GEOMETRY & ALGEBRA IN THE ALGORITHMS LAB

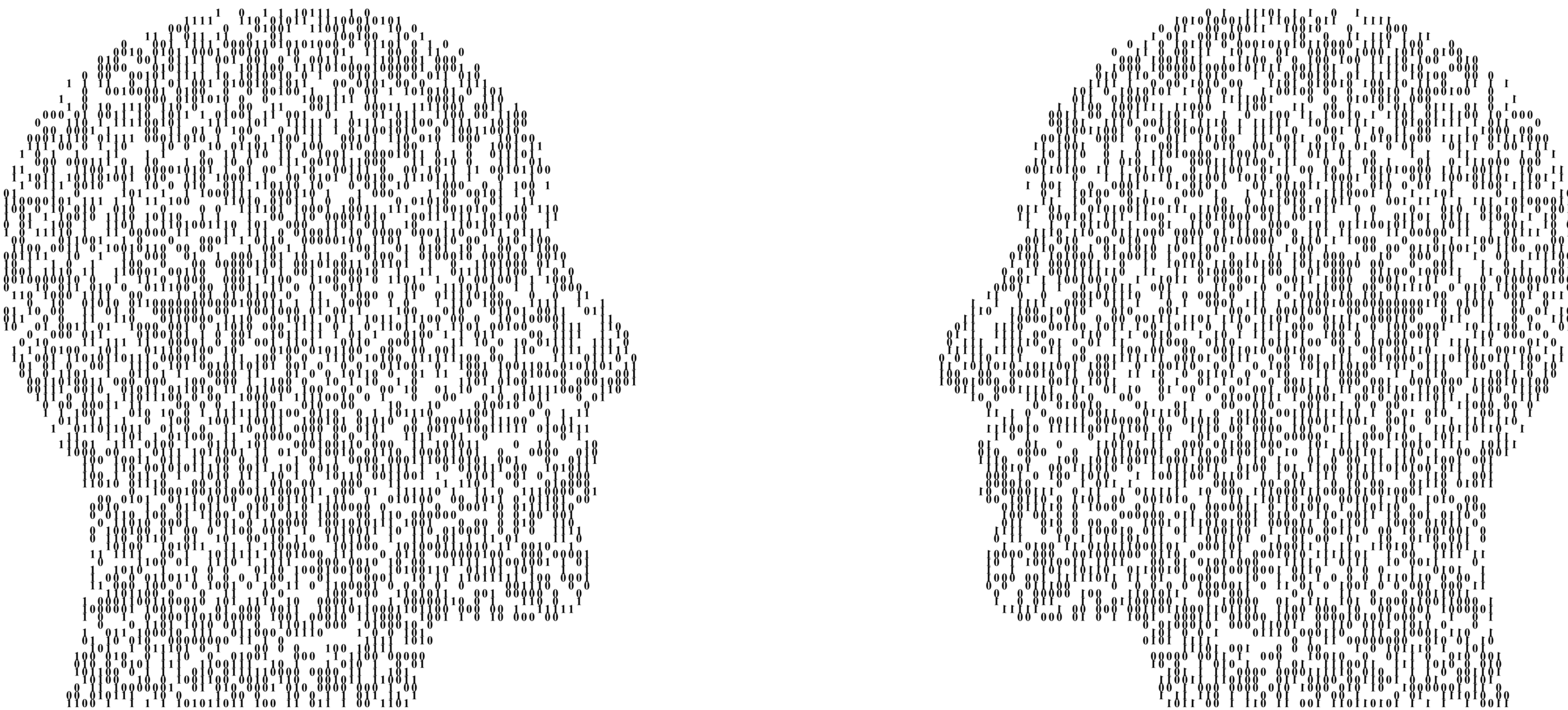
Michael Hoffmann <hoffmann@inf.ethz.ch>

I: Numbers: Representation & Computation
II: Basic Geometric Computing using CGAL

Also check out the (draft) lecture notes on moodle.

RECOMMENDATIONS…

- Check out "A Short Introduction to C++ for the Algorithms Lab": It is essential for some problems.
- Multiplicative constants matter: Asymptotics provide a baseline, but being 5 times slower does not cut it in most cases.
- Avoid "weird" optimizations: Think how to make your solution more efficient and avoid unnecessary computations. But only where it makes sense, also from a "theoretical" perspective.



PART I:

Numbers: Representation & Computation

OUTLINE

When working with numbers, choose an appropriate datatype for representation and computation.

- Does the input fit?
- Do the results of computations fit?
- Do not blindly trust limited precision arithmetic!
- Use exact algebraic computing where needed.

Here: Black box tool

http://www.esa.int/ESA_Multimedia/Images/1998/01/Ariane_501_explosion

GOALS

Awareness of discrepancy: infinite abstraction vs. finite computer.

- Consequences of using limited precision arithmetic for discrete decisions.
- How to ensure correctness.

Basic knowledge of limited precision arithmetic.

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

DOES THE INPUT FIT?

Check Section 2.5 in "A Short Introduction to C++ for the Algorithms Lab".

- If possible, read as an `int`
- else read as a `long`

32bit on Code Expert

64bit on Code Expert

Typical for 64-bit computers, but not universally true....

Sanity check

`double` literal for 2^25

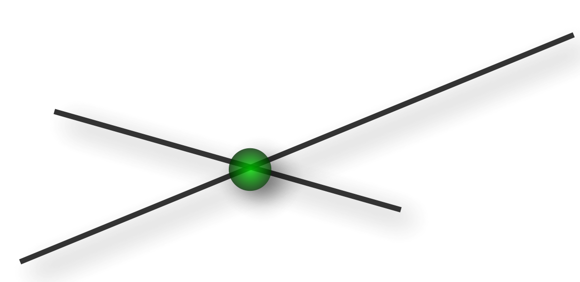
```
#include <limits>
#include <stdexcept>

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

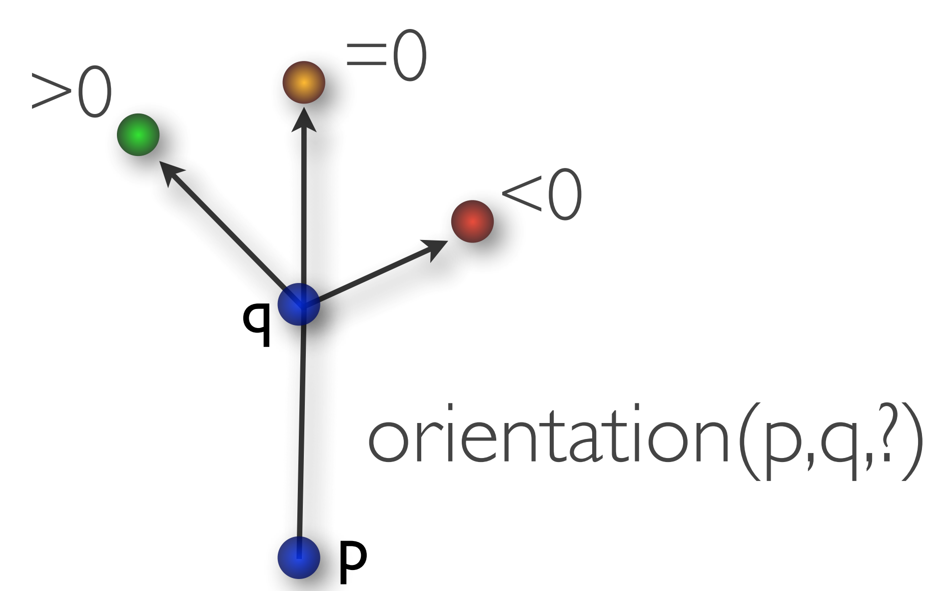
COMPUTATIONS

Some tasks require nontrivial computations, e.g.,

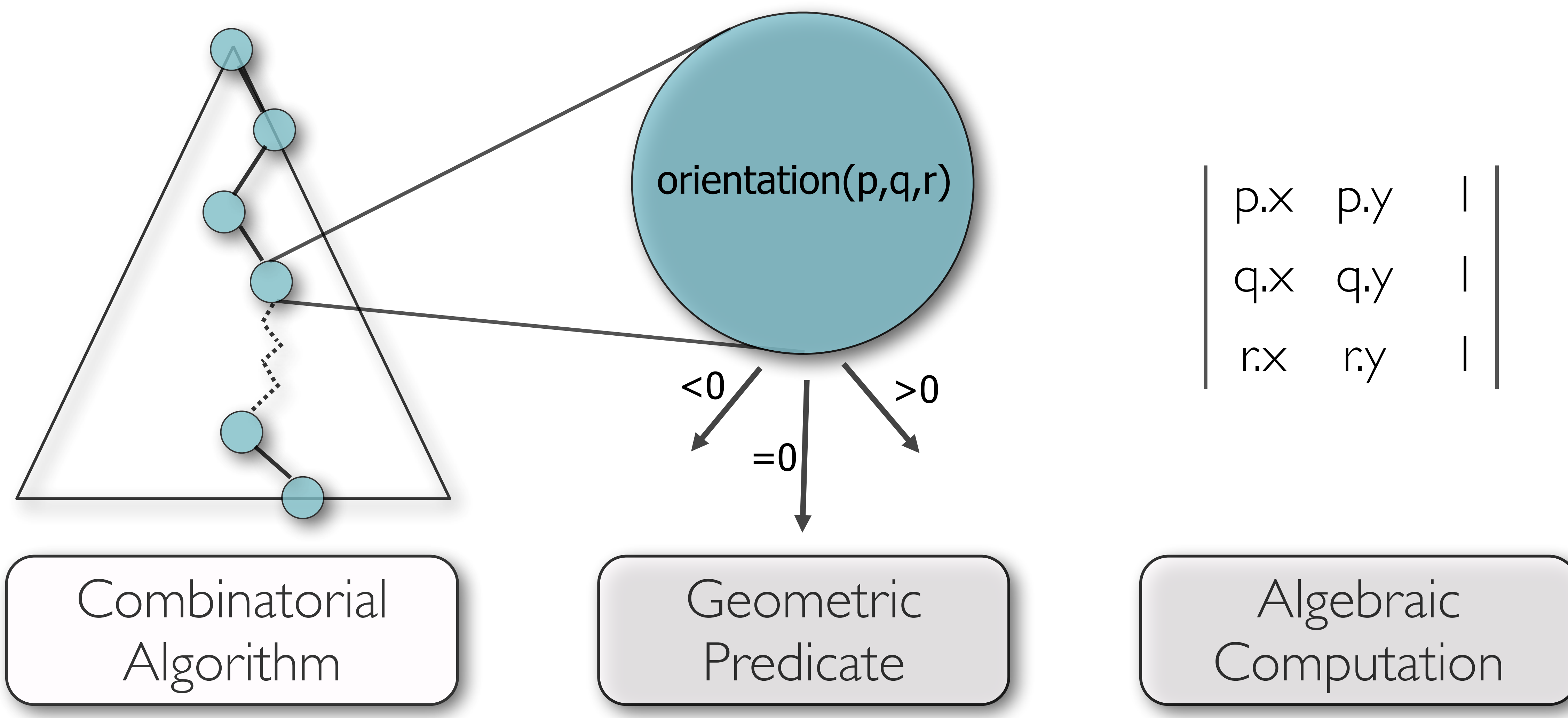
- computing Euclidean distances
- solving a linear system
- computing orientations of point triples

$$a_1 \cdot x + b_1 \cdot y = c_1$$
$$a_2 \cdot x + b_2 \cdot y = c_2$$





LAYERS OF GEOMETRIC ALGORITHMS



Control flow depends on nontrivial algebraic computations.

How to do these efficiently and consistently?

ARITHMETIC

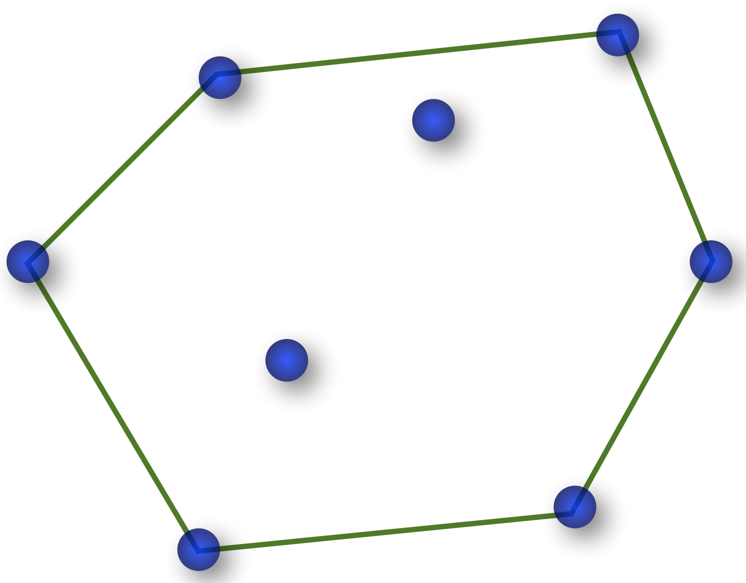
If computations are done using limited precision (floating point) arithmetic…

→ Results may be incorrect due to roundoff.

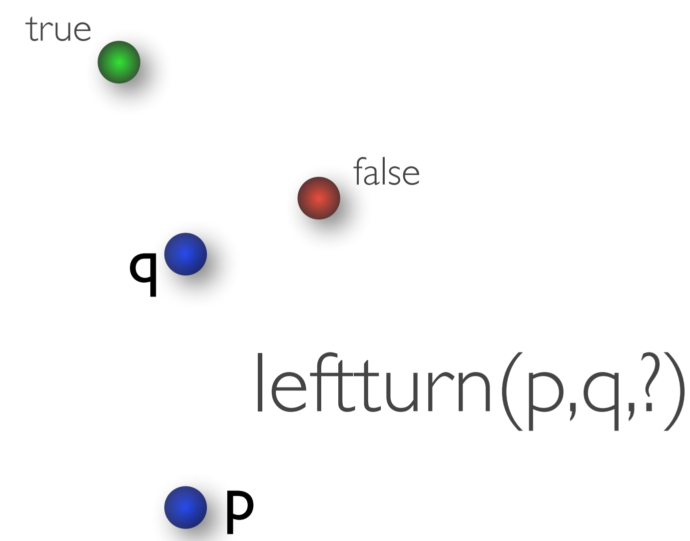
Difference to numeric computing:
Results are interpreted combinatorially: yes or no.

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

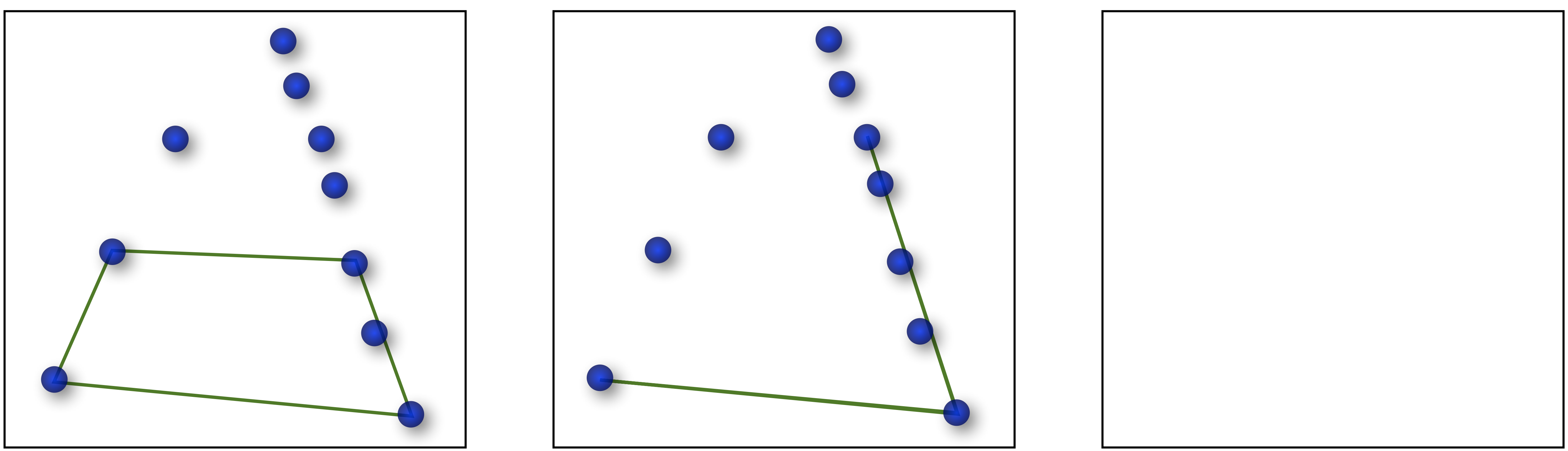
CONVEX HULL



Based on
orientation test.



Possible results with an unreliable orientation test:



ROUNDOFF ERRORS?

```
#include <iostream>

int main()
{
    double x = 1.1;
    x -= 1;
    x -= 0.1;
    std::cout << x << "\n";
}
```

Output: `8.32667e-17`

Why?

BUILTIN NUMBER TYPES

Type specifier	Standard	Code Expert	Min Integer	Max Integer
`int`	$\geq$ 16 bits	32 bits	-2^{31}	$2^{31}-1$
`long`	$\geq$ 32 bits	64 bits	-2^{63}	$2^{63}-1$
`double`	64 bits	64 bits	-2^{53}	2^{53}

IEEE 754 double precision

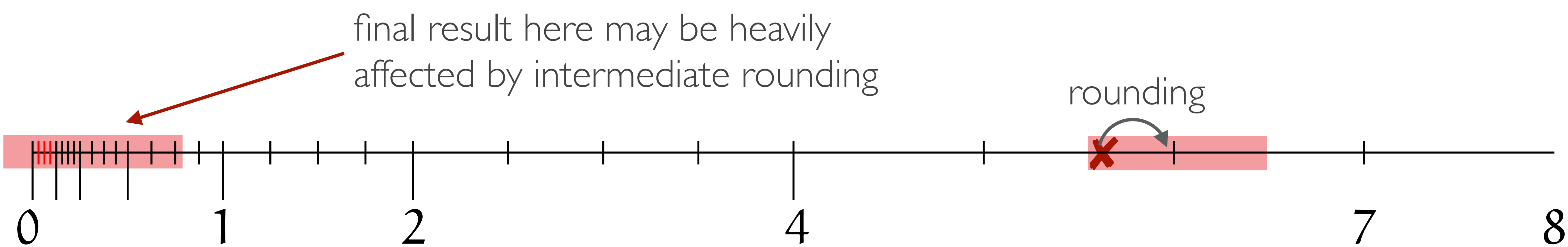
+/-	exponent	mantissa
1 bit	11 bits	53 bits

53 binary digits

Numbers $\pm m \cdot 2^x$, $1 \leq m < 2$, $-1022 \leq x \leq 1023$.

Results are rounded to nearest representable number.

FLOATING POINT NUMBERS



2bit m, 3bit x: $m \cdot 2^x$ with $m=(1.D_1D_2)_2$ and $-3 \leq x \leq 2$.

- Mantissa width ≈ #ticks between 2^i and 2^{i+1}
- Uniform relative error
- But absolute error of large numbers can be large
- If intermediate results are large but the final result is small…

ORIENTATION

Leftturn(p, q, r) $\Leftrightarrow$

$$\begin{vmatrix} p.x & p.y & 1 \\ q.x & q.y & 1 \\ r.x & r.y & 1 \end{vmatrix} = (q.x-p.x)(r.y-p.y)-(q.y-p.y)(r.x-p.x) > 0$$

maybe may comparatively large

…. even if the final result is very small.

$\longrightarrow$ Roundoff errors may lead to wrong results.

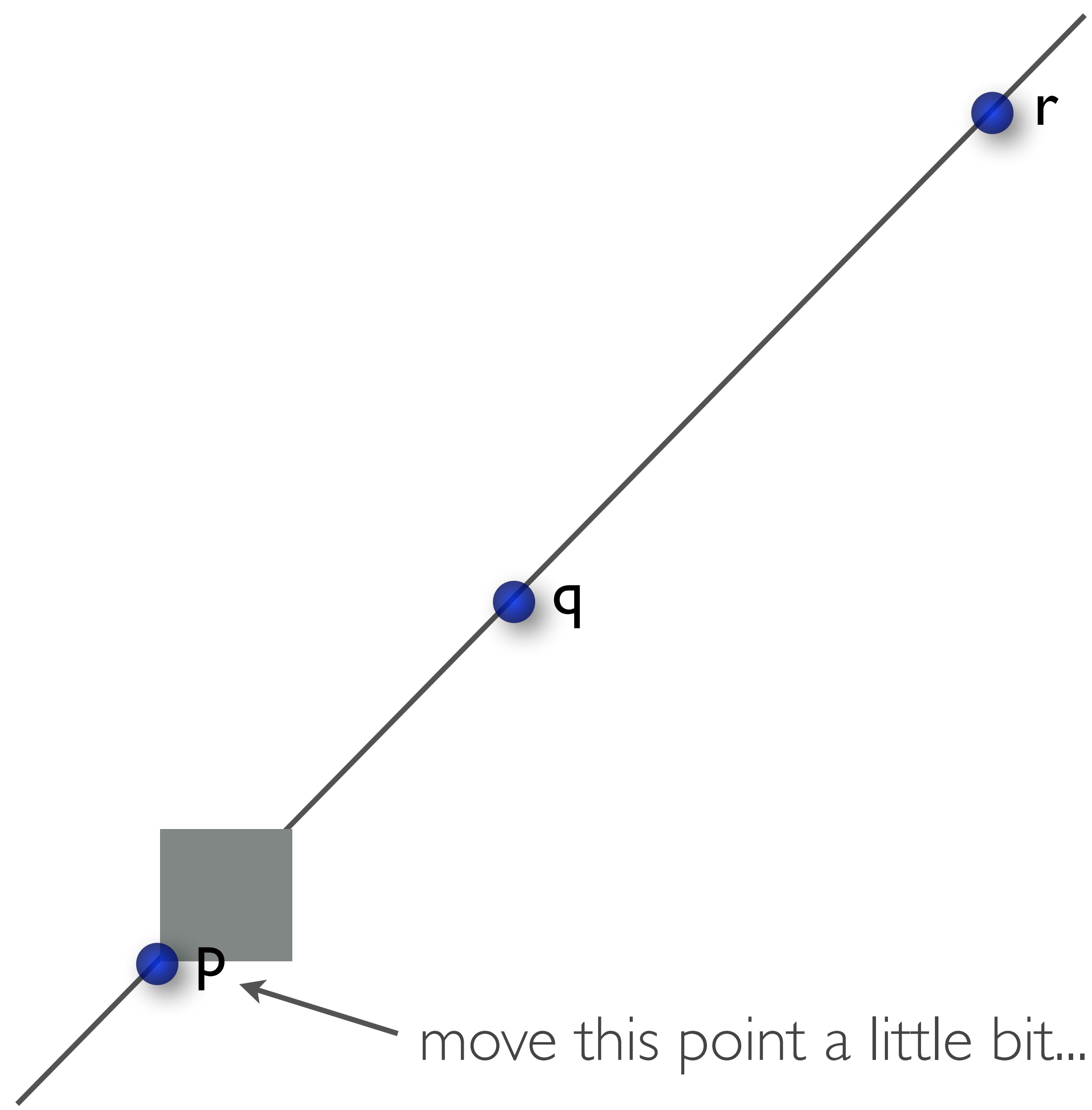
STRAIGHT LINES ?

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

$p = (0.5+x \cdot u, 0.5+y \cdot u)$
$q = (12, 12)$
$r = (24, 24)$

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

256x256 pixel image
red: <0, yellow: =0, blue: >0



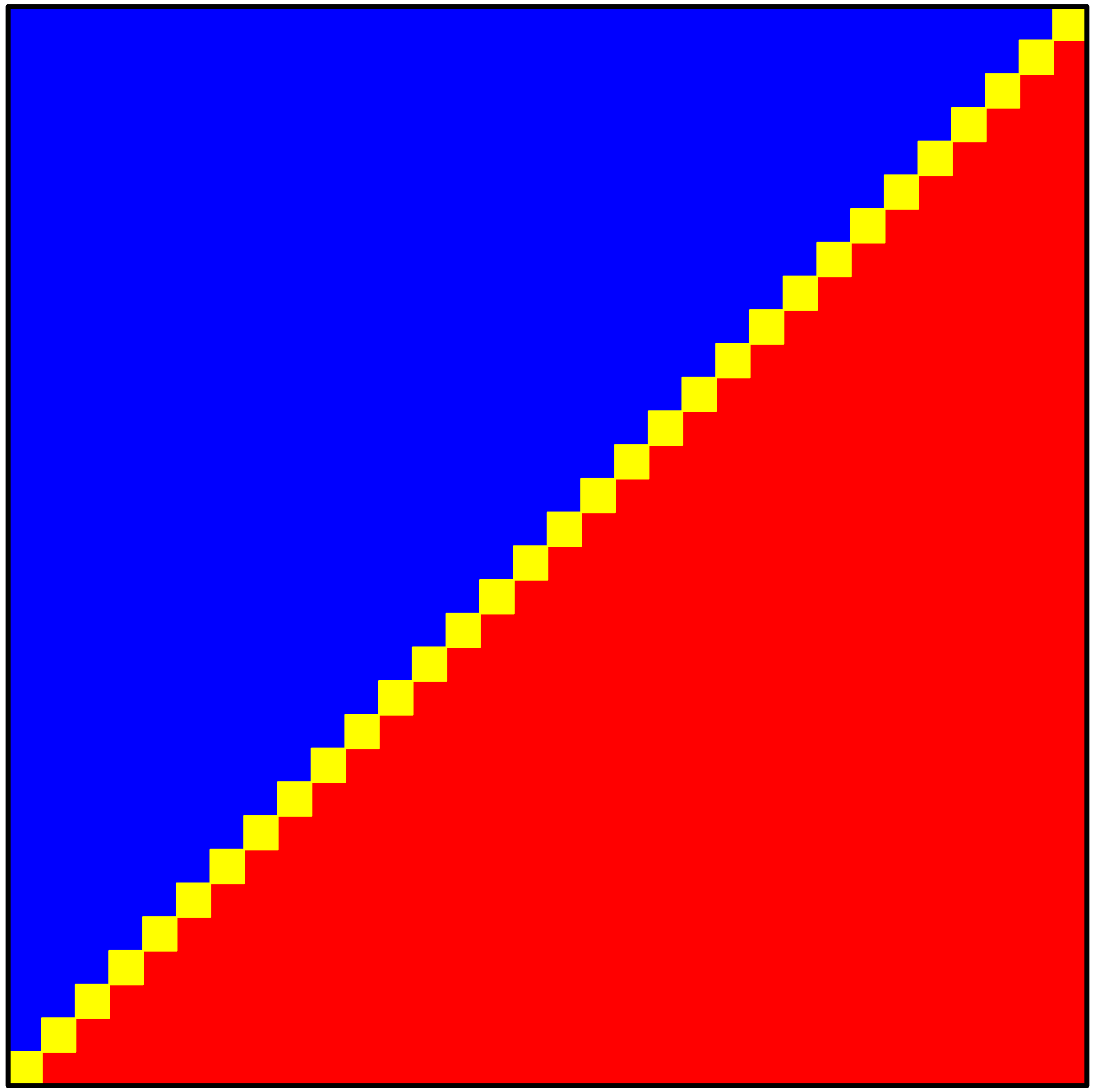
STRAIGHT LINES ?

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

$p = (0.5+x \cdot u, 0.5+y \cdot u)$
$q = (12, 12)$
$r = (24, 24)$

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

256x256 pixel image
red: <0, yellow: =0, blue: >0



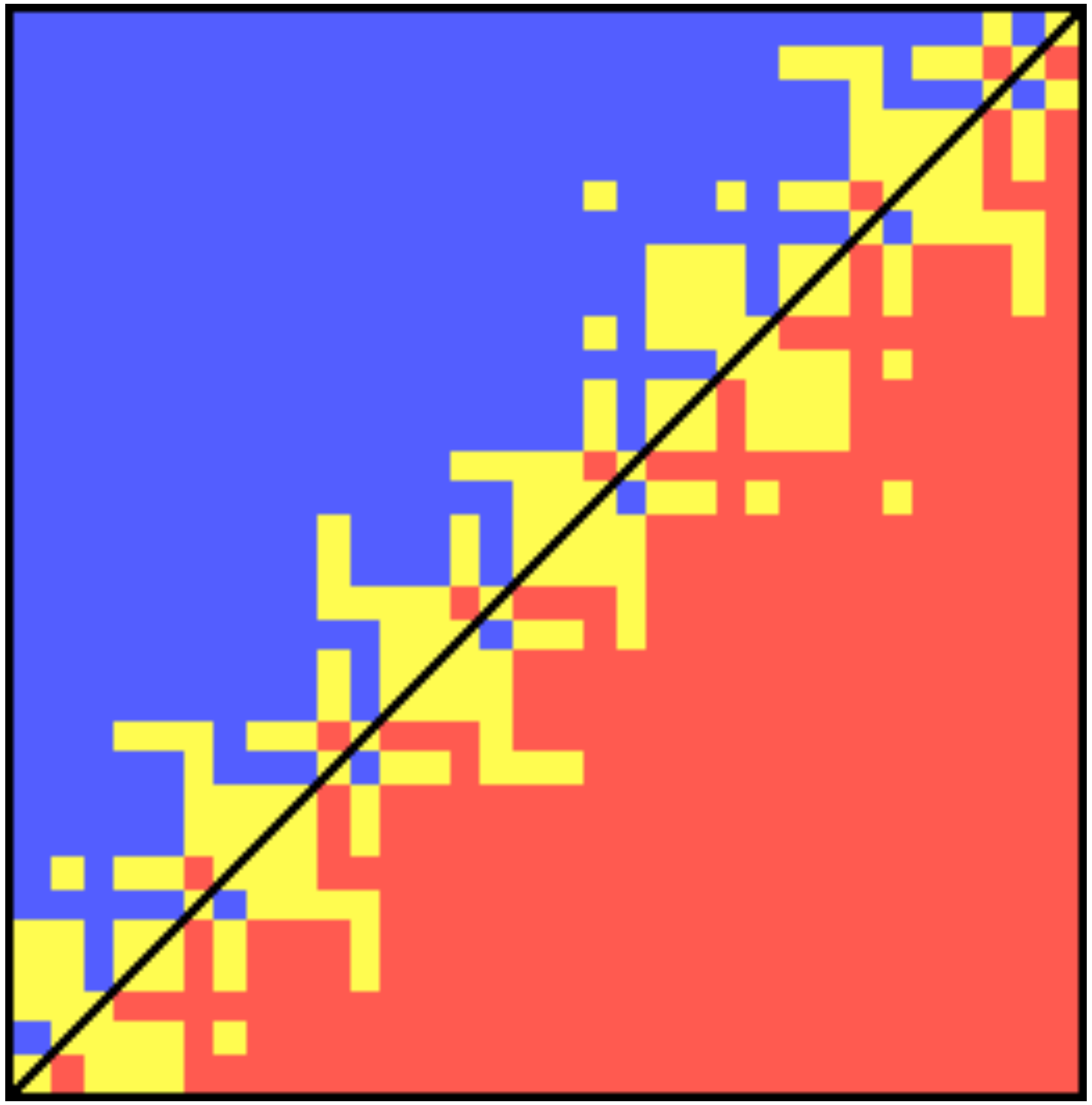
STRAIGHT LINES ?

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

$p = (0.5+x \cdot u, 0.5+y \cdot u)$
$q = (12, 12)$
$r = (24, 24)$

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

256x256 pixel image
red: <0, yellow: =0, blue: >0
evaluated with **double**



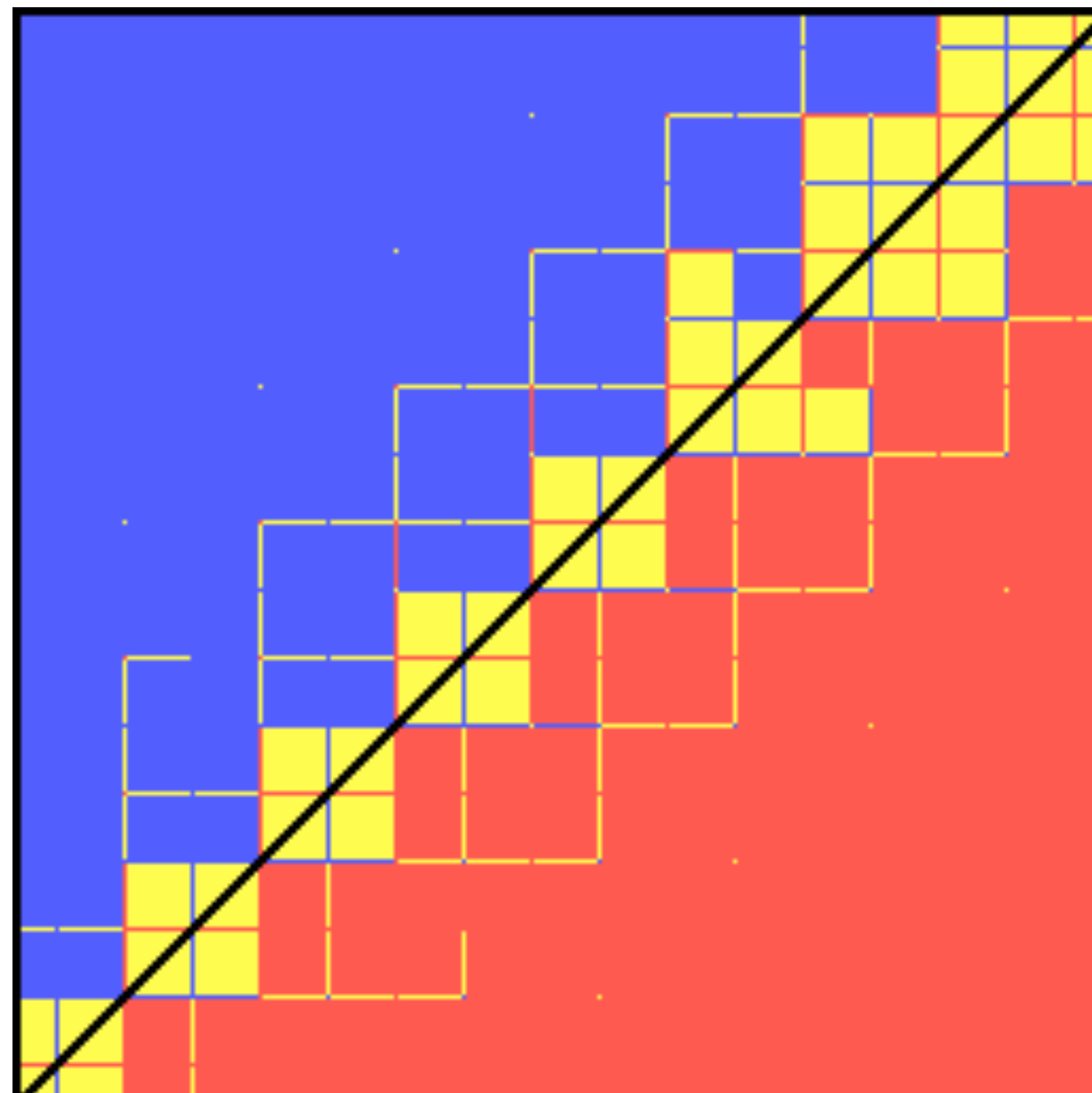
STRAIGHT LINES ?

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

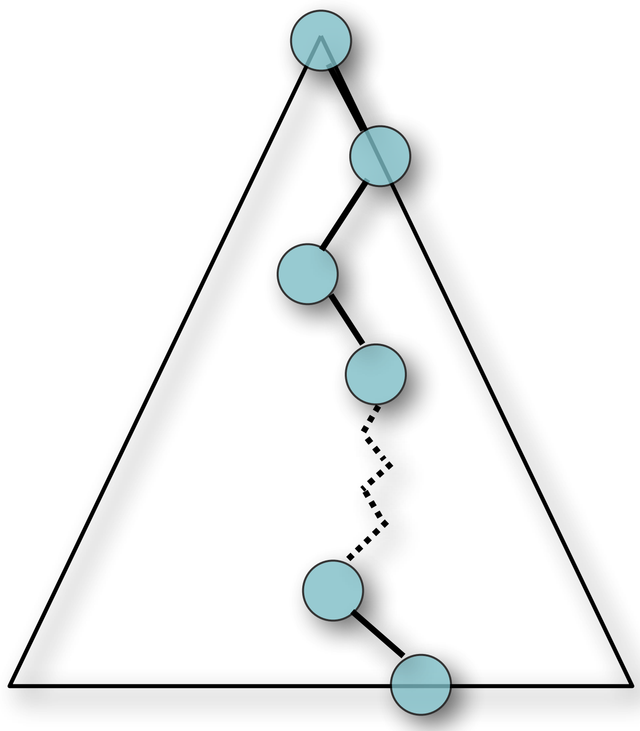
$p = (0.5+x\cdot u, 0.5+y\cdot u)$
$q = (8.8000000000000007, 8.8000000000000007)$
$r = (12.1, 12.1)$

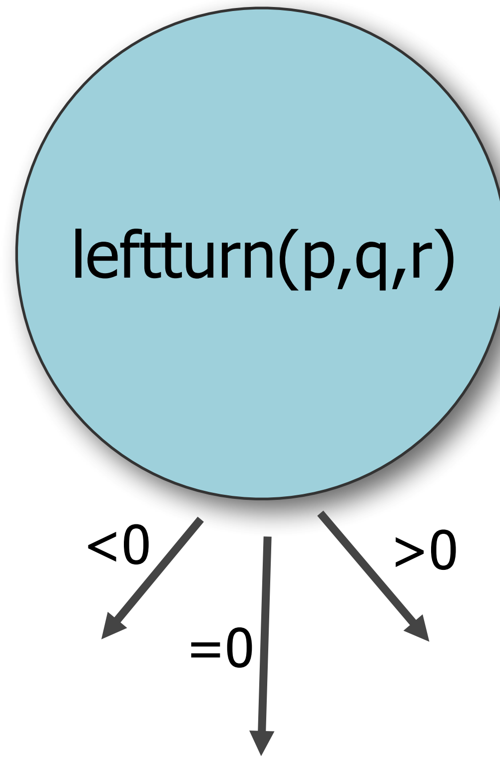
$0 \leq x, y < 256, \; u = 2^{-53}$

256x256 pixel image
red: <0, yellow: =0, blue: >0
evaluated with **`double`**



EXACT COMPUTATION





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

Combinatorial Algorithm

Geometric Predicate

Algebraic Computation

Using tools from algebra, ensure that all predicates are computed correctly.

Correctness

SO, THAT'S IT? 😀

Not quite… 😞

Exact algebraic computation comes at a cost.

Usually, arithmetic operations are assumed to have unit cost.

If numbers grow, this assumption becomes invalid.

⟶ Use only as much algebra as needed.

HOW TO AVOID ROOTS

$p = (p_x, p_y)$  $q = (q_x, q_y)$

$$d(p, q) = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}$$

For Euclidean distances, we need squareroots! (?)

Not necessarily! For comparison (e.g., to compute an MST) squared distances suffice.

Guideline #1: Avoid (square)roots where possible!

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

COMPUTING WITH FLOATING POINT NUMBERS

Guideline #1: Avoid (square)roots!

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

Guideline #2: Avoid divisions!

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

You saw in the 1.1-1-0.1 example how a division (here by 10) can create trouble.

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

ON THE SIZE OF NUMBERS

Type specifier	Standard	Code Expert	Min Integer	Max Integer
`int`	≥ 16 bits	32 bits	-2^{31}	$2^{31}-1$
`long`	≥ 32 bits	64 bits	-2^{63}	$2^{63}-1$
`double`	64 bits	64 bits	-2^{53}	2^{53}

When computing with numbers, they may grow.

We can get easy upper bounds using:

$$\boxed{\text{a bits}} \pm \boxed{\text{b bits}} \leq \boxed{\max\{a,b\}+1 \text{ bits}}$$

$$\boxed{\text{a bits}} \cdot \boxed{\text{b bits}} \leq \boxed{\text{a+b bits}}$$

EX. EUCLIDEAN DISTANCES

$p = (p_x, p_y)$  $q = (q_x, q_y)$

How large is d^2 for b-bit input coordinates?

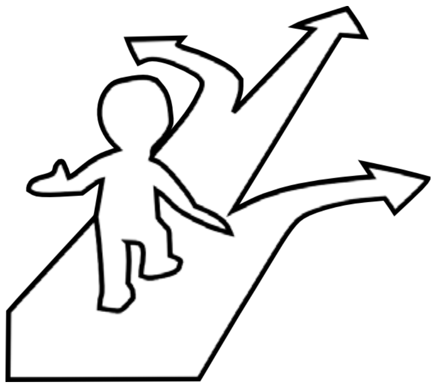
$$d^2(p, q) = \underbrace{\underbrace{(\underbrace{q_x - p_x}_{b+1})^2}_{2b+2} + \underbrace{(\underbrace{q_y - p_y}_{b+1})^2}_{2b+2}}_{2b+3}$$

$\longrightarrow$ For $b \leq 25$ `double` suffices.

PART II:

Basic Geometric Computing using CGAL

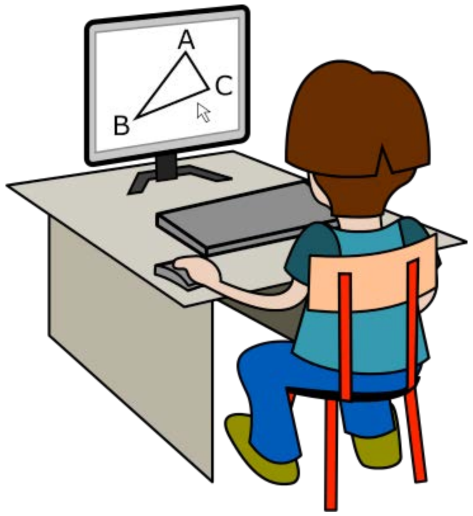
GOALS



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

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

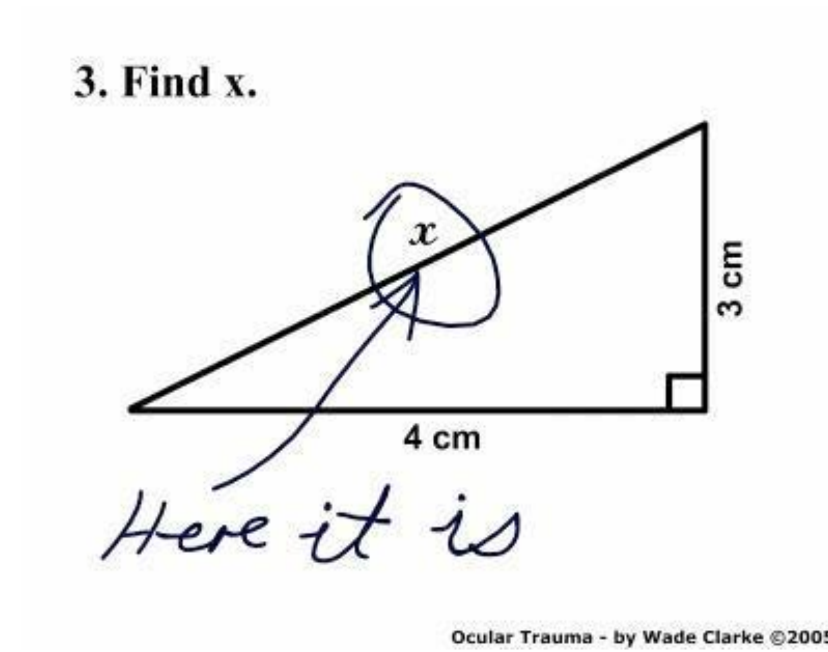
You are able to do some basic geometric computations using CGAL.



- 2D kernel objects
- Intersections
- Minimum enclosing circles

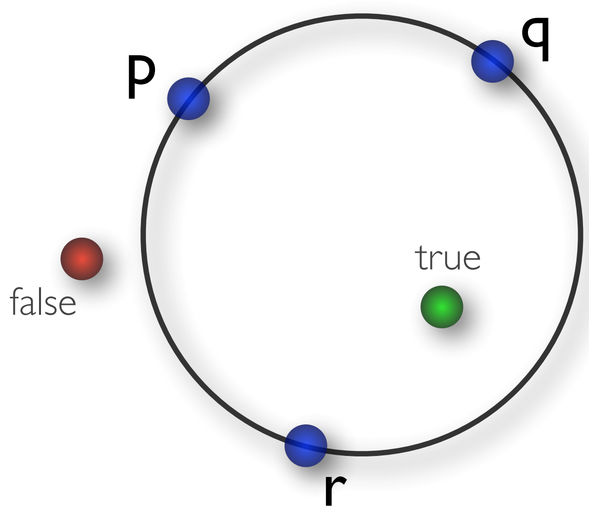
PREREQUISITES

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.

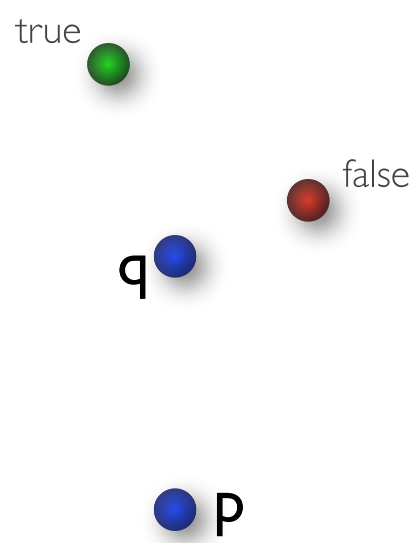


You know basic algorithmic techniques (e.g., D.P., binary search, sorting, union-find…). → You skillfully combine them with the geometric techniques discussed here.

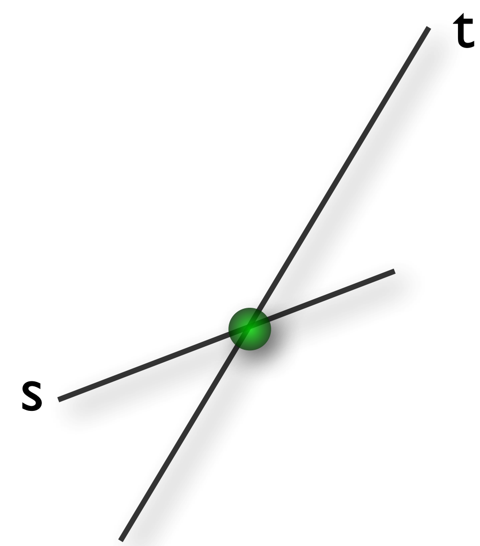
GEOMETRIC OPERATIONS



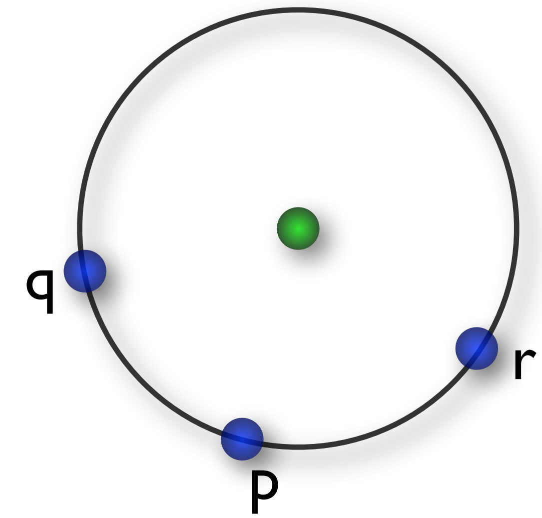
incircle(p,q,r,?)



leftturn(p,q,?)



intersection(s,t)



circumcenter(p,q,r)

Predicates

Result is constant size (e.g., true/false)

Constructions

Result is not necessarily constant size (e.g., a real number or a geometric object)

PREDICATES VS. CONSTRUCTIONS

If a program uses predicates only, numbers do not grow and it remains in the unit cost model.

Sometimes exact constructions are needed. Still, try to use as few as possible, and try to make do with elementary operations +,-,*,/.

Sometimes you need also need roots.

KERNELS

Collection of geometric data types and operations.

There is no single true way to do geometric computing.

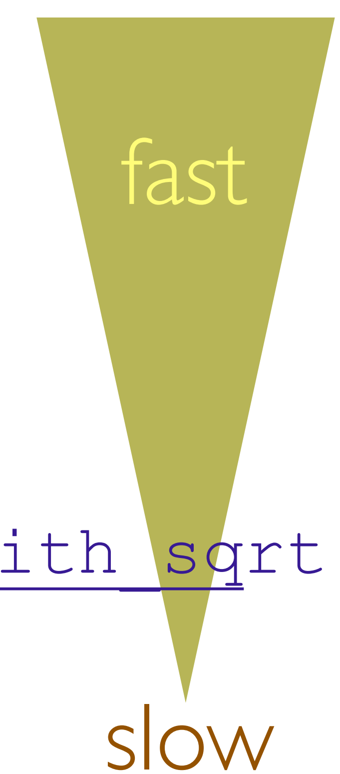
CGAL offers different kernels to serve various needs

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

Predefined defaults:

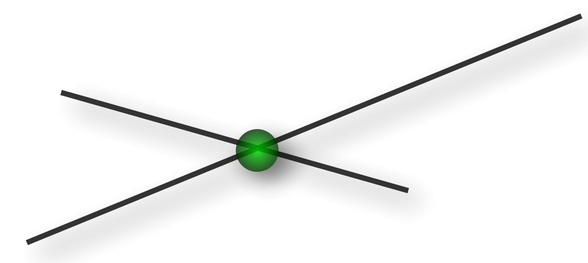
All three compute all predicates exactly.

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



WHY ARE PREDICATES EASIER?

$$a_1 \cdot x + b_1 \cdot y = c_1$$
$$a_2 \cdot x + b_2 \cdot y = c_2$$



Testing for intersection/solution:

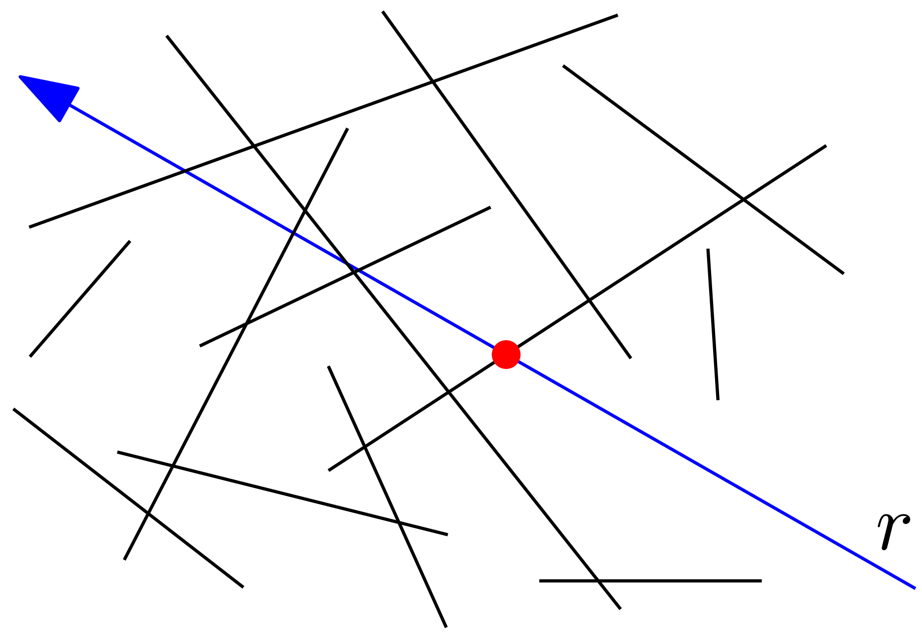
$a_1 \cdot b_2 \neq b_1 \cdot a_2$

Constructing intersection/solution:

$$x = \frac{c_1 \cdot b_2 - c_2 \cdot b_1}{a_1 \cdot b_2 - a_2 \cdot b_1} \qquad y = \frac{a_1 \cdot c_2 - a_2 \cdot c_1}{a_1 \cdot b_2 - a_2 \cdot b_1}$$

$\longrightarrow$ Three times as many computations.
Numbers grow by a factor of four.

EX. FIRSTHIT PROBLEM



Given: A set of n segments and a ray r in $\mathbb{R}^2$.
Want: The first point along r on a segment (if any).

Hard to avoid constructing intersections (needed for output). But can you reduce the number of intersection constructions (e.g., in favor of cheaper intersection tests)?

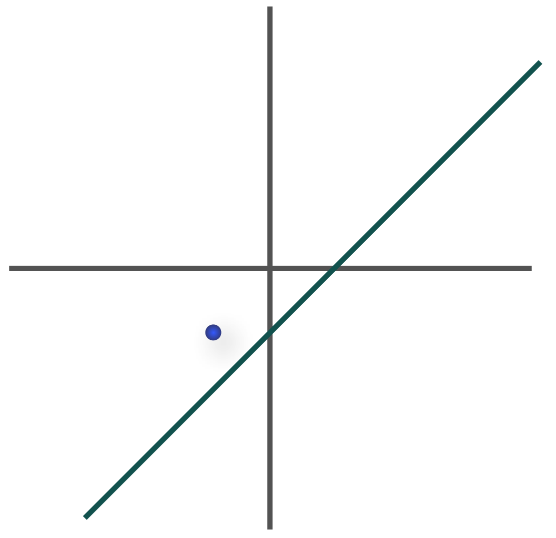
HELLO POINT

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

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 << "\n";
}
```

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

Here: `double`

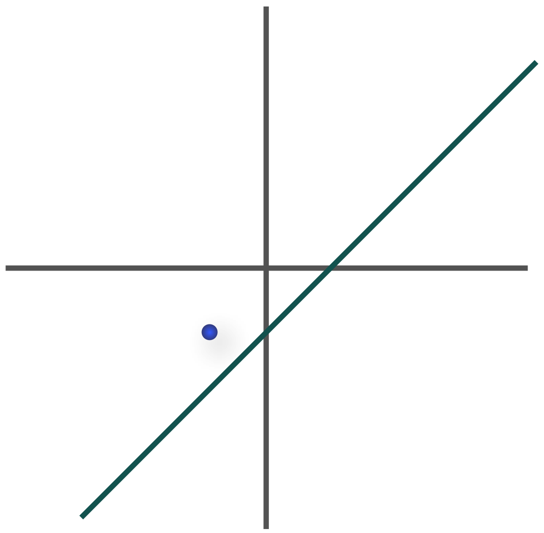
The number type used for the underlying algebra. Supports all field operations, i.e., +-*/.

Some (few) field types also support exact roots.

avoids square root computation

HELLO POINT

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

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 << "\n";
}
```



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 => nontrivial 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 nontrivial construction.
(Squared distance may be considerably larger than input coordinates, which may lead to overflow.)

CGAL::Line_2<Kernel>

Class

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*.

HELLO POINT (EXACTLY)

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

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));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

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

Compute approximation of the closest integer ≤ x. (Usually, this is ok. But there are no guarantees…)

Compare to the exact value to be sure.

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

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>

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

two FTs (Cartesian coordinates)

- Line_2 three FTs (coefficients of line equation)
- Ray_2
- Segment_2

two points

- Triangle_2 three points (corners)
- Iso_rectangle_2 (two points, opposite corners)
- Circle_2 point and FT (center and squared radius)

So that you can tell which constructions are trivial…

2D KERNEL FUNCTIONALITY

See the CGAL Manual: http://www.cgal.org

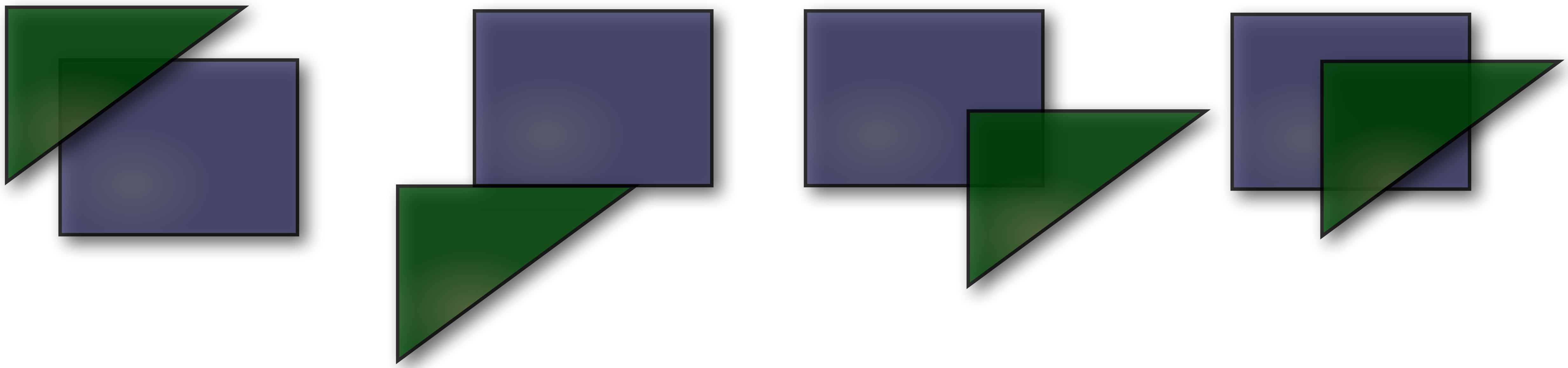
Most manual chapters have two parts:

- 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

https://algolab.inf.ethz.ch/doc/cgal/doc_html/Kernel_23/group__PkgKernel23Ref.html

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

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

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";
}
```

Needs #include <type_traits>

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

Test for intersection (predicate)

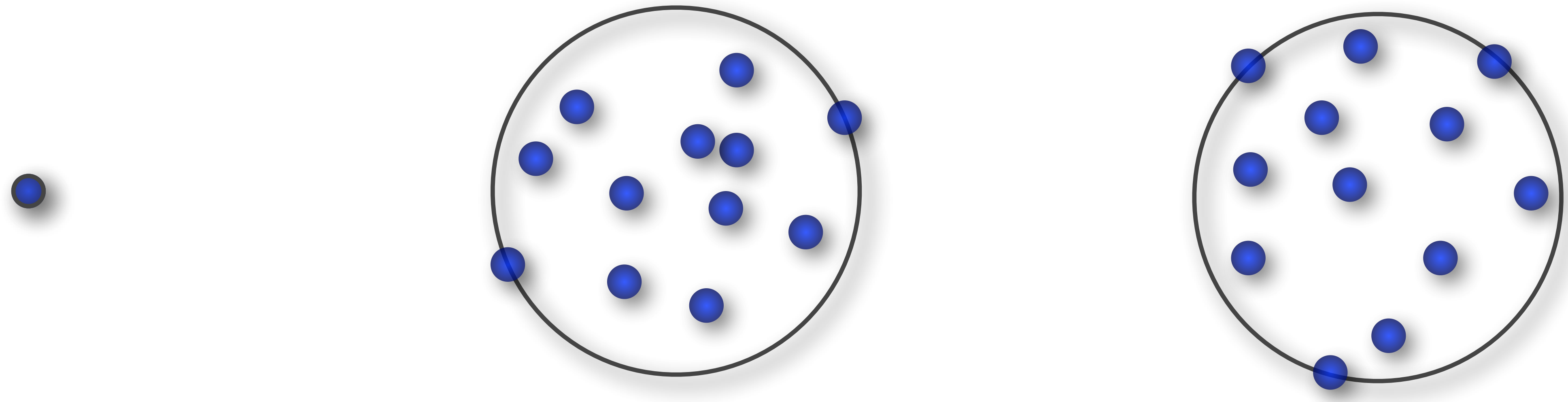
Construct intersection (construction :-))

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

Output:

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

MINIMUM ENCLOSING CIRCLE



Given: A set of n points in $\mathbb{R}^2$.
Want: Their minimum enclosing circle.

- determined by ≤ 3 support points on its boundary
- can be computed in expected linear time using

CGAL::Min_circle_2<Traits>

(uses constructions internally => Epec*)

Use this data structure, not Min_sphere_of_spheres_d.

https://algolab.inf.ethz.ch/doc/cgal/doc_html/Bounding_volumes/

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 <vector>

// 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;

int main()
{
    const int n = 100;
    std::vector<K::Point_2> P;

    for (int i = 0; i < n; ++i)
        P.push_back(K::Point_2((i % 2 == 0 ? i : -i), 0));
    // (0,0), (-1,0), (2,0), (-3,0), ...

    Min_circle mc(P.begin(), P.end(), true);
    Traits::Circle c = mc.circle();
    std::cout << c.center() << " " << c.squared_radius() << "\n";
}
```

Many data structures and algorithms have their own traits concept.
It defines the geometric primitives needed, beyond those in the kernel.

Build from a range of points.

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

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

Construct and return the circle.

Output:
`-0.5 0 9702.25`

USING CGAL

Check Section 5.2 in “A Short Introduction to C++ for the Algorithms Lab”.

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.

Then build your program using `make`

That’s it!