LGS 3D Geometric Solver Overview

Version 6.0

Bricsys NV
Bellevue 5/201
9050 GENT
Belgium

phone: +32 (0)9 244 01 90
fax: +32 (0)9 244 01 99
web, support: http://www.bricsys.com

Copyright © 2001 - 2013 Menhirs NV
All rights reserved
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>2</td>
</tr>
<tr>
<td>1.1</td>
<td>Purpose of LGS 3D</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Products that benefit from the LGS 3D</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>LGS 3D strengths</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Using LGS 3D</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>APPLICATIONS</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Assembly design</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>3D sketching</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>3D direct modelling</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Motion simulations</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Data transfer applications</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>FUNCTIONALITY</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Geometric objects</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Constraints</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Rigid sets</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>User-defined curves, surfaces and swept surfaces</td>
<td>10</td>
</tr>
<tr>
<td>3.5</td>
<td>Orientations, alignments and measuring modes</td>
<td>10</td>
</tr>
<tr>
<td>3.6</td>
<td>Help points and help parameters</td>
<td>11</td>
</tr>
<tr>
<td>3.7</td>
<td>Variables and equations</td>
<td>11</td>
</tr>
<tr>
<td>3.8</td>
<td>Moving under constraints</td>
<td>11</td>
</tr>
<tr>
<td>3.9</td>
<td>Well-, under- and overdefined models</td>
<td>11</td>
</tr>
<tr>
<td>3.10</td>
<td>Tolerances</td>
<td>13</td>
</tr>
<tr>
<td>3.11</td>
<td>Interrupting calculations</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>LGS 3D INTERFACE</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Supported platforms</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Data exchange</td>
<td>14</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Callbacks interface</td>
<td>14</td>
</tr>
<tr>
<td>4.2.2</td>
<td>External application identifiers</td>
<td>15</td>
</tr>
<tr>
<td>4.3</td>
<td>API functions</td>
<td>15</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Context</td>
<td>15</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Objects specific functions</td>
<td>15</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Model manipulation functions</td>
<td>15</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Service functions</td>
<td>16</td>
</tr>
<tr>
<td>4.4</td>
<td>Sample example of API using</td>
<td>16</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

LGS 3D constraint solver is a state-of-the-art software engine that will help anyone who needs a strong variational (constraint-based) functionality in CAD, 3D modeling, engineering and computer graphics software. LGS 3D is a part of the Bricsys LGS products family, which contains both 2D and 3D variational solvers developed continuously since 2001. Numerous software development companies embedded LGS 3D in their commercial software products, including such noticeable examples as Ascon (Russia), Bricsys (Belgium), Cimatron (Israel), and JSOL Corporation (Japan). Successful commercial implementations of LGS 3D proved its ability to compete in the market of dimensional and geometric constraint management software components.

1.1 Purpose of LGS 3D

Geometric modeling is widely used in many software products, but mainly in CAD/CAM/CAE/PLM systems. Other domains of applications include computer games, geometric theorem proving, molecular modeling, publishing, etc. A Geometric Constraint Solver is a computation engine that supports creation and modification of geometric models by means of (explicit or implicit) constraints. It moves and rotates objects to positions where all constraints are satisfied and which are not far from the original positions.

Typical geometric objects are points, lines, circles, curves (both 2D and 3D), planes and arbitrary surfaces. Objects can be fixed in an absolute coordinate system or with respect to each other (so-called rigid sets of objects). Geometric constraints include logical constraints between geometric entities (like coincidence, parallelism, tangency, symmetry, etc.), dimensional constraints (that specify the required values for given distances, radii, angles), or engineering constraints (like equal distances and radii, other mathematical relations on dimensions).

Here are some typical problems solved by geometric constraint solvers:

- Find a configuration for a set of geometric objects which satisfies a given set of constraints over geometric elements, e.g. construct some detail of a mechanism from its parts,
- If such a configuration does not exist, provide both a partial solution (that satisfies only a subset of the given constraints) and information about over-constrained parts of the geometric model,
- Drag a geometric object along a given trajectory keeping all the constraints satisfied and animate this motion with respect to the constraints.

An example of a 3D variational model is shown in Fig 1.

There are several parts connected primarily by the concentricity constraint, but also there are some dimensional constraints preserving correct positioning of details to each other. When all these constraints are satisfied, we get a model of an arm manipulator built from the parts.

Fig. 1. Example of a 3D variational model
LGS 3D supports the most prevalent types of geometrical objects in 3D-space, such as points, lines, planes, circles, cylinders, spheres, parametric (or user-defined) curves and surfaces and also swept surfaces (defined by a parametric curve and a sweep direction). A wider set of geometrical objects can be easily modeled with the help of constraints: for example, in order to create a line segment, an application can use a line, 2 points and 2 coincidence constraints. To model more complex parts, a rigid set feature can be used. It allows one to define a set of objects as a rigid body. All objects included in a rigid set are moved and rotated altogether, while their relative positions within the body are preserved.

Moreover, unlike 2D sketches, 3D assembly models (the most popular application of 3D solvers) consist primarily of rigid sets – all objects, except for some auxiliary elements like symmetry axes, are in some rigid set. Rigid sets are details from which the whole detail or mechanism is built.

Almost all constraints restrict the rigid sets. However constraints are attached to specific objects inside a rigid set. The supported constraints are essential for such applications; they include measures – distances and angles – and boolean constraints – coincidence and tangency. The semantics of all constraints relates to the objects they constrain.

LGS 3D allows easy modification of already created models by changing the parameters of constraints or interactive “dragging” of objects.

### 1.2 Products that benefit from the LGS 3D

Many geometry-based applications can benefit from using LGS through enhanced functionality, for example:

- mechanical and architectural CAD,
- CAM,
- CAE,
- knowledge-based engineering,
- computer graphics,
- games, multimedia and computer animation,
- data exchange between all above applications.

Variational modeling is now a standard part of most high-end mechanical CAD applications. Through the scalable architecture of LGS 3D, variational/parametric techniques become available in a wide range of products, including desktop CAD and 3D modeling tools, architectural CAD, chemical and biological software and many others. But LGS, powered with strong computational algorithms, can benefit high-end CAD/CAM systems as well, providing a strong basis for variational modeling (assembly and part design), knowledge-based engineering, and constraint-based geometry.

In combination with other intellectual solutions by Bricsys, LGS 3D can support the following applications:

- modeling tools – by capturing both geometric and behavioral aspects of a model via the set of behavioral constraints;
- engineering systems – by adding not only variational power, but also generic constraint-based modeling and computational functionality;
- all-in-one industry solutions – by serving as a core for CAD/CAM and adding the features of lifecycle management, such as production chain optimization, scheduling, and workforce management;
- 3D modeling, virtual reality and computer animation can benefit from LGS 3D and Bricsys Physical Simulation Engine.

### 1.3 LGS 3D strengths

Variational functionality is requested in various software systems dealing with geometric objects. But it is hard to implement a variational solver without a solid constraint-solving background. Relying on the library of state-of-the-art computational methods, the LGS team has implemented some dedicated geometry-oriented methods and optimized the existing methods to use them for geometric models. This development, along with strong analytical methods incorporated into LGS, ensures that performance will be rather good for interactive applications.

LGS 3D is the right choice for those who develop assembly design, kinematic simulation and direct modeling applications, because it supports a wide range of 3D constraints and has an extension module for parametric modification of “dumb” geometry (which construction history might be lost in the translation from one CAD system to another).
Version 6.0 of LGS 3D introduced new functions aimed at supporting variational direct modeling applications. Users of such applications often move or rotate interactively one or more objects. LGS 3D is able to support this end-user operation by finding the transformations, which do not violate constraints imposed on the objects. LGS 3D Version 6.0 also supports so-called extended transformations, which may include the changes of parameters (radii). This extension allows to solve efficiently constraints on complex fillets. Version 6.0 also delivers significant performance improvement for move under constrains function.

The secret of efficiency of LGS 3D (which is able to solve problems with thousands constraints in less than a second) is mainly located in its sophisticated decomposition methods, which split a large initial problem into a set of smaller ones. Version 6.0 contains a lot of new decomposition patterns introduced by analysis of different industrial cases. There are also several important improvements in the non-linear equation solver. Version 6.0 also fixes several problems reported by customers, including some cases with unnatural solutions, which are now solved naturally.

The extendable architecture of LGS 3D allows the project team to build dedicated configurations for different application areas, where the power of the generic LGS 3D core is probably insufficient.

Simplicity of the LGS 3D API allows its easy integration into software of any kind. A minimal effort is then needed to build a powerful and intuitive parametric design, as well as modeling tools of exceptional computational power.

For more information on the underlying technologies and the functionality, refer to chapter 3 Functionality.

1.4 Using LGS 3D

LGS 3D can be integrated into the existing or newly developed software. LGS 3D has its own data representation and does not require any support from the data structures of the host system. All the geometric information is passed to LGS 3D through a simple, straightforward and compact interface.

LGS 3D is a core technology designed to empower the host application by providing advanced variational technology. Adoption of LGS 3D does not somehow affect application’s functionality, user interfaces and other aspects that make application unique. LGS 3D is just the base for some geometry- and constraint-related functions. The application authors have a complete control on what to use and what not to use from LGS 3D and how to use LGS 3D.

LGS 3D does not share any data structures with application. Any application-related data passed to LGS 3D is just associated with LGS 3D objects and are not used internally. No provisions are made about the application data contents. Application is completely responsible for interpretation and use of the results provided by LGS 3D. Having LGS 3D as a variational core enables application to focus on the higher level and end-user oriented features.
2 APPLICATIONS

In this chapter, different applications of LGS 3D are discussed. Its usual applications are essentially different from those of LGS 2D. If in the latter case the most common applications deal with drawing (sketching), in LGS 3D constraints are most often applied to entire parts (modeled in a 3D-modelling tool using primitives or wire-frame).

2.1 Assembly design

The most usual application of a 3-dimensional variational solver is assembling a whole mechanism from its parts. Namely, a user, with the help of tools for creating solid 3D models, designs each part of his product. Then he generates the assembly constraints that state how these parts relate to each other. The constraints can leave some degrees of freedom, which will be the degrees of freedom of the final mechanism. When all constraints are generated, LGS 3D is launched and the product is gathered from its parts with respect to all constraints applied. A simple assembly model is shown in Fig. 2. The solver strikingly simplifies the assembly modeling by saving a user from the manual stitching parts to each other. Instead a user just generates the constraints. And even more, the constraints once created will hold the information about interrelations between the parts. A user may disassemble the product, modify its individual parts (to some extent) or dimensions and then just launch the solver again in order to assemble the product back.

![Fig. 2. A simple assembly model](image)

The parts can be added incrementally, allowing the hierarchical creation of the model from smaller ones and so enabling collaboration within the design team. For example, one engineer can create a model of a car engine, another one - of the car body, and so on. Then all these models are assembled in the car model.

If some degrees of freedom are left unfixed, a user may try to move some part of the mechanism and LGS 3D, according to the constraints, will show only admissible motion of the parts.

2.2 3D sketching

By now most of the drawing tasks are accomplished in LGS 2D: parts are drawn from 2D primitives and then a 2D sketch is extruded or by some other transformation made 3-dimensional. But there is another possibility – to draw in 3D from the scratch using 3D primitives. This is the so-called 3D sketching. It has some obvious advantages:

- You don’t need to imagine what does your part look like when projected on a plane;
- Operations with different 3D primitives having the same projection are easier, e.g. a cone, cylinder and sphere have the same projections to the base plane of the cylinder/cone;
- 3D modeling is more straightforward.

This approach has its disadvantages as well, but it becomes more and more popular nowadays. Many developers of 3D modeling and CAD systems have already included 3D sketching tools in their systems or have plans to do it in the nearest future.

The tasks of the solver in 3D sketching are similar to the 2D case:
• Computing the overall precise model built from imprecisely positioned primitives and constraints among them. The solver should preserve configuration as close to the designed one as possible.
• Maintaining consistency of all constraints during the model modifications: movements, rotations and changing the constraint parameters.
• Reporting the inconsistency when constraints cannot be satisfied.

3D sketching is a challenge for the variational solver, because it generates more independent objects and much more constraints than assembly design. 3D sketching also demands for more types of constraints: while for the assembly you primarily need coincidence, tangency, etc., for sketching you will also need symmetry, equal distance, etc. Even for the types of objects you need a wider set here. Primitives like cones and tori are not needed for the assembly but may be needed primarily for sketching.

### 2.3 3D direct modelling

A huge problem of mainstream mechanical computer-aided design (MCAD) products presented on the market is rooted in the fact that they implement the same approach to 3D design: history-based parametric modelling. Engineers who used to design in 2D for years can hardly adopt this method, which main drawback consists in impossibility of direct manipulations with geometry. Users of history-based system operate parameters, which are used to generate the geometry. That’s totally different from 2D drawing environment, where the users can directly manipulate geometric entities (lines, arcs, polylines, splines) and their subentities by simply dragging them and directly changing their shapes.

Complicated design methodology is not the only drawback of history-based MCAD. Another important problem arises with the need to modify multi-CAD and legacy CAD data – imported from other CAD systems or from neutral CAD formats. Design history cannot be fully translated from one CAD format to another one just because each CAD system uses its unique set of 3D modelling functions – with different parameters and semantics.

These two problems (inability of direct manipulations and working with multi-CAD data) are addressed within a novel 3D design methodology called **direct modelling**. It allows users to directly edit the elements of the waterproof boundary of a 3D solid body – very similarly to the way of dragging 2D geometry. However, existing implementations of direct modelling have other drawbacks: they do not allow users to express their design intent and store it with geometry. Design intent of a CAD model can be thought as a set of rules that defined admissible modifications of its geometry. In history-based systems it is expressed by the design history, which is absent in direct modellers. So traditional history-based systems cannot be replaced with direct modelling ones, the latter can only supplement the former.

Variational direct modelling implemented on top of LGS 3D in combination with a solid modelling kernel (ACIS, Parasolid, Open CASCADE or similar) radically solves these problems, because it combines the best of two approaches – parametric history-based design and direct modelling. Here geometric and dimensional constraints are applied directly to the boundary elements of a solid model (faces, edges, vertices) and allow users to have the full parametric control on any model with no design history available.

This approach also allows users to change 3D solid geometry dynamically by moving, rotating and transforming its boundary elements. To take into account all transformations applied and to preserve the initial design intent, a variational solver like LGS 3D is absolutely needed.

Decomposition of the model into well-defined parts and inclusion of all the existing constraints ensure that only valid configurations will be allowed in dragging. The incremental recalculation along with the above-mentioned decomposition provides performance sufficient to support true interactivity when dragging and rotating.

Achieving such a performance and at the same time ensuring a correct behavior in any circumstances is impossible without a dedicated software, e.g. LGS 3D.

### 2.4 Motion simulations

LGS 3D can also benefit the mechanical modeling tools. Due to its computational efficiency, it allows easy computation of valid configurations for systems of rigid bodies, various consistency-preserving manipulations with such systems, and modeling their behavior by recomputing efficiently the configurations after a parameter change. This approach allows users to solve both forward and inverse kinematic problems in real time.

Fig.3 shows a door hinge, which has one internal degree of freedom – rotation around its axis. So if someone will try to drag one ear of the hinge, he will see not a movement but rotation around the axis.
2.5 Data transfer applications

In the variational design, solutions are derived only from the current model. No reference is made to the sequential design history. LGS 3D is based on the approach of simultaneous, non-sequential, variational solving.

In the history-based design, solutions are derived by replaying a record of the design history. Finding a solution depends on maintaining this record. LGS-technology is not history-based.

It is difficult to exchange history-based models within different CAD systems. The reason is that the system that receives a history-based model from a different application most probably cannot import and replay the design history. Therefore, further parametric modification is not possible; the imported model becomes a static non-parametric part. LGS 3D only uses simple information about dimensions and constraints that can be easily stored along with the current state of the model. Reproduction of this information in different CAD systems is simple, far simpler than the (non-LGS) problem of actual transfer of a geometric model.

Many CAD companies have an end-user (customer) database that contains a large quantity of archived older models. These models are often not constructed in a modern parametric system. When these customers migrate to a modern CAD product, it is desirable to enable the archived models to be dimensioned, if required, without being reconstructed in a new system. This process is difficult with history-based systems, since the necessary design history is rarely stored with the archived models. These models usually have to be constructed again in the new system. By contrast, because there is no requirement for a design history, LGS 3D does enable the post-parameterization of these models not requiring their re-construction.
3 FUNCTIONALITY

The main functionality of the LGS 3D is to solve geometrical model with constraints, i.e. to find positions of the geometrical objects (as well as values of the scalar variables), which satisfy all of the imposed constraints and equations. Along with this main function there are additional powerful functions including

- Moving under constraints functionality
- Diagnostics of degrees of freedom and contradictory constraints

In this chapter we describe which entities can be used to define the models. Additional functionality is also described.

3.1 Geometric objects

LGS 3D supports the following types of geometric objects:

- **Point** is specified by three coordinates.
- **Line** is specified by point and direction.
- **Plane** is specified by point and direction of its normal vector.
- **Circle** is specified by center point, radius and vector of normal direction of the circle plane.
- **Cylinder** is specified by point, direction of its axis and by radius.
- **Sphere** is specified by point and radius.
- **User-defined curve** is specified by callback functions returning curve points, derivatives in this points and parameter boundaries.
- **User-defined surface** is specified by callback functions returning surface points, derivatives in this points and boundaries of the parameters.
- **Swept surface** is defined by sweeping the user-defined curve in sweep direction.

3.2 Constraints

LGS 3D supports the following types of geometrical constraints, which can be imposed on supported geometrical objects:

- **Position fixation**: a constraint fixing current position of an object.
- **Coincidence**: a constraint linking two objects. Some examples of possible combinations are: two points coincide; a point lies on a line or circle.
- **Distance**: a parametric constraint fixing the distance between two objects. Some examples are: distance between two points, distance between two planes (which implies that they are parallel).
- **Angle**: a parametric constraint with two arguments that are objects with directions. It fixes the angle between the objects. Special cases are:
  - **Planar angle**: is specified by an axis around which the angle is measured.
  - **Parallelism**: directions of objects are parallel.
  - **Perpendicularity**: directions of objects are perpendicular.
- **Tangency**: a constraint stating that two objects touch each other at a boundary point.
- **Concentricity**: a constraint stating that centers (direction axes or central points) of two objects coincide or a point lies on an object’s axis. For example, line-cylinder concentricity means that the line and the cylinder axis become equal, as well as point-cylinder concentricity means that the point lies on the cylinder axis.

Along with geometrical entities LGS 3D supports variables. The following type of constraint can be imposed on such entities:

- **White-box equation**: a constraint used to link the variables associated with some constraint parameters with the help of some mathematical equations.

Sometimes there is more than one possible solution of a given geometry. For instance, the directions of two lines in a parallelism constraint could be either aligned or anti-aligned, or the distance to a circle,
cylinder or sphere could be measured to its center or its boundary. By using help points and help parameters as well as additional constraint attributes such as alignments, orientations and measuring modes an application can specify which solution is required in each specific case.

3.3 Rigid sets

LGS 3D supports rigid sets, i.e., the sets of objects that are rigidly attached to each other. Objects contained in the same rigid set cannot change their relative positions during computations; in other words, LGS 3D treats such sets as rigid bodies and can only move and rotate them. The use of rigid sets allows an application to specify the sets of objects in the model that are already positioned with respect to each other and should not be recalculated.

The rigid set is specified either by listing the objects included or by creating an empty set and its further extension by new objects. It is possible to add one rigid set to another in order to construct tree-like hierarchies of rigid sets that can represent complicated models containing many parts.

A rigid set can be treated as a typical geometric object and so it can be fixed with the help of fixation constraint.

3.4 User-defined curves, surfaces and swept surfaces

In addition to the predefined set of geometric objects described above, the application can define an arbitrary parametric curve, surface and a swept surface defined by sweeping usual parametric curve in sweep direction. The current version supports rigid curves, surfaces and swept surfaces depending on one (or two for user-defined surface) continuous parameter. Such curves and surfaces are controlled by a callback function returning the curve's or surface's points in the local coordinate system. This function is called many times during computations to obtain the points of the curve or surface and to compute the derivative.

The initial coordinate system of curve or surface is specified upon object creation. During computations only origin and direction of objects coordinate system can change, not the form of curve or surface.

3.5 Orientations, alignments and measuring modes

3D objects are quite complex entities and there are a lot of relations between them that an engineer needs to express. For example, there are several possible circle-cylinder relations: circle is tangent to cylinder and is outside cylinder; circle is tangent to cylinder and is inside cylinder; minimal distance between circle and cylinder is equal to, say, 10 units; the circle center is inside cylinder at the distance of 5 units from the cylinder, etc. Some of the relations are very similar and there is no need to invent different types of constraints in order to specify each relation. Therefore, relations are naturally grouped into classes, each class is represented by a single constraint, and constraint attributes are used for exact specification of particular relation to be held by LGS 3D. Three types of constraint attributes are supported by LGS 3D: orientations, alignments and measuring modes.

An orientation is a flag that specifies on which side of the object another object is located. This attribute can have “positive” and “negative” values and is associated not with a constraint itself but with its particular argument. For example, a user is able to specify positive orientation for a plane object in a tangency constraint between sphere and plane. It implies that the sphere have to be in the positive half-space of plane, which means - in the same half-space as the normal of the plane.

Another type of constraint attributes is alignment. Alignment is specified for a constraint itself and can have two values: “align” and “antialign”. This attribute is used for constraints on directed geometries (like lines or planes) that imply parallelism and set type of the parallelism – identical directions of vectors of geometries for the “align” value or opposite directions for the “antialign” value.

The “measuring mode” constraint attribute defines a part of a geometric entity that is directly involved into constraint specification. This attribute is associated with a particular argument of a constraint and can be assigned to “boundary”, “center” or “center point” values. For example, in case of concentricity between circle and point, a constraint may be imposed on the central circle axis (the “center” value of the measuring mode) or on the center circle point (the “center point” value). In practice, it is more efficient to use concentricity with additional attribute between these circle and point instead of modeling of the desired configuration by presenting a new point or line in the same rigid set as circle and implying a coincidence constraint.
3.6 Help points and help parameters

When defining the tangency constraint for a circle and plane, line, cylinder or other circle, an application can fix via help point the side of a circle where the tangency should take place – the nearest or the farthest side. When tangency constraint is imposed for a pair of circles, help points can be specified for each separately.

The help point is not a real geometrical object. It is associated with tangency constraint and a circle object, so the application is unable to specify geometrical constraints for help points directly. If no help points were created, the tangency can occur in any point of circle.

The application needs to provide only an approximate position of the help point. LGS 3D will choose the side for which help point is closer and will move it to its accurate position during the evaluation.

When defining the coincidence constraint for a point and a parametric curve or surface or for a point and a swept surface, an application can retrieve the value of the parameter at which the coincidence occurs, i.e., retrieve the help parameter. LGS 3D tries to calculate the initial value of the coincidence parameter using its own techniques, based on the current location of the objects.

3.7 Variables and equations

Along with geometrical entities LGS 3D supports variables, each variable represents a real number and can be associated with a parameter of a constraint.

Variables can be used as parameters of geometrical constraints (such as distance or angle) or as free variables playing auxiliary role. Algebraic relations can be imposed on variables, these relations are equations of type \( f(x_1, ..., x_n) = 0 \), where function \( f \) is given with the help of white-box equation constraint: an equation expressed via simple algebraic operations and elementary functions. During the calculations LGS 3D will solve the equations to find the values of the variables, and consequently the values of the constraint parameters.

Application is able to create fixed variables (constants) as well as unfixed variables that can be changed upon recalculating the model. A value of a fixed variable can be changed by an application only via call of the correspondent function.

3.8 Moving under constraints

LGS 3D allows moving groups of objects and rigid sets so that the specified constraints remain satisfied. This functionality provides easy modification of the underdefined models previously created.

To move a group of objects, the application must pass to LGS 3D the vector of a relative displacement of this group and the rotational matrix. LGS 3D moves the objects in accordance with the transformations specified, if this is possible. If given transformations conflict with some constraints, then resulting transformations of objects will be closest to given ones, and all constraints will be satisfied.

Performance of LGS 3D is usually sufficient to support dragging objects interactively, e.g., with the mouse. To implement this functionality in the end-user application, one should pass to the LGS 3D a small displacement of objects for every small movement of a mouse pointer.

3.9 Well-, under- and overdefined models

Part of geometrical model can be characterized as welldefined, underdefined and overdefined. This notions concern with degrees of freedom, which still remain in a part of a model. Geometrical objects add degrees of freedom to a model, while constraints remove them, for example, in 3D, point adds three degrees of freedom and simple distance constraint between points removes one degree of freedom.

The definition of a sufficient number of non-conflicting constraints for a set of objects makes it welldefined. The relative location of the objects in a welldefined set is fixed, but the entire set has six degrees of freedom that a rigid body has, i.e., all of its objects can be moved by the same vector or rotated by the same angle around a specified point without breaking the constraints. The position of objects in a welldefined part of the model is fully determined by the parameters of the constraints, and so such subsets can be predictably modified via changing constraint parameters.
If welldefined part of a geometrical model is fixed due to constraints, it implies that there are no any degrees of freedom left for all objects of the part. LGS 3D will indicate this fact by assigning correspondent welldefined statuses for all objects of the part. In other words, this status will arise for each object which has no degrees of freedom. This functionality allows end-user to easily control constraints definition process and to create sketches that are fully driven by parameters of constraints.

If a part of the model does not contain enough constraints, i.e., if this part has internal degrees of freedom and the relative position of its objects may be changed, then it is said to be underdefined.

For underdefined cases, LGS 3D tries to find a solution by analyzing the sketch configuration, i.e., without unnecessary rotation or movement of the geometric objects.

If some part of the model contains conflicting constraints, this part is said to be overdefined. An example of an overdefined part can be any welldefined part with an additional constraint on its objects. However, one can construct an example in which an overdefined part becomes underdefined whenever any constraint is removed.
added in order to make the sketch rigid (or in other words well-defined).

3.10 Tolerances

A tolerance is a small positive real number that characterizes the accuracy of LGS 3D computations. LGS 3D has two kinds of tolerances: angle and linear. The former is used to verify angular measurements, and the latter – linear dimensions. The tolerances are specified by an application and are used by LGS 3D in the following situations:

- when testing special cases. For example, if LGS 3D is positioning a line that passes through two points, then it can select any direction of the line if the distance between the points is less than the linear tolerance.
- when testing constraints. A constraint is assumed to be satisfied, and the corresponding status is assigned to it, if its discrepancy is less than the corresponding tolerance.

When setting tolerances, the application should take into account the selected scale for angles and linear dimensions, as well as the computation accuracy on this computer.

3.11 Interrupting calculations

Recalculation of a model or movement of objects can be interrupted by the application with the help of a special callback function. The pointer to this function is passed to LGS 3D, and this function is then called during the calculation. If the function returns a nonzero value, LGS 3D stops the calculation in an ordered way and returns a special return code.
4 LGS 3D INTERFACE

Application programs interact with LGS 3D through its API (Application Programming Interface). An API object has:

• a type,
• parameters – an array of real numbers,
• arguments – a list of identifiers of other objects,
• attributes – a list of constants defining properties that are specific for this object.

The object creation function returns the identifier of an object (it is unique within a context – an abstraction of a geometrical model). The API entity is a geometric object, variable or a constraint. A short overview of the API entities is given below.

4.1 Supported platforms

LGS 3D is a cross-platform software. It is a set of libraries that runs under 32- and 64-bi Windows and UNIX (including Linux and Mac OS X) operating systems but due to its cross-platform architecture LGS 3D can be built on any platform with C/C++ compiler. Being written in C++, LGS has a C-based API that allows integrating it into a broad range of software applications (even written not in C/C++).

4.2 Data exchange

LGS 3D can work with simple data types only. All input data are represented as the values of numerical types and arrays of such values. LGS 3D objects are visible to applications as integer handles.

Almost all geometric data necessary for calculations (including coordinates of objects and parameters of constraints) are stored by LGS 3D in its own data structures. The application must supply this information when creating LGS 3D objects. After recalculation of the geometric model, the application can obtain coordinates (or transformations) of the resulting objects with the help of special interface functions.

To achieve better performance and true interactivity, LGS 3D supports incremental data definition, which means that data can be added as they appear in the application and all the needed modifications can be made on the fly, ensuring consistency of the new data with constraints and the previously entered data.

The following diagram demonstrates the data exchange between an application and LGS 3D.

![Data exchange between application and LGS 3D](image)

**Fig. 7. Data exchange between application and LGS 3D**

4.2.1 Callbacks interface

While working, LGS 3D uses a portion of data that is defined and stored by the application. This data is accessible via special callback function; its implementation must be provided by the application. In
particular, callbacks are used for evaluation of user-defined curves and surfaces. A pointer to a callback function is passed to the context constructor.

4.2.2 External application identifiers

Sometimes an application has a necessity to perform a search of its own structure that corresponds to the LGS 3D object with fixed handle. For example, this necessity occurs when

- application must update positions of its own objects after the correspondent LGS 3D objects have changed their positions,
- application must handle a call of a callback function that inquires some data about user-defined curve or surface.

To be able to perform a fast search of its structures by a given LGS 3D object handle an application has an ability to store arbitrary integer value near the LGS 3D object (e.g. this value can be a pointer to the application structure). This stored value can be obtained later via passing the LGS 3D object handle to a special API function and application structure can be found via dereferencing operation.

4.3 API functions

This chapter shortly describes functions of the LGS 3D API.

4.3.1 Context

The construction of a geometric model starts with creation of LGS 3D context. A context is a container of geometrical entities; all other objects are created and exist only inside their context, and all operations are defined only for objects, which belong to the same context. There are several context-related functions:

- **Context creation**: context is an abstraction for the geometric model. All the computations are performed within a certain context and only on objects and constraints belonging to the context. Computational tolerances are passed upon context creation.
- **Getting the state of a context**: the application can obtain the current state of the geometric model represented by the context, i.e. check whether the model is solved, overconstrained or inconsistent.
- **Context saving/restoring**: LGS 3D allows an application to save a context into a file or an array of binary data and to restore it later.

4.3.2 Objects specific functions

LGS 3D supports the following functionality for objects manipulations:

- **Objects creation functions**: the application creates geometrical object or variable by passing its position and parameters or initial value to the corresponding LGS 3D function. The function returns the handle of the newly created object or variable.
- **Constraint creation function**: the application creates constraint by passing its arguments and parameters to the corresponding LGS 3D function. The function returns the handle of the newly created constraint.
- **Getting parameters of an object**: at any moment, the application can obtain current position and parameters of an object or current value of a variable.
- **Getting state of an object**: at any moment, the state of a geometrical object or constraint can be examined. States like “changed” can be reported for objects and states like “overdefined” can be reported for constraints.

4.3.3 Model manipulation functions

The following functionality for model manipulation is supported:

- **Recalculate model**: since LGS 3D is incremental, objects and constraints can be added to the model or removed from it. For these changes to affect the current values of other objects, the model recalculation function must be called explicitly.
• **Move or rotate objects**: LGS 3D allows moving or rotating of a group of geometrical objects under constraints, i.e. LGS 3D can move or rotate a group of geometrical objects according to given transformations, so that all constraints will be kept satisfied.

• **Obtain transformations**: after recalculation of the model, the application can obtain transformations of geometrical objects and rigid sets.

• **Change constraint parameter**: for parameterized constraints (such as distance or angle) the value of the parameter can be changed.

• **Change object parameter**: Modification of parameters of a geometric object affects the position of the object and, thus, the initial (sketch) configuration; during recalculation of the model LGS 3D takes these changes into account and looks for an appropriate solution.

• **Run diagnostics**: the application can run diagnostics of over- and under- defined parts of a model by calling diagnostic function.

• **Halt calculation**: a callback function may be supplied to control calculation.

### 4.3.4 Service functions

There are some functions that perform generic operations with LGS 3D objects:

• **Generic objects creation function**: any LGS 3D object can be created with the one generic function that accepts the object type as a parameter.

• **Generic removal function**: any object can be removed at any moment. When the geometric object is deleted, all the constraints having this object as an argument are deleted as well.

• **Iterators**: allow consecutive iterations over all objects of the specified type and state (possibly with a “mask” of an abstract type, i.e., “all”, “geometric objects”, “constraints”), or all objects modified during the last recalculation.

### 4.4 Sample example of API using

The text below is a simple example of using the LGS 3D API.

```cpp
#include <iostream>
#include "LGS3DPublicAPI.h"

using namespace std;

// printing a point
void print_point(LGS3DContext ctx, LGS3DPoint iPoint) {
    LGS3DReal x, y, z;
    // of course, one can get all the parameters of object in a loop,
    // because the index of a parameter ranges from 0 to the value
    // returned by LGS3DGetObjectParamsCount - 1, but here we use
    // predefined constants as indices of parameters for
    // demonstration purposes
    LGS3DGetObjectParamValue(ctx, iPoint, LGS3D_LOCATION_X, &x);
    LGS3DGetObjectParamValue(ctx, iPoint, LGS3D_LOCATION_Y, &y);
    LGS3DGetObjectParamValue(ctx, iPoint, LGS3D_LOCATION_Z, &z);
    cout << "point \[" << iPoint << "]: x = " << x <<", y = " << y <<", z = " << z << endl;
}

// printing a line
void print_line(LGS3DContext ctx, LGS3DLine iLine) {
    LGS3DReal x, y, z, dx, dy, dz;
    LGS3DGetObjectParamValue(ctx, iLine, LGS3D_LOCATION_X,  &x);
    LGS3DGetObjectParamValue(ctx, iLine, LGS3D_LOCATION_Y,  &y);
    LGS3DGetObjectParamValue(ctx, iLine, LGS3D_LOCATION_Z,  &z);
    LGS3DGetObjectParamValue(ctx, iLine, LGS3D_DIRECTION_X, &dx);
    LGS3DGetObjectParamValue(ctx, iLine, LGS3D_DIRECTION_Y, &dy);
    LGS3DGetObjectParamValue(ctx, iLine, LGS3D_DIRECTION_Z, &dz);
    cout << "line \[" << iLine << "]: x = " << x <<", y = " << y <<", z = " << z <<;
    cout << "dx = " << dx <<", dy = " << dy <<", dz = " << dz << endl;
}

// printing a plane
void print_plane(LGS3DContext ctx, LGS3DPlane iPlane) {
    LGS3DReal x, y, z, dx, dy, dz;
    LGS3DGetObjectParamValue(ctx, iPlane, LGS3D_LOCATION_X, &x);
    // ...
LGS3DGetObjectParamValue(ctx, iPlane, LGS3D_LOCATION_Y, &y);
LGS3DGetObjectParamValue(ctx, iPlane, LGS3D_LOCATION_Z, &z);
LGS3DGetObjectParamValue(ctx, iPlane, LGS3D_DIRECTION_X, &dx);
LGS3DGetObjectParamValue(ctx, iPlane, LGS3D_DIRECTION_Y, &dy);
LGS3DGetObjectParamValue(ctx, iPlane, LGS3D_DIRECTION_Z, &dz);
cout << "plane [" << iPlane << "]:
  x = " << x <<", y = " << y <<", z = " << z
  dx = " << dx <<", dy = " << dy <<", dz = " << dz << endl;
}

// printing an object
void print_object(LGS3DContext ctx, LGS3DGeoObject iObj) {
  switch (LGS3DGetObjectType(ctx, iObj)) {
  case LGS3D_GEOOBJECT_POINT:
    print_point(ctx, iObj);
    break;
  case LGS3D_GEOOBJECT_LINE:
    print_line(ctx, iObj);
    break;
  case LGS3D_GEOOBJECT_PLANE:
    print_plane(ctx, iObj);
    break;
  default:
    cout << "unknown object" << endl;
  }
}

// printing a constraint
void print_constraint(LGS3DContext ctx, LGS3DConstraint iConstr) {
  cout << "constraint [" << iConstr << "]:
  unsigned i=0;
  for (i=0;i<LGS3DGetObjectParamsCount(ctx, iConstr);i++) {
    LGS3DReal value;
    LGS3DGetObjectParamValue(ctx, iConstr, i, &value);
    cout << "(" << value << ") " ;
  }
  for (i=0; i<LGS3DGetObjectArgsCount(ctx, iConstr); i++)
    cout << "[" << LGS3DGetObjectArg(ctx, iConstr, i) << "] " ;
  cout << endl;
}

// printing the model
void print_model(LGS3DContext ctx) {
  cout << "----------- Printing model -----------" << endl;
  LGS3DGeoObject obj = LGS3DGetContextFirstObject(ctx, LGS3D_TYPE_GEOOBJECT, LGS3D_STATE_ANY);
  while (obj) {
    print_object(ctx, obj);
    obj = LGS3DGetContextNextObject(ctx, LGS3D_TYPE_GEOOBJECT, LGS3D_STATE_ANY, obj);
  }
  LGS3DConstraint constr = LGS3DGetContextFirstObject(ctx, LGS3D_TYPE_CONSTRAINT, LGS3D_STATE_ANY);
  while (constr) {
    print_constraint(ctx, constr);
    constr = LGS3DGetContextNextObject(ctx, LGS3D_TYPE_CONSTRAINT, LGS3D_STATE_ANY, constr);
  }
}

// printing the objects that have changed during calculations
void print_changed_objects(LGS3DContext ctx) {
  cout << "----------- Printing changed objects -----------" << endl;
  LGS3DGeoObject obj = LGS3DGetContextFirstObject(ctx, LGS3D_TYPE_GEOOBJECT, LGS3D_STATE_CHANGED);
  while (obj) {
    print_object(ctx, obj);
    obj = LGS3DGetContextNextObject(ctx, LGS3D_TYPE_GEOOBJECT, LGS3D_STATE_CHANGED, obj);
  }
}

int main () {
  /* This is an example how to use LGS 3D public API. We will create
  and run simple model, a tetrahedron. */
  cout << "::LGS3DExample - tetrahedron::" << endl;
  /* First of all, we need to create a context, which is living
  place for all geometric objects of our model. */
  LGS3DContext ctx = LGS3DCreateContext (LGS3D_DEFAULT_LINEAR_TOLERANCE, LGS3D_DEFAULT_ROTATIONAL_TOLERANCE);
  cout << "Context created" << endl;
}
/* Now we will create vertices, edges and faces of tetrahedron */

// vertices of first face
const LGS3DReal pt1_1[] = { 1., 0., -1. };
const LGS3DReal pt1_2[] = { 0., 1., -1. };
const LGS3DReal pt1_3[] = { 0., 0., -1. };

// vertices of second face
const LGS3DReal pt2_1[] = { 1., 0., 0. };
const LGS3DReal pt2_2[] = { 0., 1., 0. };
const LGS3DReal pt2_3[] = { 0., 0., 1. };

// vertices of third face
const LGS3DReal pt3_1[] = { 1., 0., 0. };
const LGS3DReal pt3_2[] = { 0., -1., 0. };
const LGS3DReal pt3_3[] = { 0., 0., 1. };

// vertices of fourth face
const LGS3DReal pt4_1[] = { -1., 1., 0. };
const LGS3DReal pt4_2[] = { -1., -1., 0. };
const LGS3DReal pt4_3[] = { -1., 0., 1. };

// normal of first face
const LGS3DReal nor1[] = { 0., 0., -1. };

// normal of second face
const LGS3DReal nor2[] = { 1., 1., 1. };

// normal of third face
const LGS3DReal nor3[] = { 1., -1., 1. };

// normal of fourth face
const LGS3DReal nor4[] = { -1., 0., 0. };

// edges of first face
const LGS3DReal aline1_1[] = { -1., 1., 0. };
const LGS3DReal aline1_2[] = { 0., -1., 0. };
const LGS3DReal aline1_3[] = { 1., 1., 0. };

// edges of second face
const LGS3DReal aline2_1[] = { -1., 1., 0. };
const LGS3DReal aline2_2[] = { 0., -1., 1. };
const LGS3DReal aline2_3[] = { 1., 0., -1. };

// edges of third face
const LGS3DReal aline3_1[] = { -1., -1., 0. };
const LGS3DReal aline3_2[] = { 0., 1., 1. };
const LGS3DReal aline3_3[] = { 1., 0., -1. };

// edges of fourth face
const LGS3DReal aline4_1[] = { 0., -1., 0. };
const LGS3DReal aline4_2[] = { 0., 1., 1. };
const LGS3DReal aline4_3[] = { 0., 1., -1. };

/* Now we will create normals of faces of tetrahedron in our context */
LGS3DPlane plane1 = LGS3DCreatePlane (ctx, pt1_1, nor1);
LGS3DPlane plane2 = LGS3DCreatePlane (ctx, pt2_1, nor2);
LGS3DPlane plane3 = LGS3DCreatePlane (ctx, pt3_1, nor3);
LGS3DPlane plane4 = LGS3DCreatePlane (ctx, pt4_1, nor4);

/* Now we will create edges of faces of tetrahedron in our context */
LGS3DLine line1_1 = LGS3DCreateLine (ctx, pt1_1, aline1_1);
LGS3DLine line1_2 = LGS3DCreateLine (ctx, pt1_2, aline1_2);
LGS3DLine line1_3 = LGS3DCreateLine (ctx, pt1_3, aline1_3);
LGS3DLine line2_1 = LGS3DCreateLine (ctx, pt2_1, aline2_1);
LGS3DLine line2_2 = LGS3DCreateLine (ctx, pt2_2, aline2_2);
LGS3DLine line2_3 = LGS3DCreateLine (ctx, pt2_3, aline2_3);
LGS3DLine line3_1 = LGS3DCreateLine (ctx, pt3_1, aline3_1);
LGS3DLine line3_2 = LGS3DCreateLine (ctx, pt3_2, aline3_2);
LGS3DLine line3_3 = LGS3DCreateLine (ctx, pt3_3, aline3_3);

LGS3DLine line4_1 = LGS3DCreateLine (ctx, pt4_1, aline4_1);
LGS3DLine line4_2 = LGS3DCreateLine (ctx, pt4_2, aline4_2);
LGS3DLine line4_3 = LGS3DCreateLine (ctx, pt4_3, aline4_3);

/* Now we will create faces of tetrahedron as rigid sets in our context */
// create first face
cout << "Creating first face..." << endl;
LGS3DGeoObject arr1[7] = {point1_1,point1_2,point1_3,plane1,line1_1,line1_2,line1_3};
LGS3DRigidSet rs1 = LGS3DCreateRigidSet(ctx,7,arr1);
// create second face
cout << "Creating second face..." << endl;
LGS3DGeoObject arr2[7] = {point2_1,point2_2,point2_3,plane2,line2_1,line2_2,line2_3};
LGS3DRigidSet rs2 = LGS3DCreateRigidSet(ctx,7,arr2);
// create third face
cout << "Creating third face..." << endl;
LGS3DGeoObject arr3[7] = {point3_1,point3_2,point3_3,plane3,line3_1,line3_2,line3_3};
LGS3DRigidSet rs3 = LGS3DCreateRigidSet(ctx,7,arr3);
// create fourth face
cout << "Creating fourth face..." << endl;
LGS3DGeoObject arr4[7] = {point4_1,point4_2,point4_3,plane4,line4_1,line4_2,line4_3};
LGS3DRigidSet rs4 = LGS3DCreateRigidSet(ctx,7,arr4);

/* Now we will create constraints, which connects faces */
cout << "Creating constraints..." << endl;
LGS3DCreateCoincidence(ctx,point1_1,point2_1);
LGS3DCreateCoincidence(ctx,point1_1,point3_1);
LGS3DCreateCoincidence(ctx,point1_2,point2_2);
LGS3DCreateCoincidence(ctx,point1_2,point4_1);
LGS3DCreateCoincidence(ctx,point1_3,point3_2);
LGS3DCreateCoincidence(ctx,point1_3,point4_2);
LGS3DCreateCoincidence(ctx,point2_3,point3_3);
LGS3DCreateCoincidence(ctx,point2_3,point4_3);
cout << "Geometry was created successfully." << endl;

/* Now we can start calculations. */
cout << "Applying all changes..." << endl;
LGS3DResult res = LGS3DApplyChanges (ctx);
cout << "applyChanges() returned " << res << endl;
/* Analysis of the result */
print_changed_objects(ctx);
print_model(ctx);

/* Finally we must free all resources. */
LGS3DRemoveContext (ctx);
cout << "Context removed" << endl;
cout << "Done." << endl;
return 0;
/* The end */}