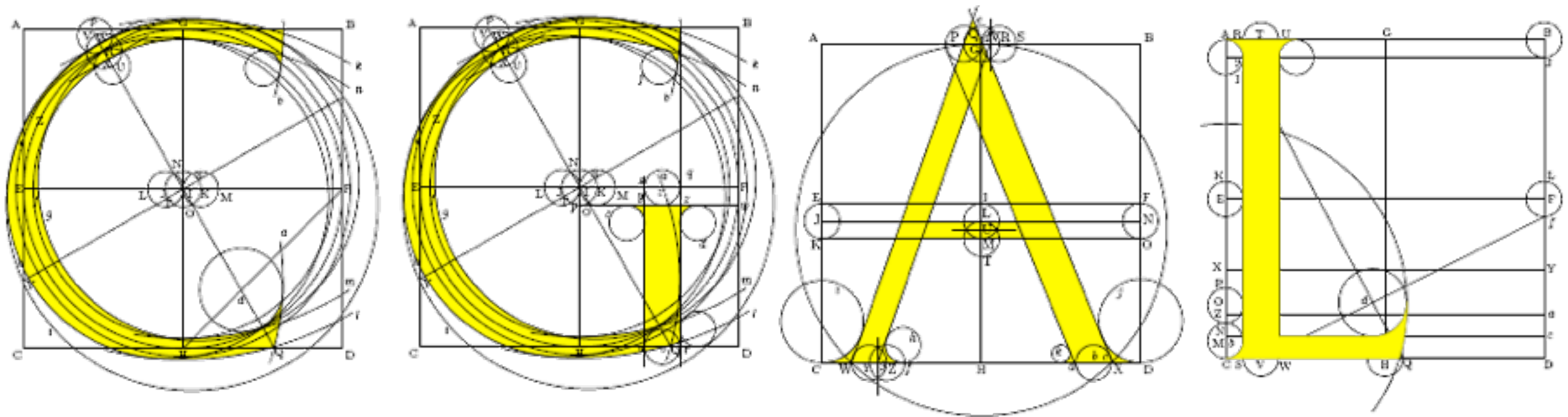


# A VERY SHORT INTRODUCTION TO



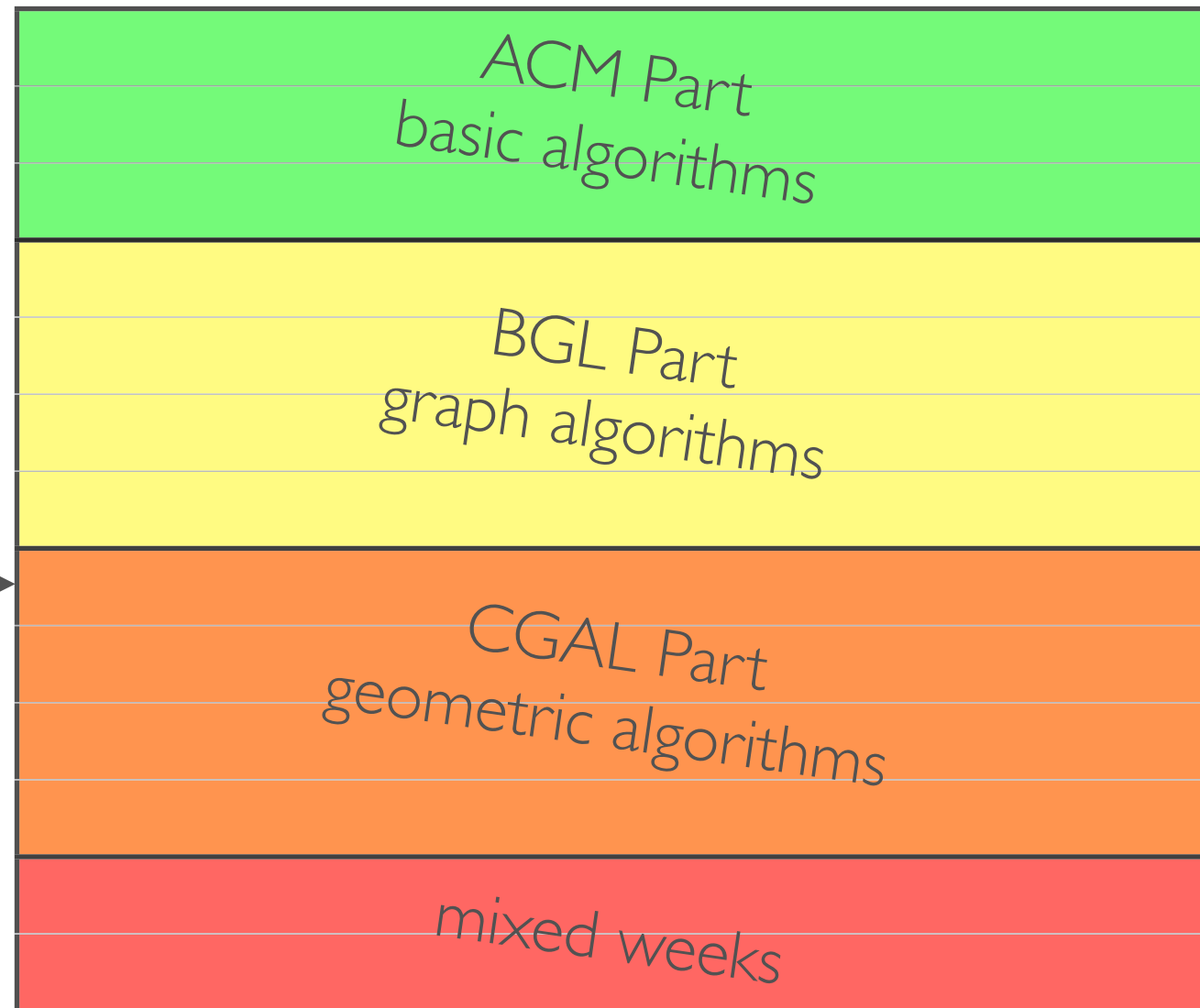
## The Computational Geometry Algorithms Library

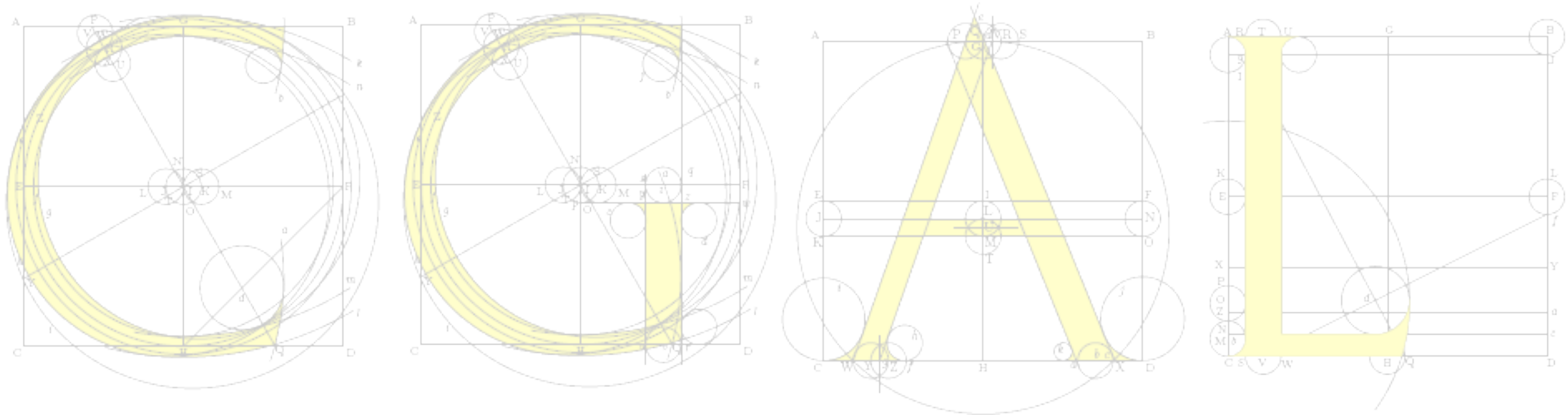
Michael Hoffmann <[hoffmann@inf.ethz.ch](mailto:hoffmann@inf.ethz.ch)>

(Based on work by Pierre Alliez, Andreas Fabri, Efi Fogel, Lutz Kettner, Sylvain Pion, Monique Teillaud, Mariette Yvinec, and probably many others.)

# ALGOLAB TIMELINE

we are here



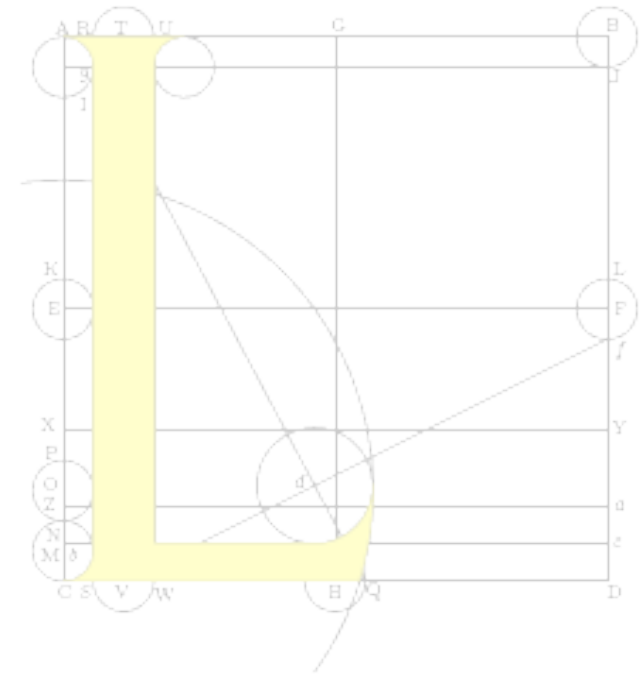
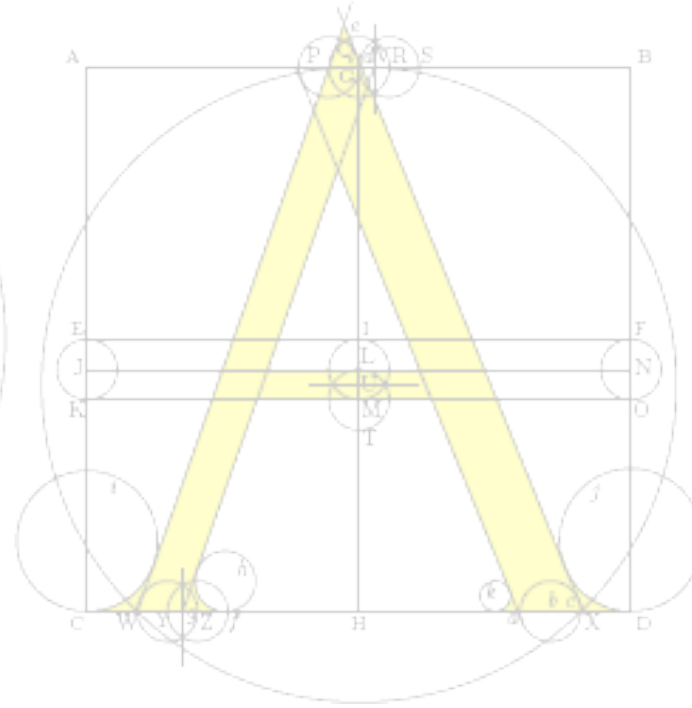
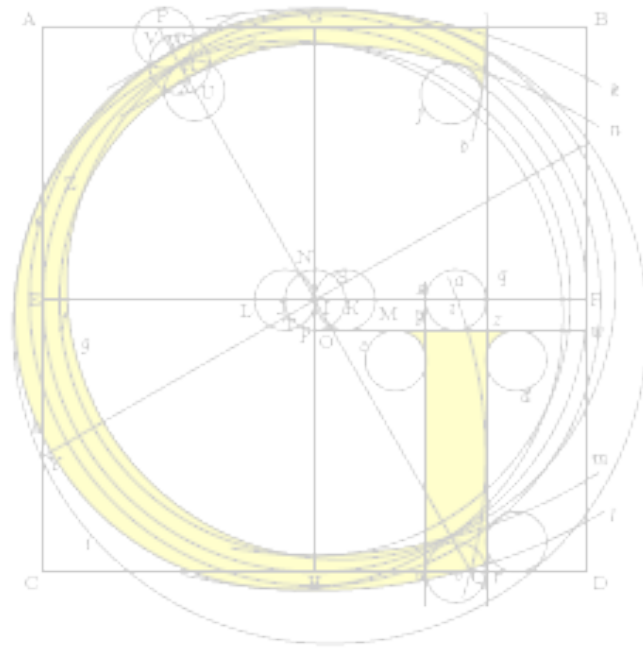
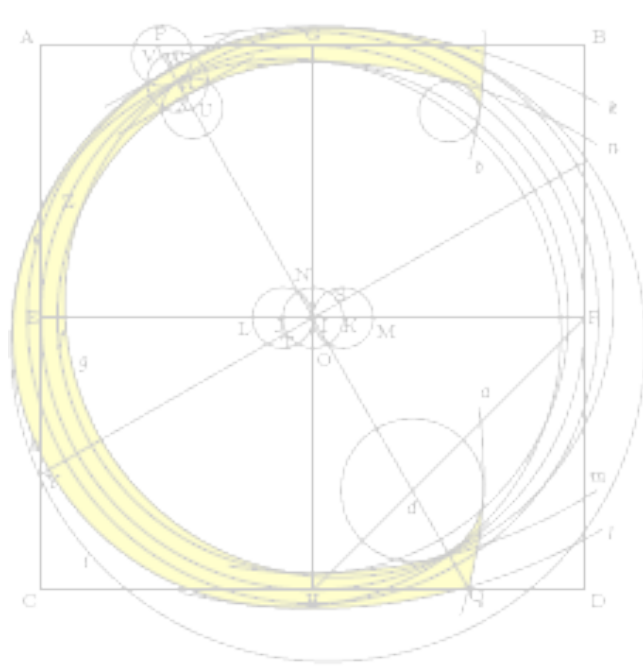


I: The CGAL Project

II: Exact Geometric Computing

III: Basic Programming using a CGAL Kernel

IV: Practical Information



# PART I:

## The CGAL Project: History and Philosophy

# THE MISSION

“Make the large body of geometric algorithms developed in the field of computational geometry available for industrial applications”

CGAL Project Proposal, 1996

Design goals: Reliability, efficiency, and flexibility.

Achieved through

- ▶ Exact geometric computing
- ▶ Generic Programming
- ▶ ISO C++

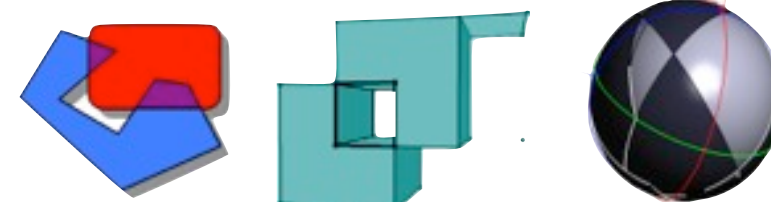
# CONTENTS



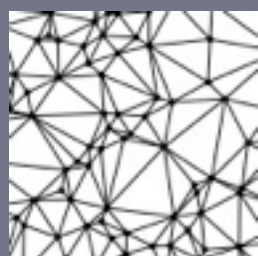
Bounding Volumes



Polyhedral Surfaces



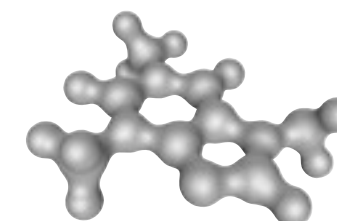
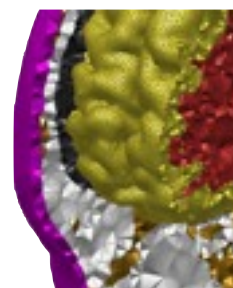
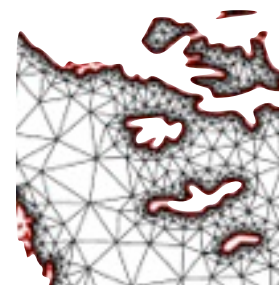
BooleanOperations



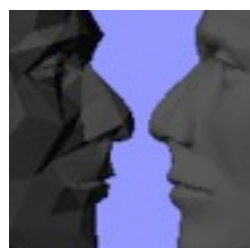
Triangulations



Voronoi Diagrams



Mesh Generation



Subdivision



Simplification



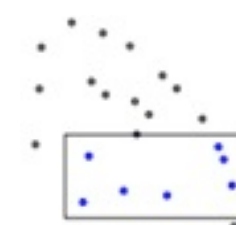
Parametrisation



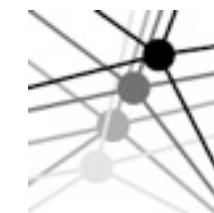
Streamlines



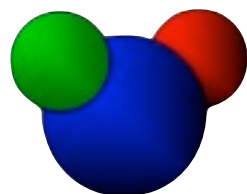
Ridge  
Detection



Neighbor  
Search



Kinetic  
Datastructures



Lower Envelope



Arrangement



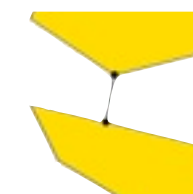
Intersection  
Detection



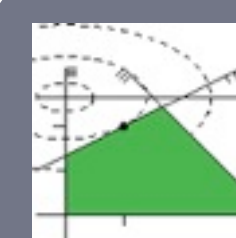
Minkowski  
Sum



PCA



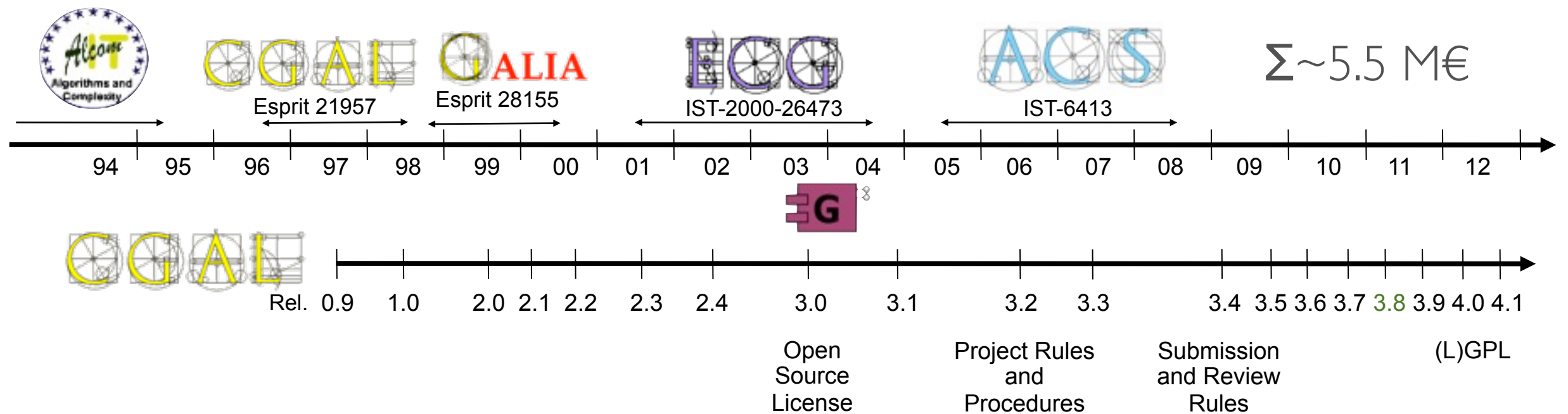
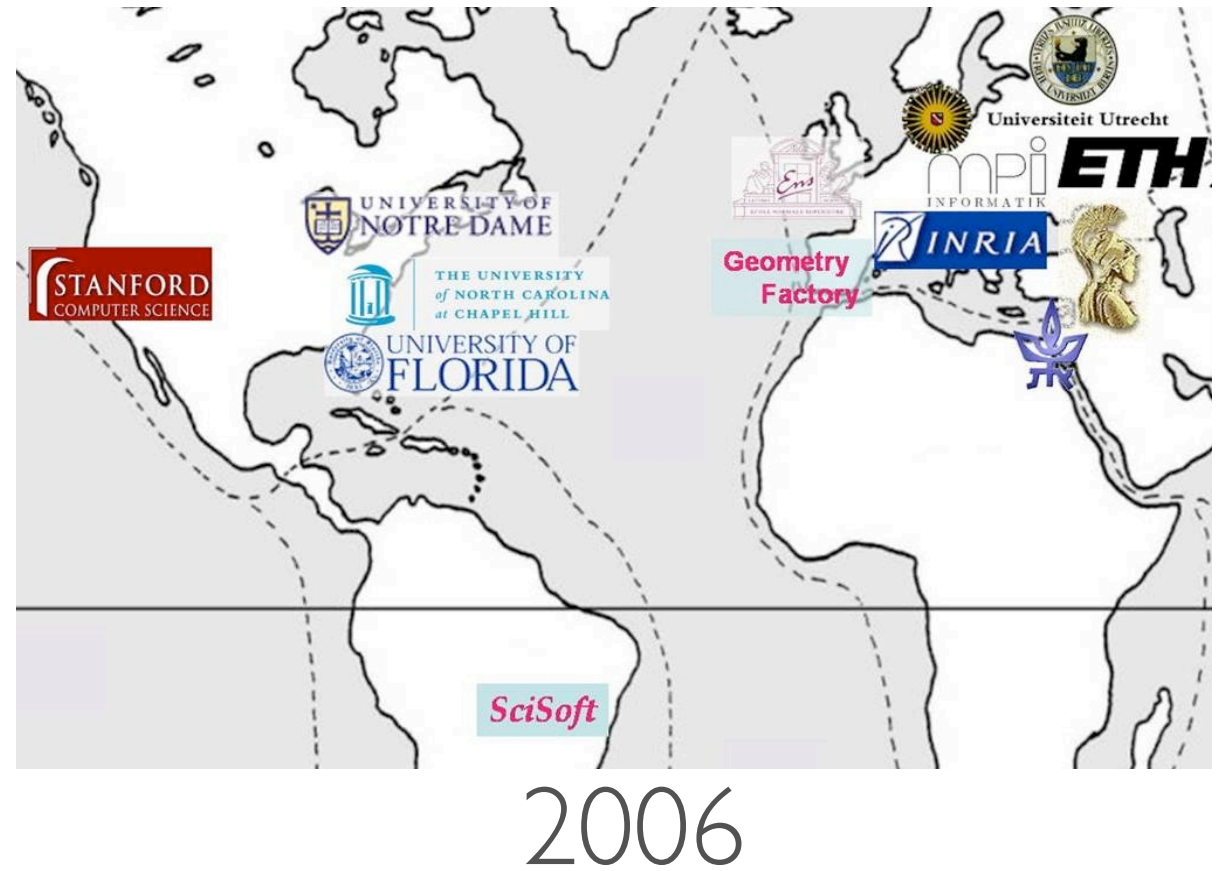
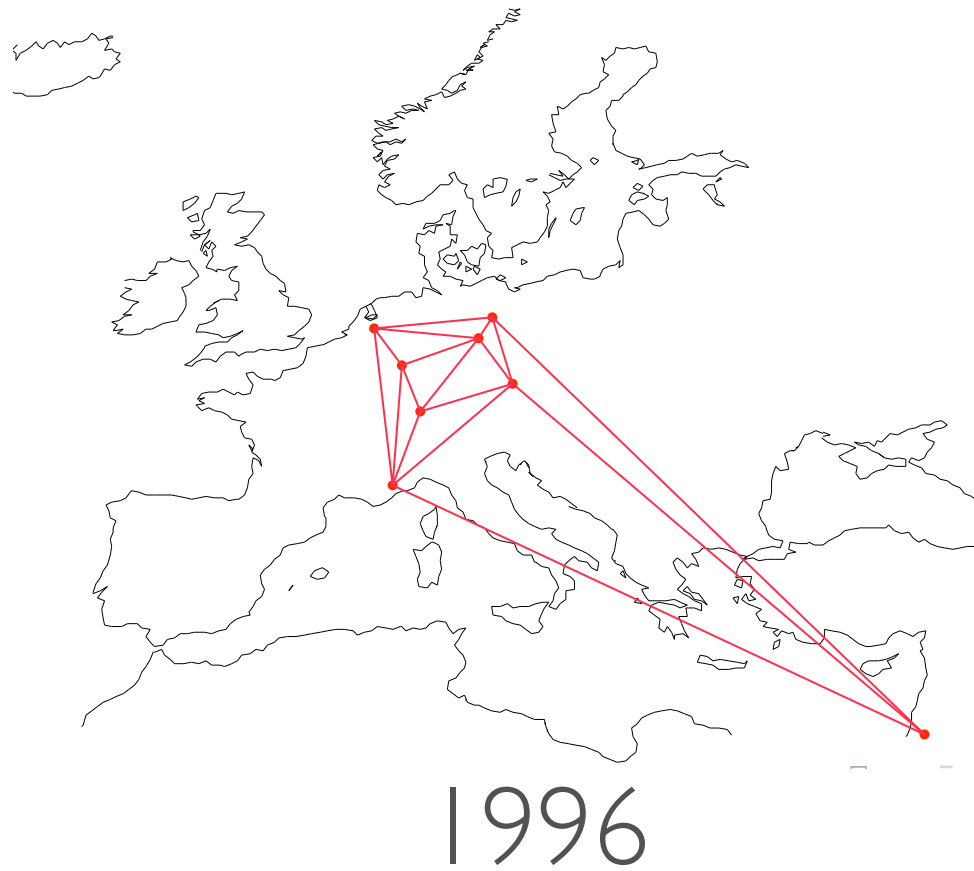
Polytope  
distance



QP Solver



# HISTORY



# IN NUMBERS

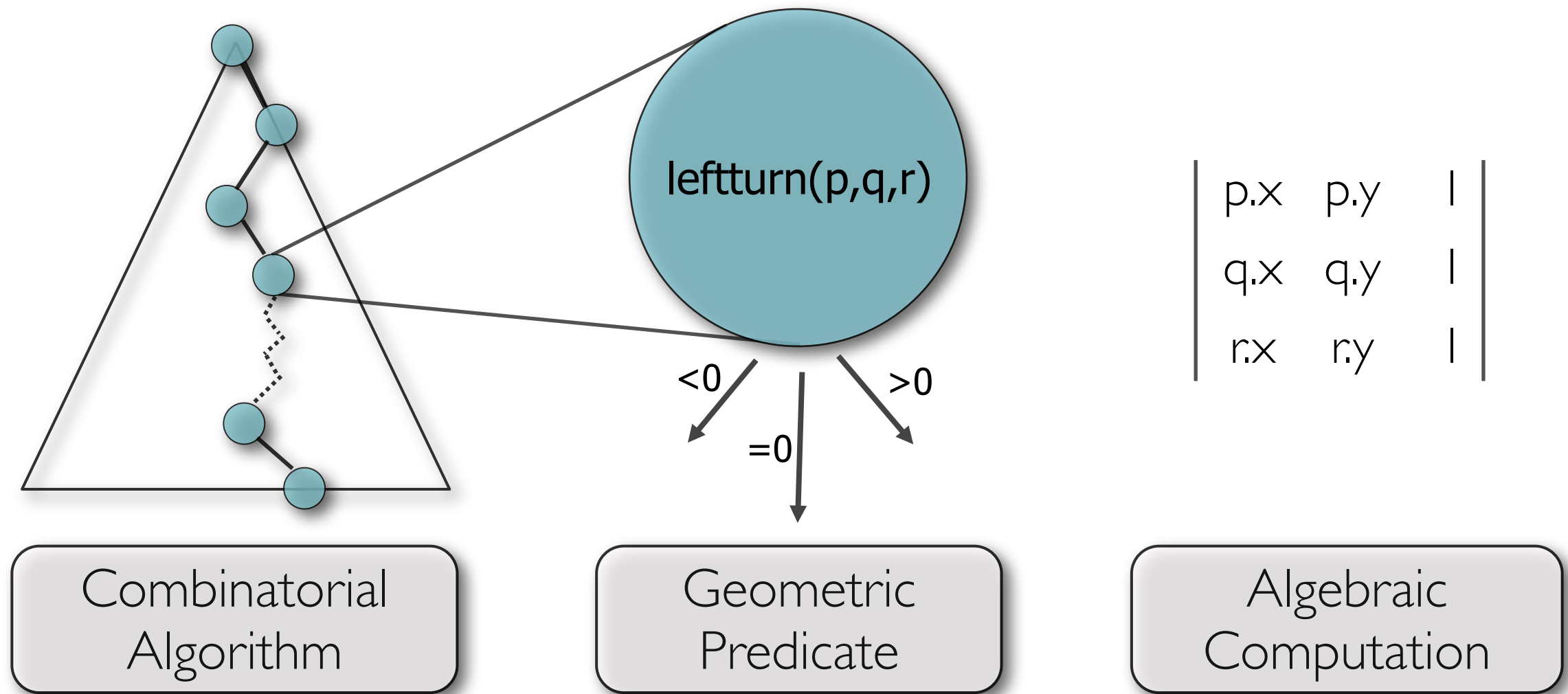
- ▶ 500'000 lines of code (40 man years)
- ▶ 10'000 downloads per year
- ▶ 3'500 manual pages
- ▶ 4'000 subscribers to cgal-announce (7'000 for gcc)
- ▶ 1'400 subscribers to cgal-discuss (600 in gcc-help)
- ▶ 120 components
- ▶ 80 commercialization licenses sold
- ▶ 24 Master Theses and 22 PhD Theses
- ▶ 20 active developers



# PART II:

# Exact Geometric Computing

# LAYERS OF GEOMETRIC ALGORITHMS



Control flow depends on non-trivial algebraic computations.  
How to do these efficiently and consistently?  
(Tough, no universally applicable solution...)

# ARITHMETIC

All operations beyond  $+$  and  $-$  are computed using limited precision floating point arithmetic.

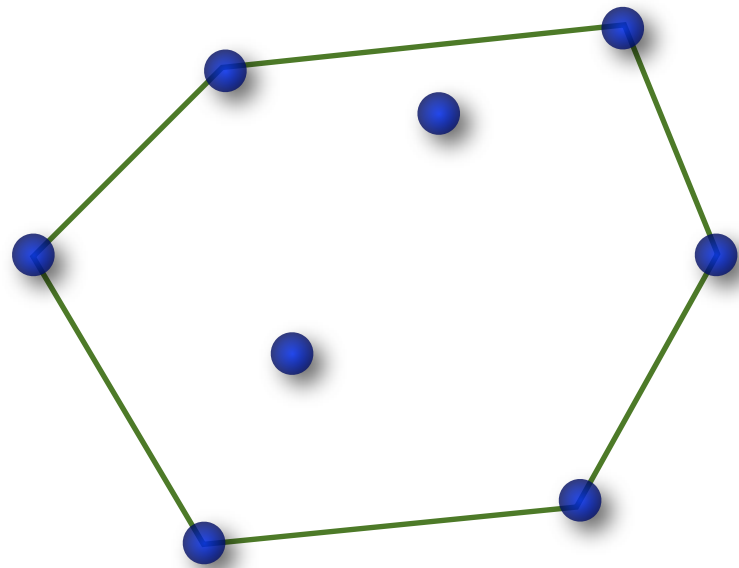
Integer multiplication and division are usually slower, often considerably. And the precision is limited regardless...

➔ Results may be **incorrect** due to roundoff.

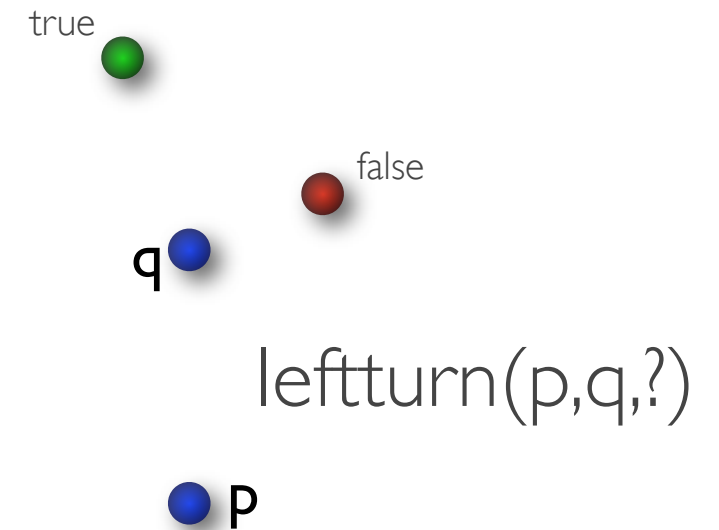
Difference to numeric computing:  
Results are interpreted combinatorially: yes or no.

Incorrect results often lead to a **complete failure** rather than to a reasonable approximation.

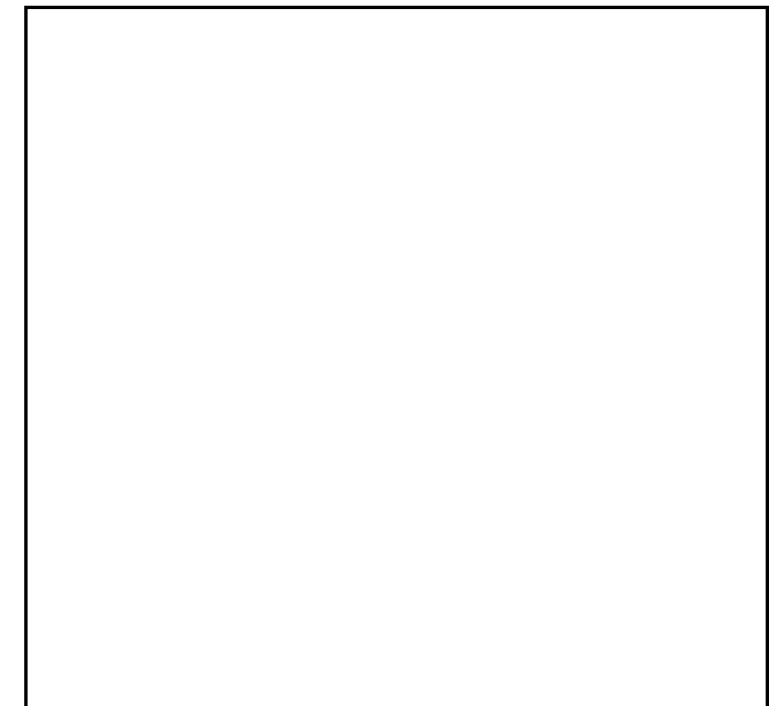
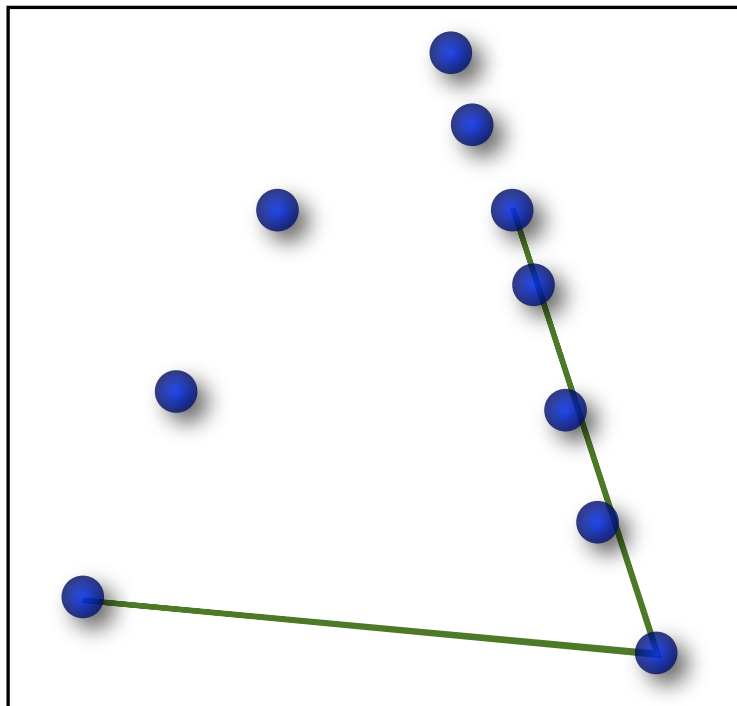
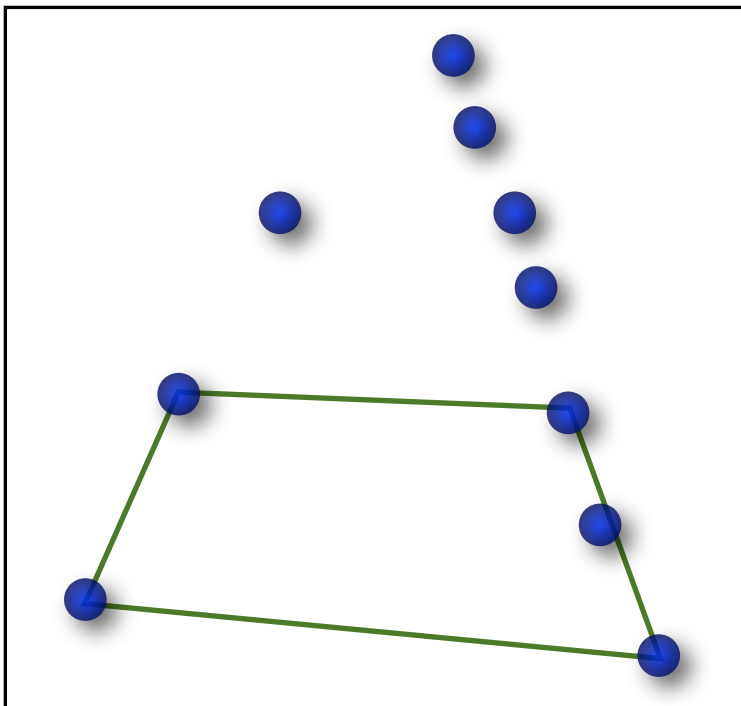
# CONVEX HULL



Based on  
orientation test.



Possible results with an unreliable orientation test:



# STRAIGHT LINES ?

$$\text{Orientation}(p, q, r) = \begin{vmatrix} p.x & p.y & 1 \\ q.x & q.y & 1 \\ r.x & r.y & 1 \end{vmatrix} = (q.x - p.x)(r.y - p.y) - (q.y - p.y)(r.x - p.x)$$

$$p = (0.5 + x \cdot u, 0.5 + y \cdot u)$$

$$q = (12, 12)$$

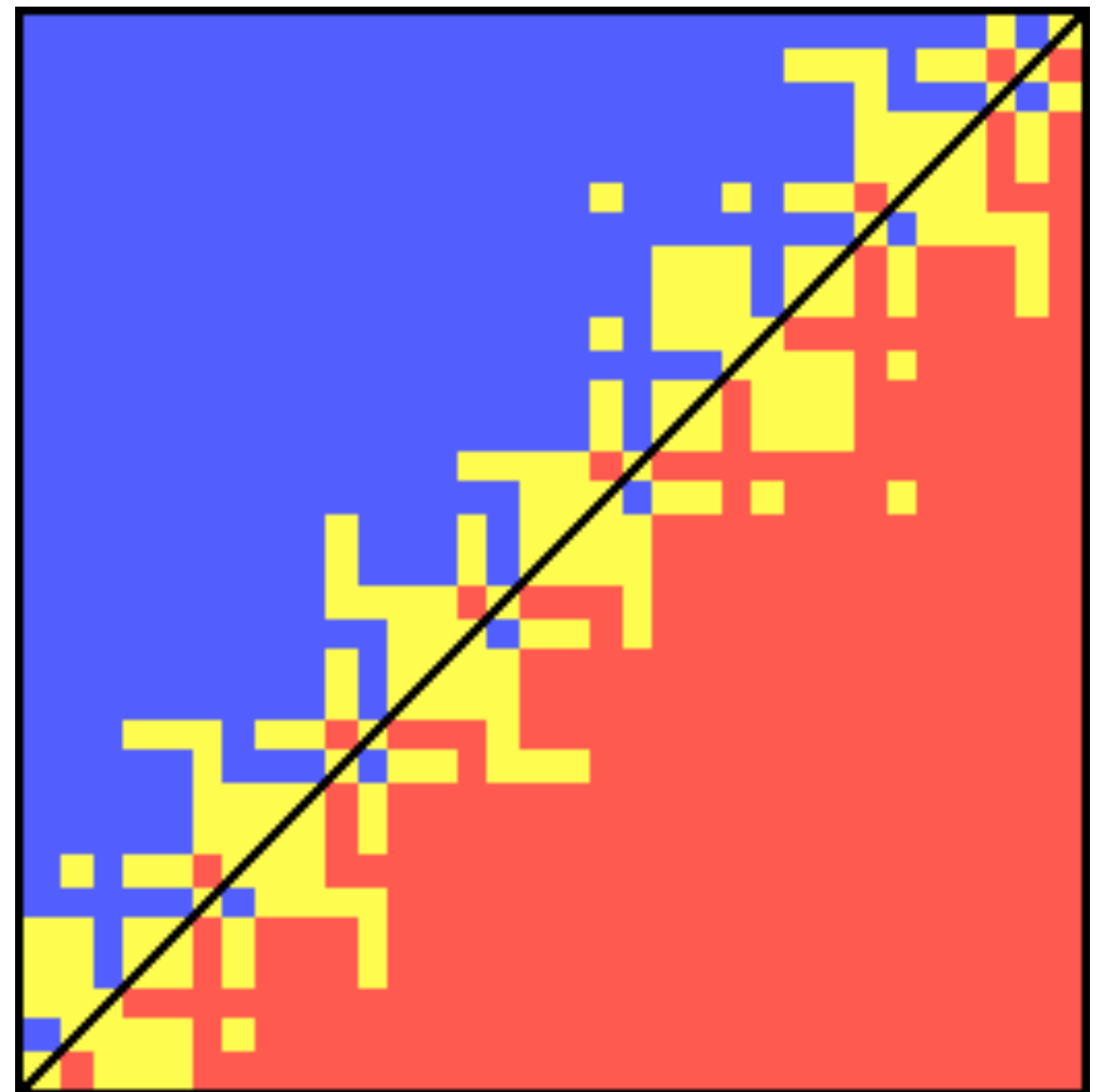
$$r = (24, 24)$$

$$0 \leq x, y < 256, u = 2^{-53}$$

256x256 pixel image

**red**: <0, **yellow**: =0, **blue**: >0

evaluated with **double**



# STRAIGHT LINES ?

$$\text{Orientation}(p, q, r) = \begin{vmatrix} p.x & p.y & 1 \\ q.x & q.y & 1 \\ r.x & r.y & 1 \end{vmatrix} = (q.x - p.x)(r.y - p.y) - (q.y - p.y)(r.x - p.x)$$

$$p = (0.5 + x \cdot u, 0.5 + y \cdot u)$$

$$q = (8.80000000000000000000007, \\ 8.80000000000000000000007)$$

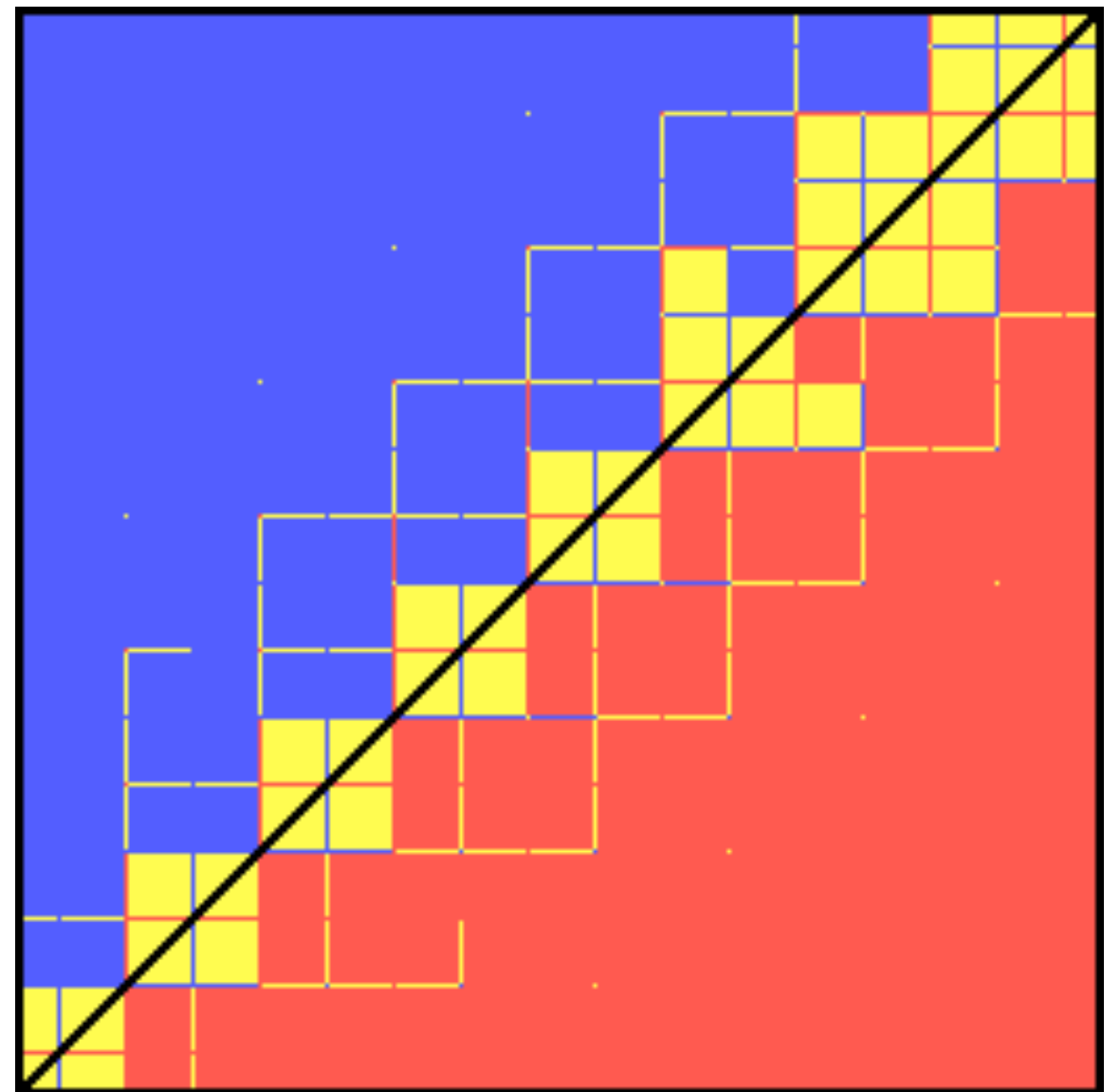
$$r = (12.1, 12.1)$$

$$0 \leq x, y < 256, u = 2^{-53}$$

256x256 pixel image

**red**: <0, **yellow**: =0, **blue**: >0

evaluated with **double**





# STRAIGHT LINES ?

$$\text{Orientation}(p, q, r) = \begin{vmatrix} p.x & p.y & 1 \\ q.x & q.y & 1 \\ r.x & r.y & 1 \end{vmatrix} = (q.x - p.x)(r.y - p.y) - (q.y - p.y)(r.x - p.x)$$

$$p = (0.5000000000000000002531 + x \cdot u, \\ 0.5000000000000000001710 + y \cdot u)$$

$$q = (17.30000000000000000001, \\ 17.30000000000000000001)$$

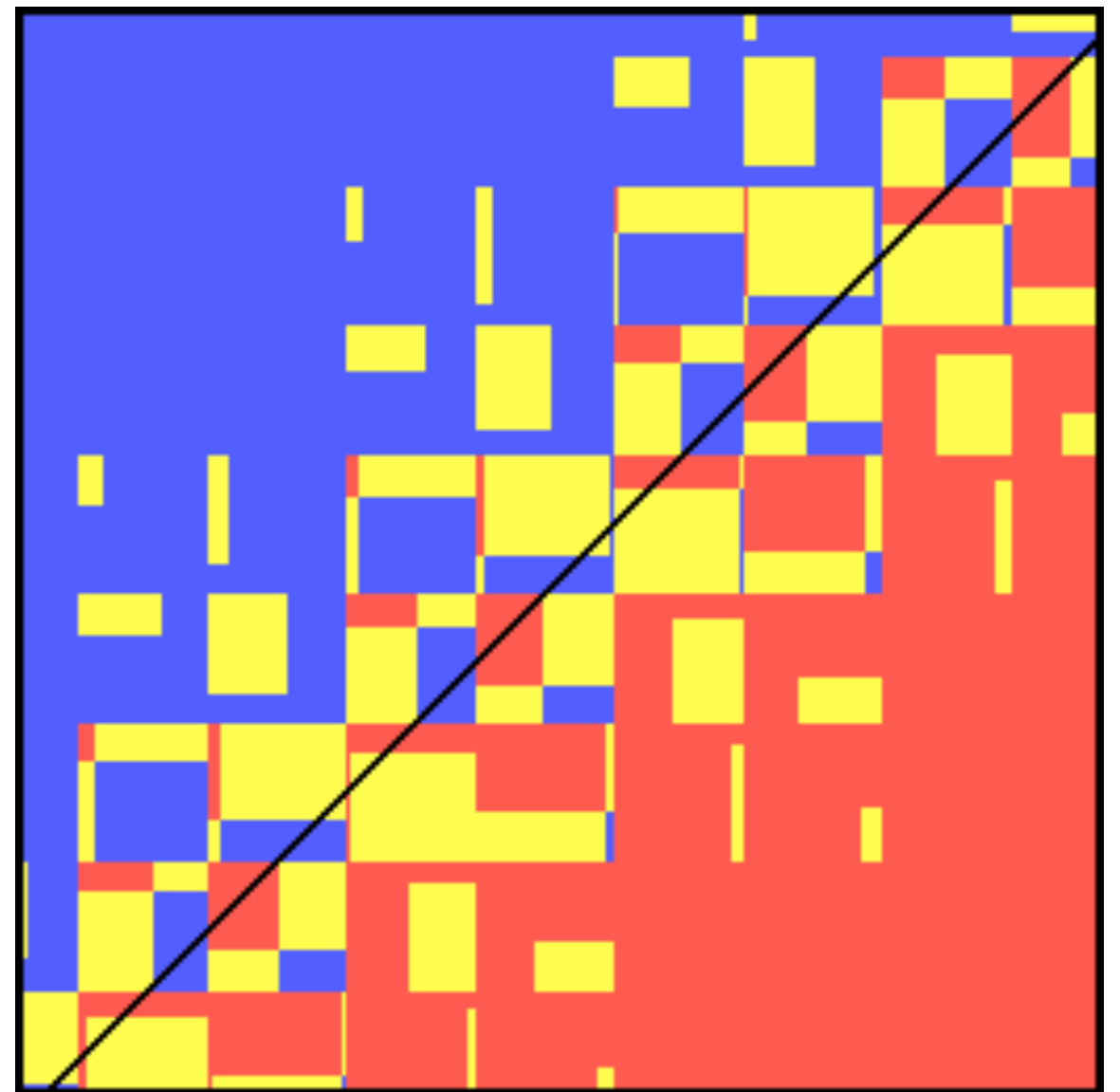
$$r = (24.000000000000000000500000, \\ 24.00000000000000000517765)$$

$$0 \leq x, y < 256, u = 2^{-53}$$

256x256 pixel image

**red**: <0, **yellow**: =0, **blue**: >0

evaluated with **ext double**



# HOW TO OBTAIN CORRECTNESS?

Several options:

▶ Hope things go fine  and fiddle around if not

Sometimes possible, often hard,  
always messy. Very problem-  
specific, no general machinery.

▶ Adapt algorithm to cope with imprecisions 

▶ Restrict input

Good in special cases, hard to impossible  
for general purpose implementations .  
Document and check properly!

▶ Use exact algebra

General approach. Easy to use.  
Can be very slow...

▶ Filtering: Check whether things go fine and use  
exact algebra only when needed.

General approach. Easy to use.  
Often quite efficient...

# FLOATING POINT NUMBERS

IEEE 754 double precision

+/-	exponent	mantissa
1 bit	11 bits	53 bits

0.1 is not exactly representable

Numbers  $\pm m \cdot 2^x$ ,  $0 \leq m < 2^{53}$ ,  $-1022 \leq x \leq 1023$ .

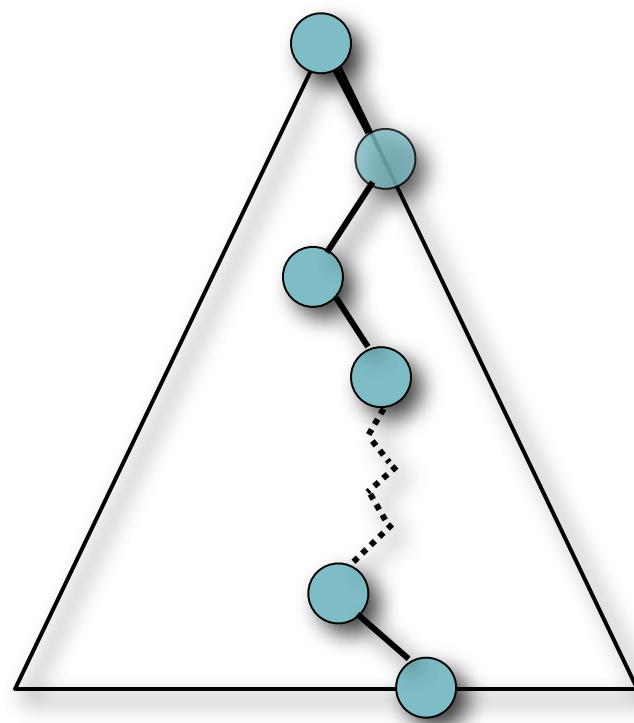
b bits	$\pm$	b bits	$\approx$	b+1 bits
b bits	$\cdot$	b bits	$\approx$	2b bits

$$(q.x-p.x)(r.y-p.y)-(q.y-p.y)(r.x-p.x)$$

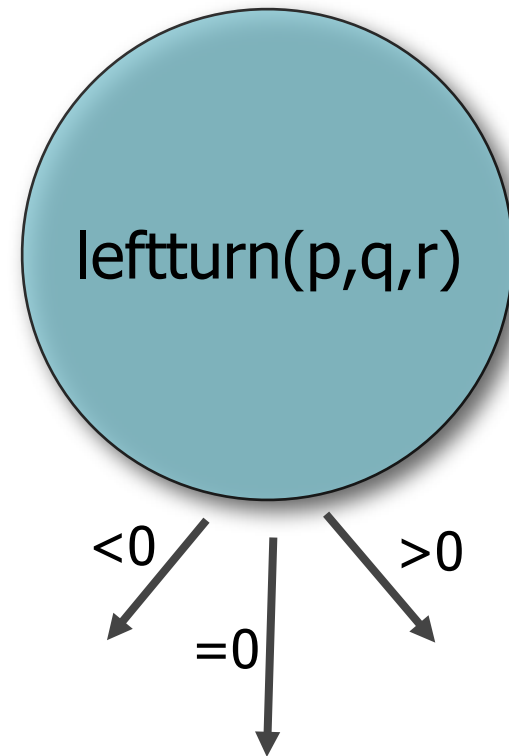


orientation test  $\approx 2b+3$  bits, can be done exactly for 25-bit integer coordinates.

# EXACT COMPUTATION



Combinatorial  
Algorithm

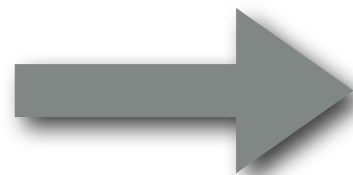


Geometric  
Predicate

$$\begin{vmatrix} p.x & p.y & 1 \\ q.x & q.y & 1 \\ r.x & r.y & 1 \end{vmatrix}$$

Algebraic  
Computation

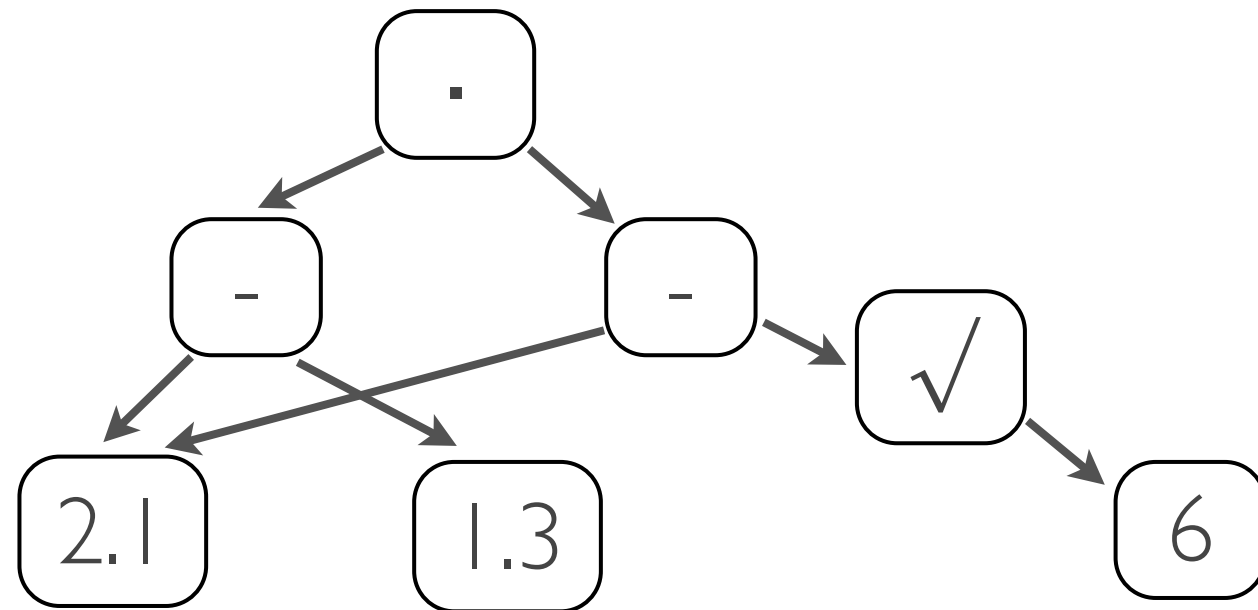
Ensure that the control flow in the algorithm is the same, as if all algebraic computations were made exactly.



Correctness

# EXACT ALGEBRAIC COMPUTATION

$$(2.1 - 1.3)(2.1 - \sqrt{6})$$



- ▶ numbers represented as expression-dags
- ▶ arbitrary precision floating point data types (array of digits) to compute approximations
- ▶  $\text{sign}(x)$ : compute finer and finer approximations for  $x$ , until it becomes clear that  $x > 0$  or  $x < 0$ ;
- ▶ for any algebraic expression there is a *separation bound* that tells where to stop and conclude  $x = 0$ .

# FLOATING POINT FILTERS

Exact algebraic computation is expensive.

→ use when absolutely necessary only.

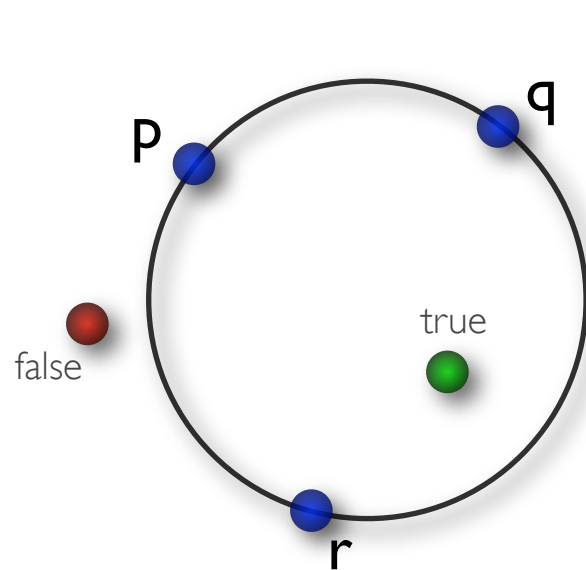
- ▶ maintain double approximation  $[l, h]$  using interval arithmetic (hardware support  $\Rightarrow$  fast)
- ▶ if  $0 \notin [l, h]$ , this is good enough to decide about sign.
- ▶ use exact machinery only if  $0 \in [l, h]$ .

Minimal overhead as long as filter works.

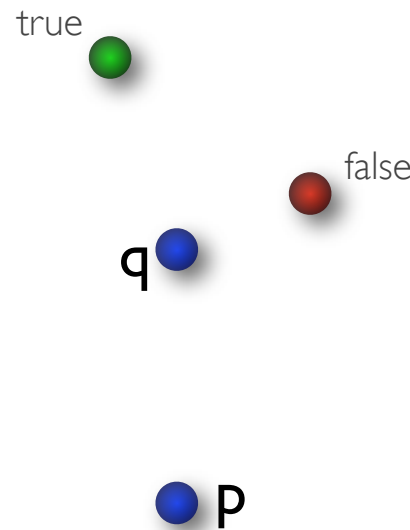
In particular, if only predicates are used and no constructions.



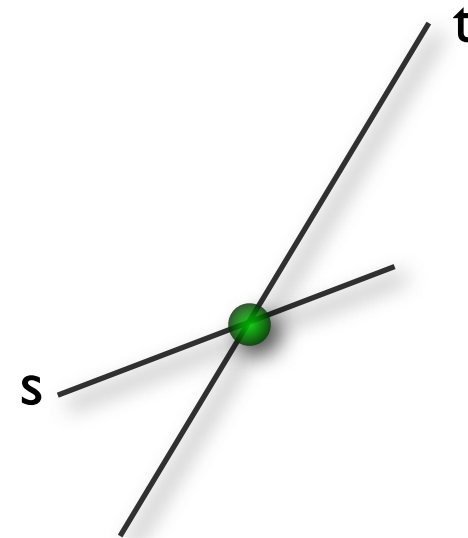
# GEOMETRIC OPERATIONS



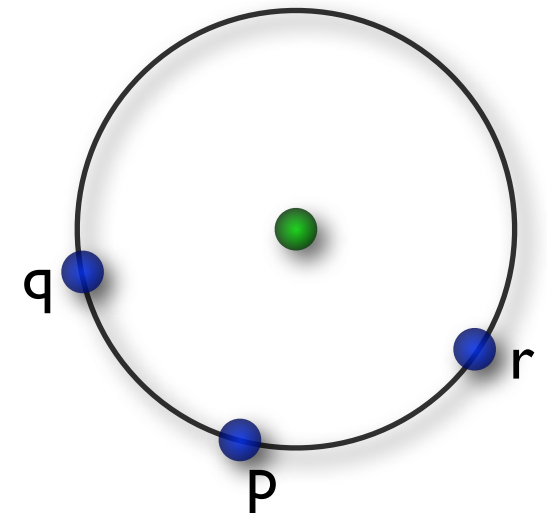
`incircle(p,q,r,?)`



`leftturn(p,q,?)`



`intersection(s,t)`



`circumcircle(p,q,r)`

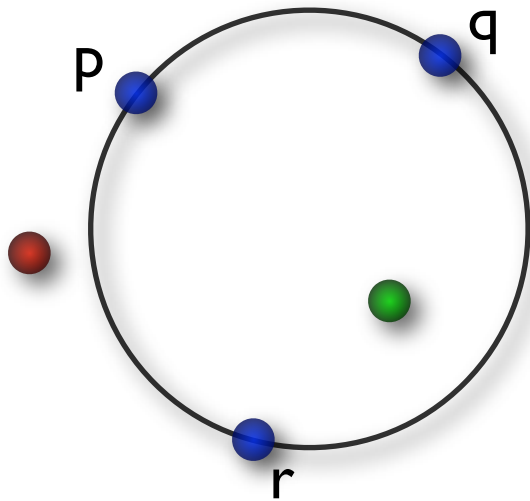
Predicates

Constructions

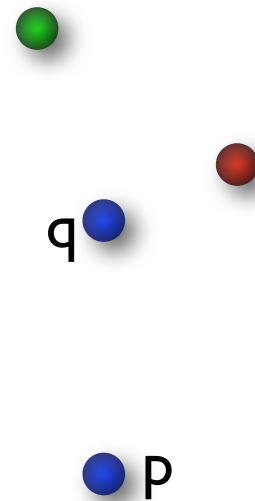


Do you need (exact) constructions?

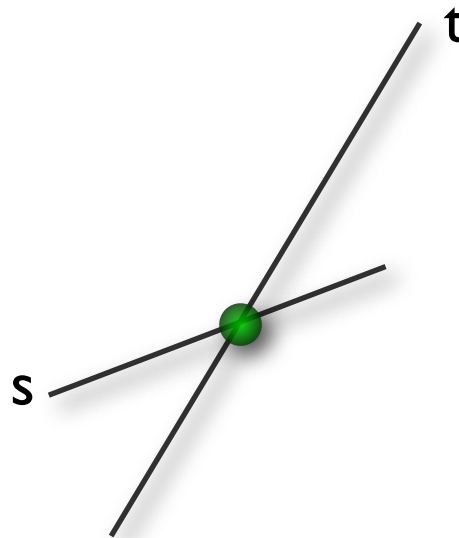
# GEOMETRIC OPERATIONS



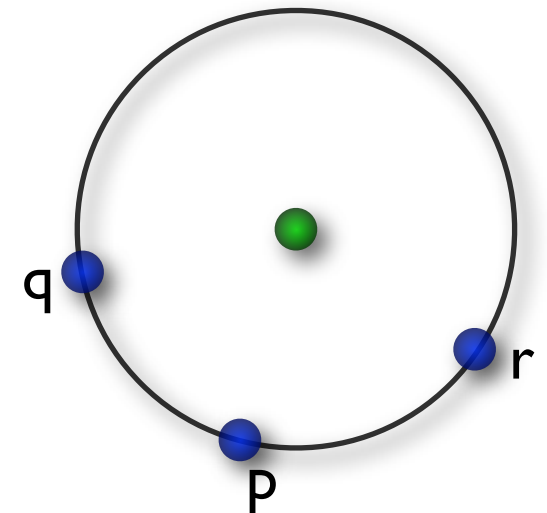
$\text{incircle}(p,q,r,?)$



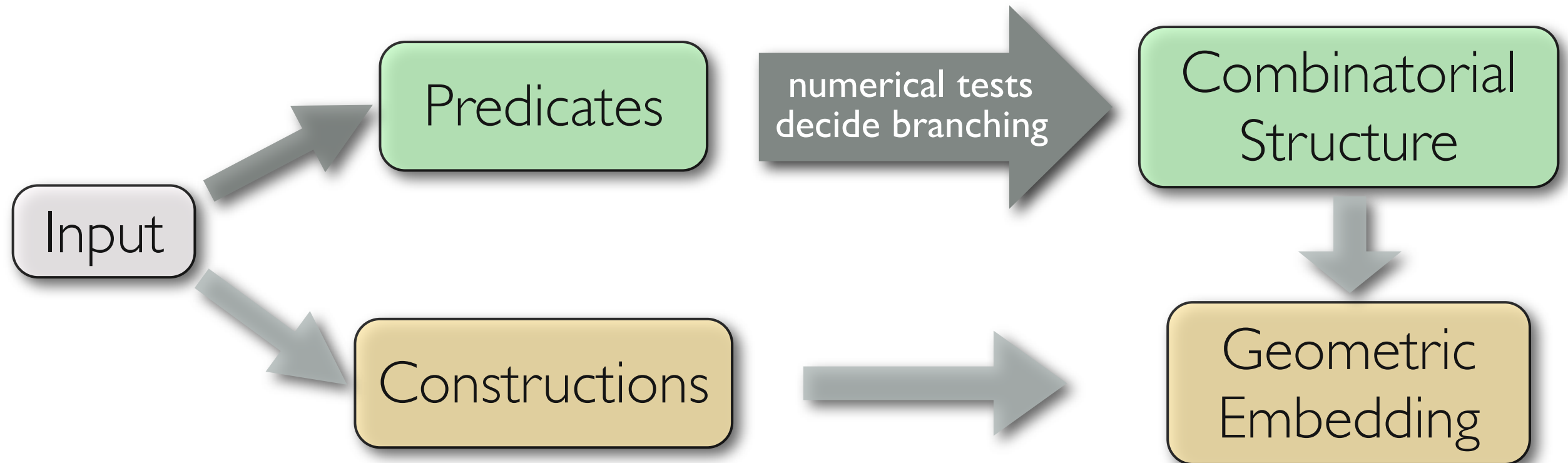
$\text{leftturn}(p,q,?)$



$\text{intersection}(s,t)$



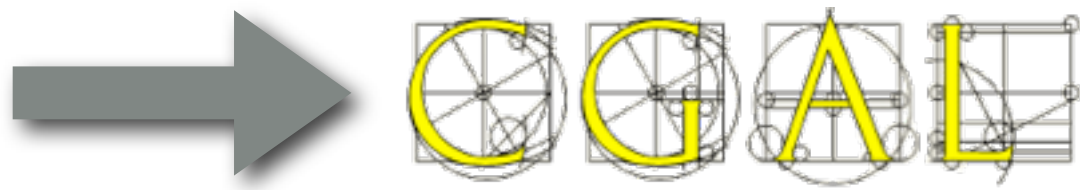
$\text{circumcircle}(p,q,r)$



# FLEXIBILITY

Collection of geometric data types and operations.

There is no single true way to do geometric computing.



offers different kernels to serve various needs

You have to choose the right one for your particular case.

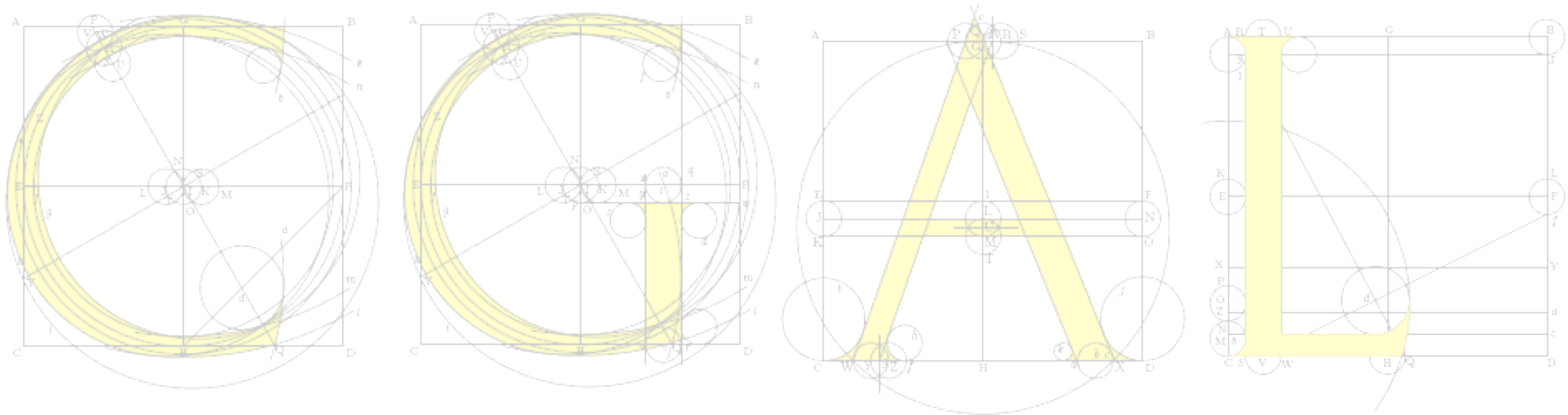
Predefined defaults:

All three compute predicates exactly using filters for efficiency.

- ▶ `CGAL::Exact_predicates_inexact_constructions_kernel`  
Constructions use `double`.
- ▶ `CGAL::Exact_predicates_exact_constructions_kernel`  
Constructions use an exact number type supporting `+, -, *, /`.
- ▶ `CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt`  
Constructions use an exact number type supporting `+, -, *, /`, and roots.

fast

slow



# PART III:

## Basic Programming using a CGAL Kernel

# HELLO POINT

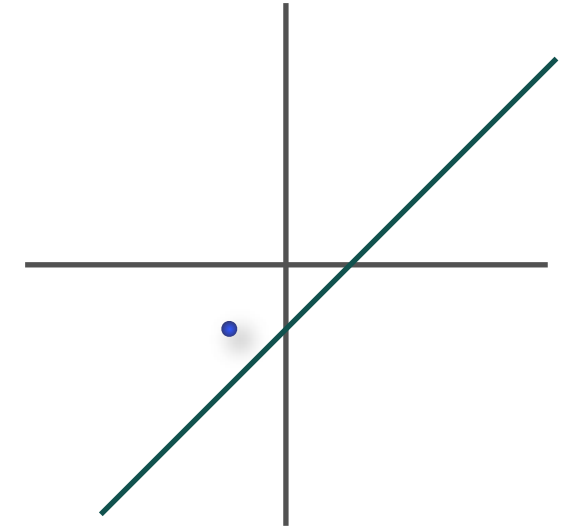
```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
```

```
int main()
{
    K::Point_2 p(2,1), q(1,0), r(-1,-1);
    K::Line_2 l(p,q);
    K::FT d = CGAL::squared_distance(r,l);
    std::cout << d << std::endl;
}
```

There is a bunch of hyperlinks here.  
Click me to get to the CGAL manual.

Does this code use  
constructions?  
YES!



Output: 0.5

FT = field type

The number type used for the  
underlying algebra. Supports all  
field operations, i.e., +-\*./.

Some (few) field types also support exact roots.

avoids square root computation

To obtain an approximation of the real distance, use

```
std::sqrt(CGAL::to_double(CGAL::squared_distance(r,l)))
```

This function must be defined for any field type.

Even if the field type supports exact square roots, in order to  
output it numerically you have to resort to an approximation...

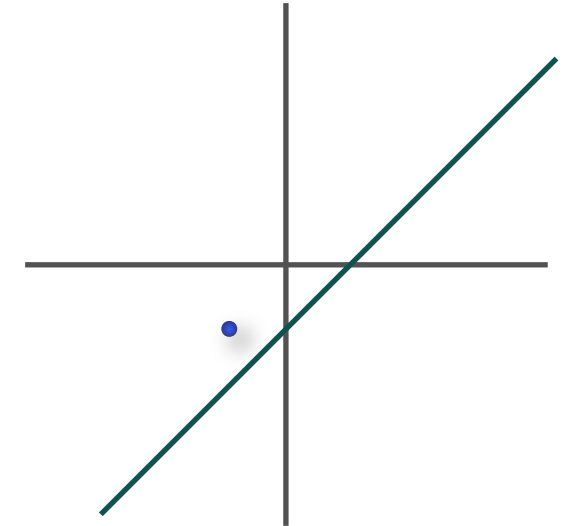
[https://elabs.inf.ethz.ch/file.php/29/CGALWeek1/Sample\\_Programs/hello.cpp](https://elabs.inf.ethz.ch/file.php/29/CGALWeek1/Sample_Programs/hello.cpp)

# HELLO POINT

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

int main()
{
    K::Point_2 p(2,1), q(1,0), r(-1,-1);
    K::Line_2 l(p,q);
    K::FT d = CGAL::squared_distance(r,l);
    std::cout << d << std::endl;
}
```



For the small coordinates used here, things are probably fine. But in general...

this code is not safe!

Constructing a line from two points.  
Trivial?

Depends on representation of lines... equation => non-trivial construction

Constructing a point from Cartesian double coordinates. All default kernels can do this exactly, by just storing the coordinates.  
=> trivial construction, no problem

Also a non-trivial construction.  
(Squared distance may be considerably larger than input coordinates, which may lead to overflow.)

## CGAL::Line\_2<Kernel>

### Definition

An object  $l$  of the data type `Line_2<Kernel>` is a directed straight line in the two-dimensional Euclidean plane  $\mathbb{E}^2$ . It is defined by the set of points with Cartesian coordinates  $(x,y)$  that satisfy the equation  $l : ax + by + c = 0$

The line splits  $\mathbb{E}^2$  in a *positive* and a *negative* side. A point  $p$  with Cartesian coordinates  $(px, py)$  is on the positive side of  $l$ , iff  $apx + bpy + c > 0$ , it is on the negative side of  $l$ , iff  $apx + bpy + c < 0$ . The positive side is to the left of  $l$ .

Class



# HELLO POINT (EXACTLY)

```
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <iostream>
#include <iomanip>
```

```
typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;
```

```
int main()
```

```
{
```

```
    K::Point_2 p(2,1), q(1,0), r(-1,-1);
```

```
    K::Line_2 l(p,q);
```

```
    K::FT d = sqrt(CGAL::squared_distance(r,l));
```

```
    std::cout << CGAL::to_double(d) << std::endl;
```

```
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(2)
    << CGAL::to_double(d) << std::endl;
```

```
}
```

Set precision (number of digits after the decimal point) for floating point number output. Round to nearest, but tie-breaking is not well defined!

Output:

0.707107

0.71

Compute squareroot (here: exactly).

Round to some **double** nearby.  
(There is no easy way to output the exact internal representation.)

**Problem:** No guarantee on precision and rounding.

Output floating point numbers in fixed point notation from now on.  
`std::resetiosflags(std::ios::fixed)` switches back to default behaviour.

# HELLO POINT (EVEN MORE EXACTLY)

```
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <iostream>
#include <cmath> ← for std::floor(...)

typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;

double floor_to_double(const K::FT& x)
{
    double a = std::floor(CGAL::to_double(x)); ← Compute approximation of the
    while (a > x) a -= 1;                       closest integer ≤ x.
    while (a+1 <= x) a += 1;                     (Usually, this is pretty good. But we
    return a;                                    cannot be sure that it is always...)
}

int main()
{
    K::Point_2 p(2,1), q(1,0), r(-1,-1);
    K::Line_2 l(p,q);
    K::FT d = sqrt(CGAL::squared_distance(r,l)); ← Compare to the exact
    std::cout << floor_to_double(d) << std::endl;    value to be sure.
}                                                    Compute squareroot exactly.
```

Output:

0

We need a precise specification for all output, in order to compare on the judge.

This is the recommended way to round down to an integer.

(The symmetric function `ceil_to_double(...)` to round up should be an easy exercise...)

# TWO KERNELS IN ONE PROGRAM

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <stdexcept>
```

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel IK;
typedef CGAL::Exact_predicates_exact_constructions_kernel EK;
```

```
int main()
```

```
{
```

```
    IK::Point_2 p(2,1), q(1,0), r(-1,-1);
```

```
    // do something that needs predicates only, e.g., ...
```

```
    std::cout << (CGAL::left_turn(p, q, r) ? "y" : "n") << "\n";
```

```
    // now we use non-trivial constructions...
```

```
    EK::Point_2 ep(p.x(), p.y()), eq(q.x(), q.y()), er(r.x(), r.y());
```

```
    EK::Circle_2 c(ep, eq, er);
```

```
    if (!c.has_on_boundary(ep))
```

```
        throw std::runtime_error("ep not on c");
```

```
}
```

This works because the coordinates of `IK::Point_2` are actually `double`.

It would not work the other way round, because the coordinates of `EK::Point_2` are of some elaborate number type.

We cannot just write `c(p, q, r)` because these are `IK::Point_2` and there is no general conversion between points from different kernels.

Output:

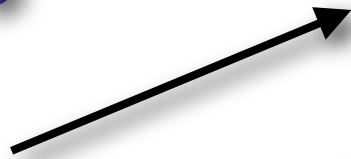
n

# 2D (LINEAR) KERNEL

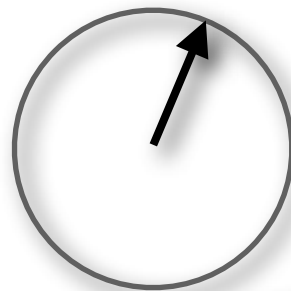
► Point\_2



► Vector\_2



► Direction\_2



► Line\_2

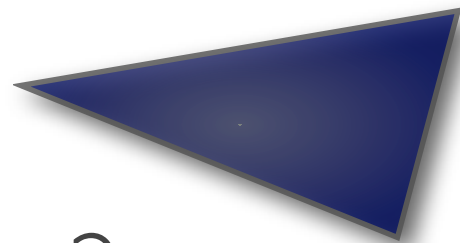
► Ray\_2



► Segment\_2

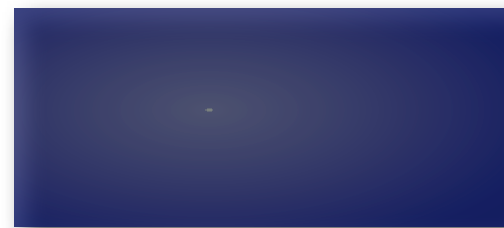
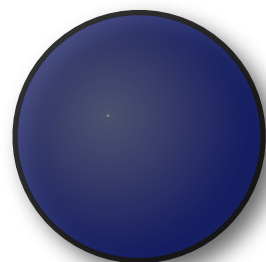


► Triangle\_2



► Iso\_rectangle\_2

► Circle\_2



Follow the links to see the manual.

# 2D KERNEL FUNCTIONALITY

See the  Manual: <http://www.cgal.org>

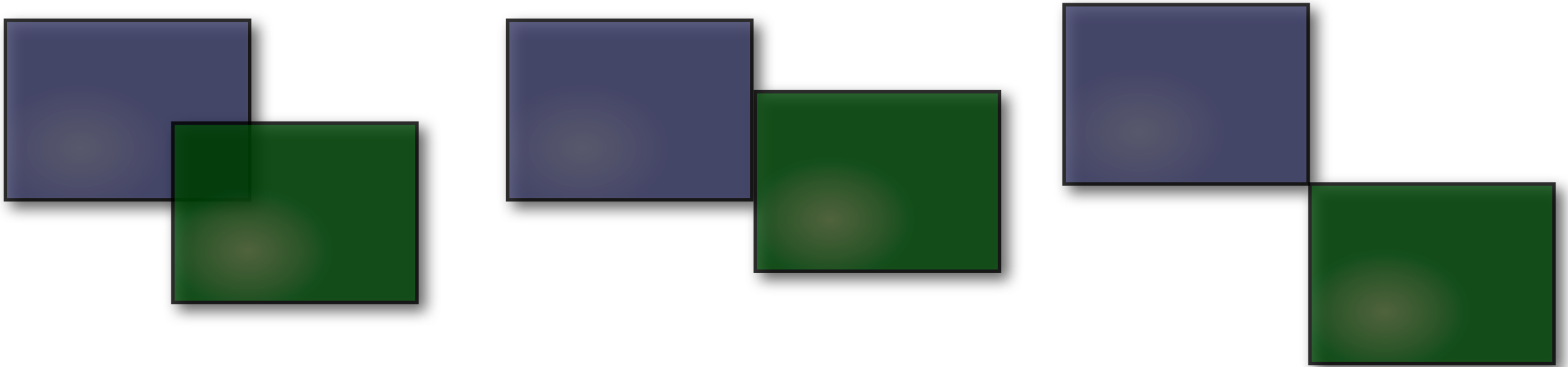
Most manual chapters have two parts:

- ▶ User Manual: general introduction and examples.
- ▶ Reference Manual: complete list of functionality.

Often one deals with several different interacting types and has to jump back and forth.

=> html is very convenient

# INTERSECTIONS



Problem: We do not know the return type.

```
K::Iso_rectangle_2 r1 = ... ;  
K::Iso_rectangle_2 r2 = ... ;  
??? i = CGAL::intersection(r1, r2);
```

You might say that a point is nothing but a degenerate rectangle. Then think about a plane and a line in  $\mathbb{R}^3$ .

Solution: Use a generic class CGAL::Object.

Test whether it contains an object of type  $T$  using `object_cast<T>`.

Note: CGAL::Object is not a common base class but just a generic wrapper.



# INTERSECTIONS

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <stdexcept>
```

```
typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 P;
typedef K::Segment_2 S;
```

```
int main()
{
```

```
    P p[] = { P(0,0), P(2,0), P(1,0), P(3,0), P(.5,1), P(.5,-1) };
```

```
    S s[] = { S(p[0],p[1]), S(p[2],p[3]), S(p[4],p[5]) };
```

```
    for (int i = 0; i < 3; ++i)
```

```
        for (int j = i+1; j < 3; ++j)
```

```
            if (CGAL::do_intersect(s[i],s[j])) {
```

```
                CGAL::Object o = CGAL::intersection(s[i],s[j]);
```

```
                if (const P* op = CGAL::object_cast<P>(&o))
```

```
                    std::cout << "point: " << *op << "\n";
```

```
                else if (const S* os = CGAL::object_cast<S>(&o))
```

```
                    std::cout << "segment: " << os->source() << " "
```

```
                        << os->target() << "\n";
```

```
                else // how could this be? -> error
```

```
                    throw std::runtime_error("strange segment intersection");
```

```
            } else
```

```
                std::cout << "no intersection\n";
```

```
    }
```

Test for intersection (predicate)

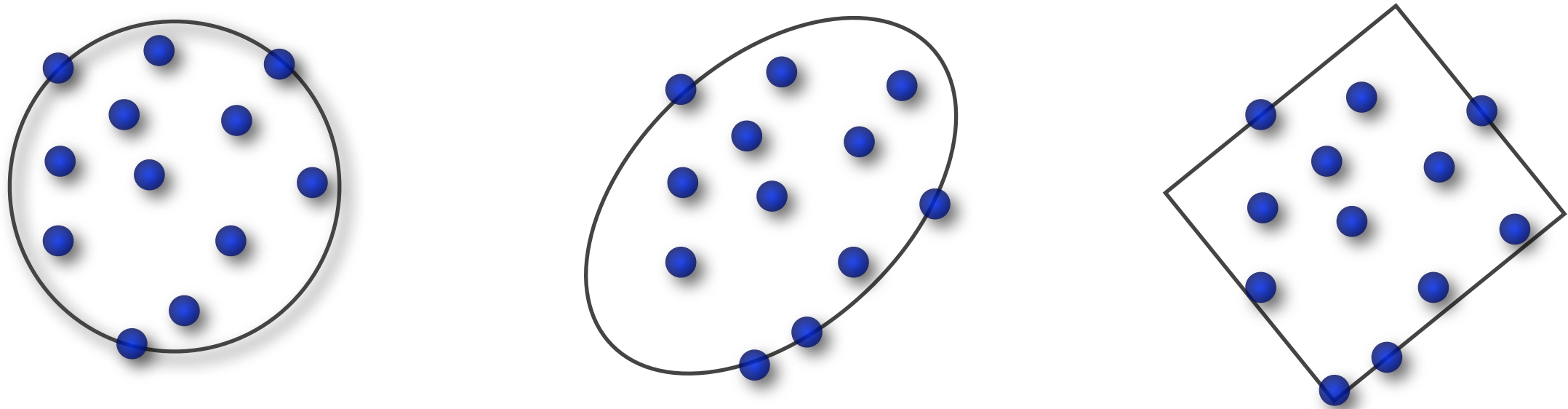
Construct intersection (construction :-)

Cast fails (=0) if o is not of type P.

Output:

```
segment: 1 0 2 0
point: 0.5 0
no intersection
```

# BOUNDING VOLUMES



Problem: Given  $n$  points in  $\mathbb{R}^2$ , what is their minimum enclosing ... ?

- ▶ Circle
- ▶ Ellipse
- ▶ (Circular) annulus
- ▶ Rectangle
- ▶ Parallelogram
- ▶ Strip



Can be computed in expected linear time.



Can be computed in linear time once the convex hull is known.

# MINIMUM ENCLOSING CIRCLE

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>
```

Many data structures and algorithms have their own traits concept. It defines the geometric primitives needed.

```
// typedefs
typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;
```

Separate: Combinatorial algorithm  $\Leftrightarrow$  geometry

```
int main()
```

```
{
```

```
    const int n = 100;
```

```
    K::Point_2 P[n];
```

Build from a range of points.

```
    for (int i = 0; i < n; ++i)
```

```
        P[i] = K::Point_2((i % 2 == 0 ? i : -i), 0);
```

```
    // (0,0), (-1,0), (2,0), (-3,0), ...
```

Randomize input order? Generally a good idea, unless input is known to be random, anyway.

This part needs the incircle predicate only...

```
    Min_circle mc(P, P+n, true);
```

```
    Traits::Circle c = mc.circle();
```

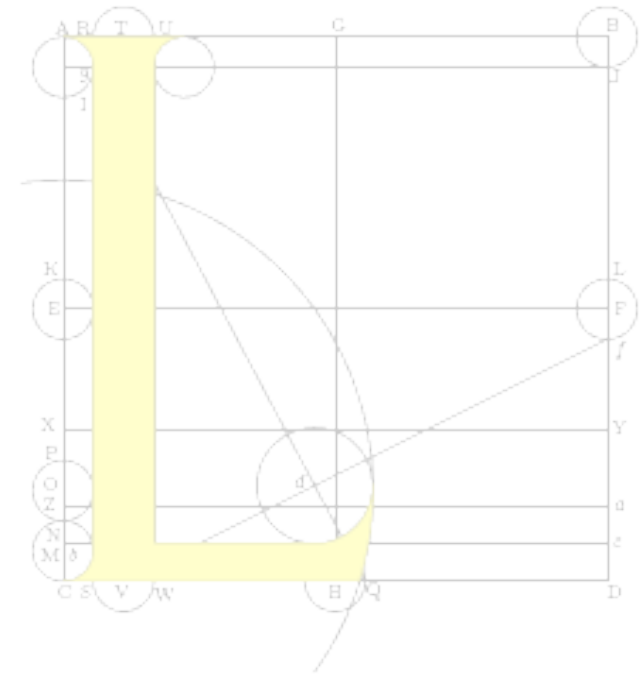
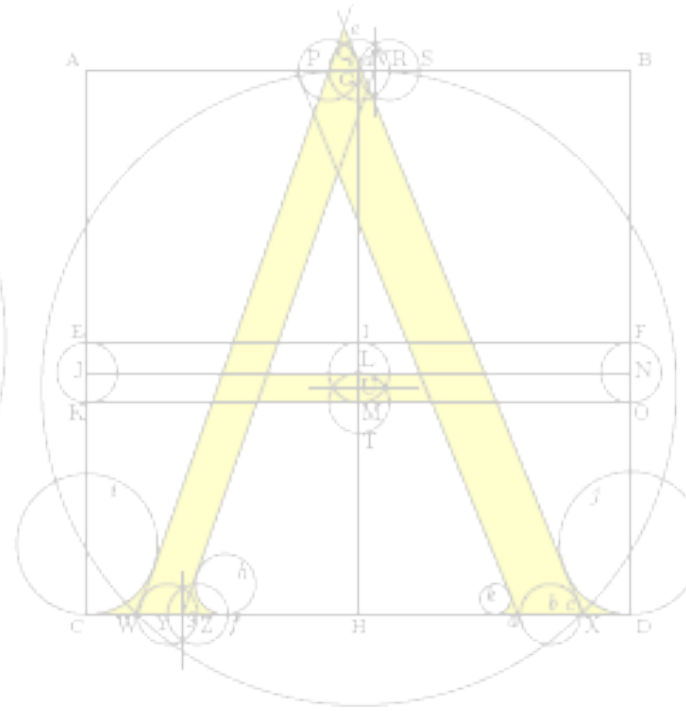
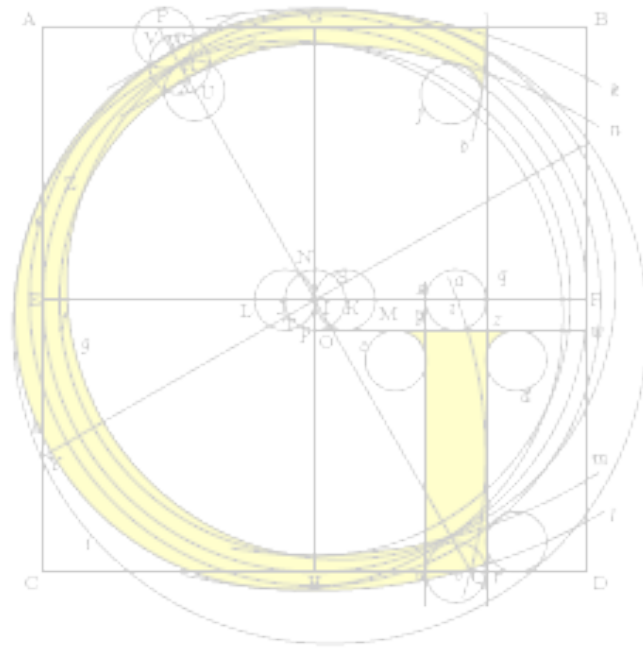
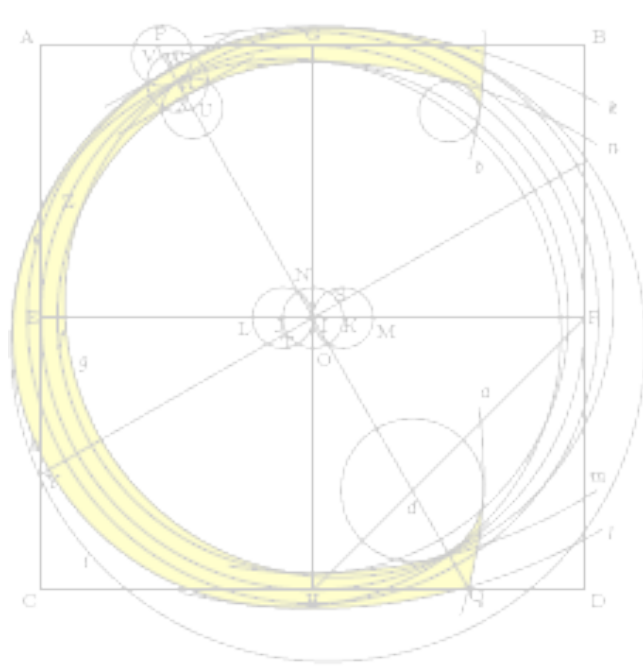
```
    std::cout << c.center() << " " << c.squared_radius() << std::endl;
```

Construct and return the circle.

Only here the construction is needed...

Output:  
-0.5 0 9702.25

[https://elabs.inf.ethz.ch/file.php/29/CGALWeek1/Sample\\_Programs/miniball.cpp](https://elabs.inf.ethz.ch/file.php/29/CGALWeek1/Sample_Programs/miniball.cpp)



# PART IV:

## Practical Information

# USING

Best start in a new directory, name source file s.t. it ends with `.cpp`.

Run **`cgal_create_cmake_script`** in this directory.  
**`cmake .`** ← Note the dot  
(current directory)!

These scripts should be in PATH on the lab PCs  
and the provided VirtualBox environment.

[http://csa.inf.ethz.ch/~trast/vm\\_cgal\\_stuff.html](http://csa.inf.ethz.ch/~trast/vm_cgal_stuff.html)

This creates a makefile with rules and targets for every `.cpp` file.  
You can then build your program using **`make`**

You have to re-run `cgal_create_cmake_script` whenever you add a new application/`.cpp` file.

No need to re-run `cmake` because that's done by `make` automatically.

As a default, makefiles are created in release mode. If you want to debug, run `cmake -DCMAKE_BUILD_TYPE=Debug .`

To go back to release mode, run `cmake -DCMAKE_BUILD_TYPE=Release .`

If you want to see the actual compiler and linker calls, run `cmake -DCMAKE_VERBOSE_MAKEFILE=ON .`

Warning: Do not use `valgrind` with CGAL.

That's it!

If you want to install CGAL on your private computer:

- Check/install prerequisites first: compiler, cmake, boost, gmp, mpfr, (qt)
- Install cgal, c.f.

[http://www.cgal.org/Manual/3.8/doc\\_html/installation\\_manual/contents.html](http://www.cgal.org/Manual/3.8/doc_html/installation_manual/contents.html)

- Or download CGAL packages of your distribution if they exist (don't forget `cgal-devel`).

For more, see...



# HTTP://WWW.CGAL.ORG

CGAL - Computational Geometry Algorithms Library - Mozilla Firefox

Echier Édition Affichage Historique Marque-pages Outils 2

http://www.cgal.org/

Les plus visités G HP Geometria GForge LP LCL cgal semir agenda transfer ijl IzIGFD forms lists

Home | Intranet

**Software**

- Overview
- Online Manual
- Tutorials
- All Manuals
- Download
- License
- The CGAL Philosophy
- Acknowledging CGAL
- Work in Progress
- Release History

**Support**

- FAQ
- Supported Platforms
- Reporting Bugs
- Mailing Lists

**Project**

- Project Members
- Getting Involved
- Project Rules
- Partners and Funding

**More information**

- Videos
- Events
- Classes

**Other Resources**

- Python Bindings
- CGAL-Ipelets
- Scilab Geometry Toolbox
- Projects Using CGAL
- 3rd Party Software
- Related Links

Google

www cgal.org

Terminé

**Computational Geometry Algorithms Library**

The goal of the CGAL Open Source Project is to provide easy access to *efficient and reliable geometric algorithms* in the form of a C++ library. CGAL is used in various areas needing geometric computation, such as: computer graphics, scientific visualization, computer aided design and modeling, geographic information systems, molecular biology, medical imaging, robotics and motion planning, mesh generation, numerical methods... More on the [projects using CGAL](#) web page.

The Computational Geometry Algorithms Library (CGAL), offers data structures and algorithms like [triangulations](#) (2D constrained triangulations and Delaunay triangulations in 2D and 3D), [Voronoi diagrams](#) (for 2D and 3D points, 2D additively weighted Voronoi diagrams, and segment Voronoi diagrams), [Boolean operations](#) on polygons and polyhedra, [arrangements of curves and their applications](#) (2D and 3D envelopes, Minkowski sums), [mesh generation](#) (2D Delaunay mesh generation and 3D surface mesh generation, skin surfaces), [geometry processing](#) (surface mesh simplification, subdivision and parameterization, as well as estimation of local differential properties, and approximation of ridges and umbilics), [alpha shapes](#), [convex hull algorithms](#) (in 2D, 3D and dD), [operations on polygons](#) (straight skeleton and offset polygon), [search structures](#) (kd trees for nearest neighbor search, and range and segment trees), [interpolation](#) (natural neighbor interpolation and placement of streamlines), [shape analysis](#), [fitting](#), and [distances](#) (smallest enclosing sphere of points or spheres, smallest enclosing ellipsoid of points, principal component analysis), and [kinetic data structures](#).

All these data structures and algorithms operate on geometric objects like points and segments, and perform geometric tests on them. These objects and predicates are regrouped in CGAL [Kernels](#).

Finally, the [Support Library](#) offers geometric object generators and spatial sorting functions, as well as a matrix search framework and a solver for linear and quadratic programs. It further offers interfaces to third party software such as the GUI libraries Qt, Geomview, and the Boost Graph Library.

**License**


CGAL is distributed under a dual-license scheme. CGAL can be used together with Open Source software free of charge. Using CGAL in other contexts can be done by obtaining a commercial license from [GeometryFactory](#). For more details see the

3D Polyhedral Surfaces - Mozilla Firefox

http://www.cgal.org/manual/3.4/doc/packages/polyhedron.html

Les plus visités G HP Geometria GForge LP LCL cgal semir agenda transfer ijl IzIGFD forms lists


Polychedral surfaces in three dimensions are composed of vertices, edges, facets and an incidence relationship on them. The organization beneath is a halfedge data structure, which restricts the class of representable surfaces to orientable 2-manifolds - with and without boundary. If the surface is closed we call it a polyhedron. For example, see the following model of a hammerhead.



The polyhedral surface is realized as a container class that manages vertices, halfedges, facets with their incidences, and that maintains the combinatorial integrity of them. It is based on the highly flexible design of the halfedge data structure, see the introduction in Chapter 22 and [\[Se99\]](#). However, the polyhedral surface can be used and understood without knowing the underlying design. Some of the examples in this chapter introduce also gradually into first applications of this flexibility.

**21.2 Definition**

A polyhedral surface `CGAL::Polyhedron_3` in three dimensions consists of vertices  $V$ , edges  $E$ , facets  $F$  and an incidence relation on them. Each edge is represented by two halfedges with opposite orientations. The incidences stored with a halfedge are illustrated in the following figure.



Terminé

User Manual Reference Manual

**Package Overview - Mozilla Firefox**

http://www.cgal.org

Les plus visités G HP Geometria GForge

**2D Boolean Operations on Nef Polygons**

Michael Seel

A Nef polygon is any set that can be obtained from a finite set of open halfspaces by set complement and set intersection operations. Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polygons are also closed under those operations. Apart from the set complement operation there are more topological unary set operations that are closed in the domain of Nef polygons interior, boundary, and closure.

**2D Boolean Operations on Nef Polygons Embedded on the Sphere**

Peter Hachenberger, Lutz Kettner, and Michael Seel

This package offers the equivalent to 2D Nef Polygons in the plane. Here halfplanes correspond to half spheres delimited by great circles.

**2D Polygon Partitioning**

Susan Hert

This package provides functions for partitioning polygons in monotone or convex polygons. The algorithms can produce results with the minimal number of polygons, as well as approximations which have no more than four times the optimal number of convex pieces but they differ in their runtime complexities.

**Introduced in:** CGAL 2.3  
**License:** GPL  
**BibTeX Key:** cgal/s-bonp2-08  
**Demo:** 2D Nef Polygons  
User Manual Reference Manual

**Introduced in:** CGAL 3.1  
**Depends on:** 2D Nef Polygon  
**BibTeX Key:** cgal/hk-bonpes2-08  
**Demo:** 2D Polygon Partitioning  
User Manual Reference Manual

**Introduced in:** CGAL 2.3  
**License:** GPL  
**BibTeX Key:** cgal/hk-bonpes2-08  
**Demo:** 2D Polygon Partitioning  
User Manual Reference Manual

Terminé