromBuffer(       c3d::ItemSPtr &        item,
                                      const C3DExchangeBuffer &    buffer,
                                            MbeModelExchangeFormat modelFormat,
                                            IConvertorProperty3D * prop = nullptr,
                                            IProgressIndicator *   indicator =␣
↪nullptr );
```

```
MbeConvResType c3d::ImportFromBuffer(       ItModelDocument &      mDoc,
                                      const C3DExchangeBuffer &    buffer,
                                            MbeModelExchangeFormat modelFormat,
                                            IConvertorProperty3D * prop = nullptr,
                                            IProgressIndicator *   indicator =␣
↪nullptr );
```

```
MbeConvResType c3d::ExportIntoBuffer( MbModel &              model,
                                      MbeModelExchangeFormat modelFormat,
                                      C3DExchangeBuffer &    buffer,
                                      IConvertorProperty3D * prop = nullptr,
                                      IProgressIndicator *   indicator = nullptr );
```

```
MbeConvResType c3d::ExportIntoBuffer( MbItem &               item,
                                      MbeModelExchangeFormat modelFormat,
                                      C3DExchangeBuffer &    buffer,
                                      IConvertorProperty3D * prop = nullptr,
                                      IProgressIndicator *   indicator = nullptr );
```

```
MbeConvResType c3d::ExportIntoBuffer( ItModelDocument &      item,
                                      MbeModelExchangeFormat modelFormat,
                                      C3DExchangeBuffer &    buffer,
```

(continues on next page)

(continued from previous page)

```
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr );
```

The function arguments are:

- **model** – the exported or imported geometric model;
- **item** – the model item;
- **mDoc** – the model document;
- **modelFormat** – the exchange file format;
- **buffer** – the buffer;
- **data** – the buffer address in memory;
- *prop* – the converter settings;
- *indicator* – the progress indicator.

The ImportFromFile(), ImportFromBuffer() functions perform import of geometric models:

– STL, OBJ formats;

– SAT/SAB format up to version 22.0 inclusive;

– IGES format version 5.3;

– JT format versions 8.0 — 11.x;

– Parasolid format up to version 25.0 inclusive;

– STEP format with supported application protocols 203, 214;

– VRML format version 2.0.

The ExportIntoFile(), ExportInfoBuffer() functions perform export of geometric models:

– STL, OBJ formats;

– SAT/SAB format up to version 22.0 inclusive;

– IGES format version 5.3;

– JT format version 9.5;

– Parasolid format up to version 10.0 inclusive;

– STEP format with supported application protocols 203, 214;

– VRML format version 2.0.

The result of FullFilePath() of the *prop* argument is used to determine the exchange file format only if the value of **modelFormat** is equal to *mxf_autodetect*.

The responsibility for deleting the buffer that is created and filled by the ExportIntoBuffer() function rests with the user. To free memory, it is recommended to use the *delete*[] operator.

Another method of importing and exporting a model is to call the methods of the IConverter3D converter.

To start working with the converter, it is necessary to obtain its instance. The function IConverter3D * GetConvertor3D() defined in the global scope creates a converter that can perform import and export of models in any of the supported formats. The function has no parameters. If successful, the function returns a converter, otherwise it returns a null pointer.

The function is declared in the *conv_model_exchange.h* file.

After creating the converter, data exchange is performed by calling the converter's methods. Converter methods (deprecated, it is recommended to use ImportFromFile(), ImportFromBuffer(), ExportInfoFile(), ExportInfoBuffer()):

```
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 );
```

```
MbeConvResType OBJRead( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0)
```

perform import and export of models in SAT, IGES, JT, X_T, STEP, STL, VRML, OBJ (import only) formats respectively.

The input parameters of the methods are:

- **property** – the converter properties;
- **stream** – the stream for receiving or writing a model in exchange format;
- **document** – the document transmitting the geometric model and other information;
- *indicator* – the indicator of the reading or writing process progress;
- *qeuryStitch* – the surface stitching or units of measurement requester during import (deprecated parameter, its use is not recommended, to control stitching you should use the converter properties).

If successful, the methods return *cnv_Success* or *cnv_PartialSuccess*, otherwise the methods return the operation result code from the MbResultType enumeration.

Upon completion of work, the obtained converter instance must be deleted.

The converter inherits the interface of the IConverter3D class, which is declared in the *conv_model_exchange.h* file.

The following methods are defined in the IConverter3D class:

- Method

```
virtual void SetConfigurationSelector( SPtr<IConfigurationSelector>␣
↪configuration_selector )
```

  sets the handler for selecting configurations. If the handler is set and there is more than one configuration (execution) in the imported file, then during the reading process the installed handler will be called to select the required configuration. (see *Model Configuration Import*).

  The input parameter **configuration_selector** is a pointer to the handler being set.

- Method

```
virtual void SetMetadataTransferCallback( SPtr<IConverterMetadataReceiver>␣
↪metadataTransferCallback )
```

  sets the metadata transfer (see *Import of Construction Format Metadata*).

- Method

```
virtual MbeConvResType LoadForeignReader( const c3d::path_string &           ␣
↪pluginName,
                                          const c3d::optionNameValuePairs_t &␣
↪pluginSpecificSettings )
```

  loads a plugin for obtaining data to build a model. Used for explicitly specifying the plugin to be used for importing proprietary formats, located at the specified path. This is an alternative method to c3d::ImportFromFile() for using plugins, when the path to the plugins is not explicitly set, assuming that the plugins are located in the same directory as the kernel.

  Input parameters:

  - **pluginName** – the name of the plugin file;
  - **pluginSpecificSettings** – the plugin-specific settings (a description of the plugin-specific settings should be obtained from the component provider).

- Method

```
virtual MbeConvResType ReleaseForeignReader()
```

  disconnects the loaded plugin for obtaining data to build a model.

- Method

```
virtual MbeConvResType ImportForeign( const c3d::path_string &    path,
                                      ItModelDocument &            idoc,
                                      IConvertorProperty3D* prop = 0,
                                      IProgressIndicator*   indicator = 0 )
```

reads a file using the plugin specified in LoadForeignReader().

Input parameters:

- **path** – the path to the file to be read;
- **idoc** – the implementation of the document interface;
- **prop** – the implementation of the converter properties interface;
- **indicator** – the process progress indicator.

8.3 Exchange Format Features

The main features of exchange formats are shown in the table below:

Формат	C3D	3MF	OBJ	STL	VRML	JT	X_T/X_B	SAT	IGES	STEP
Основное назначение	Формат экосистемы C3D	3D-печать	Визуализация	3D-печать	Визуализация	Обмен данными в Parasolid CAD-PLM	Формат экосистемы Parasolid	Формат экосистемы ACIS	Обмен данными, хранение	Хранение данных, машиностроение
Единица длины	мм	Задаются в файле	-	-	-	На компонент	м	На весь файл	На весь файл	На компонент
Особенности физического устройства	Бинарный	Набор XML-файлов	-	Есть	Текстовый	Бинарный	bin+txt	bin+txt	Текстовый	Текстовый
Ссылки на другие файлы	Нет	Нет	Нет	Нет	Нет	Есть	Нет	Нет	Есть	Есть
Особенности сборочных моделей	-	-	-	Одна сетка на модель	Простр-ые трансформации	-	-	Развертывается в лин-й список	Разные сп. групп-ки объектов	только правые СК
Визуальные свойства	Есть	Есть	Есть	Нет	Есть	Есть	Есть	Есть	Есть	Есть
GD&T	Есть	Нет	Нет	Нет	Нет	Есть	Нет	Нет	Нет	Есть
PMI	Есть	Нет	Нет	Нет	Нет	Есть, тесселяция	Нет	Нет	Есть в 2D	Есть
Полигональное представление без топологии	Есть	Есть	Есть	Есть	Есть	Есть	Нет	Нет	Нет	Есть
Полигональное представление с топологией	Есть	Есть	-	Нет	Нет	Есть	Есть	Нет	Нет	Есть
Точки	Есть	Нет	Нет	Нет	Нет	Есть	Нет	Нет	Есть	Есть
Кривые	Есть	Нет	-	Нет	Нет	Есть	Есть	Есть	Есть	Есть
Поверхности без топологии	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Есть	Есть
Граничное представление формы	Есть	Нет	Нет	Нет	Нет	Есть	Есть	Есть	Есть	Есть

8.4 IConvertorProperty3D Converter Property

The IConvertorProperty3D converter property is one of the parameters of data exchange methods. Since the same IConvertorProperty3D interface is used to control the operation of converters for all formats, it contains both methods common to all formats and methods specific to a particular format. In addition, the set of data that controls the operation of export and import differs.

The methods that are called when exporting to any format are responsible for providing a number of information about the document as a whole, about the coordinate system relative to which the geometric model should be oriented, and also request whether the exchange format objects should be forcibly transformed so that their shape and relative position are pre-

served, but only right-handed coordinate systems and the scale factor of length units are used, by which all linear dimensions are multiplied during export:

```
bool GetPropertyString( MbeConverterStrings nString, std::string & propertyString );

MbPlacement3D GetOriginLocation();

bool ReplaceLocationsToRight() (for formats with left-handed coordinate systems);

double LengthUnitsFactor().
```

The method

```
void SetIoPermission( MbeIOPermiss nPermission, bool setF )
```

allows setting a filter for the transferred objects.

The method that is called when importing from any format and is used to request the scale factor of length units in the user model, by which all linear dimensions are multiplied:

```
double AppLengthUnitsFactor();
```

The method that is called when importing from the STEP format and is used to control the import of instance attributes:

```
virtual bool ImportInstanceAttributes() const { return true; }
```

The method that is called when importing from SAT, X_T, X_B, STEP and IGES formats, as well as when importing proprietary formats through plugins, provides a prohibition or permission for automatic stitching of individual surfaces into solids and specifies the stitching precision:

```
bool EnableAutoStitch( double& stitchPrecision );
```

When importing from IGES and SAT, a method is called that allows or prohibits the converter from accessing the requester:

```
bool CanShowMessages();
```

When exporting to STL, a method is called that determines whether the data will be provided in text or binary form:

```
bool IsFileAscii();
```

When exporting to STEP, a method is called that has a default implementation and is used to specify the text representation in annotation elements and the STEP application protocol that will be used:

```
eTextForm GetAnnotationTextRepresentation();
```

When exporting to IGES, a method is called that determines whether topology information will be exported:

```
bool IsOutOnlySurfaces();
```

When exporting to polygonal formats (STL and VRML), methods are called that have a default implementation and control the parameters of triangulation calculation:

```
MbStepData TesselationParameters();

bool DualSeams();

void DualSeams( bool dSeams );
```

When exporting to STEP and SAT, a method is called that has a default implementation and returns the format version. To specify the version, it is recommended to use predefined values.

```
long int GetFormatVersion();
```

When exporting to the ACIS format, the method that has a default implementation and controls the removal of seams from loops is called:

```
virtual bool ExportWithSeamRemoval() const { return false; }
```

The method

```
virtual bool ImportMeshesWithTopology()
```

imports polygonal objects as objects with topological information.

The method

```
virtual bool RepairInconsistentMesh( MbRepairInconsistentMeshParams & )
```

enables post-processing of meshes.

The C3D geometric kernel provides a standard implementation of the IConvertorProperty3D interface.

The ConvConvertorProperty3D class represents the standard implementation of the converter property. The property inherits the interface of the IConvertorProperty3D class.

In its publicly accessible fields, the ConvConvertorProperty3D class stores the following information:

- **fileName** – the path to the exchange format file (by default an empty string);
- **fileASCII** – the flag for export to a text file STL (by default true);
- **formatVersion** – the format version during export (by default EXPORT_DEFAULT);
- **exportIGESTopology** – the flag for exporting topology to IGES format (by default true);
- **ioPermission** – the object filter by types (provides permission to export and import objects of any type);
- **propertyStrings** – special information about the document (an empty container);
- **annotTextReprSTEP** – the text representation of annotation elements (by default exf_TextOnly);
- **originLocation** – the local coordinate system of the model;
- **enableAutostitch** – the flag for enabling auto-stitching when importing a model (by default true);
- **autostitchPrecision** – the precision of face stitching (by default 0.3);
- **tesseleationStepData** – the triangulation parameters;

- **showMessages** – the control of accessing the requester (by default false);
- **dualSeams** – the flag for preserving coincident seam points (for STL and VRML converter);
- **lengthUnitsFactor** – the scale factor of length units during import and export;
- **appUnitsFactor** – the scale factor of length units of the user application model;
- **logRecords** – the container of messages for the user during import/export.

The IConvertorProperty3D interface and the ConvConvertorProperty3D class are declared in the *conv_exchange_settings.h* file.

Character Encoding Transformation

The IC3DCharEncodingTransformer interface is responsible for character encoding transformation, the implementation of which is returned by the GetUserCharEncodingTransformer() method. The method has standard implementations:

- IC3DCharEncodingTransformerUTF8 - transformation according to utf-8 rules;
- IC3DCharEncodingTransformerWindows1251 – transformation corresponding to cp-1251 encoding (Windows Cyrillic);
- C3DCharEncodingTransformerLocale - transformation according to the specified localization rules;
- IC3DCharEncodingTransformerStep - transformation according to STEP notation.

If the transformation is not explicitly specified, then for the STEP format the IC3DCharEncodingTransformerStep transformation is used by default, and for other formats C3DCharEncodingTransformerLocale.

8.5 ItModelDocument Model Document

The ItModelDocument interface is one of the interfaces that allow passing data with a complex tree-like multicomponent structure, in which multiple reuse of components is allowed. The document provides two ways to work with a model:

- Direct use of the model constructed with C3D geometric kernel objects.
- A user implements the ItModelPart, ItModelAssembly, ItModelInstance interfaces and the methods that work with their implementations. The converter, in turn, calls interface methods of the returned objects.

The interface methods are grouped as follows:

a) Group of methods intended for working with the model directly:

1. Method for setting the root element of a model:

```
virtual void C3dModelDocument::SetContent( MbItem* content );
```

2. Method for obtaining the model content:

```
virtual MbItem* GetContent();
```

3. Method (deprecated)

```
virtual map_of_visual_items C3dModelDocument::GetAnnotationItems( eTextForm form );
```

returns annotation items corresponding to the geometric model elements.

Arguments:

eTextForm – text representation form.

4. Method (deprecated)

```
virtual void SetAnnotationItems( const map_of_visual_items& visItems );
```

allows setting dimensions.

b) Group of methods intended for constructing a model during import:

1. Method

```
virtual ModelAssemblyPtr C3dModelDocument::CreateAssembly(
    const std::ItemsSPtrVector& componentItems,
    const c3d::string_t&        fileName
);
```

allows creating a document with a new assembly during import.

Arguments:

fileName – assembly name;

componentItems – bodies added to the assembly.

2. Method

```
virtual ModelPartPtr C3dModelDocument::CreatePart(
    const c3d::ItemsSPtrVector& componentItems,
    const c3d::string_t&        fileName
);
```

allows creating a document with a new part on import and increases the reference counter of the resulting document by 1.

Arguments:

componentItems – bodies added to the part.

fileName – part name.

3. Method

```
virtual bool C3dModelDocument::FinishImport( IProgressIndicator * indicator );
```

allows completing import and saving the document. Returns *true* if the operation is successful, *false* otherwise.

Arguments:

indicator – an object for displaying the process progress.

c) A group of methods intended for obtaining related model components during export:

1. bool IsAssembly(); // whether the model is an assembly
2. bool IsEmpty(); // whether the model is empty (contains neither a part nor an assembly)
3. The method

```
virtual ModelAssemblyPtr C3dModelDocument::GetInstanceAssembly();
```

allows you to get the assembly associated with the model document.

4. The method

```
virtual ModelPartPtr C3dModelDocument::GetInstancePart();
```

allows you to get the part associated with the model document.

5. The method

```
bool C3dModelDocument::MakeAssembliesUnique();
```

converts all assemblies and instances into unique components.

Since it is not known in advance which implementation will be used for data transfer, on import both the SetContent() method and one of the CreateAssembly() or CreatePart() methods are called, depending on the import result. Accordingly, on export the GetInstanceAssembly() or GetInstancePart() methods are called first depending on the IsAssembly() result, and only if the converter fails to construct the model, the GetContent() method is called.

The C3D geometric kernel provides a standard implementation of the ItModelDocument interface: C3DModelDocument. During import, the model is constructed using standard implementations of the ItModelPart, ItModelAssembly, ItModelInstance interfaces. On export, you can either define a model using the SetContent method or using implementations of the ItModelPart, ItModelAssembly, ItModelInstance interfaces.

The ItModelDocument interface and the C3DModelDocument class are declared in the *conv_model_document.h* file.

8.6 IProgressIndicator Progress Indicator

The progress indicator (or simply indicator) provides feedback from the converter to the user. During data exchange, the converter calls various methods.

The method for setting the range of steps, step size, and displayed information about the process is

```
bool Initialize( size_t range, size_t delta, IStrData & massage ).
```

The method for reporting the completion of a certain number of steps is

```
bool Progress( size_t n ).
```

The method returns an indication of whether the operation should continue.

The method for obtaining information about whether the operation should be interrupted at the user's request is

```
bool IsCancel().
```

The method for informing the user that the operation has been completed is

```
void Success().
```

The method for informing the user about an emergency stop of the process is

```
void Stop().
```

The user can define a method for transmitting information about whether a forced stop of the process should be performed. As a standard method, the user is suggested to implement the method

```
void SetCancel( bool cancel );
```

The IProgressIndicator interface is declared in the *alg_indicator.h* file.

In the test application, the ProgressIndicatorImp class represents one of the possible implementations of the indicator.

The ProgressIndicatorImp class is declared and implemented in the *test_converter.cpp* file of the C3D geometric kernel test application.

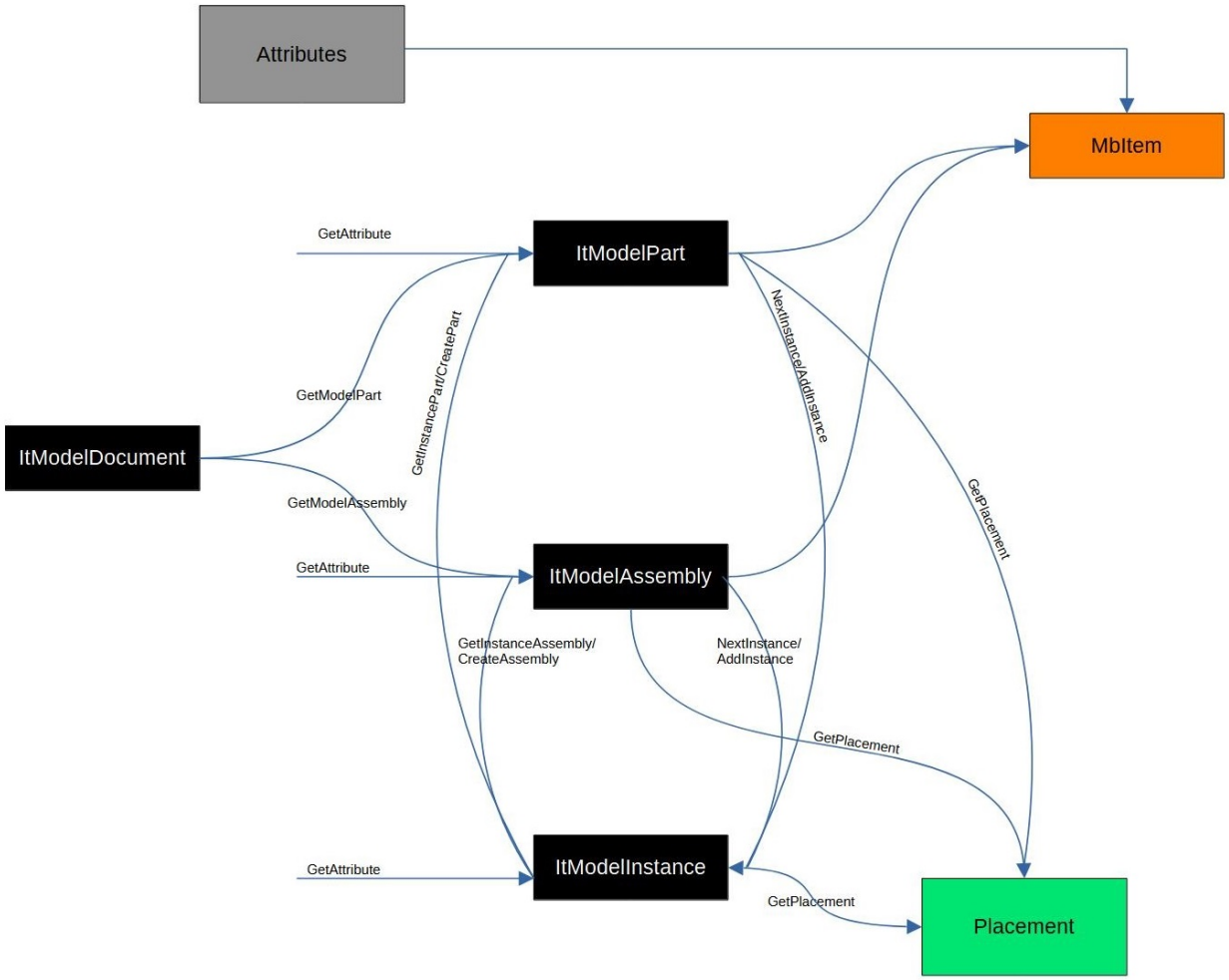
8.7 Model Tree Architecture

For a model created by using C3D geometric kernel objects, the following conventions are adopted:

- A document is equivalent to a root component (a top-level component).
- A component always corresponds to an object of type MbAssembly in the case of a boundary representation format. In the case of a polygonal representation format, an object of type MbMesh is allowed as a component. The component is expected to have assigned properties, which is expressed by the fact that the component has attributes of type MbProductInfo and MbPersonOrganizationInfo.
- Own objects of a component can be objects of type MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance (both with geometry objects inside and containing MbPMI annotation elements), and MbPlaneInstance for boundary representation formats, or MbMesh for polygonal formats, which reside in MbAssembly.
- Nesting of components and their positioning are performed by using objects of type MbInstance. MbInstance resides in a higher-level component and references a lower-level component. Cyclic dependencies are not allowed.

The approach in which a data exchange model is created by means of the C3D geometric kernel has several drawbacks. The most significant one is that the use of specific objects and a specific configuration makes it difficult to evolve the functionality, because any changes will require corresponding changes on the user side. The use of model, component, and instance interfaces implemented on the user side gives users independence from a specific implementation.

In terms of interfaces, the model tree architecture looks as follows:



- A document ItModelDocument() contains a top-level component.
- Components are of two types: parts ItModelPart and assemblies ItModelAssembly. The difference between them lies only in interpretation, and both types can reference lower-level components.
- Own objects of components can be objects of type MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance, and MbPlaneInstance.
- Nesting of components and their positioning are performed by using ItModelInstance objects. Cyclic dependencies are not allowed.

ItModelInstanceProperties is a base class, the ancestor of the ItModelPart, ItModelAssembly, ItModelInstance interfaces. The following methods are declared in the interface:

```
virtual bool SetAttributes( const c3d::AttrSPtrVector & /*attributes*/ );

virtual c3d::AttrSPtrVector GetAttributes();

virtual void GetRequirements( AnnotationSptrVector &, eTextForm ); (deprecated)

virtual void SetRequirements( const AnnotationSptrVector & ); (deprecated)
```

The ItModelPart and ItModelAssembly interfaces have the same set of methods, so they can be considered identical up to interpretation. The assignment of component properties, visual properties, and technical requirements is implemented through the methods of the ItModelInstanceProperties base class.

The PureFileName() method is called during export and returns a string whose meaning is the component file name.

Methods for working with own items:

```
void GetItems( std::vector< SPtr<MbItem> > & items,
               MbeGettingItemType            itemType,
               bool                          includeInvisible );

void AddItems( const std::vector< SPtr<MbItem> > & items );

GetItems();

AddItems();
```

The GetItems() method is intended to output items into a container using type and visibility filters, which are formed based on the values returned by the IConvertorProperty3D:: GetIoPermission() method.

The AddItems() method is used during import and is intended for adding items to a component.

Methods for working with instances:

```
SPtr<ItModelInstance> PrepareInstance();

SPtr<ItModelInstance> NextInstance( bool includeInvisible );

PrepareInstance();

NextInstance();
```

The PrepareInstance() method is intended for inserting a component during import.

The NextInstance() method is an iterator of instances during export. It is assumed that the iterator is ready to output instances from the beginning. The sign of completion is returning a null pointer. The visibility filter value during traversal corresponds to the result of calling IConvertorProperty3D:: GetIoPermission(iop_wInvisible).

The GetAnnotationItems(eTextForm **textForm**) method is deprecated.

Methods for import via plugins:

- The method

```
virtual void ItModelPart::AddMappedItem( MbItem &                    item,
                                         MbPlacement3D const &       placement,
                                         c3d::AttrSPtrVector const & attributes );
```

allows you to add an item with a given local coordinate system to the root of an assembly during import. The method is relevant only for the IFC format (see *IFC Import*).

Arguments:

item – an item added to the model.

placement – a local coordinate system.

- The method

```
virtual SPtr<MbInstance> ItModelPart::NextMappedItem() const;
```

returns an item with a given local coordinate system or nullptr. The method is relevant only for the IFC format (see *IFC Import*).

- The method

```
virtual void ItModelPart::AddAxesGrid( c3d::ItemSPtr const & );
```

allows you to add a grid of construction axes. The method is relevant only for the IFC format (see *IFC Import*).

- The method

```
virtual c3d::ItemSPtr ItModelPart::NextAxesGrid() const;
```

returns the next grid of construction axes or null in case of depletion. The method is relevant only for the IFC format (see *IFC Import*).

- The method

```
virtual c3d::ItemSPtr NextSpatialAxis() const { return c3d::ItemSPtr{}; }
```

returns the next spatial axis or null in case of depletion.

- The method

```
virtual void AddSpatialAxis( c3d::ItemSPtr const & );
```

allows you to add the next spatial axis.

- The method

```
virtual c3d::ItemSPtr ItModelPart::NextBoundaryItem();
```

returns the next item that bounds an area or a volume. The method is relevant only for the IFC format (see *IFC Import*).

- The method

```
virtual SPtr<MbAssistingItem> ItModelPart::NextCoordinateSystem();
```

returns the next coordinate system or null in case of depletion. Enabling and disabling this method during import and export is possible by using the SetIoPermission(iop_rCS / iop_wCS) filtering.

Methods for working with a model document and assemblies:

```
virtual void ItModelPart::AddBoundaryItem( c3d::ItemSPtr const &);
```

adds an item that bounds an area or a volume.

```
virtual void ItModelPart::AddCoordinateSystem( SPtr<MbAssistingItem> const &);
```

allows you to add a coordinate system. Enabling and disabling this method during import and export is possible by using the SetIoPermission(iop_rCS / iop_wCS) filtering.

The ItModelPart and ItModelAssembly interfaces are declared in the *conv_model_document.h* file.

ItModelInstance Instances

The inheritance of the ItModelInstance instance interface from ItModelInstanceProperties has been established historically and is preserved for compatibility. Within the framework of the established approach, only the assignment of custom visual properties to an instance is correct.

The void * GetId() method allows distinguishing instances of equivalent components. If different instances return the same value, the equivalence is determined by the results of the content analysis. During the export process, the converter seeks to optimize the model by reusing equivalent components. A user can provide information about which components are definitely not equivalent to reduce computational costs during export.

During export, the following methods are used to obtain such information about an instance as the local coordinate system of the instance, the content of the instance, and the absence of instance content:

```
bool GetPlacement( MbPlacement3D & place );

bool IsAssembly();

bool IsEmpty();
```

If an instance is not empty, then depending on the result of the IsAssembly() method, one of the following methods is called:

```
SPtr<ItModelAssembly> GetInstanceAssembly();

SPtr<ItModelPart> GetInstancePart();
```

These methods return the interface of the component referenced by the instance.

The following methods are intended for creating components during import:

```
SPtr<ItModelAssembly> CreateAssembly(const MbPlacement3D &              place,
                                     const std::vector<SPtr<MbItem> > &␣
↪componentItems,
                                     const c3d::path_string &           fileName );
```

```
SPtr<ItModelPart> CreatePart(      MbPlacement3D &              place,
                             const std::vector<SPtr<MbItem> > & componentItems,
                             const c3d::path_string &           fileName );
```

The result of these methods is a component interface whose placement in the top-level component is described by the local coordinate system **place**, the component is filled with elements from the **componentItems** container, and it corresponds to the file name **fileName**.

The following methods are used to place previously created components in an instance:

```
bool SetAssembly( const MbPlacement3D &   place,
                  const ItModelAssembly * existing );
```

```
bool SetPart( const MbPlacement3D & place,
              const ItModelPart *   existing );
```

The placement of the **existing** component is defined by the local coordinate system **place**.

The ItModelInstance interface is declared in the *conv_model_document.h* file.

8.8 Product Attributes

The product data transferred through the converter includes:

- Information about a component: type (part or assembly unit), identifier, name, and comment.
- Visual properties, material properties, and elementary attributes.
- Information about an author and an organization.

This information is transferred by means of MbProductInfo and MbPersonOrganizationInfo attributes and any elementary attributes (MbElementaryAttribute) respectively.

MbProductInfo Component Information

The MbProductInfo class must be assigned to a MbItem model element that should be marked as a component, or to a model element of type MbInstance when the name, designation, or comment of the instance differs from the corresponding data of the used component.

The main constructor of the MbProductInfo class has the following signature:

```
MbProcuctInfo( const c3d::string_t & initId,
               const c3d::string_t & initName,
               const c3d::string_t & initDesc,
                     bool            isAssm );
```

The **initId** argument is used to pass the component identifier (designation), **initName** is the component name, **initDesc** is the comment, and **isAssm** is a flag indicating whether the component is an assembly unit.

The main way to obtain string data about a product is to call the method

```
void GetData( c3d::string_t & oId,
              c3d::string_t & oName,
              c3d::string_t & oDesc ) const;
```

The input parameters **oId**, **oName**, and **oDesc** are used to return the identifier, the name, and the comment respectively.

To obtain information on whether a component is an assembly unit, use the method

```
bool IsAssembly() const;
```

The MbProductInfo class also contains methods that take strings of type *std::string*. These methods perform conversions using the c3d::ToSTDstring() and c3d::ToC3Dstring() functions.

The MbProductInfo class contains the following constructor:

```
MbProductInfo(       bool           isAssm,
              const std::string & initId,
              const std::string & initName,
              const std::string & initDesc );
```

Information about a product is obtained using the method

```
void GetDataStd( std::string & oId,
                 std::string & oName,
                 std::string & oDesc ) const.
```

8.9 Model Configuration Import

When importing models in the c3d format, as well as models available through the SolidWorks, NX, Creo, Inventor, and Solid Edge plugins, it is possible to read the model in different configurations (variants). Configuration import from Catia is not implemented, because configurations in Catia are stored in a parametric form that C3D Converter does not support.

There are two ways to specify a configuration selector:

- Pass a pointer to IConfigurationSelector to the LoadForeignReader() method when loading a plugin for subsequent use in import methods that explicitly specify the plugin.
- Use the IConverter3D::SetConfigurationSelector() method for subsequent use in import methods that work through an automatically loaded plugin.

To work with these methods, it is necessary to implement the IConfigurationSelector interface (*conv_requestor.h*).

IConfigurationSelector interface methods:

```
// Add a configuration for selection by the user.
virtual void AddConfiguration( const c3d::string_t & configurationName ) = 0;

// Add a configuration without a model for selection by the user.
virtual void AddEmptyConfiguration( const c3d::string_t & configurationName )
{
    AddConfiguration( configurationName );
}

// Add a configuration that needs to be updated for selection by the user.
virtual void AddNeedsUpdateConfiguration( const c3d::string_t & configurationName )
{
    AddConfiguration( configurationName );
}

// Specify the index of the active configuration.
virtual void SetActiveConfiguration( const size_t index ) = 0;

// Get the index of the configuration selected by the user.
virtual size_t GetConfiguration() const = 0;

// Index of the configuration that should be returned to cancel reading on user␣
↪request.
static const size_t configurationIdToCancel = SYS_MAX_T;
```

- If GetConfiguration() is not explicitly used, the default configuration specified via SetActiveConfiguration() will be loaded.
- The AddEmptyConfiguration() and AddNeedsUpdateConfiguration() methods are primarily informational, because unlike AddConfiguration() they do not allow obtaining a full model due to limitations of saving the model in the source CAD system.

8.10 C3D Converter Plugins

Plugins make it possible to read models from formats that are not directly supported by the C3D kernel, primarily proprietary formats. Technically, a plugin is a library that implements a given API and is explicitly loaded by the kernel on request from an application.

The plugin is a part of C3D Toolkit and is tightly integrated with the geometric kernel. In particular, this is manifested in the fact that the kernel automatically loads the plugin and activates it with a single license key if:

1. The license key includes the required format ([nx], [sw], [inv], [cat], [se], [cre], [ifc]),
2. The plugin binary files are located in the same directory as *c3d.dll*.

Some specifics of working with plugins:

- Plugins can be updated independently of the C3D geometric kernel.
- As a simplified workflow, the native plugins can be used through the ImportFromFile() method (provided that there is a license and the plugin binary files are located in the same directory as *c3d.dll*).
- If it is necessary to pass any settings to the conversion manager instance returned GetConverter3D(), then the plugin usage scheme is two-stage: GetConverter3D() + ImportFromFile() (if specific callback objects are required for configuration selection and similar tasks).
- If it is necessary to pass additional settings, such as working with model configurations or metadata, it is required to use the GetConverter3D() function to set specific callback objects, and then call ImportFromFile().
- Geometric models cannot be read through plugins by using the ImportFromBuffer() method.
- To use third-party plugins, or to use C3D plugins located in a directory different from the one containing *c3d.dll*, the following call sequence is used: GetConverter3D() - LoadForeignReader() - ImportForeign().

List of supported C3D plugin versions:

For the nwd, rvt, rvm, and sp formats there are partner plugins compatible with *c3d_plugin*; working with them is similar to working with the IFC format (see *Auxiliary Converter Methods*).

8.11 IFC Import

When importing IFC, it is possible to obtain IFC-specific data types:

- named planar axes (NextAxesGrid());
- items that bound a volume (NextBoundaryItem());
- reusable items (NextMappedItem(), see ItModelPart and ItModelAssembly Interfaces).

To obtain this data, the model must be imported into ItModelDocument. In the case of import into MbItem or MbModel, the domain specific is not taken into account, and the listed types are converted to ordinary geometric objects.

Using reusable items is similar to using regular instances in an assembly, with the difference that in the assembly structure each inserted item is a separate logically complete part with an MbProductInfo attribute, whereas a reusable item is only a geometric block (for example, a fence fragment that forms a complete fence when instantiated multiple times).

Items that bound a volume are essentially not standalone geometric objects. They are a set of geometric data that defines the boundaries of a volume, for example a room or a terrace. Such items can be filtered during import by using SetIoPermission(iop_rBounds).

Construction axis groups are planar objects that represent axes and have attributes, including axis names.

When importing IFC, as well as other construction formats (nwd, rvt, rvm, sp, etc.), it is possible to import metadata (see *Import of Construction Format Metadata*).

8.12 Import of Construction Format Metadata

Construction format metadata refers to heterogeneous data that cannot be represented in terms of the geometric kernel due to excessive complexity and the domain-specific nature, and therefore is processed separately. This includes complexly structured object properties, their relationships, and model representation trees. Metadata import is available for the *ifc* format from *c3d_plugin*, as well as for the *nwd*, *rvt*, *rvm*, and *sp* formats, which are also processed via *c3d_plugin*.

8.13 Auxiliary Converter Methods

A number of auxiliary methods are provided in C3D Converter to obtain information about supported formats and loaded plugins.

Getting the list of supported formats

The list of supported formats can be obtained either through the IConverter3D interface or by using a separate API function. The list includes both exchange formats available with the standard [cnv] license and formats provided by plugins.

The following IConverter3D methods are intended for this purpose:

```
virtual std::vector<c3d::FormatInfo> GetSupportedImportFormats() const;
```

```
virtual std::vector<c3d::FormatInfo> GetSupportedImportForeignFormats() const;
```

The GetSupportedImportFormats() method returns a list of supported formats, taking into account the license key and the presence of format-specific converter libraries. If some plugin does not support format listing, then its formats are not included in the result.

The GetSupportedImportForeignFormats() method returns a list of formats supported for import via ImportForeign(). The list will be empty if LoadForeignReader() has not been called or has failed. If some plugin does not support format listing, then its formats are not included in the result.

A separate API function is also provided to obtain the list of supported formats:

```
CONV_FUNC( std::vector<FormatInfo> ) GetSupportedFormats();
```

The GetSupportedFormats() function returns a list of supported formats, taking into account the license key and the presence of format-specific converter libraries. If some plugin does not support format listing, then its formats are not included in the result.

The FormatInfo structure contains brief information about a format: its name and the list of supported file extensions.

Getting information about loaded plugins

To obtain information about loaded and activated plugins, the following function is used:

```
CONV_FUNC( std::vector<PluginVersionInfo> ) GetLoadedPluginsVersions();
```

The GetLoadedPluginsVersions() function returns versions of loaded plugins. The version value is defined by the plugin implementation. If a plugin does not report its version, the string “n/a” is used instead.

The PluginVersionInfo structure contains the full path to the plugin and the plugin version.

Plugin versions differ for different formats. For mechanical formats (*catia*, *creo*, *nx*, *sw*, *se*) and *ifc*, a single plugin version is used, for example 22.0.29.0. Partner plugins for construction formats (*nwd*, *rvt*, *rvm*, *sp*) have their own version numbering schemes, for example 1.4.0.0, since they have been developed independently.

8.14 Creating a Solid from User Objects

The ImportSolid() function is intended for creating a solid from data stored in a custom object container ObContainer. The function constructs the imported solid solid based on a set of ob-structures describing solid geometry and topology and returns an operation completion code of type MbeConvResType. The function signature is as follows:

```
CONV_FUNC( MbeConvResType ) ImportSolid( c3d::SolidSPtr &                  solid,
                                         std::shared_ptr<ObContainer const>␣
↪obContainer,
                                         IConvertorProperty3D *            prop,
                                         IProgressIndicator *              indicator␣
↪= nullptr );
```

Function parameters:

solid – the imported solid body created from the data stored in the container;

obContainer – the custom object container ObContainer;

prop – a pointer to the converter properties interface implementation IConvertorProperty3D;

indicator – a pointer to the process progress indicator IProgressIndicator (can be *nullptr* if progress tracking is not required).

The ObContainer container is filled with user data represented by a set of ob-structures describing solid geometry and topology. After it is filled, the container is passed to the ImportSolid() function, which creates a solid model from the source data. The function fixes some typical errors specific to import from various formats and also performs stitching and merging of similar faces; the behavior is configured in the standard way through IConvertorProperty3D.

If questions or errors arise on the user side when using this functionality, the source data can be transferred to the technical support team as a debug text file generated by the ObContainer::WriteIntoFile() method.

CHAPTER 9

Reference Materials

9.1 Operation Result Codes

9.1.1 MbResultType Enumeration

rt_Success = 0 – Normal completion.

Curve contour error types

rt_Empty – Empty result.

rt_ToManyAxis – Too many axes.

rt_ToFewAxis – Not enough axes.

rt_ToManyContours – Too many contours.

rt_Stars – A “star” configuration is detected.

rt_SelfIntersection – Contour self-intersection.

rt_SelfIntWhenExtended – Self-intersection in the contour extension.

rt_Intersection – Contour intersection.

rt_NoIntersectSolid – Generating contour does not intersect solids.

rt_NoIntersectSection – Generating contour does not intersect sections (for the solid-by-sections operation).

rt_LibNotFound – Fragment not found in the library.

rt_MustBeClosed – Must be closed.

rt_MustBeOpen – Must be open.

rt_AxisIntersection – Intersection with the axis.

rt_DegenerateAxis – Degenerate axis.

Solid error types

rt_MultiSolid – The solid consists of separate parts.

rt_CurveError – Invalid curve.

rt_ContourError – Invalid contour.

rt_SurfaceError – Invalid surface.

rt_SolidError – Invalid solid.

rt_ParameterError – Invalid parameter.

rt_ThicknessError – Incorrect thickness value.

rt_NoSequenceCurveAndSections – Nonsequential arrangement of sections along the curve (for the solid-by-sections operation).

rt_SelfIntersect – Object self-intersection.

rt_NoIntersect – Objects do not intersect.

rt_OffsetIntersectError – It is impossible to construct an offset with the given parameters.

rt_BooleanError – Boolean operation error.

rt_NoEdges – Edges not found.

rt_PrepareError – Error while preparing the operation.

rt_ChamferError – Error while creating an edge chamfer.

rt_FilletError – Error while creating an edge fillet.

rt_PartlyChamfer – Chamfers were not created on all edges.

rt_PartlyFillet – Not all edges were filleted.

rt_ChamferSurfaceError – Error while creating the chamfer surface.

rt_FilletSurfaceError – Error while creating the fillet surface.

rt_TooLargeChamfer – Chamfer legs are too large.

rt_TooLargeFillet – Fillet radius is too large.

rt_SemiChamfer – Chamfers were not created for all edges.

rt_SemiFillet – Not all edges were filleted.

rt_CuttingError – Error while cutting by a surface.

rt_ThinError – Error while creating a thin-walled solid.

rt_OffsetError – Wall thickness is too large when creating a thin-walled solid.

rt_FaceError – Invalid face.

rt_RibError – Unknown error while placing a stiffening rib.

rt_DraftError – Unknown draft error on solid faces.

rt_NoObjectForDirection – No face is present in the selected direction.

rt_AbsorptionSolid – A local solid absorbs the result.

rt_Error – Unknown error.

rt_None – No messages.

rt_Intersect – Objects intersect.

rt_InvalidType – Improper curve type.

rt_NoConvertTextToNurbs – Text cannot be converted to NURBS.

rt_SplitWireNotSplitFace – The contour does not split any face or coincides with a face edge.

rt_SplitWireNotIntersectFace – The contour does not intersect the selected set of faces or coincides with a face edge.

rt_MustBeOnlyOnePoint – For this operation, the sketch must contain only one point.

rt_InvalidPoleUsage – A sketch with one point can be used only for an end section.

rt_ThinWithPole – Thin-wall construction is impossible if one of the sections is a point.

rt_TopologyError – Invalid topology.

rt_NoObjectInDirNormal – No object in the forward direction.

rt_NoObjectInDirReverse – No object in the reverse direction.

rt_SolidAffectedByBoolean – Solids have been modified by a Boolean operation.

rt_HaveDegenerateSegment – The curve contains zero-length segments.

rt_NotAllSourcesFound – One or more source operations were not found.

rt_InvalidEmptyContour – The contour consists of two segments lying one over another.

rt_UnnecessaryVariables – Redundant number of variables.

rt_DomainMismatch – Domain does not match the specified values.

rt_UnknownTranslatorError – Unknown error in the translator.

rt_UnknownParserError – Unknown error in the parser.

rt_UnknownSymbol – Unknown symbol in the string.

rt_NoClosingBracket – Closing bracket is missing.

rt_NoOpeningBracket – Opening bracket is missing.

rt_ImpossibleOperation – Impossible operation.

rt_LostObject – Operation lost its reference objects.

rt_NotAllBendsProcessed – Not all bends were folded or unfolded.

rt_ValueScalingError – Scaling with the specified factor is impossible.

rt_RatioScalingError – Scaling with the specified ratio of factors is impossible.

rt_MultiSolidDeflected – This operation is not applicable to multipart solids.

rt_MultiSolidDefused – The result of this operation cannot be a multipart solid.

rt_TransitionError – Boundary condition of blending is no longer satisfied.

rt_MeshCrossingError – Curve intersection points do not form a regular grid.

rt_ClosedError – Closure cannot be performed.

rt_TooGreatCurve – The curve used for blending is longer than the surface itself.

rt_ClosedOrUnClosedError – All curves must be either closed or open.

rt_SketchNoSheet – The sketch must lie only on an outer or inner planar face of a sheet solid.

rt_BadSketch – The sketch does not meet the operation requirements.

rt_ConnectionError – Object connectivity is broken.

rt_MeshSmoothError – At intersection points, curves from opposite families are tangent.

rt_NoIntersectContour – No intersection contours.

rt_ErShMtHeight – Total height must be greater than sheet thickness.

rt_DegenerateSurface – Surface is degenerate.

rt_SurfaceEdgesIntersect – Surface cannot be created: internal partition edges order is violated.

rt_TooLargeExtension – Extension length is too large.

rt_TooSmallExtension – Extension length is too small.

rt_VertexExtensionError – Failed to extend the surface to the given vertex.

rt_SurfaceExtensionError – Failed to extend the surface to the given surface.

rt_NoEdgesConection – Edges do not form a connected chain.

rt_TooManyPoints – Too many points. Reduce their number.

rt_TooManyPoints_1 – Too many points.

rt_DirectionExtensionError – Failed to extend the surface in the given direction.

rt_ExtensionPoleError – The face contains a pole: select another edge or another extension type.

rt_MustBeOpenOrClosed – All contours must be either closed or open.

rt_TooComplicatedItemsSet – Too complex set of items for processing.

rt_NoAxesIntersection – Axes do not intersect.

rt_TooFarItems – Objects are too far apart.

rt_ProcessIsStopped – Process was stopped.

rt_ContourSweptError – The contour cannot be used for the specified motion.

rt_SomeContourError – One of the contours cannot be used for the specified construction.

rt_SplitWireNotAllFaces – Split lines (intersection edges) are not created on all selected faces or do not reach face boundaries.

rt_GeneratrixColinearGuide – In some points, the generatrix is parallel to the guide.

rt_NotEnoughMemory – Not enough memory.

rt_BorderColinearCurve – Side boundary direction is parallel to the tangent at the end of the generating curve.

rt_ObjectNotFound – Object not found.

rt_PoleBrokenError – In the spline surface, some pole points were moved and some remained coincident.

rt_ApproxError – Approximation failed.

rt_AccuracyError – Construction accuracy requirements are not satisfied.

rt_BadVariable – Invalid variable value.

rt_BadEdgesForChamfer – Impossible to create a chamfer on the specified edges.

rt_RevokeStopFillet – Stopping the fillet is impossible; fillet was created without stopping.

rt_AdjacentTransitionError – Blends on adjacent boundaries are inconsistent.

rt_ObjectAccessDenied – Object access denied.

rt_TooManySegments – Number of segments is too large.

rt_GapShiftError – Invalid gap position of the shell.

rt_CutBySilhouetteError – Silhouette edge line does not split the face.

rt_DegeneratedProjection – Degenerate projection of the reference object.

rt_NotAllContoursUsed – Not all contours (curves) are used.

rt_ChangedParameter – Operation parameter has been changed.

rt_NotAllObjectsUsed – Not all objects are used.

rt_ZeroJumperError – Error due to forming a zero-thickness jumper.

rt_FullFilletError – Error while creating a face fillet.

Smooth spline construction errors

rt_IncorrectData – Invalid data.

rt_IncorrectPolylines – Invalid polyline shapes or tangent directions.

rt_IncorrectStructure – Invalid structure of a polyline with straight segments.

rt_TooFewPoints – Too few points.

rt_CoincidentPoints – Coincident points.

rt_TooSharpAngle – Angle between polyline segments is too sharp.

rt_ReverseMotion – Reverse motion of a segment.

rt_SharpTorsion – Sharp torsion of the polyline.

rt_IncorrectFirstDirection – Invalid tangent direction at the start point.

rt_IncorrectLastDirection – Invalid tangent direction at the end point.

rt_FirstTangentVector – First tangent is assigned to a straight segment.

rt_LastTangentVector – Last tangent is assigned to a straight segment.

rt_TangentVectorsUnsuitable – Tangent vectors are invalid for a locally convex region.

rt_IncorrectBezier – Invalid structure of the source Bézier curve.

rt_NoIntersection – Tangents do not intersect; invalid polyline structure.

rt_CriticalConfiguration – Critical configuration for B-spline approximation.

rt_IncorrectInitialData – Invalid initial data.

rt_NotImplemented – This command is not implemented.

rt_InsufficientMemory – Insufficient memory.

rt_IncorrectFirstTangent – Invalid first tangent direction.

rt_IncorrectLastTangent – Invalid last tangent direction.

rt_StraightenLastSite – Final straight segment; tangent vector is not required.

rt_IncorrectPolylineStructure – Invalid structure of a polyline with straight segments.

rt_IncorrectDataStructure – Invalid data structure in the model.

rt_ObjectIsNotPolyline – Object is not a spatial polyline.

rt_MissingKnots – Knot vector is missing.

rt_ClosedSpline – Closed spline in an external file.

rt_ObjectNotSelected – Required object is not selected.

rt_SharpChangeCurvature – Sharp curvature change at an inflection region.

rt_CloseToStraightness – Close to straightness.

rt_TooFewStartPoints – Too few points on the convex part of the initial region.

rt_BanClosedConfiguration – Closed configuration is forbidden when the number of points is less than 5.

rt_IncorrectDeterminant – Invalid structure of the source geometric determinant.

rt_DimensionsPointsArrays – Point array dimensions.

rt_TangentsNotIntersect – Tangents do not intersect.

rt_PointsCoincided – Points coincide.

rt_FewPointsForClosed – Too few points for a closed polyline.

rt_TooFewPointsOnSite – Too few points on a convex region.

rt_PointsOnLine – Points lie on a straight line.

rt_TooManyApexes – Too many points.

rt_SiteNotConvex – Three-segment region is not convex.

rt_CurvatureRange – Curvature is outside the allowed range.

rt_ObjectNotHermiteGD – Object is not a Hermite G^1 object.

rt_ObjectNotNURBS – Object is not a NURBS.

rt_DerivativeContinuityErr – Failure to ensure derivative continuity.

rt_NotEnoughBuildData – Not enough data for construction.

rt_InternalError – Unknown error.

rt_DerivativeGap = 198 – No continuity of curve derivative with respect to parameter.

rt_DerivativeGapU – No continuity of surface derivative with respect to the first parameter.

rt_DerivativeGapV – No continuity of surface derivative with respect to the second parameter.

rt_ContourGapError – The contour has gaps between segments.

rt_ContourSegmentsOverlapError – The contour has overlapping segments.

rt_ContourSegmentsNoTangentJoint – The contour has segments with non-smooth junction.

rt_ExtensionOutOfRange – Extension cannot be limited by the given surface.

rt_BuildSheetBySolidError – Error while constructing a sheet solid from an arbitrary solid.

rt_BendEdgeError – Edge cannot be used as a bend edge.

rt_StampToolPositionError – Invalid tool position relative to the sheet solid.

Variable section surface construction errors

rt_NoEdgesForBuild – No edges for construction.

rt_NoCurvesForBuild – No curves for construction.

rt_NotCorrectDataForBuild – Invalid data.

rt_NotEnoughDataForBuild – Not enough data for construction.

rt_ReferenceCurveError – Invalid reference curve.

rt_GuideCurveError – Invalid guide curve.

rt_ApexCurveError – Invalid apex curve.

rt_DiscriminantFunctionError – Invalid discriminant function.

rt_DiscriminantCurveError – Invalid discriminant curve.

rt_DiscriminantSurfaceError – Invalid discriminant surface.

rt_AngleFunctionError – Invalid angle function.

rt_ReferenceCurveBuildFailed – Failed to construct the reference curve.

rt_GuideCurveBuildFailed – Failed to construct the guide curve.

rt_ApexCurveBuildFailed – Failed to construct the apex curve.

rt_VertexCurveBuildFailed – Failed to construct the control curve.

rt_ApexDiscriminantBuildFailed – Failed to construct the apex curve and the discriminant.

rt_DiscriminantBuildFailed – Failed to construct the discriminant function.

rt_PatternCurveError – Invalid generating curve.

rt_SynchronizeParameterError – Failed to synchronize parameters.

rt_ReparamBuildFailed – Failed to perform reparameterization.

rt_TriangleSectionFailed – Section triangle of the surface cannot be determined.

rt_DiscriminantFunctionWrong – Discriminant function used to define the section shape does not allow the surface to be constructed.

rt_DiscriminantCurveWrong – Curve used to define the section shape does not allow the surface to be constructed.

rt_DiscriminantSurfaceWrong – Tangent surface used to define the section shape does not allow the surface to be constructed.

rt_SectionMovementWrong – Section motion is incompatible with the guides.

rt_DiscriminantSurfaceFar – Discriminant surface is far from the section.

rt_CrossOrderError – Order of mutual curve intersections is violated.

rt_WrongExtensionLength – Invalid extension length.

rt_WrongExtensionWayValue – Unknown extension method.

rt_StampToolHoleError – Error: stamping area must not contain cutouts.

rt_CurveClosedAtStart – Curve is closed at the start.

rt_CurveClosedAtEnd – Curve is closed at the end.

rt_CurveClosedBothSides – Curve is closed on both sides.

rt_BeyondLimitsExtension – Extended surface curve goes beyond the surface bounds.

rt_NonBijectiveFunc – Mapping of function parameter values to curve parameter values is not bijective.

rt_LicenseError – License error: invalid or expired license.

rt_NoRayIntersection – No ray–second object intersection in the selected direction.

rt_ExtremesRequestExceeded – Number of extrema exceeds the number of dimensions.

rt_ConstraintSurfaceDomainError – Invalid situation in the element domain with respect to the external constraint object (surface).

rt_ConstraintCurveEmptyDomain – Empty domain with respect to the external constraint object (curve).

rt_ConstraintCurveDomainError – Invalid situation in the element domain with respect to the external constraint object (curve).

rt_ConstraintCurveNoBijection – Violation of bijection between the trajectory and the external constraint object (curve).

rt_FacesGroupsOverlapError – Arrays of replacement and replaced faces intersect.

rt_StitchShellsError – Shell stitching error.

rt_NoFacesForModification – No faces for modification.

rt_NoReplacementFaces – No replacement faces.

rt_ErrorTotal – General operation error.

9.1.2 MbeStitchResType Enumeration

```
enum MbeStitchResType {
    stch_Success = 0,
    stch_PrepareError,
    stch_CoorientFaceError,
    stch_SomeEdgesUnstitched,
    stch_OutwardOrientError,
    stch_NoEdgeWasStitched,
    stch_SeparatePartsResult,
    stch_EdgeStitchError,
    stch_InputTopologyError
};
```

stch_Success = 0 – No errors.

stch_PrepareError – Error while preparing the stitching operation.

stch_CoorientFaceError – Failed to set consistent orientation of faces.

stch_SomeEdgesUnstitched – Some edges remain unstitched.

stch_OutwardOrientError – Failed to orient face normals outward from the solid.

stch_NoEdgeWasStitched – Not a single edge was stitched.

stch_SeparatePartsResult – After stitching, separate unconnected parts remain.

stch_EdgeStitchError – Edge stitching error.

stch_InputTopologyError – Critical topology errors in input shells.

9.2 Blend Surface Construction Methods (Fillet or Chamfer)

```
enum MbeSmoothForm
{
    st_Unknown     = -3, // Undefined blend
    st_Ellipse     = -2, // Fillet by an ellipse arc with radius recalculation
    st_Span        = -1, // Fillet with a given chord
    st_Fillet      =  0, // Fillet with given radii
    st_Chamfer     =  1, // Chamfer with given legs
    st_Slant1      =  2, // Chamfer by leg and angle (leg distance2 is computed for a␣
↪right angle between faces and defines the angle adjacent to leg distance1)
    st_Slant2      =  3, // Chamfer by angle and leg (leg distance1 is computed for a␣
↪right angle between faces and defines the angle adjacent to leg distance2)
    st_ByCurvature =  4  // Fillet with curvature continuity at the boundary with␣
↪supporting surfaces
};
```