User and Reference Manual: CGAP for CGAL

Separate Build

July 21, 2007

Contents

1 Introduction **1**

2 Compiler/Linker installation **3**

2.1 Bug fixing in CGAL 5

3 Using CGAP **7**

3.1 Getting started with CGA Types 7

3.2 Getting started with CGAP primitives and operations 8

4 Demos **13**

5 Reference Pages **15**

5.1 Global Functions 16

Chapter 1

Introduction

In this user and reference manual a small introduction to the CGAP library is given.

For people who are not accustomed with Conformal Geometric Algebra, it is recommended to get accustomed to it by reading the GA-tutorial of Leo Dorst and Daniël Fontijne[1].

In this user manual several demos and examples of the CGAP library will be given. The complete code of the examples can be found in the directory `CGAP/examples`, the demos can be found in `CGAP/demo`.

[1] Downloadable from `http://staff.science.uva.nl/ leo/ga/`

Chapter 2

Compiler/Linker installation

The demos were tested in a Microsoft Visual Studio 2005 (MSVC8) environment and in the Dev-C++ editor (mingw, gcc compiler). The Dev-C++ project files can be found in the folder `/CGAP/demo/CGAP`. The Visual Studio solutions can be found in the individual folders in `/CGAP/demo/CGAP/MSVC8`.

The applications have been tested in a Windows 2000, XP and Vista environment. Since FLTK was used for the GUI part of the project, the code is also assumed to work on Linux machines.

To define your own projects, several settings should be set.

Compiler

Compiler include directories[1]:

- standard include directory of your compiler
- $CGAL_ROOT/include
- $CGAL_ROOT/include/CGAL/config/msvc7 (when using Microsoft Visual Studio C++ 7.0 or the MinGW C++-compiler)
- $FLTK (for this project we used Fast Light ToolKit 2.0 in the demos)
- $FLTK/fltk/compat
- $GAIGEN. Those files can be generated with the GAIGEN GA code-generator, which can be downloaded from `http://www.science.uva.nl/ga/gaigen/` . The source files of the GAIGEN-output should be located in algebrasGAIGEN/c*x*ga_element. In this case *x* denotes the number of dimension.

There are two ways to add the CGAP-package library: by adding `/CGAP/include` to the compiler include directories or by adding all files to your project directly.

Compiler options:

- disable warnings

[1] In the include directives of compiler and linker we use several constants: $CGAL_ROOT equals `/CGAL-3.2` in the CGAL-package. The latest version of CGAL can be downloaded from `http://www.cgal.org`. $FLTK is an additional library that can be downloaded from `http://www.fltk.org`. For this version the release `fltk-2.0.x-r5187` was used and the corresponding fluid-GUI-editor was used to build the viewer.

Linker

Linker libraries:
The CGAL-lib:

- $CGAL_ROOT/lib/msvc7/CGAL.lib (for Microsoft Visual Studio and the MinGW C++-compiler; when you use another compiler, see the CGAL user manual for more instructions)

When using FLTK[2]:

- $FLTK/lib/libfltk.a
- $FLTK/lib/libfltk_gl.a
- $FLTK/lib/libfltk_png.a
- $FLTK/lib/libfltk_images.a
- $FLTK/lib/libfltk_jpeg.a

Instead of this, in MSVC8 FLTK can also be loaded by adding:

- $FLTK/lib/fltkd.lib
- $FLTK/lib/fltkdlld.lib

In the options of some linkers the default library `Libc.lib` has to be excluded, because otherwise the linker will complain about several functions that occur twice in the code. In some other cases the files `libcd, msvcrt` and `msvcprt` have to be excluded. (depending on debugging settings)

For the MinGW/gcc-compiler the following libraries should also be included[3]:

- libgdi32.a
- libwsock32.a
- libole32.a
- libuuid.a
- libcomctl32.a
- libopengl32.a

Makefile The files gaigenhl.cpp, gaigenhl.h, and c*x*ga_element.h of the GAIGEN-library should not be put as a rule in the makefile. Otherwise this will result in some compiler errors. When using a C++ GUI-editor: disable the building of those files.

[2]Remark: fltk should be correctly installed on your computer
[3]Remark: these libraries should be included AFTER the fltk-libraries (and not before, otherwise the linker will complain).

2.1 Bug fixing in CGAL

In the file `Delaunay_d` is a bug. The line:
`const R& lifted_kernel() const { return Base::kernel(); }`
should be replaced by the line:
`const Lifted_R_& lifted_kernel() const { return Base::kernel(); }`

Chapter 3

Using CGAP

3.1 Getting started with CGA Types

To discuss all basic components of an application based on CGA types, we give the code of simple program.

```
// File: /CGAP/examples/CGAP/Simple GA_Calculations_2d.cpp
#include <iostream>
using namespace std;

// Load and define GA-numbertype
#include <algebrasGAIGEN/c2ga_element/c2ga_element.h>
typedef c2ga_element GA_Type_2;

int main(int argc, char *argv[])
{
  cout << "Some simple GA-calculations in 2D. ";

  // Simple addition
  GA_Type_2 a = 1;
  GA_Type_2 b = 1;
  GA_Type_2 result = a + b;
  a.print(); cout << "plus\n"; b.print();
  cout << "equals"; result.print(""); cout << "\n\n";

  // Define two CGA-points p = (2,3) and q = (1,1)
  GA_Type_2 p(GRADE1, 2.0, 3.0, 1.0, (2*2+ 3*3)/2);
  GA_Type_2 q = 1.0 * GA_Type_2::e1 + 1.0 * GA_Type_2::e2 +
                GA_Type_2::no + ((1*1+ 1*1)/2) * GA_Type_2::ni;
  p.print("p: ");
  q.print("q: ");
  cout << "The CGA-line through p and q has CGA-form: ";
  GA_Type_2 result2 = p^q^GA_Type_2::ni;
  result2.print();
}
```

Specification of the number type
The first important choice in a CGAP program is the use of a CGA type (`GA_Type`). Basically a GAIGEN based CGA-type can be used.

```
#include <algebrasGAIGEN/c2ga_element/c2ga_element.h>
typedef c2ga_element GA_Type_2;
```

Simple additions
As we see in the code, several calculations can be made with CGA-elements.

Construction of CGA-elements
There are two ways to construct CGA-elements:

- By calling the CGA-element constructor. For example:
  `GA_Type_2 p(GRADE1, 2.0, 3.0, 1.0, (2*2+ 3*3)/2);`
- By using the CGA-formula and base element constants. For example:
  `GA_Type_2 q = 1.0 * GA_Type_2::e1 + 1.0 * GA_Type_2::e2 + GA_Type_2::no + ((1*1+ 1*1)/2) * GA_Type_2::ni;`.

Operators
Which operators are available depends on the CGA type. When GAIGEN is used, the geometric product is available as the `*`-operator, the addition as the `+`-operator, the outer product as `^`-operator. The inner product does not have an operator, but should be called by the `hip`-function.

3.2 Getting started with CGAP primitives and operations

3.2.1 Primitives

Examples of the construction of CGAP-primitives can be found in the following code:

```
// File: /CGAP/examples/CGAP/Primitives_Example_3d.cpp
#include <iostream>
using namespace std;

// Define number type
#include <algebrasGAIGEN/c3ga_element/c3ga_element.h>
typedef c3ga_element GA_Type_3;

// Define 3D-kernel
#define C3GA_KERNEL
#include <CGAL/CGAP/GA_Kernel.h>
// Use GA_Type_3 as GA type and float as field number type
typedef CGAL::C3GA<GA_Type_3, float> FK_3;
class K3 : public FK_3 {};

// Define types
typedef K3::Point       Point;
typedef K3::Segment     Segment;
```

```
typedef K3::Line        Line;
typedef K3::Plane       Plane;
typedef K3::Vector      Vector;
typedef K3::Ray         Ray;
typedef K3::Triangle    Triangle;
typedef K3::Tetrahedron Tetrahedron;

// Extra CGAL includes
#include <CGAL/origin.h>

int main(int argc, char *argv[])
{
// POINT EXAMPLES
    // Construct a point based on cartesian coordinates
    Point pt1(1.0,2.1,3.2);
    // Construct a point based on GA-type
    GA_Type_3 g3(GRADE1, 2.0, 3.0, 7.0,  1.0, (2*2+ 3*3 + 7*7)/2);
    Point pt2(g3);
    // Construct a point at the CGAL-Origin
    Point pt3(CGAL::ORIGIN);

    // Extract Cartesian coordinates from CGA-points
    cout << "Pt2 is located at (" << pt2.x() << ", " << pt2.y() << ", " <<
        pt2.z() << ")\n";

    // Compare equality of points
    if (pt1==pt2) cout << "Pt1 and pt2 are equal";

// VECTOR EXAMPLES
    // Contruct the (positionless) vector with direction pt1->pt2
    Vector v1(pt1,pt2);
    // Calculate the result vector pt1-pt2
    Vector v2 = pt1 - pt2;
    // Initialize as Null vector
    Vector v3(CGAL::NULL_VECTOR);
    // Add the vector to a point
    Point pt4 = pt3 + v1;

// SEGMENT EXAMPLES
    // Construct the segment pt1-pt3
    Segment seg(pt1, pt3);
    // Extract start and endpoint
    Point p = seg.startPoint(), q=seg.endPoint();

// LINE EXAMPLES
    // Construct the line through pt1 and pt2
    Line ln1(pt1, pt2);
    // Construct the line through the segment seg
    Line ln2(seg);

// RAY EXAMPLES
    // Construct a ray originating at pt1 and directing to pt2
    Ray r1(pt1, pt2);
    // Construct a ray originating at pt2 and directing in direction of ln1
```

```
    Ray r2(pt2, ln1);
    // Construct a ray originating at pt2 and directing in direction of vector v1
    Ray r3(pt4, v1);

// PLANE EXAMPLES
    // Construction from 3 points
    Plane pl1(pt1,pt2,pt3);
    // Construction from a line and a point
    Plane pl2(ln2, pt2);

// TETRAHEDERON AND TRIANGLE EXAMPLES
   // Construct triangle of points pt1, pt2 and pt3
   Triangle tr1 = Triangle(pt1,pt2,pt3);

    return 0;
}
```

3.2.2 Operations

Examples of the operations on CGAP-primitives can be found in the following code:

```
// File: /CGAP/examples/CGAP/Operations_Example_3d.cpp
#include <iostream>
using namespace std;

// Define number type
#include <algebrasGAIGEN/c3ga_element/c3ga_element.h>
typedef c3ga_element GA_Type_3;

// Define 3D-kernel
#define C3GA_KERNEL
#include <CGAL/CGAP/GA_Kernel.h>
// Use GA_Type_3 as GA type and float as field number type
typedef CGAL::C3GA<GA_Type_3, float> FK_3;
class K3 : public FK_3 {};

using namespace CGAL;

// Define types
typedef K3::Point       Point;
typedef K3::Segment     Segment;
typedef K3::Line        Line;
typedef K3::Plane       Plane;
typedef K3::Vector      Vector;
typedef K3::Ray         Ray;
typedef K3::Triangle    Triangle;
typedef K3::Tetrahedron Tetrahedron;

int main(int argc, char *argv[])
{
    // Construct a few points
    Point pt1(1.0,2.1,3.2), pt2(-4.3, 5.4, 8.2), pt3(-3.2,-5.2, 9.2),
```

```
            pt4(-5.3, 5.4, 0.3), pt5(8.2,-3.4,-4.3), pt6(1.2,3.2,-4.7);
    Plane pl1(pt1, pt2, pt3);

    // Calculate the centre of the circle through pt1, pt2 and pt3
    Point centre = K3().construct_circumcenter_3_object()(pt1, pt2, pt3);

    // Determine wheter pt2 or pt3 is closer to pt1
    Comparison_result result = K3().compare_distance_3_object()(pt1,pt2,pt3);

    // Construct the line perpendicular to plane pl1, through point pt4
    Line ln = K3().construct_perpendicular_line_3_object()(pl1, pt4);

    return 0;
}
```

For further examples of use of the kernel we refer to the chapter about the demos. In the reference manual, the individual predicate function objects will be discussed.

Chapter 4

Demos

The CGA-package has 3 demo applications to show the strengths of CGA:

- A **Delaunay viewer application** that shows the use of CGAP for 2D and 3D graphics. It illustrates chapter 6 of the master thesis.
- A **Voronoi viewer application** that shows the use of CGAP in combination with flags for 2D and 3D graphics. This application is discussed in chapter 7 of the thesis.
- A **Voronoi file application** that works with CGAP in combination with flags for nD. Due to the limitations of GAIGEN yields $2 \leq n \leq 5$. It reads an input file with points and outputs the result to a specified output file.

The demos can be found in the directory `/CGAP/demo`.

4.0.3 Format of the input file

All three demo applications can work with point input files. An example input file has the following format:

```
4 2
1.0 2.5
2.5 1.0
3.5 3.6
1.0 6.0
```

At the first line the number of points n (here: 2) and the dimension d (here: 4) are specified. At the succeeding lines, the n points should be present. The different coordinates of the points are separated by spaces. The `.` is used as a decimal separator.

Some example input files can be found in the directory `/CGAP/demo/CGAP/input`

Chapter 5

Reference Pages

The following pages give an overview on the functionality provided in the kernel.

5.1 Global Functions

Function Object Class CGAL::Coplanar_orientation_3

Orientation *Coplanar_orientation_3.coplanar_orientation(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

CGAL requirements: If *p,q,r* are collinear, then *COLLINEAR* is returned. If not, then *p,q,r* define a plane *P*. The return value in this case is either *POSITIVE* or *NEGATIVE*, but we don't specify it explicitely. However, we guarantee that all calls to this predicate over 3 points in *P* will return a coherent orientation if considered a 2D orientation in *P*.

GA implementation: The plane P has the formula $P = p \wedge q \wedge r \wedge \mathbf{e}_\infty$. The plane with opposite orientation is $-P$. The dual of the plane P has the form: $\widetilde{P} = \mathbf{n} + d\mathbf{e}_\infty$, where $\mathbf{n}$ is the 3D-tangent vector to the plane with generic formula $\mathbf{n} = a\mathbf{e}_1 + b\mathbf{e}_2 + c\mathbf{e}_3$, and where d is the distance to from the plane to the origin point $\mathbf{e}_0$.
The direction of the normal vector (positive or negative also determines its orientation. So a consistent orientation (*POSITIVE* or *NEGATIVE*) can be given by looking at the sign of one of the normal vector coëfficients (a, b or c). When such a coëfficient is zero, we should look at another coëfficient. We do this in the order c, b, a, because in that way we get the same results as the CartesianKernel-version of this function. When *p,q,r* are collinear, then $P = p \wedge q \wedge r \wedge \mathbf{e}_\infty$ returns zero.

See Also

CGAL::orientation_3 . . . . . page **??**

Function Object Class CGAL::Orientation_3

Orientation *Orientation_3.operator()(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r,*

Point_3<Kernel> s)

returns *POSITIVE*, if *s* lies on the positive side of the oriented plane *h* defined by *p*, *q*, and *r*, returns *NEGATIVE* if *s* lies on the negative side of *h*, and returns *COPLANAR* if *s* lies on *h*.
GA implementation: The GA-formula $S = p \wedge q \wedge r \wedge s$ defines an oriented sphere. This orientation can be determined by looking at the sign of the $\mathbf{e}_0$ coëfficient. This coëfficient *c* can easily be calculated by taking the inner product of S with $\mathbf{e}_\infty$: $c = -S \cdot \mathbf{e}_\infty$ (because $\mathbf{e}_0 \cdot \mathbf{e}_\infty = -1$, while $\mathbf{e}_i \cdot \mathbf{e}_\infty = 0$ where $1 \leq i \leq 3$ and $\mathbf{e}_\infty \cdot \mathbf{e}_\infty = 0$).

Function Object Class CGAL::CompareDistance_3

Comparison_result *CompareDistance_3(Point_3 p, Point_3 q, Point_3 r)*

compares the distances of points *q* and *r* to point *p*
GA implementation: In CGA the distance between two normalized points *p* and *q* can be calculated with the following formula $d(p,q) = \sqrt{-2(p \cdot q)}$. This function only returns which of the two distances is the smallest; therefore taking the square roots is not necessary, because that does not affect the result of the function.

Function Object Class CGAL::CompareXYZ_3

Comparison_result *CompareXYZ_3(Point_3 p, Point_3 q)*

Compares the Cartesian coordinates of points *p* and *q* lexicographically in *xy* order: first *x*-coordinates are compared, if they are equal, *y*-coordinates are compared. If they are equal, *z*-coordinates are compared.
GA implementation: This function uses cartesian coördinates to perform this comparison (see function requirements above). In the GA-kernel this method has a more generic implementation. To calculate the cartesian coördinates a special C3GA-kernel function *default_base_element(index)* is used. This function gives 3 orthogonal unit base vectors. By default it returns the cartesian unit base vectors. Other orthogonal vectors can be used by adapting this function.

Function Object Class CGAL::CompareX_2

Comparison_result *CompareX_2(Point_2 p, Point_2 q)*

compares the Cartesian *x*-coordinates of points *p* and *q*
GA implementation: This function translates the GA-points to the cartesian x-coördinates and performs the comparison. This can not be done in a 'pure' GA way, because the requirements demand Cartesian coördinates.

Function Object Class CGAL::CompareY_2

Comparison_result *CompareY_2(Point_2 p, Point_2 q)*

Compares the Cartesian *y*-coordinates of points *p* and *q*
GA implementation: This function translates the GA-points to the cartesian y-coördinates and performs the comparison. This can not be done in a 'pure' GA way, because the requirements demand Cartesian coördinates.

Function Object Class CGAL::Construct_circumcenter_3

Point_3 *Construct_circumcenter_3(Point_3 p, Point_3 q, Point_3 r)*

compute the center of the circle passing through the points *p*, *q* and *r*.
Precondition: *p*, *q* and *r* are not collinear.
GA implementation: First the circle *C* through the points *p*, *q* and *r* is constructed. After that the centre of this circle is calculated. The circle has GA-formula: $C = p \wedge q \wedge r$. The middlepoint of the circle is $m = \frac{(C\mathbf{e}_\infty C)}{2(\mathbf{e}_\infty \cdot C)^2}$.

Function Object Class CGAL::ConstructPerpendicularLine_3

Line_3 *ConstructPerpendicularLine_3(Plane_3 pl, Point_3 p)*

returns the line that is perpendicular to *pl* and that passes through point *p*. The line is oriented from the negative to the positive side of *pl*
GA implementation: This is done by calculating the normal vector to the plane *pl*: f (in GA-free-vector form). $f = \widetilde{\mathrm{pl}} \wedge \mathbf{e}_\infty$. The line is calculated by taking the join (i.e. outer product) of f and *p*.

See Also

CGAL::Plane_3<Kernel> . . . . . page **??**

Function Object Class CGAL::ConstructPlane_3

Plane_3 *ConstructPlane_3(Point_3 p, Point_3 q, Point_3 r)*

creates a plane passing through the points *p*, *q* and *r*. The plane is oriented such that *p*, *q* and *r* are oriented in a positive sense (that is counterclockwise) when seen from the positive side of the plane. Notice that is degenerate if the points are collinear.
GA implementation: uses the corresponding PlaneGA-constructor.

Plane_3 *ConstructPlane_3(Point_3 p, Direction_3 d)*

introduces a plane that passes through point *p* and that has as an orthogonal direction equal to *d*.
GA implementation: uses the corresponding PlaneGA-constructor.

Plane_3 *ConstructPlane_3(Point_3 p, Vector_3 v)*

introduces a plane that passes through point *p* and that is orthogonal to *v*.
GA implementation: uses the corresponding PlaneGA-constructor.

Plane_3 *ConstructPlane_3(Line_3 l, Point_3 p)*

introduces a plane that passes through the line *l* and the point *p*.
GA implementation: uses the corresponding PlaneGA-constructor.

Plane_3 *ConstructPlane_3(Ray_3 r, Point_3 p)*

introduces a plane that is defined through the ray *r* and the point *p*.
GA implementation: uses the corresponding PlaneGA-constructor.

Plane_3 *ConstructPlane_3(Segment_3 s, Point_3 p)*

introduces a plane that is defined through the segment *s* and the point *p*.
GA implementation: uses the corresponding PlaneGA-constructor.

See Also

CGAL::PlaneGA . . . . . page **??**

Function Object Class CGAL::ConstructRay_3

Ray_3 *ConstructRay_3(Point_3 p, Vector_3 v)*

introduces a ray with source *p* and with the direction given by *v*.
GA implementation: uses the corresponding RayGA-constructor.

Ray_3 *ConstructRay_3(Point_3 p, Line_3 l)*

introduces a ray with source *p* and with the same direction as *l*.
GA implementation: uses the corresponding RayGA-constructor.

See Also

CGAL::RayGA . . . . . page **??**

Function Object Class CGAL::ConstructTetrahedron_3

Tetrahedron_3 *ConstructTetrahedron_3(Point_3 p, Point_3 q, Point_3 r, Point_3 s)*

introduces a tetrahedron with vertices *p*, *q*, *r* and *s*.

See Also

CGAL::TetrahedronGA . . . . . page **??**

Function Object Class CGAL::ConstructTriangle_2

Triangle_2 *ConstructTriangle_2(Point_2 p, Point_2 q, Point_2 r)*

introduces a triangle with vertices *p*, *q* and *r*.
GA implementation: Uses TriangleGA-constructor

See Also

CGAL::TriangleGA . . . . . page **??**

Function Object Class CGAL::ConstructTriangle_3

Triangle_3 *ConstructTriangle_3(Point_3 p, Point_3 q, Point_3 r)*

introduces a triangle with vertices *p*, *q* and *r*.

See Also

CGAL::TriangleGA . . . . . page **??**

Function Object Class CGAL::Coplanar_side_of_bounded_circle_3

Bounded_side *Coplanar_side_of_bounded_circle_3(Point_3 p, Point_3 q, Point_3 r, Point_3 t)*

returns the bounded side of the circle defined by *p*, *q*, and *r* on which *t* lies.
Precondition: *p*, *q*, *r*, and *t* are coplanar and *p*, *q*, and *r* are not collinear.
GA implementation: First this function calculates the circumcenter point *m* of the given point. Then it compares the distance from *m* to *p* with the distance from *m* to *t*.

Function Object Class CGAL::Orientation_2

Orientation *Orientation_2(Point_2 p, Point_2 q, Point_2 r)*

returns *LEFT_TURN*, if *r* lies to the left of the oriented line *l* defined by *p* and *q*, returns *RIGHT_TURN* if *r* lies to the right of *l*, and returns *COLLINEAR* if *r* lies on *l*.
GA implementation: The GA-formula $C = p \wedge q \wedge r$ defines an oriented circle. This orientation can be determined by looking at the sign of the $\mathbf{e}_0$ coëfficient. This coëfficient *c* can easily be calculated by taking the inner product of *C* with $\mathbf{e}_\infty$: $c = -C \cdot \mathbf{e}_\infty$ (because $\mathbf{e}_0 \cdot \mathbf{e}_\infty = -1$, while $\mathbf{e}_i \cdot \mathbf{e}_\infty = 0$ where $1 \leq i \leq 2$ and $\mathbf{e}_\infty \cdot \mathbf{e}_\infty = 0$).

Function Object Class CGAL::SideOfOrientedCircle_2

Oriented_side *SideOfOrientedCircle_2(Point_2 p, Point_2 q, Point_2 r, Point_2 t)*

returns the relative position of point t to the oriented circle defined by p, q and r. The order of the points p, q and r is important, since it determines the orientation of the implicitly constructed circle.
Precondition: p, q and r are not collinear.
GA implementation: First the oriented circle is calculated: $C = p \wedge q \wedge r$. (a circle always has an orientation) Then the sign of the orientation can be determined by looking at the sign of the scalar $t \cdot \widetilde{C}$.

Function Object Class CGAL::SideOfOrientedSphere_3

Oriented_side *SideOfOrientedSphere_3(Point_3 p, Point_3 q, Point_3 r, Point_3 s, Point_3 t)*

returns the relative position of point t to the oriented sphere defined by p, q, r and s. The order of the points p, q, r, and s is important, since it determines the orientation of the implicitly constructed sphere. If the points p, q, r and s are positive oriented, positive side is the bounded interior of the sphere.
Precondition: p, q, r and s are not coplanar.
GA implementation: First the sphere is calculated: $B = p \wedge q \wedge r \wedge s$. (a sphere always has an orientation) Then the sign of the orientation can be determined by looking at the sign of the scalar $t \cdot \widetilde{B}$.

#include <../../include/CGAL/CGAP/LineGA.h>

Class CGAL::Linear_algebraHd<RT>

Definition

The class *Linear_algebraHd<RT>* serves as the default traits class for the LA parameter of *CGAL::Homogeneous_d<RT,LA>*. It implements linear algebra for Euclidean ring number types *RT*.

#include <CGAL/Linear_algebraHd.h>

Is Model for the Concept

LinearAlgebraTraits_d . . . . . page **??**

Requirements

To make a ring number type *RT* work with this class it has to provide a division *operator/* with remainder.

Operations

Fits all operation requirements of the concept.