Robustness issues in CGAL : arithmetics and the kernel

Sylvain Pion

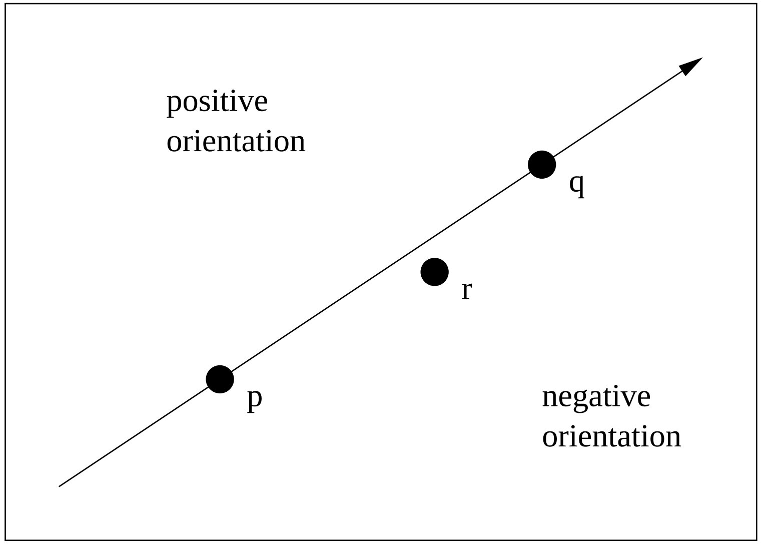
INRIA Sophia Antipolis

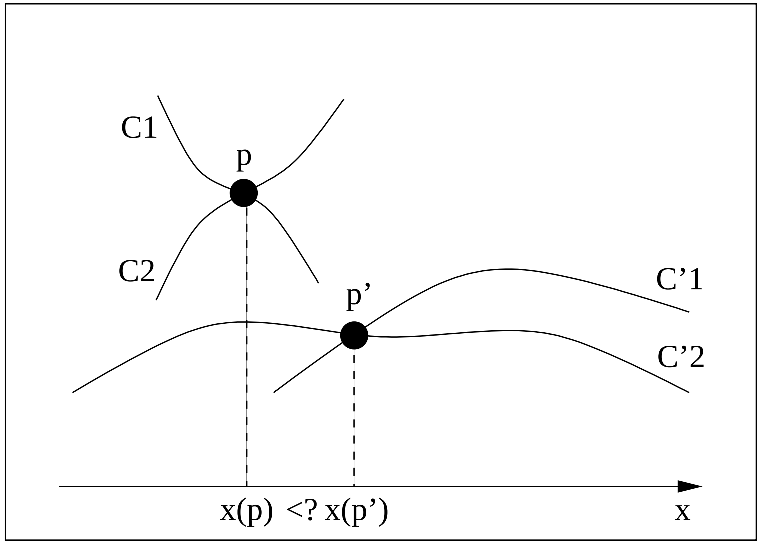
Plan

- Links between geometry and arithmetics
- Floating point arithmetic
- Exact arithmetic
- Arithmetic filters
- CGAL implementation

Introduction

Examples of geometric predicates

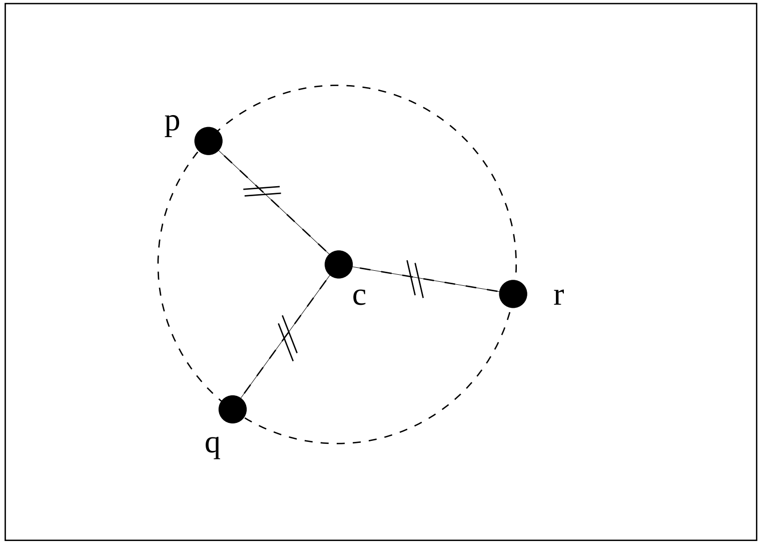


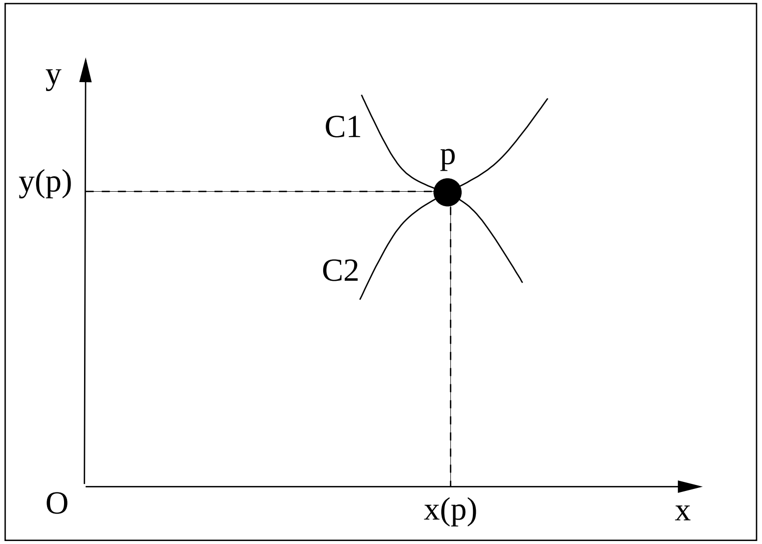


orientation$(p, q, r) =$
sign$((x(p) - x(r)) \times (y(q) - y(r)) - (x(q) - x(r)) \times (y(p) - y(r)))$

Predicate of degree 2.

Examples of geometric constructions





From geometry to arithmetic

Geometric algorithm

$\Rightarrow$ Geometric operations (predicates and constructions)

$\Rightarrow$ Algebraic operations over coordinates/coefficients

$\Rightarrow$ Arithmetic operations $(+, -, \times, \div, \sqrt{} \ldots)$

Arithmetic $\Rightarrow$ Geometry

Cost of arithmetic $\Rightarrow$ Time complexity of geometric algorithms

Approximate arithmetic $\Rightarrow$ robustness problems of geometric algorithms

The Real-RAM model

Real computer model with random access (RAM = Random access machine).

Theoretical model specifying the behavior of real arithmetic on computers.

- All arithmetic operations over reals cost O(1) time (and are exact).
- All real variables take O(1) memory space.

Complexity analyses of geometric algorithms are traditionnaly performed within this model.

Relationship with the reality of computers ?

Two approaches :

- Floating point arithmetic, approximate.
- Exact arithmetic, slower.

For geometry : which approach is the best in practice ?

What is the precise cost of the exact approach ?

Floating point arithmetic

IEEE 754 Standard

Standardization of basic FP operations on computers (1985).

Machine representation of $(-1)^s \times 1.m \times 2^e$ (for double precision, 64 bits):

s	exponent	mantissa
1	11	52

- 5 operations : $+, -, \times, \div, \sqrt{}$
- 4 rounding modes : to nearest (representable number), towards 0, towards $+\infty$, towards $-\infty$.
- Special values : $+\infty$, $-\infty$, denormals, NaNs.
- Relatively well supported by the industry (languages, compilers, processors).

Ref : `http://stevehollasch.com/cgindex/coding/ieeefloat.html`

Rounding errors

Definition : x being a positive FP value, and y the smallest FP value greater than x, we define $\texttt{ulp}(x) = y - x$ (Unit in the Last Place).

Remark 1 : $\texttt{ulp}(x)$ is a power of 2 (or ∞).
Remark 2 : In normal cases : $\texttt{ulp}(x) \simeq x2^{-53}$

Property : For all operations $+, -, \times, \div, \sqrt{}$,
the difference between the computed value r and the exact value,
the rounding error, is smaller than :
$\texttt{ulp}(r)/2$ for the rounding to nearest mode, and
$\texttt{ulp}(r)$ otherwise.

Attention : This is only true for operations taken one at a time.

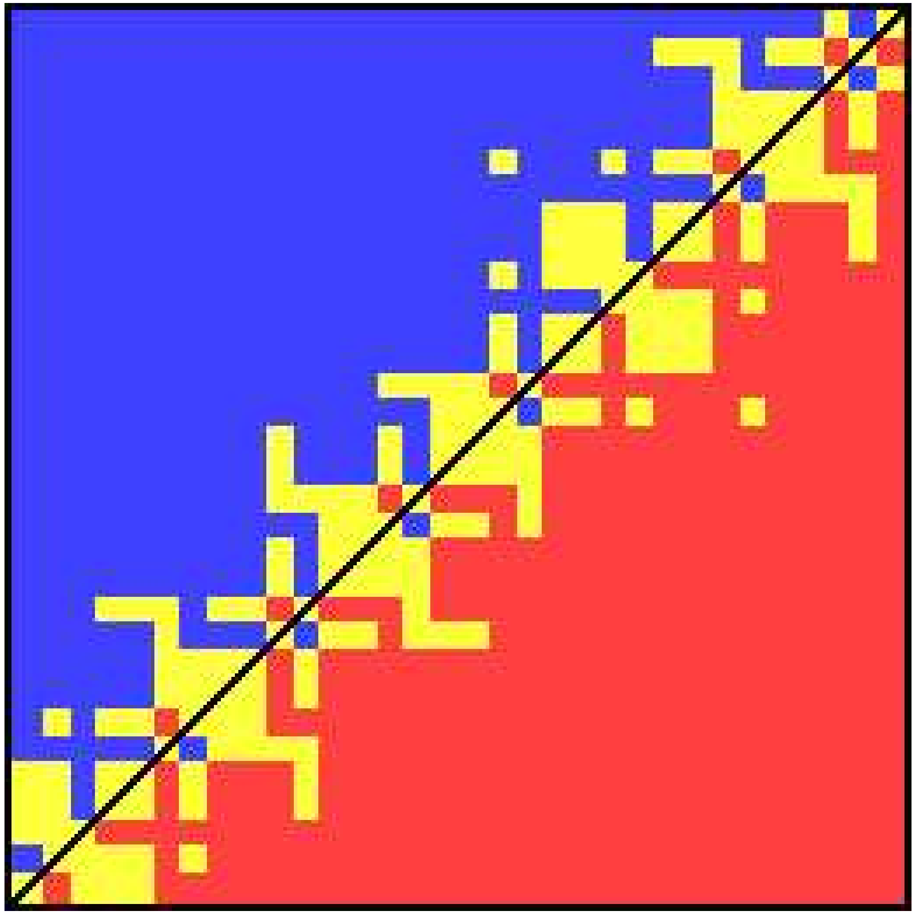
Some properties of FP arithmetic

The is no underflow for $+, -$:
$a - b = 0 \Longleftrightarrow a = b$

Detection of NaNs:
$a = a \Longleftrightarrow a$ is not a NaN

Monotonicity for a given rounding mode:
$a + b <= c + d$ computed $\Longleftrightarrow a + b <= c + d$ exact
(idem for the other operations)

Geometry of the approximate `orientation` predicate



[Kettner-Mehlhorn-Schirra-P-Yap 04]

Multiple precision computation

Multiple precision

Exact computing over integers ($\mathbb{Z}$) :

- O($n = \log N$) memory
- $+, -$: O(n) time.
- $\times, \div$:

O(n^2)	if n small
O($n^{\approx 1.5}$)	if n average (Karatsuba)
O($n \log n \log\log n$)	if n large (Schönhage Strassen)

Exact evaluation of polynomials over integral inputs of size O(n) : $\geq$ O(nd)

Libraries : GMP, LEDA, CGAL, BigNum...

Karatsuba multiplication

We cut the operands x and y in two parts of equal size (most and least significant bits) :

MSBs	LSBs
x_1	x_0

Let b the power of 2 such that $x = x_1 b + x_0$ and $y = y_1 b + y_0$. We see that :

$$xy = (b^2 + b)x_1 y_1 - b(x_1 - x_0)(y_1 - y_0) + (b + 1)x_0 y_0$$

So, we use 3 multiplications of numbers of size $n/2$ (instead of 4).

Asymptotic complexity : O($n^{log(3)/log(2)=1.585}$)

To know more : `http://www.swox.com/gmp/manual/Algorithms.html`

Rational numbers

Just a pair of exact integers : numerator / denominator.

Attention : even the addition doubles the number of bits !

Normalization can be used (not free...) to reduce the size :

- Either we are lucky (small probability).
- Either we missed an algebraic simplification.
- Other cases ?

Otherwise : exponential growth with the depth of operations.

Multiple precision floating point numbers

$m2^e$, where m and e are multiple precision integers.

It's possible to add a precision p to x such that :

$m2^e - 2^p \leq x \leq m2^e + 2^p$

p can be specified to each operation, or globally.

p can be propagated.

Libraries : MPFR, CGAL::MP_Float.

Error propagation

Let (x, p_x) be a multiprecision FP number and an associated precision corresponding to a real X. Similarly for (y, p_y).

Then we can get an approximation of $X + Y$ by $(x + y, p_{x+y})$, where:

$$\begin{aligned} |(X - x) + (Y - y)| &<= |X - x| + |Y - y| \\ |(X - x) + (Y - y)| &<= 2^{p_x} + 2^{p_y} \\ |(X + Y) - (x + y)| &<= 2^{p_{x+y}} \end{aligned}$$

$$\Longrightarrow p_{x+y} = 1 + max(p_x, p_y)$$

This is true if $x + y$ is not rounded. Otherwise, it has to be taken into account.

Other arithmetic techniques in brief

- Modular arithmetic
- Separation bounds

The other extreme : filters

Optimize easy cases

Separation bounds : treat the worst cases.

Most expected case : "easy" cases, to be optimized.

Control the FP rounding errors $\Rightarrow$ we use the costly exact computations rarely.

In the "good cases", we get a solution geometricaly exact for nearly the cost of FP computation.

Dynamic filters : interval arithmetic

Idea : we replace each FP operation by an operation over an interval of FP values $[\underline{x}; \overline{x}]$ which encodes the rounding error.

Inclusion property : at each operation, the interval contains the exact value X.

Operations : we use the IEEE 754 rounding modes :

$$X + Y \longrightarrow [\underline{x}\underline{+}\underline{y}; \overline{x}\overline{+}\overline{y}]$$

$$X - Y \longrightarrow [\underline{x}\underline{-}\overline{y}; \overline{x}\overline{-}\underline{y}]$$

Optimization :

$$X + Y \longrightarrow [-((-\underline{x})\overline{-}\underline{y}); \overline{x}\overline{+}\overline{y}]$$

Less rounding mode changes.

Multiplication and division of intervals

Multiplication :

$$X \times Y \longrightarrow \left[\min(\underline{x}\,\underline{\times}\,\underline{y},\ \underline{x}\,\underline{\times}\,\overline{y},\ \overline{x}\,\underline{\times}\,\underline{y},\ \overline{x}\,\underline{\times}\,\overline{y});\ \max(\underline{x}\,\overline{\times}\,\underline{y},\ \underline{x}\,\overline{\times}\,\overline{y},\ \overline{x}\,\overline{\times}\,\underline{y},\ \overline{x}\,\overline{\times}\,\overline{y})\right]$$

In practice, we use comparison tests for the different cases before doing the multiplications.

Division : similar.

Division by zero treatment.

Comparisons

Thanks to the inclusion property, if

$$[\underline{x}; \overline{x}] \cap [\underline{y}; \overline{y}] = \emptyset$$

then we can decide if $X < Y$ or $X > Y$.

Otherwise, we can not decide the comparison.

$\Longrightarrow$ Filter failure

Static filters

Static analysis of the rounding error propagation over the evaluation of a polynomial, supposing bounds on the inputs.

Notations : x is a real variable, $\mathtt{x}$ its value computed with doubles, $\mathtt{e_x}$ and $\mathtt{b_x}$ are doubles such that :

$$\begin{cases} \mathtt{e_x} \geq |x - \mathtt{x}| \\ \mathtt{b_x} \geq |\mathtt{x}| \end{cases}$$

Initially, we can get a rounded value to the nearest (if the values are not representable by a `double`) :

$$\begin{cases} \mathtt{b_x} = |\mathtt{x}| \\ \mathtt{e_x} = \frac{1}{2}\mathtt{ulp}(\mathtt{x}) \end{cases}$$

Addition and subtraction

Error propagation over an addition $z = x + y$ is the following :

$$\begin{cases} \mathtt{b_z} = \mathtt{b_x} + \mathtt{b_y} \\ \mathtt{e_z} = \mathtt{e_x} \overline{+} \mathtt{e_y} \overline{+} \frac{1}{2}\mathtt{ulp(z)} \end{cases}$$

Indeed :

$$\begin{aligned} |z - \mathtt{z}| &= \big|\underbrace{(z - (x + y))}_{=0} + \underbrace{((x + y) - (\mathtt{x} + \mathtt{y}))}_{\leq \mathtt{e_x} + \mathtt{e_y}} + \underbrace{((\mathtt{x} + \mathtt{y}) - \mathtt{z})}_{\leq \frac{1}{2}\mathtt{ulp(z)}}\big| \\ &\leq \mathtt{e_x} \overline{+} \mathtt{e_y} \overline{+} \frac{1}{2}\mathtt{ulp(z)} \end{aligned}$$

Multiplication

Error propagation for a multiplication $z = x \times y$ is the following :

$$\left\{ \begin{array}{l} \mathtt{b_z} = \mathtt{b_x} \times \mathtt{b_y} \\ \mathtt{e_z} = \mathtt{e_x} \overline{\times} \mathtt{e_y} \mathbin{\overline{+}} \mathtt{e_y} \overline{\times} |\mathtt{x}| \mathbin{\overline{+}} \mathtt{e_x} \overline{\times} |\mathtt{y}| \mathbin{\overline{+}} \frac{1}{2}\mathtt{ulp}(\mathtt{z}) \end{array} \right.$$

Indeed :

$$\begin{array}{rcl} |z - \mathtt{z}| & = & |\underbrace{(z - (x \times y))}_{=0} + \underbrace{((x \times y) - (\mathtt{x} \times \mathtt{y}))}_{=(\mathtt{x}-x)(\mathtt{y}-y)-(\mathtt{x}-x)\times\mathtt{y}-(\mathtt{y}-y)\times\mathtt{x}} + \underbrace{((\mathtt{x} \times \mathtt{y}) - \mathtt{z})}_{\leq\frac{1}{2}\mathtt{ulp}(\mathtt{z})}| \\ & \leq & \mathtt{e_x} \overline{\times} \mathtt{e_y} \mathbin{\overline{+}} \mathtt{e_x} \overline{\times} \mathtt{y} \mathbin{\overline{+}} \mathtt{e_y} \overline{\times} \mathtt{x} \mathbin{\overline{+}} \dfrac{1}{2}\mathtt{ulp}(\mathtt{z}) \end{array}$$

Application : `orientation` predicate

Approximate FP code :

```
int orientation(double px, double py,
                double qx, double qy,
                double rx, double ry)
{
  double pqx = qx - px,  pqy = qy - py;
  double prx = rx - px,  pry = ry - py;

  double det = pqx * pry - pqy * prx;

  if (det > 0)  return  1;
  if (det < 0)  return -1;
  return 0;
}
```

Application : `orientation` predicate

Code with static filters (for inputs bounded by 1) :

```
int filtered_orientation(double px, double py,
                         double qx, double qy,
                         double rx, double ry)
{
  double pqx = qx - px,  pqy = qy - py;
  double prx = rx - px,  pry = ry - py;

  double det = pqx * pry - pqy * prx;

  const double E = 1.33292e-15;

  if (det >  E)  return  1;
  if (det < -E)  return -1;

  ... // can't decide => call the exact version
}
```

Variants : Ex : computing the bound at run time

```
int filtered_orientation(double px, double py,
                         double qx, double qy,
                         double rx, double ry)
{
  double b = max_abs(px, py, qx, qy, rx, ry);

  double pqx = qx - px,  pqy = qy - py;
  double prx = rx - px,  pry = ry - py;

  double det = pqx * pry - pqy * prx;

  const double E = 1.33292e-15;

  if (det >  E*b*b)  return  1;
  if (det < -E*b*b)  return -1;

  ... // can't decide => call the exact version
}
```

Filter failure rates probabilities

Theoretical study : [Devillers-Preparata-99]

Inputs uniformly distributed in a unit square/cube :

`orientation 2D`	10^{-15}
`orientation 3D`	5.10^{-14}
`in_circle 2D`	10^{-11}
`in_sphere 3D`	7.10^{-10}

... for data homogeneously distributed.

On more degenerate cases

	Dynamic	Semi-static
Random	0	870
$\varepsilon = 2^{-5}$	0	1942
$\varepsilon = 2^{-10}$	0	662
$\varepsilon = 2^{-15}$	0	8833
$\varepsilon = 2^{-20}$	0	132153
$\varepsilon = 2^{-25}$	10	192011
$\varepsilon = 2^{-30}$	19536	308522
Grid	49756	299505

Number of failures of dynamic and static filters during the computation of Delaunay (10^5 points). Inputs on a integer grid of 30 bits, with relative perturbation.

Comparison : dynamic vs static filters

Can fail more often that interval arithmetic (less precise), but faster.

Static filters harder to write : needs analysis of each predicate.

Fastest scheme : cascade several methods.

Filters : remarks

Fragile : try to avoid bad cases in algorithms !

- Avoid cascaded computations (use original inputs)

- Avoid testing degenerate cases if you know them (created by the algorithm).

- Avoid constructions, because faster solutions are available for predicates.

Current work

- Automatic code generation, from a generic version, for the best methods.
- Filtering of geometric constructions.
- Rounding of constructions.

Implementation in CGAL

Algorithms and traits classes

Algorithms are parameterized (templates) by geometric traits classes, which provide :

- types of the objects manipulated by the algorithm : Point_2, Tetrahedron_3...
- predicates that the algorithm applies to the objects : Orientation_2, Side_of_oriented_sphere_3...
- constructions : Mid_point_2, Construct_circumcenter_3, Compute_squared_length_2...

The last 2 are provided as function objects.

Needs of algorithms are described towards its trais parameter as a concept.

Kernels

The kernel gathers many objects types, predicates and constructions, and can be used as parameter for the traits classes directly to many algorithms.

Classical kernels, parameterized by number types :
Cartesian<FT>
Homogeneous<RT>

Ex : Triangulation_3<Cartesian<double> >

Cartesian<double> is a model for the concept TriangulationTraits_3.

The kernel functionality is also available via global functions : `CGAL::orientation(p, q, r)`..

Number types

Valid parameters for the kernels Cartesian...

FP :
double, float

Multi-precision :
Gmpz, Gmpq, CGAL::MP_Float, leda::integer...

Number types including some filtering :
leda::real, CORE::Expr, CGAL::Lazy_exact_nt<>

Internal tools

Interval arithmetic : CGAL::Interval_nt, boost::interval

Generator of filtered predicates (dynamic) using C++ exceptions : CGAL::Filtered_predicate<>

Filtered kernels

CGAL::Filtered_kernel$< K >$ provides some predicates with static filters, and all others with dynamic filters.

Recommended kernels :
CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Exact_predicates_inexact_constructions_kernel

Example

```
template < typename K >
struct My_orientation_2
{
  typedef typename K::RT          RT;
  typedef typename K::Point_2   Point_2;

  CGAL::Orientation
  operator()(const Point_2 &p, const Point_2 &q,
             const Point_2 &r) const
  {
    RT prx = p.x() - r.x();    RT pry = p.y() - r.y();
    RT qrx = q.x() - r.x();    RT qry = q.y() - r.y();
    return static_cast<CGAL::Orientation> (
             CGAL::sign( prx*qry - qrx*pqy ) );
  }
};
```

Example

```
// Using it
typedef CGAL::Cartesian<double>  Kernel;
Kernel::Point_2 p(1, 2), q(2, 3), r(4, 5);
My_orientation_2<Kernel> orientation;
CGAL::Orientation ori = orientation(p, q, r);
```

Using Filtered_predicate

```
typedef CGAL::Simple_cartesian<double> K;
typedef CGAL::Simple_cartesian<CGAL::Interval_nt_advanced> FK;
typedef CGAL::Simple_cartesian<CGAL::MP_Float> EK;
typedef CGAL::Cartesian_converter<K, EK> C2E;
typedef CGAL::Cartesian_converter<K, FK> C2F;

typedef CGAL::Filtered_predicate<My_orientation_2<EK>,
                                 My_orientation_2<FK>,
                                 C2E, C2F> Orientation_2;

...
  K::Point_2 p(1,2), q(2,3), r(3,4);
  Orientation_2 orientation;
  orientation(p, q, r);
  return 0;
```