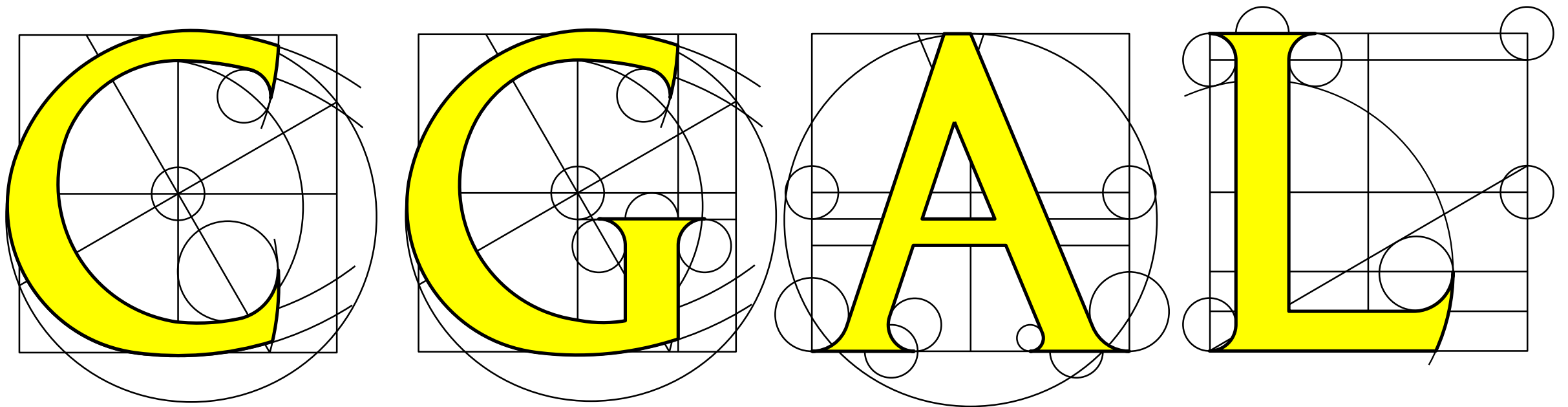


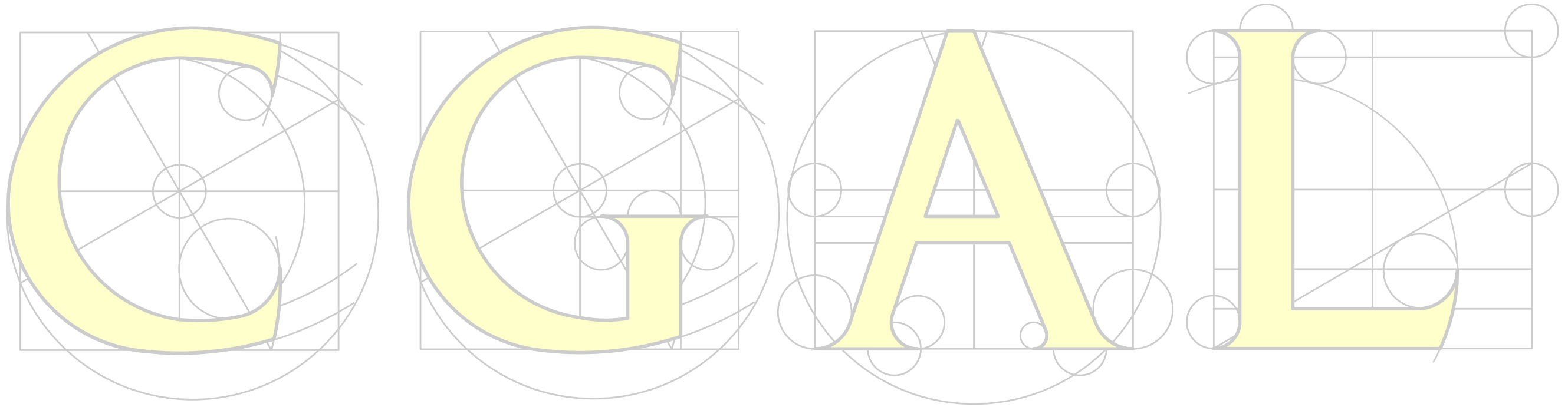
A VERY SHORT INTRODUCTION TO



The Computational Geometry Algorithms Library

Michael Hoffmann <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.)

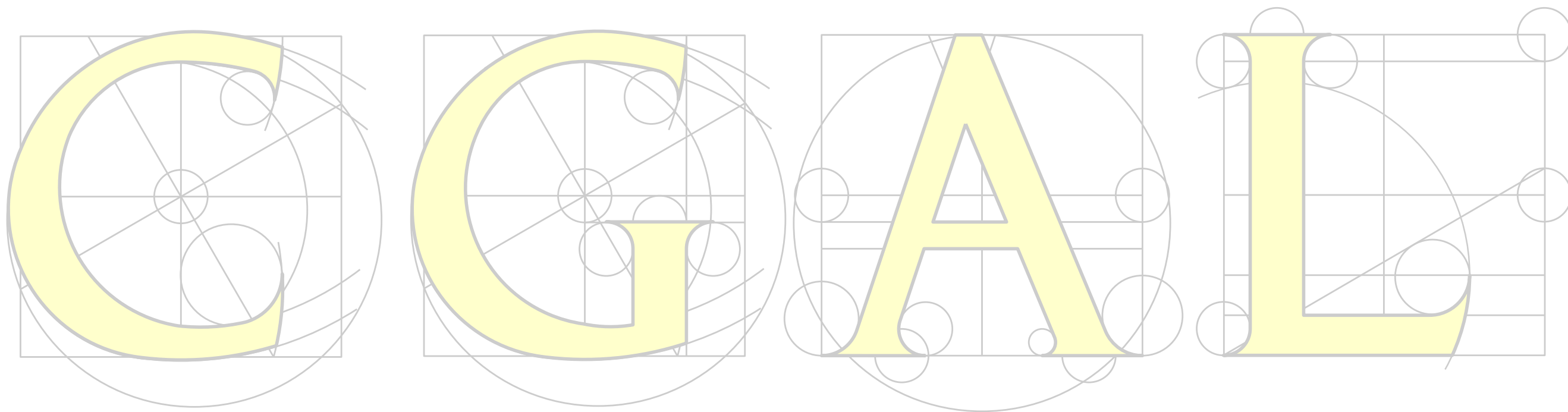


I: The CGAL Project

II: Exact Geometric Computing

III: Basic Programming using a CGAL Kernel

IV: Practical Information



PART II:

Exact Geometric Computing

GOALS

Awareness of challenges for implementing (geometric) algorithms.

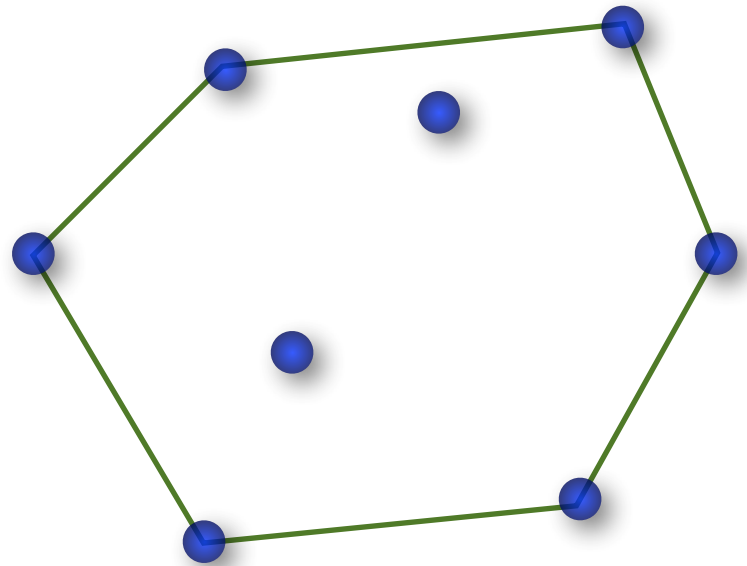
- ▶ Consequences of using limited precision arithmetic for discrete decisions.
- ▶ Exact (geometric) computing: benefits and limitations

Basic knowledge of limited precision arithmetic (in C++).

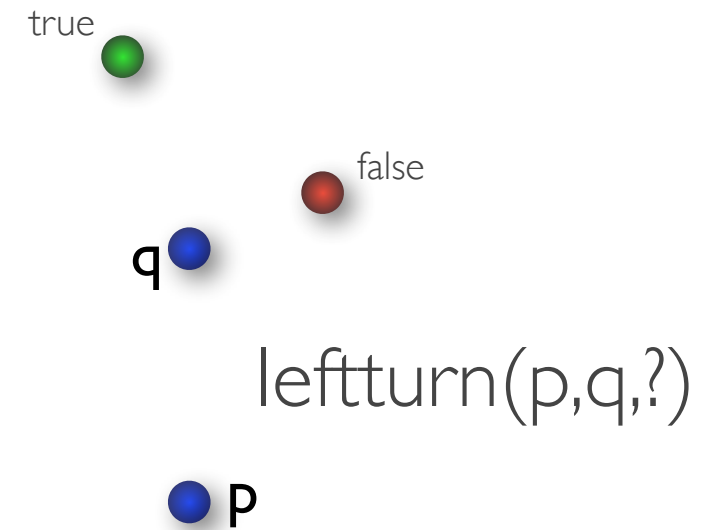
- ▶ How large is `int`, `long`, `double`, ...?
- ▶ How to bound results of a computation in terms of the input numbers.



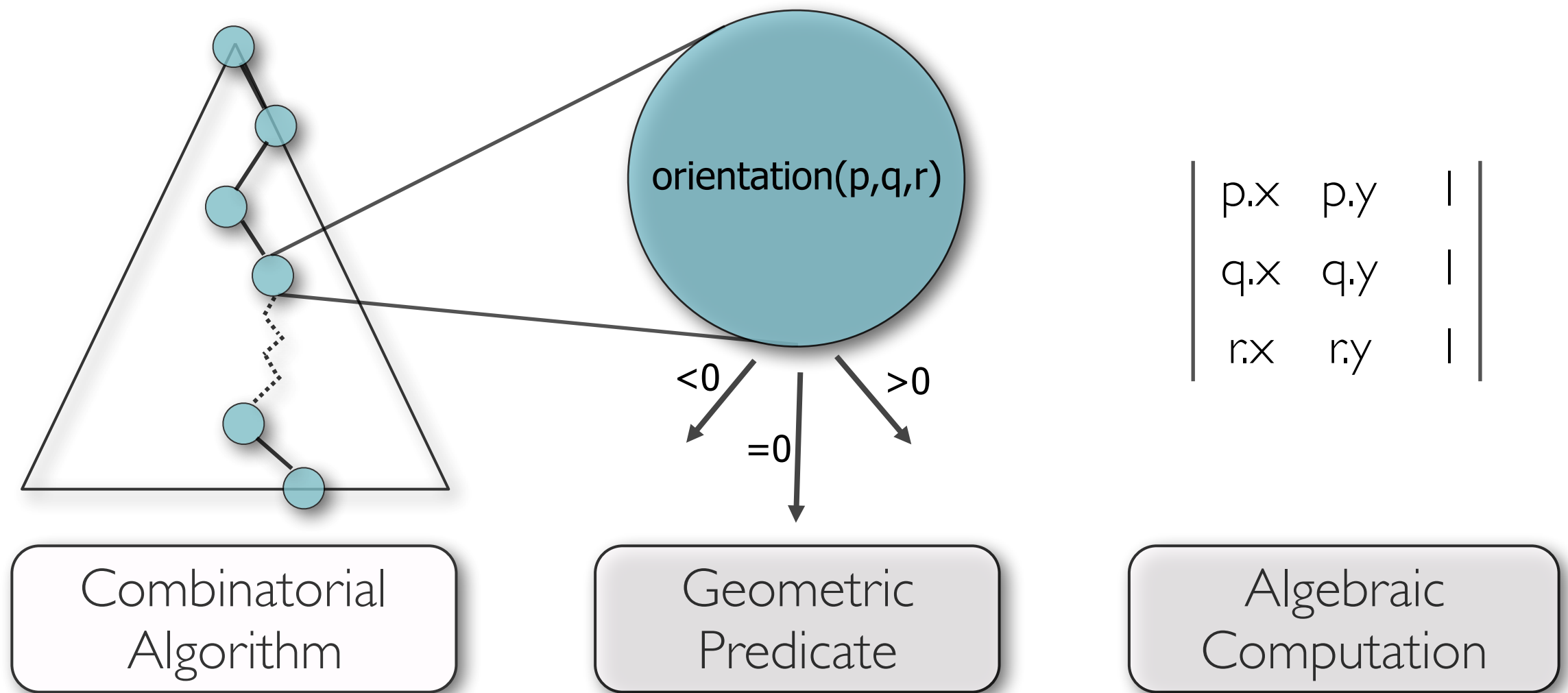
CONVEX HULL



Based on
orientation test.



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.

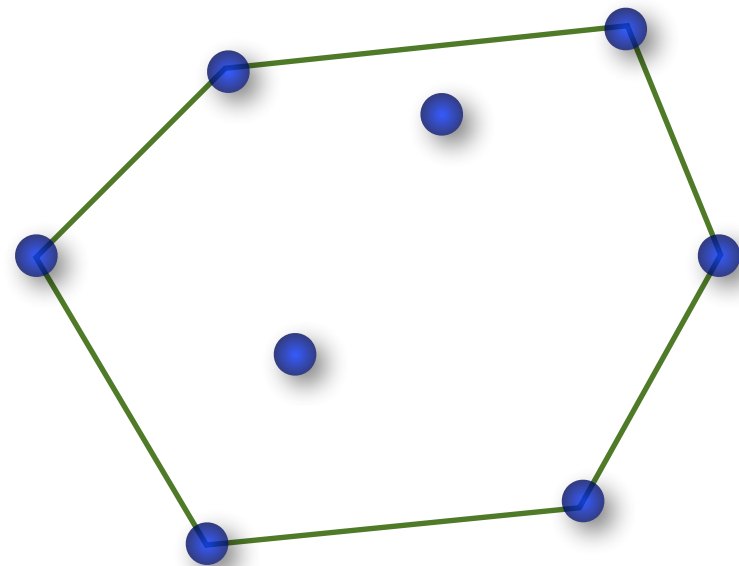
Integer multiplication and division are usually slower, often considerably. And the precision is limited regardless...

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

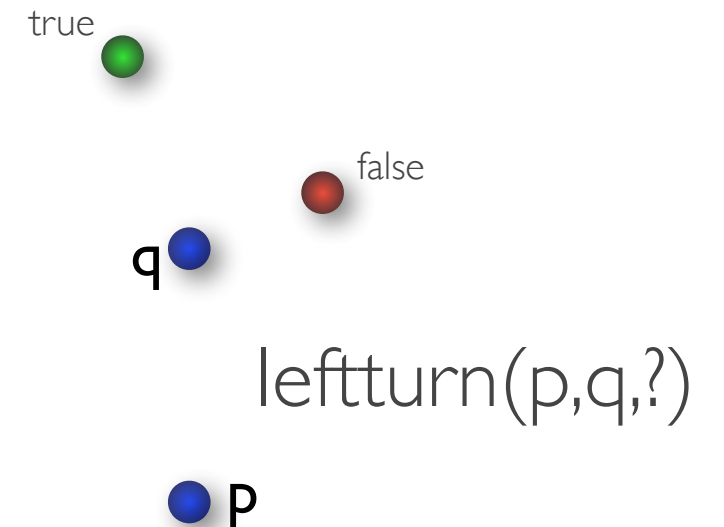
Difference to numeric computing:
Results are interpreted combinatorially: yes or no.

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

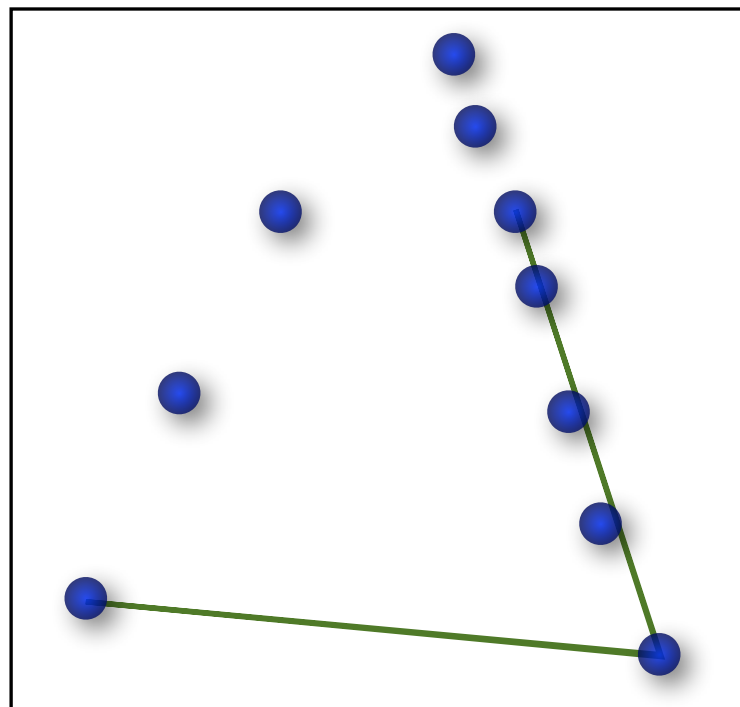
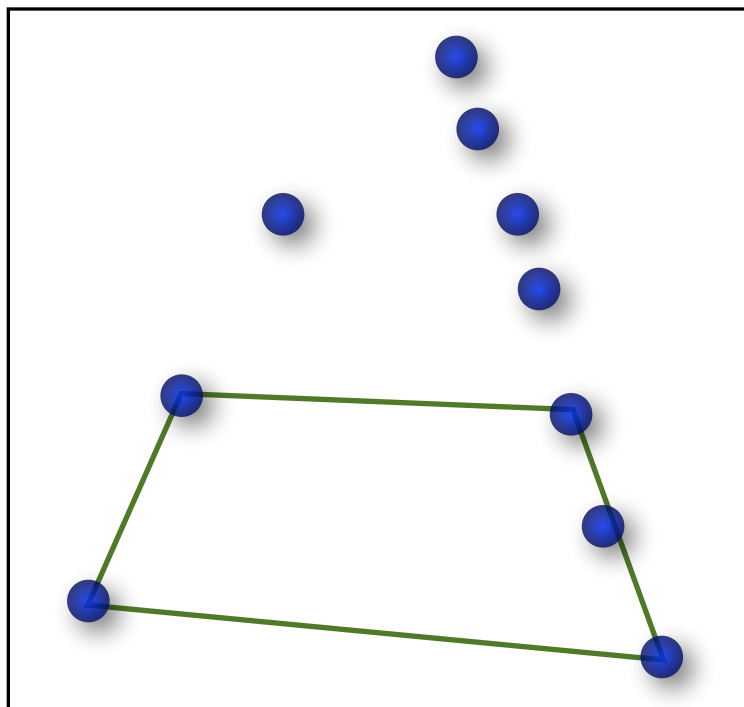
CONVEX HULL



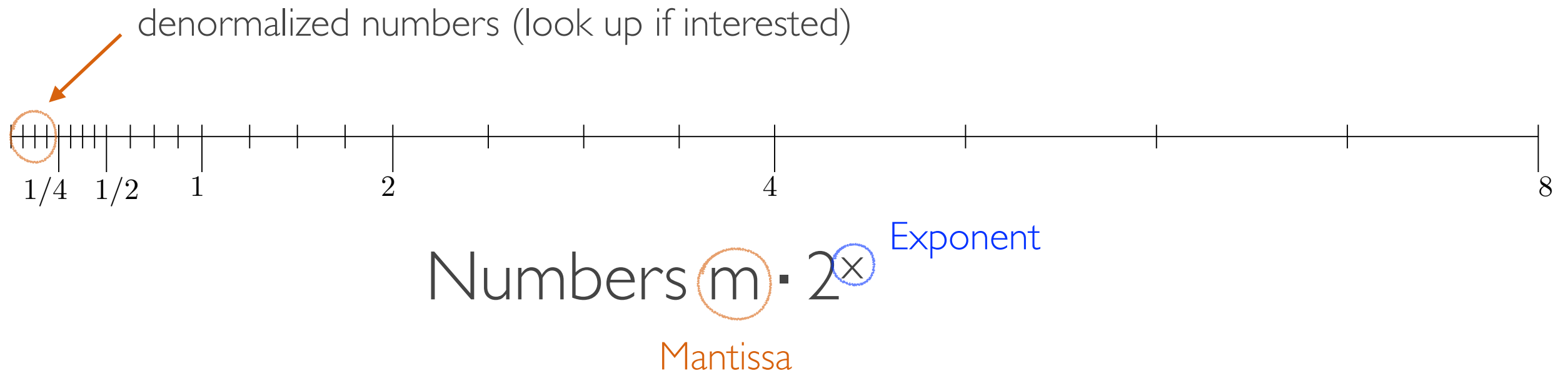
Based on
orientation test.



Possible results with an unreliable orientation test:



FLOATING POINT NUMBERS



- ▶ Precision corresponds to relative error
- ▶ But not for the result of a computation
- ▶ In particular, if intermediate results are large but the final result is small...

FLOATING POINT NUMBERS

IEEE 754 double precision

+/-	exponent	mantissa
1 bit	11 bits	53 bits

0.1 is not exactly representable

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

$$\boxed{b \text{ bits}} \pm \boxed{b \text{ bits}} \approx \boxed{b+1 \text{ bits}}$$

$$\boxed{b \text{ bits}} \cdot \boxed{b \text{ bits}} \approx \boxed{2b \text{ bits}}$$

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



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

COMPUTING WITH FLOATING POINT NUMBERS

Guideline #1: Avoid (square)roots!

$$\text{For } x, y \geq 0 : \sqrt{x} < \sqrt{y} \iff x < y.$$

Guideline #2: Avoid divisions!

$$\text{For } b, d > 0 : \frac{a}{b} < \frac{c}{d} \iff ad < bc.$$

These are just general guidelines, not hard rules. For instance, integer division can be useful to get rid of common factors.

Guideline #3: Estimate to check if loss of precision may occur! (See previous slide...)

STRAIGHT LINES ?

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

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

$$q = (12, 12)$$

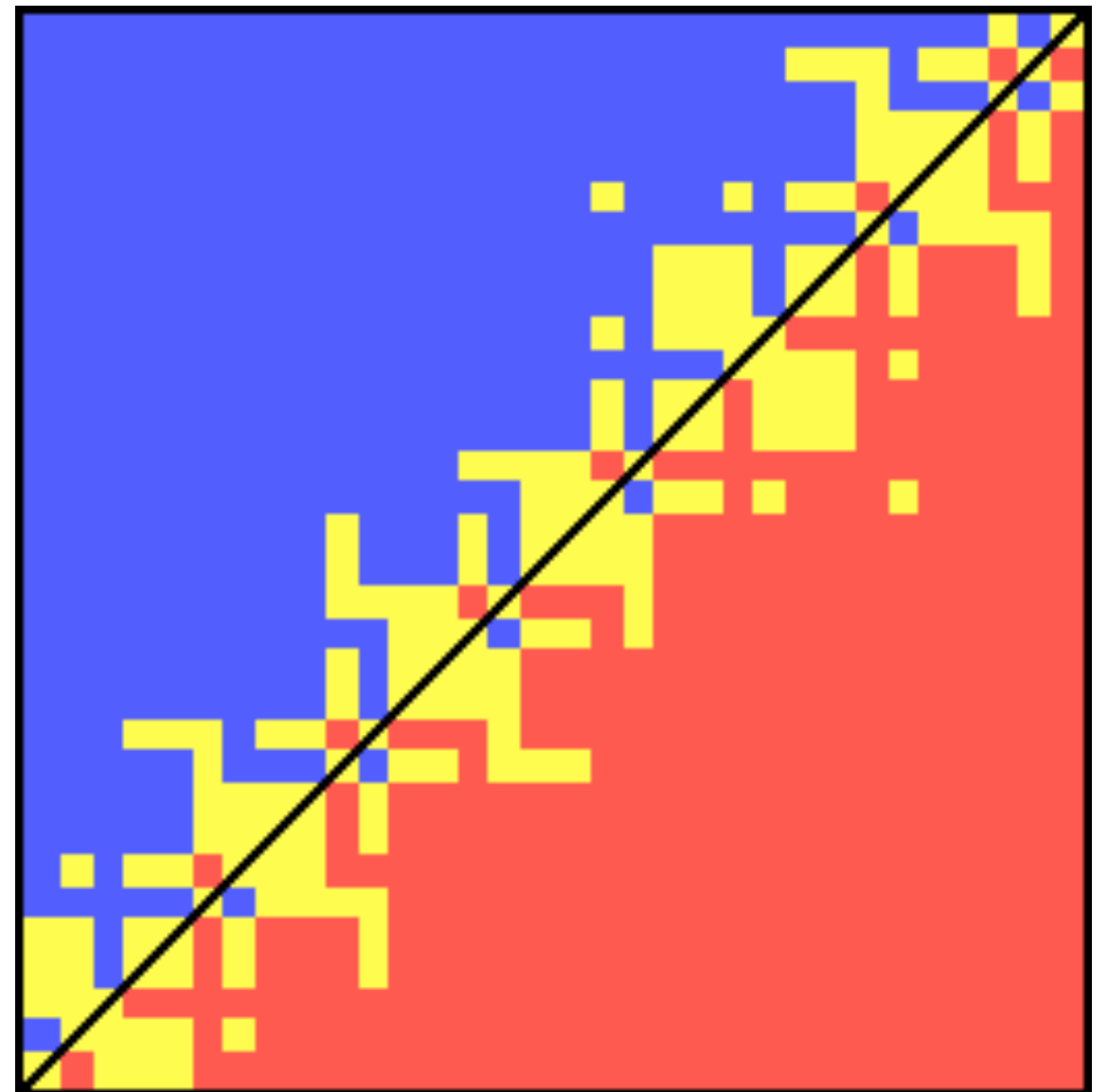
$$r = (24, 24)$$

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

256x256 pixel image

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

evaluated with **double**



STRAIGHT LINES ?

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

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

$$q = (8.800000000000000000000007, \\ 8.800000000000000000000007)$$

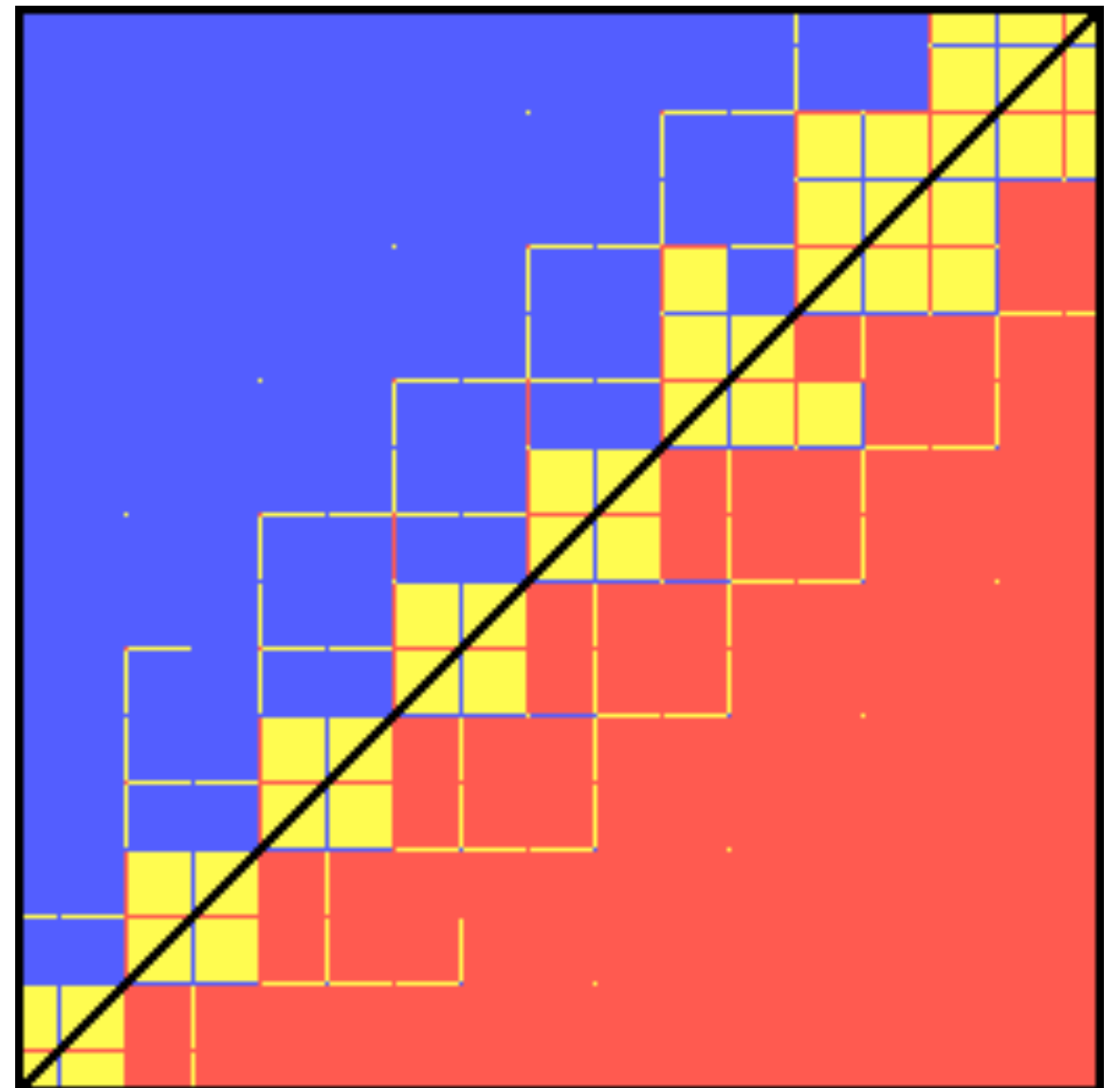
$$r = (12.1, 12.1)$$

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

256x256 pixel image

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

evaluated with **double**



STRAIGHT LINES ?

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

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

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

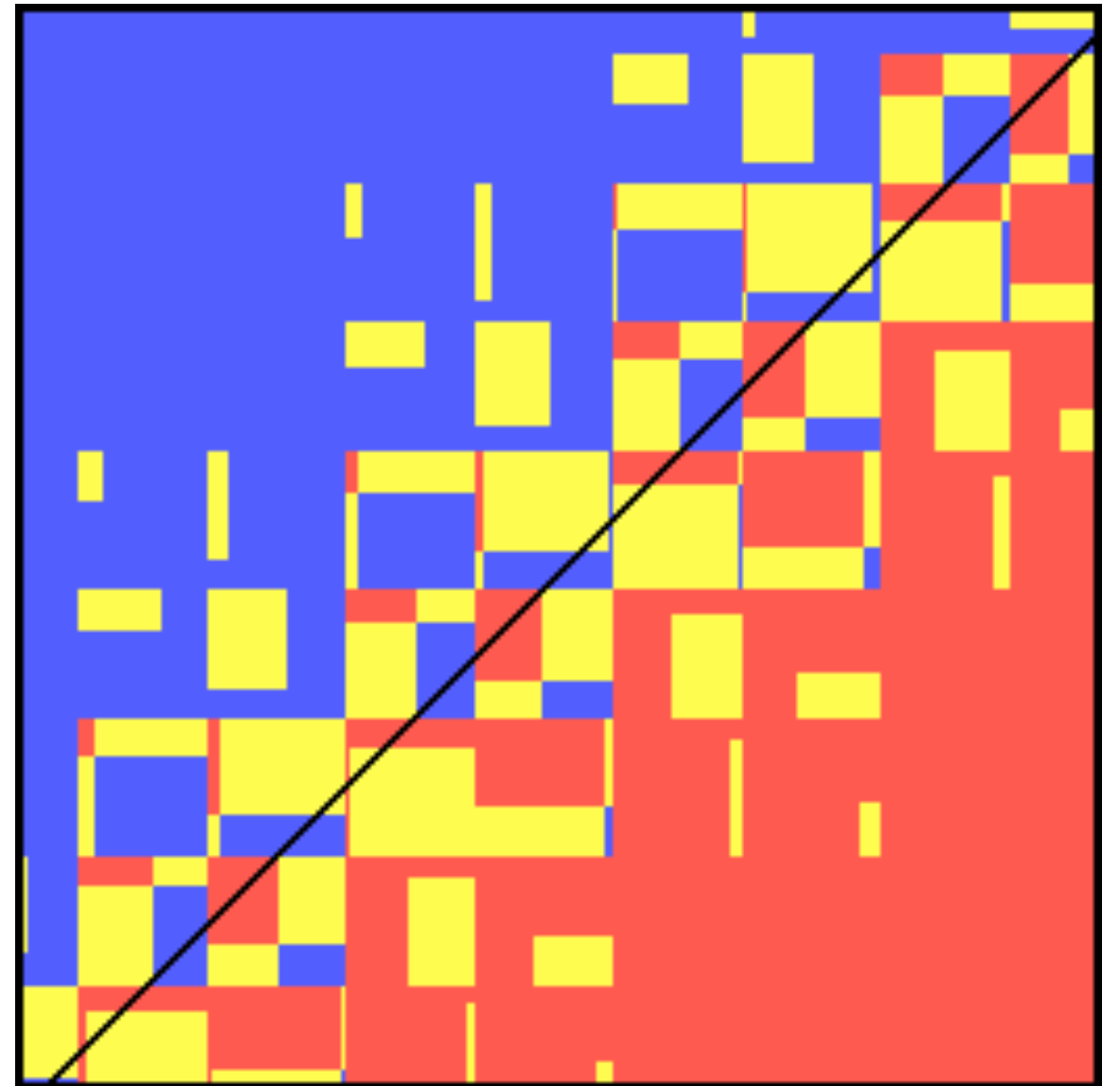
$$r = (24.000000000000000000500000, \\ 24.000000000000000000517765)$$

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

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.

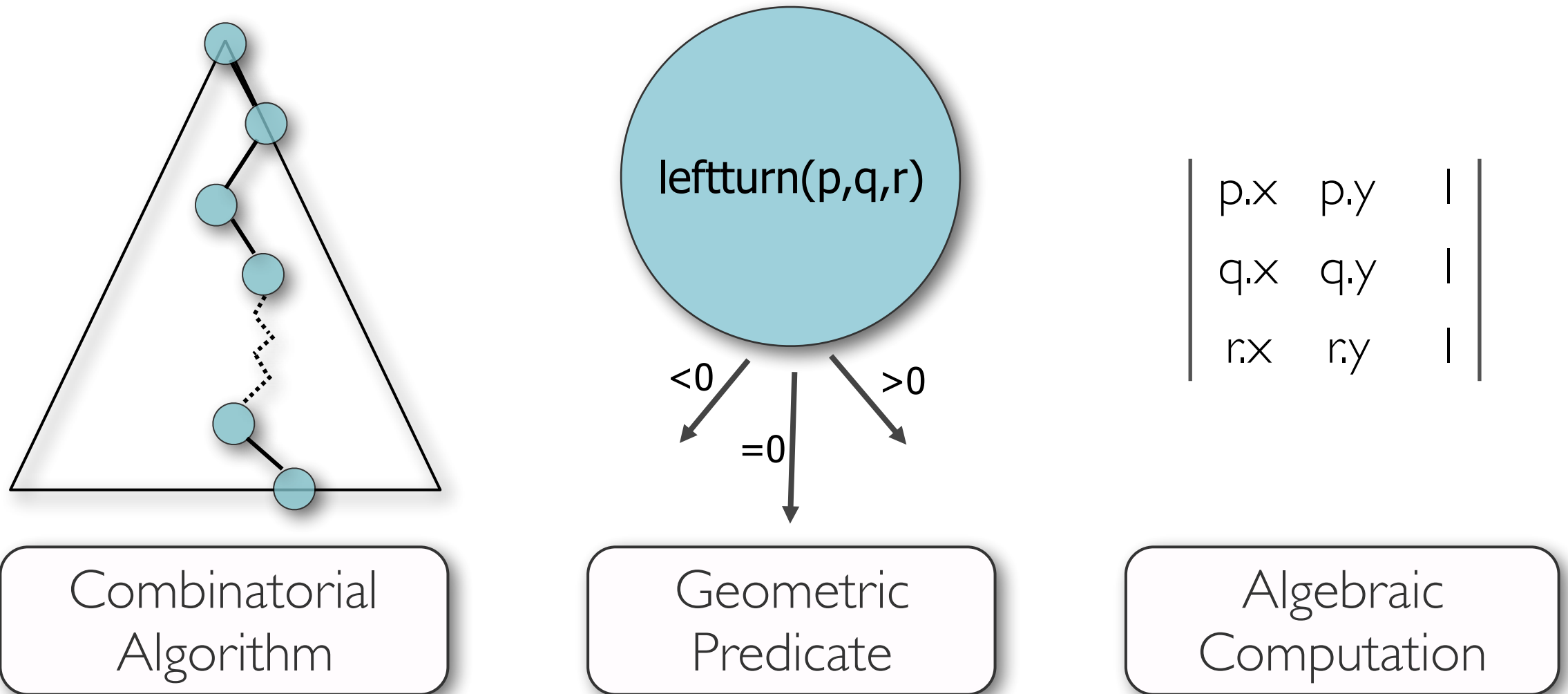
▶ Adapt algorithm to cope with imprecisions 

▶ 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...

EXACT COMPUTATION



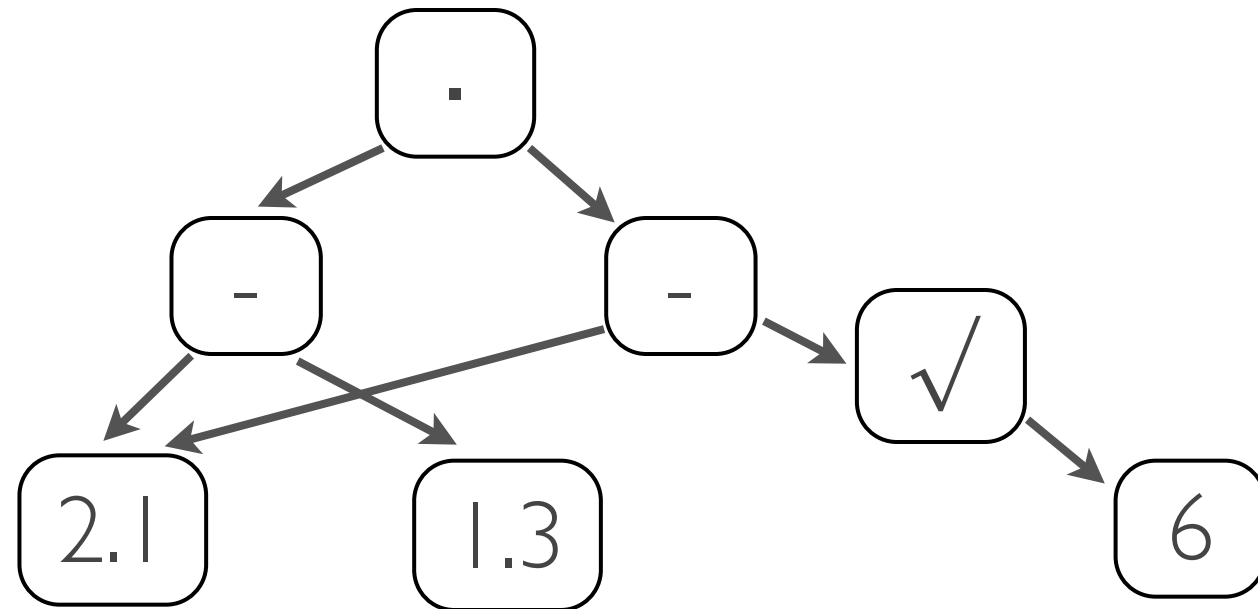
Ensure that the control flow in the algorithm is the same as if all algebraic computations were made exactly.



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.

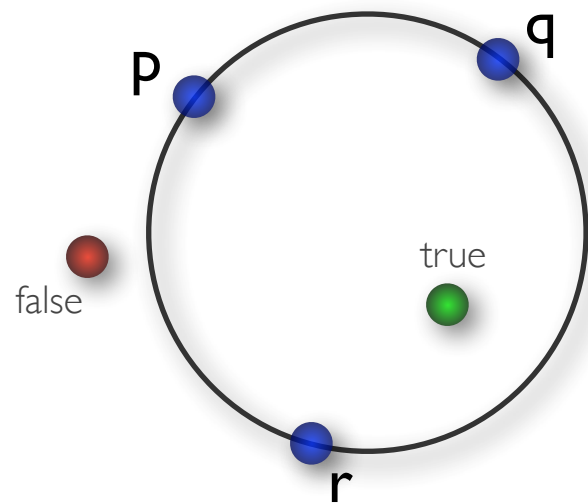
→ use when absolutely necessary only.

- ▶ 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 $0 \in [l, h]$.

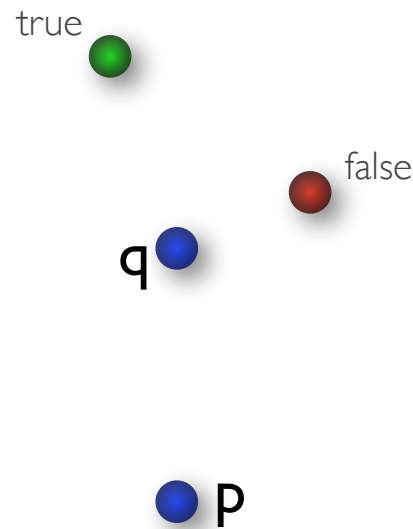
Minimal overhead as long as filter works.

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

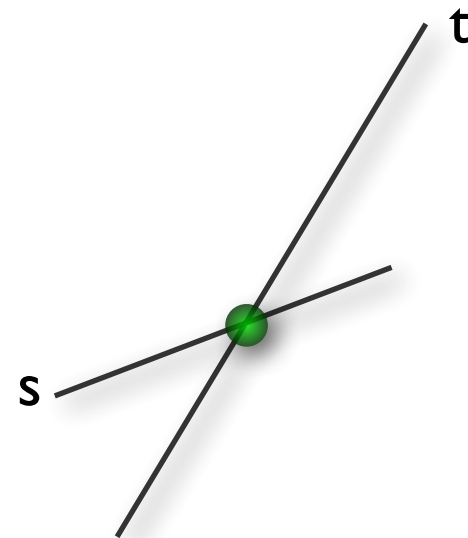
GEOMETRIC OPERATIONS



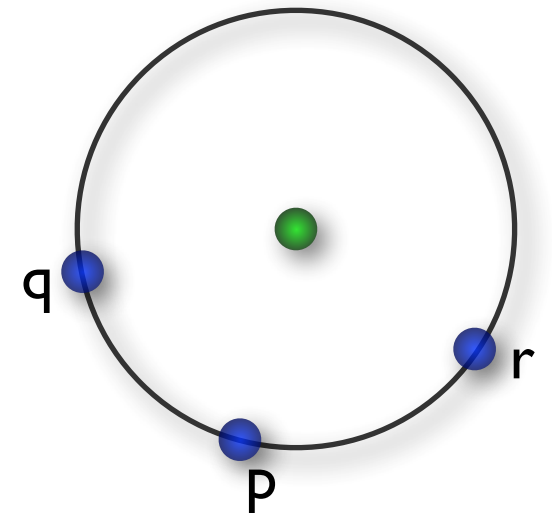
$\text{incircle}(p,q,r,?)$



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



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



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

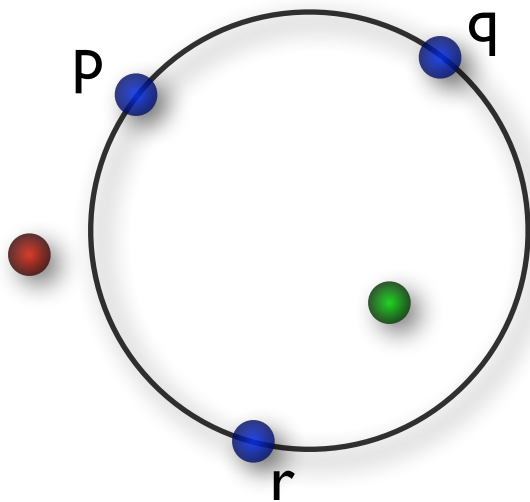
Predicates

Constructions

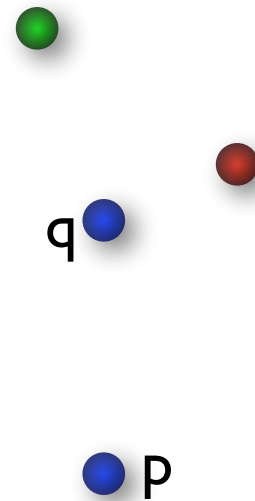


Do you need (exact) constructions?

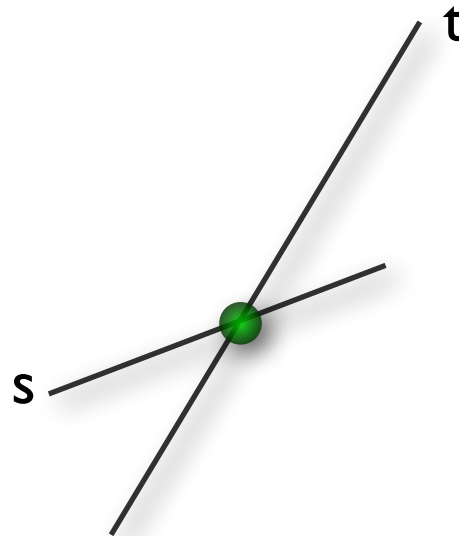
GEOMETRIC OPERATIONS



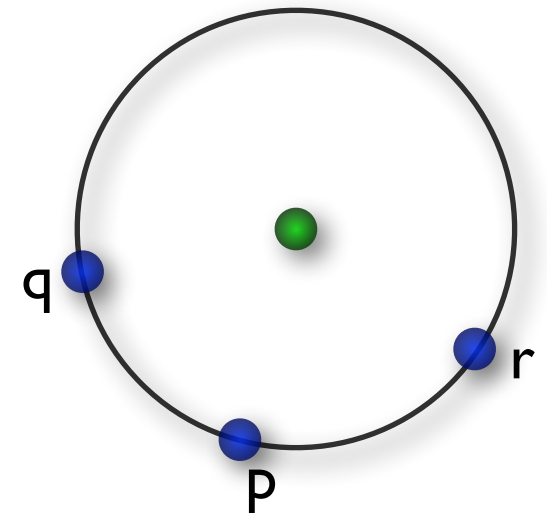
$\text{incircle}(p,q,r,?)$



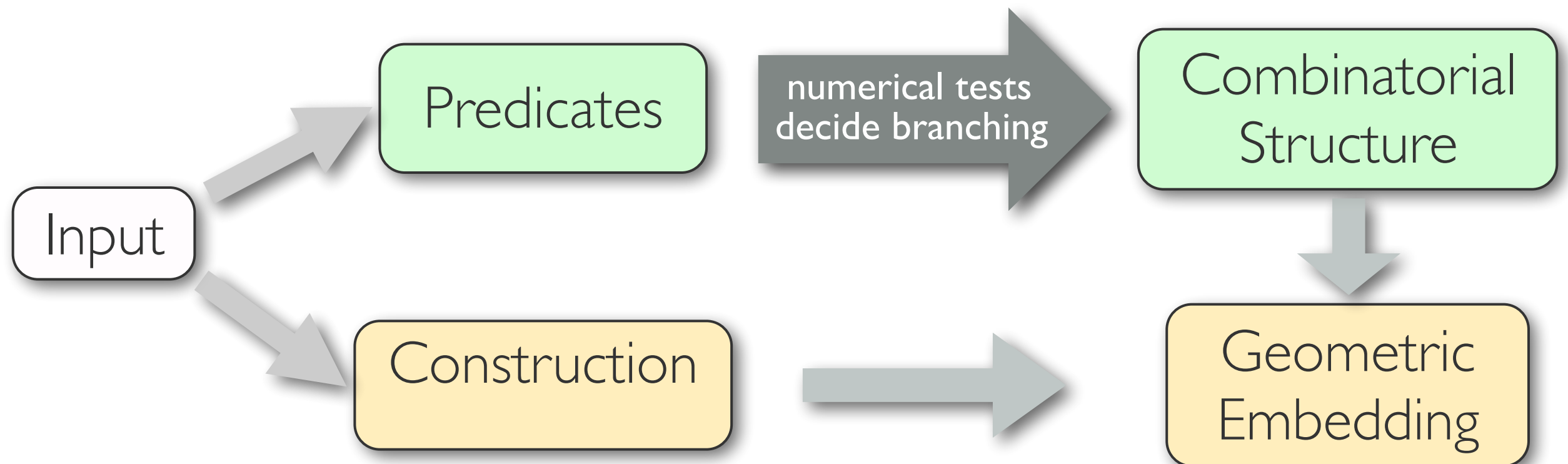
$\text{leftturn}(p,q,?)$



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



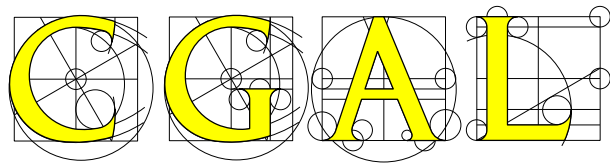
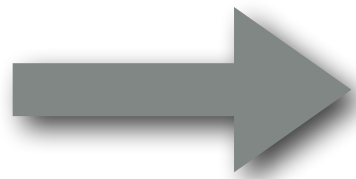
$\text{circumcircle}(p,q,r)$



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.

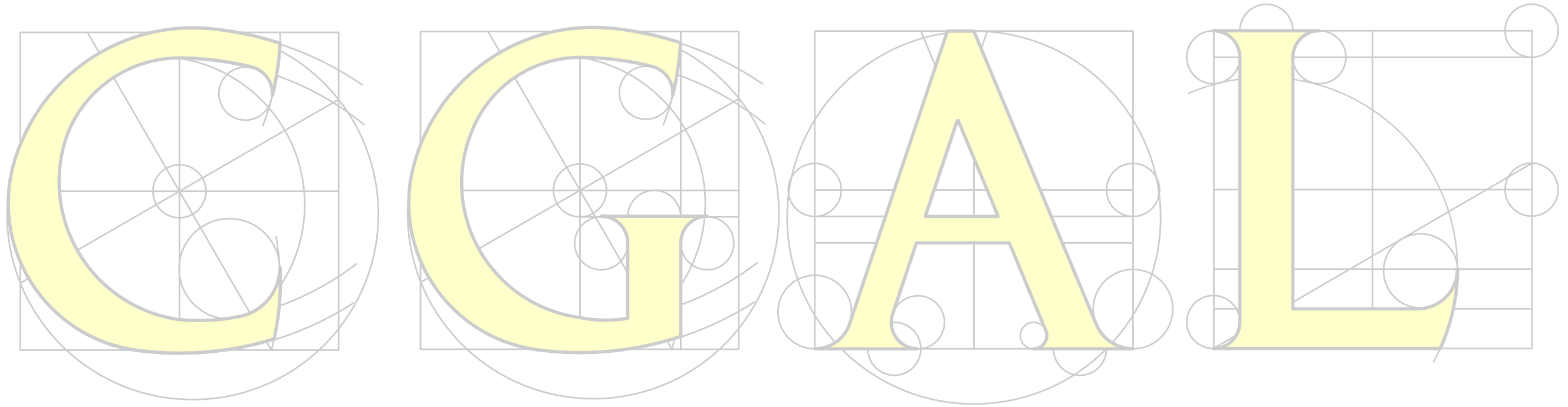
Predefined defaults:

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

GOALS

For a geometric algorithm, you are able to pick an adequate CGAL kernel.



- ▶ Are non-trivial geometric constructions needed?
- ▶ Are exact roots needed?

You are able to do some basic geometric computations using CGAL.

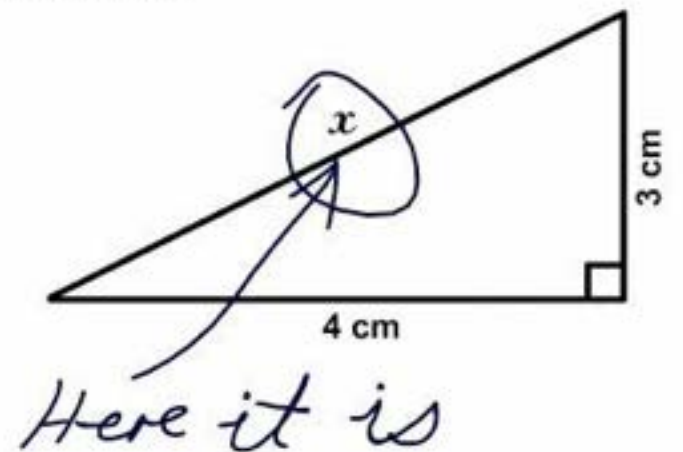
- ▶ 2D kernel objects
- ▶ Intersections
- ▶ Bounding Volumes



PREREQUISITES

You know basic Euclidean geometry (e.g., distance/area/volume, angles, Pythagoras, ...) and can apply this knowledge to describe and analyze problems, to design models and algorithms.

3. Find x .



Ocular Trauma - by Wade Clarke ©2005

You know basic algorithmic techniques (e.g., D.P., binary search, sorting, line sweep...). ➡ You skillfully combine them with the geometric techniques discussed here.

HELLO POINT

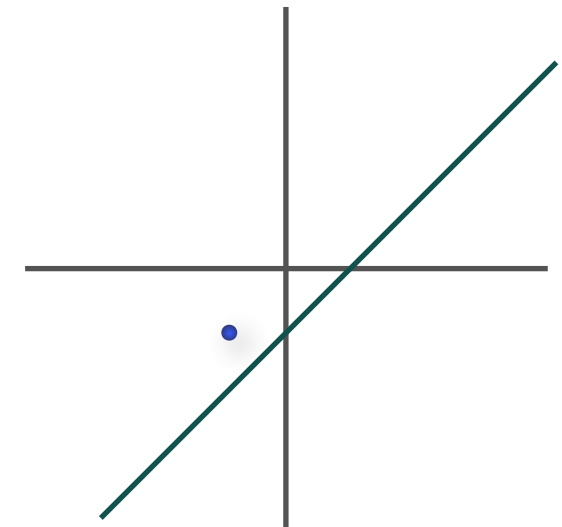
```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
```

```
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...

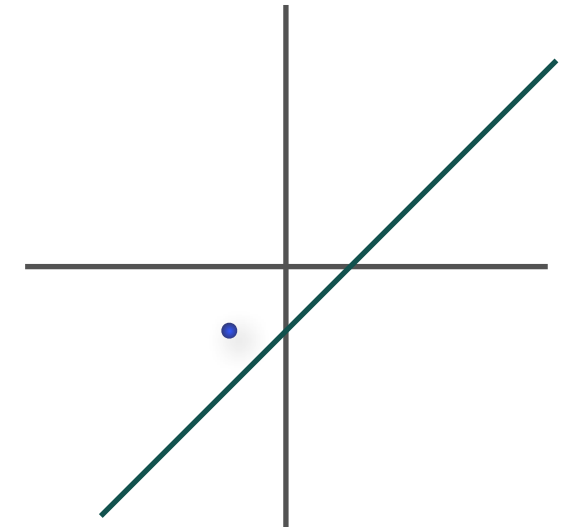
HELLO POINT

```
#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 $a px + b py + c > 0$, it is on the negative side of l , iff $a px + b py + c < 0$. 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; ← (Usually, this is pretty good. But we
    while (a+1 <= x) a += 1; ← Compare to the exact
    return a; ← value to be sure.
}

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)); ← Compute squareroot exactly.
    std::cout << floor_to_double(d) << std::endl;
}
```

(This assumes that x is somewhere within the range of double, which will be the case in all our problems.)

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 REPRESENTATIONS

- ▶ Point_2
 - ▶ Vector_2
 - ▶ Direction_2
 - ▶ Line_2
 - ▶ Ray_2
 - ▶ Segment_2
 - ▶ Triangle_2
 - ▶ Iso_rectangle_2
 - ▶ Circle_2
- } two FTs (Cartesian coordinates)
- three FTs (coefficients of line equation)
- } two points
- three points (corners)
- (two points, opposite corners)
- point and FT (center and squared radius)

So that you can tell which constructions are trivial...

2D KERNEL FUNCTIONALITY

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

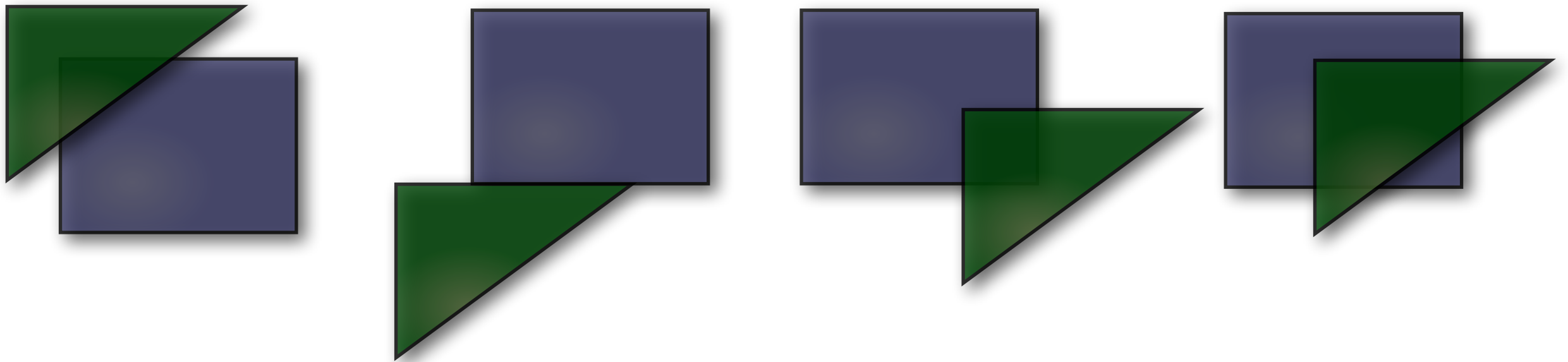
Most manual chapters have two parts:

- ▶ 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 r = ... ;  
K::Triangle_2 t = ... ;  
??? i = CGAL::intersection(r, t);
```

Solution: Use a generic wrapper class (based on boost::variant).
Test whether it contains an object of type T using `boost::get<T>`.

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])) {
```

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

```
                if (const P* op = boost::get<P>(&*o))
```

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

```
                else if (const S* os = boost::get<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";
```

```
}
```

The actual type is `std::result_of<K::Intersect_2(S,S)>::type`

Needs `#include <type_traits>`

Test for intersection (predicate)

Construct intersection (construction :-))

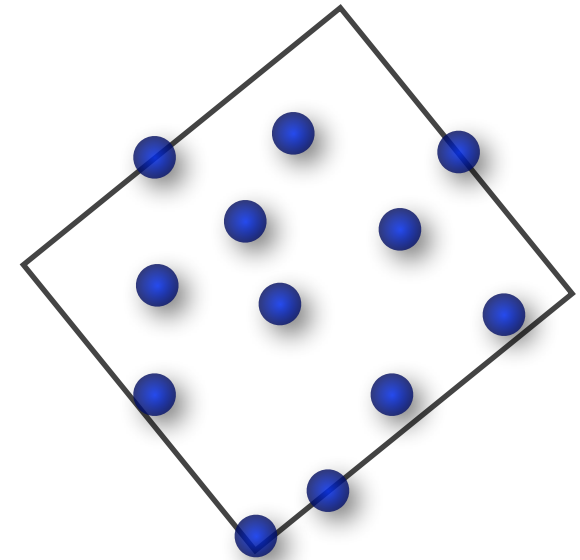
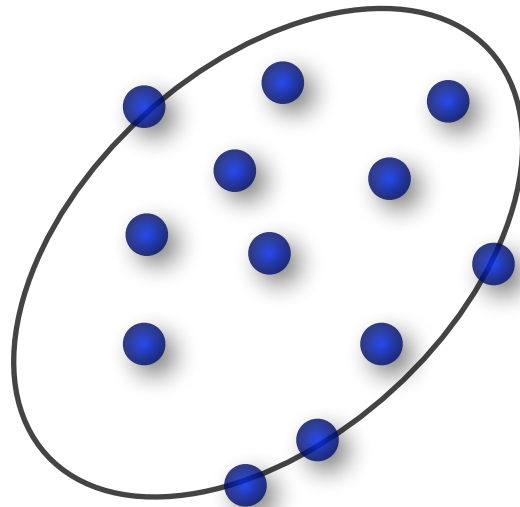
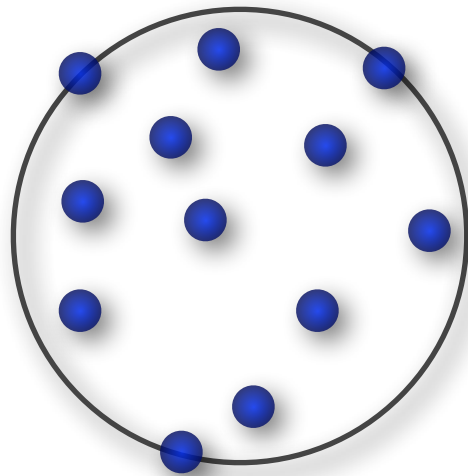
Cast fails (`=0`) if `o` is not of type `P`.

Use `auto` with caution! It disables static type checks that can be quite useful...

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.

Attention! Constructions (circumcircle of three points) used inside...

```
    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.

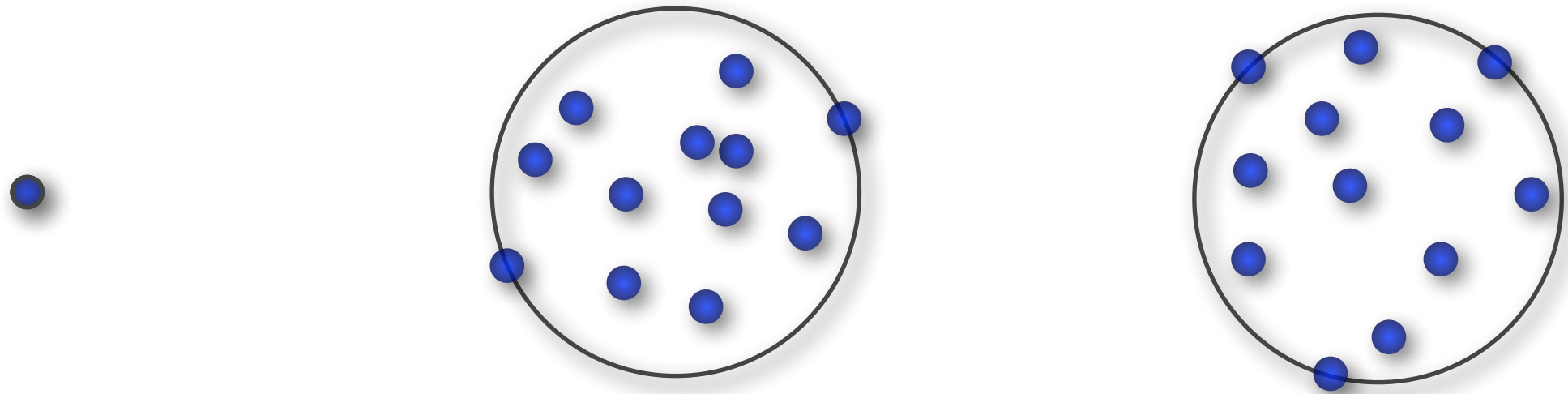
```
    Min_circle mc(P, P+n, true);
    Traits::Circle c = mc.circle();
    std::cout << c.center() << " " << c.squared_radius() << std::endl;
```

Construct and return the circle.

Output:
-0.5 0 9702.25

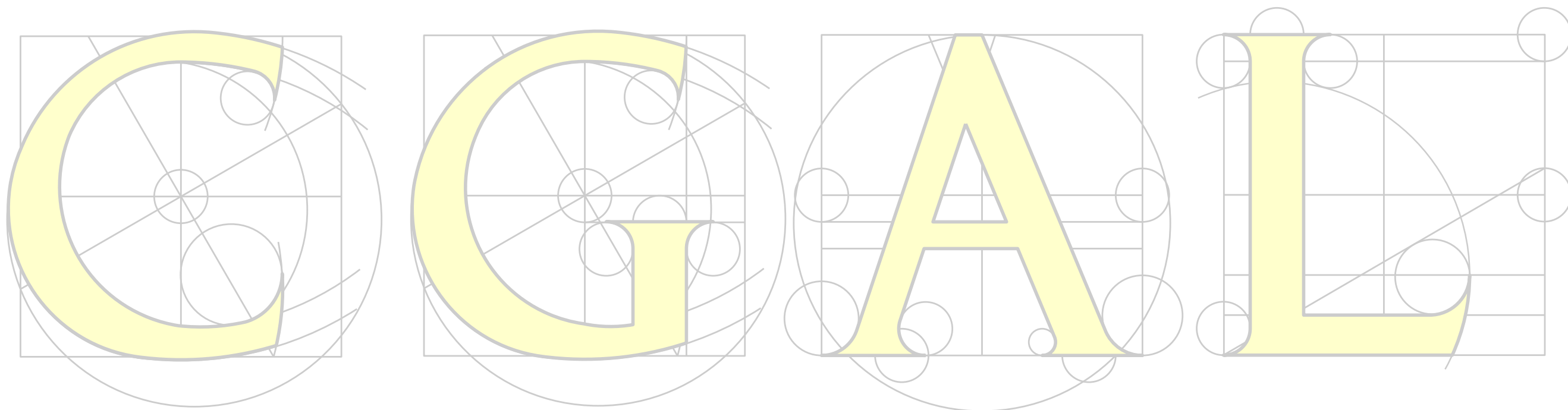
```
}
```

MINIMUM ENCLOSING CIRCLE



The minimum enclosing circle for a set of $n \geq 1$ points in \mathbb{R}^2 is determined ... by at most three points on its boundary.

These so-called support points can be obtained using corresponding member functions and iterators of `CGAL::Min_circle_2`.



PART IV:

Practical Information

INTEGER IO

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

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
...
// this is nicer ... (and usually ok for the inexact_constructions kernel)
K::Point_2 p;
std::cin >> p;
...
// this is faster ... assuming the input fits in a double
double x, y;
std::cin >> x >> y;
K::Point_2 p(x, y);
...
// this is even faster ... assuming the input fits in an int
int x, y;
std::cin >> x >> y;
K::Point_2 p(x, y);
```

IO PERFORMANCE

- ▶ If possible, read as an **int**
 - ▶ else read as a **long**
- 32bit on the judge
- 64bit on the judge
- Typical for 64-bit computers, but not universally true....

Sanity check

```
#include <limits>
```

```
if (std::numeric_limits<int>::max() < 33554432.0)  
    throw std::range_error("max(int) < 2^(25)");
```

double literal for 2^{25}

USING CGAL

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)!

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

If you want to use C++11 features, add the line
set(CMAKE_CXX_FLAGS "\${CMAKE_CXX_FLAGS} -std=c++11")
somewhere in the **CMakeLists.txt** file.

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 .**

That's it!

For more, see...

If you want to install CGAL on your private computer:

- Check/install prerequisites first: compiler, cmake, boost, gmp, mpfr, (qt)
- Install cgal (on the judge we run CGAL-4.7-1)
https://judge.inf.ethz.ch/doc/cgal/doc_html/Manual/installation.html
- Or download CGAL packages of your distribution if they exist (don't forget cgal-devel).

HTTP://WWW.CGAL.ORG

The image shows two overlapping browser windows. The background window displays the CGAL website, which features a header with navigation links (Project, Download, Documentation, Packages) and a main section titled 'The Computational Geometry Algorithms Library'. Below this is a large image of a Voronoi diagram with the text 'Voronoi Diagram' and a code snippet: `VD = CGAL::make_voronoi(points);`. A text box below the image describes CGAL as a software project providing efficient and reliable geometric algorithms in C++, used in various fields like GIS, CAD, molecular biology, and robotics. It lists the library's capabilities, including data structures and algorithms for triangulations, Voronoi diagrams, Boolean operations, and more. The foreground window shows the 'CGAL 4.9 - Manual' page, specifically the 'Triangulations and Delaunay Triangulations' section. This page includes a table of contents, a list of packages (CGAL 4.9 - Manual, Getting Started, Tutorials, Package Overview, etc.), and detailed descriptions of various triangulation packages: 2D Triangulation, 2D Triangulation Data Structure, 2D Periodic Triangulations, and 3D Triangulations. Each package description includes its author(s), a brief overview of its functionality, and links to user and reference manuals. The bottom of the foreground window shows a footer with the text 'Generated on Fri Sep 18 2016 21:24:33 for CGAL 4.9 - Manual by doxygen 1.8.4'.

The screenshot displays two overlapping browser windows showing the CGAL (Computational Geometry Algorithms Library) website and its documentation.

The background window shows the CGAL homepage, titled "The Computational Geometry Algorithms Library". It features a navigation bar with links: Project, Download, Documentation, Packages. The main content area displays a Voronoi diagram and a code snippet: `VD = CGAL::make_voronoi(points);`. Below the diagram, a text box describes CGAL as a software project providing efficient and reliable geometric algorithms in the form of a C++ library, used in various areas like geographic information systems, computer-aided design, molecular biology, medical imaging, computer graphics, and robotics. It lists the library's capabilities, including data structures and algorithms for triangulations, Voronoi diagrams, Boolean operations on polygons and polyhedra, point set processing, arrangements of curves, surface and volume mesh generation, geometry processing, alpha shapes, convex hull algorithms, shape analysis, AABB and KD trees.

The foreground window shows the "CGAL 4.9 - Manual" page, specifically the "Triangulations and Delaunay Triangulations" section. The page includes a table of contents, a list of packages (CGAL 4.9 - Manual, Getting Started, Tutorials, Package Overview, etc.), and detailed descriptions of various triangulation packages:

- 2D Triangulation**: Mariette Yvinec. This package allows building and handling various triangulations for point sets in two dimensions. Any CGAL triangulation covers the convex hull of its vertices. Triangulations are built incrementally and can be modified by insertion or removal of vertices. They offer point location facilities. The package provides plain triangulation (whose faces depend on the insertion order of the vertices) and Delaunay triangulations. Regular triangulations are also provided for sets of weighted points. Delaunay and regular triangulations offer nearest neighbor queries and primitives to build the dual Voronoi and power diagrams. Finally, constrained and Delaunay constrained triangulations allow forcing some constrained segments to appear as edges of the triangulation. Several versions of constrained and Delaunay constrained triangulations are provided: some handle intersections between input constraints segment while others do not. **Introduced in:** CGAL 0.9. **Depends on:** 2D Triangulation Data Structure. **BibTeX:** cgal-y-03-16b. **License:** GPL. **Windows Demos:** Delaunay Triangulation, Regular Triangulation, Constrained Delaunay Triangulation. **Common Demo Dlls:** dlls.
- 2D Triangulation Data Structure**: Sylvain Pion and Mariette Yvinec. This package provides a data structure to store a two-dimensional triangulation that has the topology of a two-dimensional sphere. The package acts as a container for the vertices and faces of the triangulation and provides basic combinatorial operations on the triangulation. **Introduced in:** CGAL 2.2. **BibTeX:** cgal-py-03-16b. **License:** GPL.
- 2D Periodic Triangulations**: Nico Kruithof. This package allows building and handling triangulations of point sets in the two-dimensional flat torus. Triangulations are built incrementally and can be modified by insertion or removal of vertices. They offer point location facilities. The package provides Delaunay triangulations and offers nearest neighbor queries and primitives to build the dual Voronoi diagrams. **Introduced in:** CGAL 4.3. **Depends on:** 2D Triangulation. **BibTeX:** cgal-k-p02-13-16b. **License:** GPL. **Windows Demo:** Periodic Delaunay Triangulation. **Common Demo Dlls:** dlls.
- 3D Triangulations**: Célement Jamin, Sylvain Pion and Monique Teillaud. This package allows building and handling triangulations for point sets in three dimensions. Any CGAL triangulation covers the convex hull of its vertices. Triangulations are built incrementally and can be modified by insertion, displacements or removal of vertices. **Introduced in:** CGAL 2.1. **Depends on:** 3D.

The bottom of the foreground window shows a footer: "Generated on Fri Sep 18 2016 21:24:33 for CGAL 4.9 - Manual by doxygen 1.8.4".