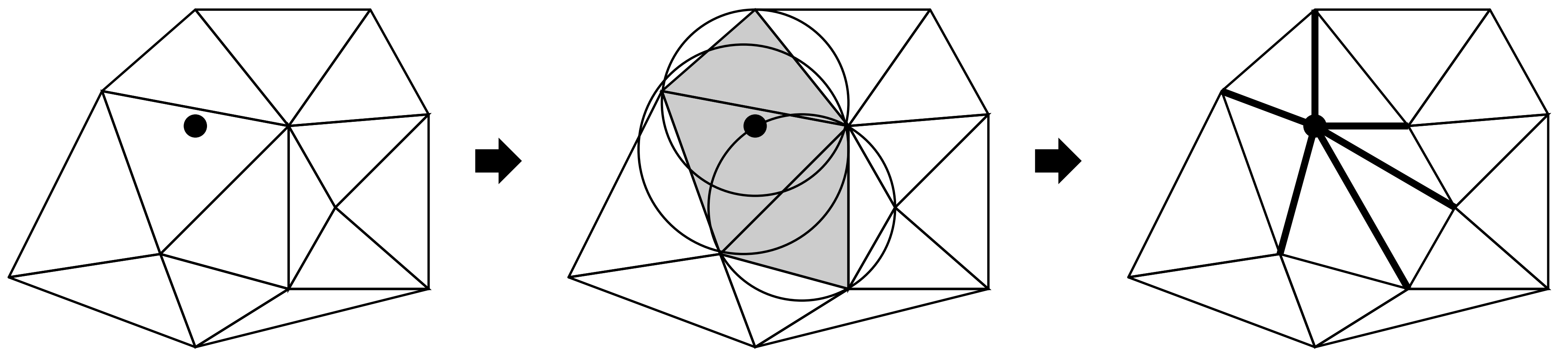
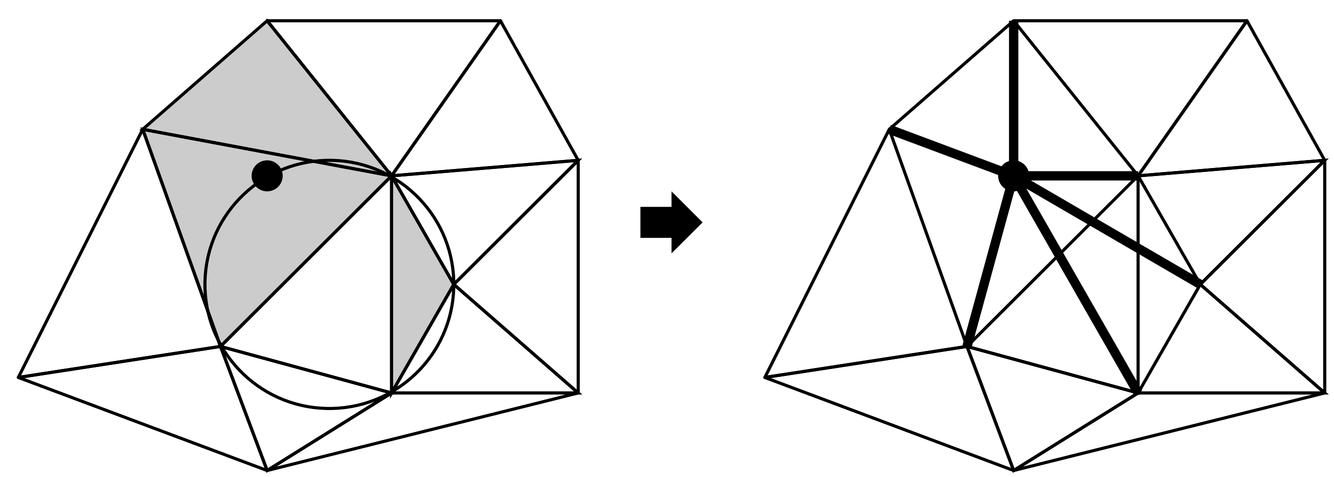
Robust and Efficient Implementation



Correct



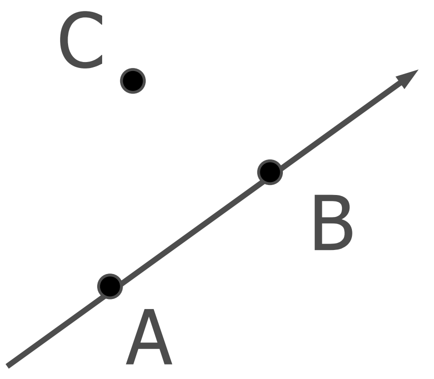
Wrong

Geometric Predicates

- Programs need to test relative positions of points based on their coordinates.
- Simple examples (in 2D):

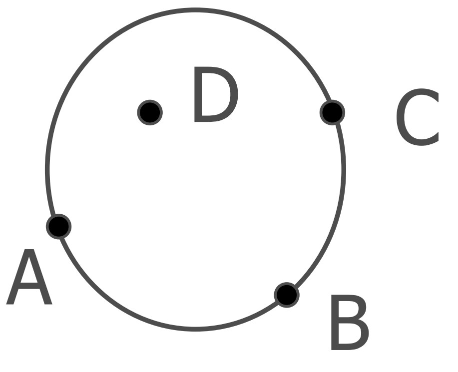
Orientation test (in convex hull)

does C lie left/right/on the line AB?

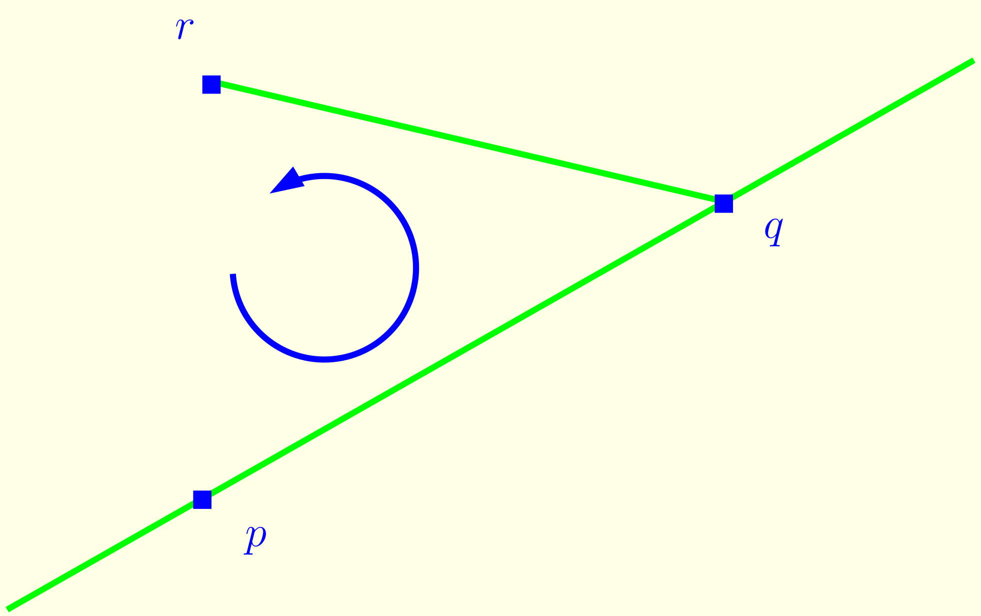


Incircle test (in Delaunay triangulation)

does D lie in/out/on the circle ABC?



Orientation of 2D points

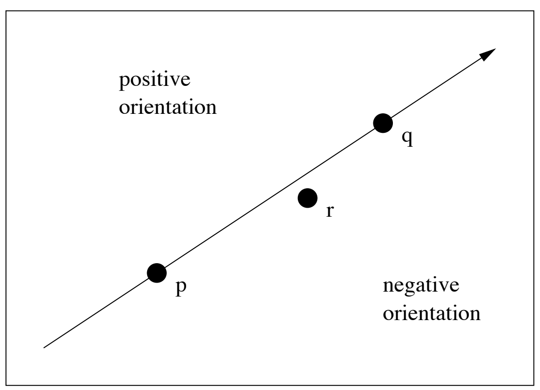


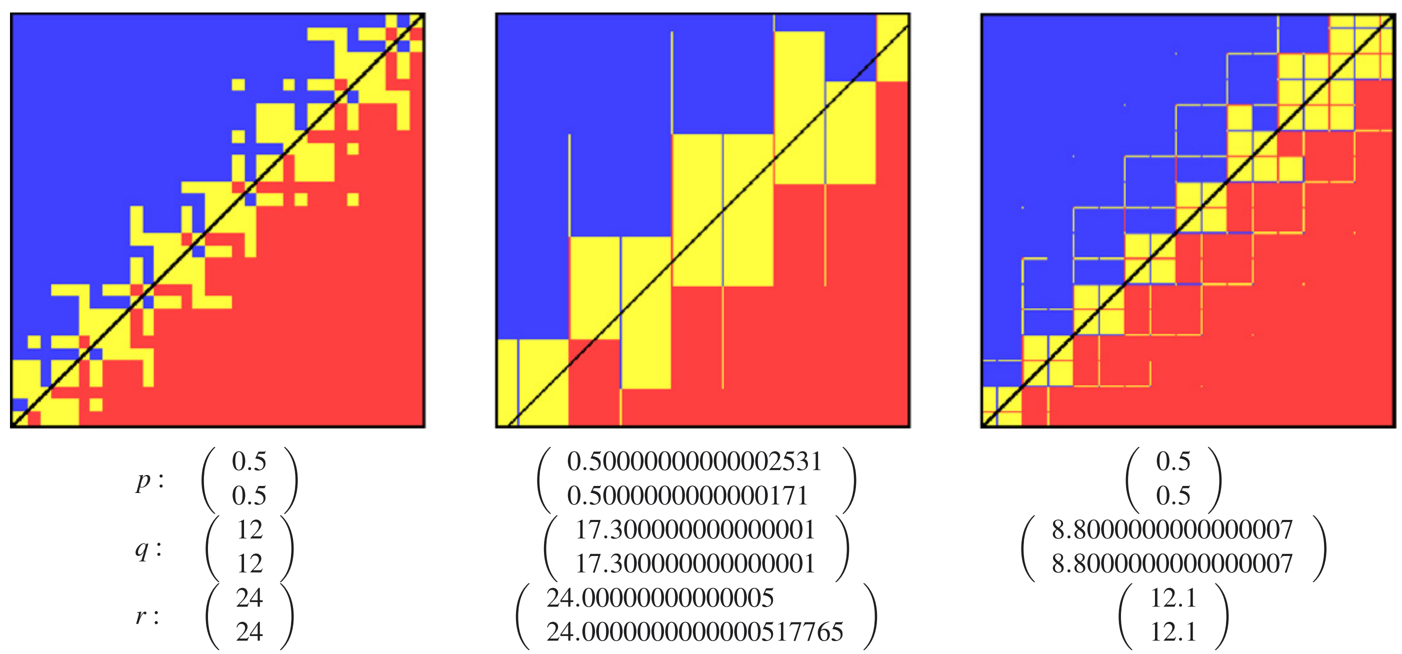
$$
\begin{aligned}
orientation(p,q,r) &= \textbf{\textit{sign}}\left(\mathsf{det}\begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}\right) \\
&= \textbf{\textit{sign}}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))
\end{aligned}
$$

- Choose three points **p**, **q**, and **r** in $\mathbb{R}^2$ and then compute

$$\mathrm{orient2d}(\mathbf{p},\mathbf{q},\mathbf{r}) = \mathrm{sign}\left(\det\begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}\right)$$

- The following figures show the results of the experiments, when the point **p** varies within a small neighborhood and the points **q** and **r** remain fixed. The blue, yellow, and red colors represent the positive, zero, and negative signs, respectively.



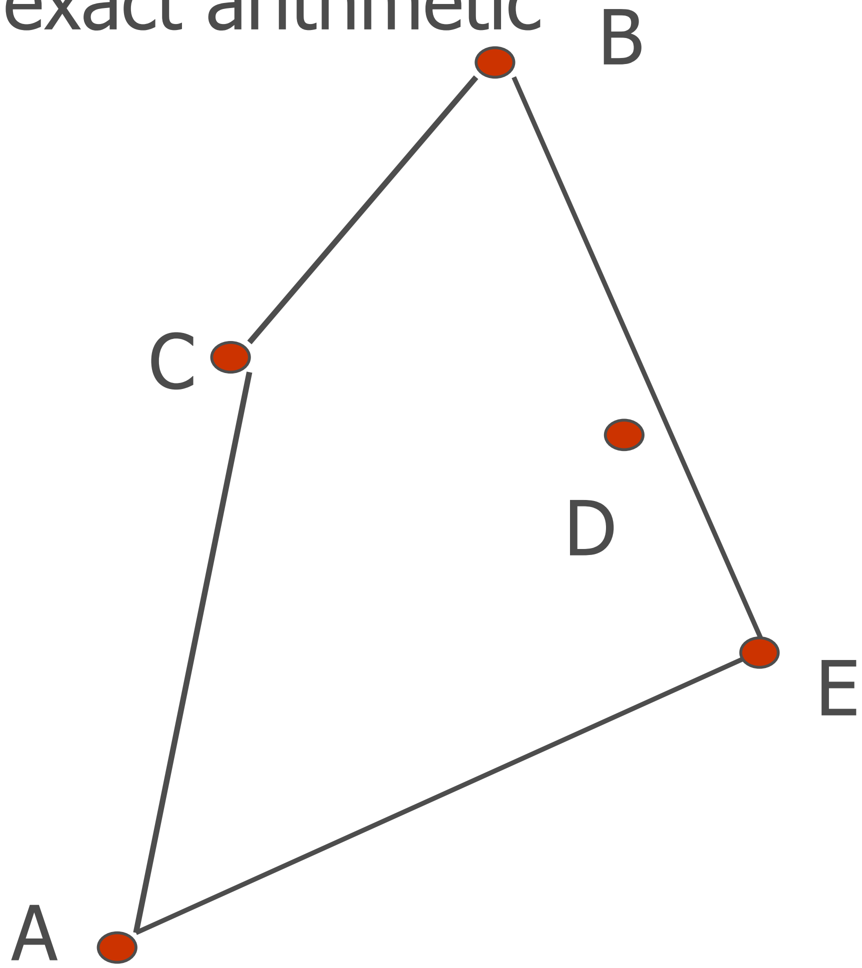


(a) (b) (c)

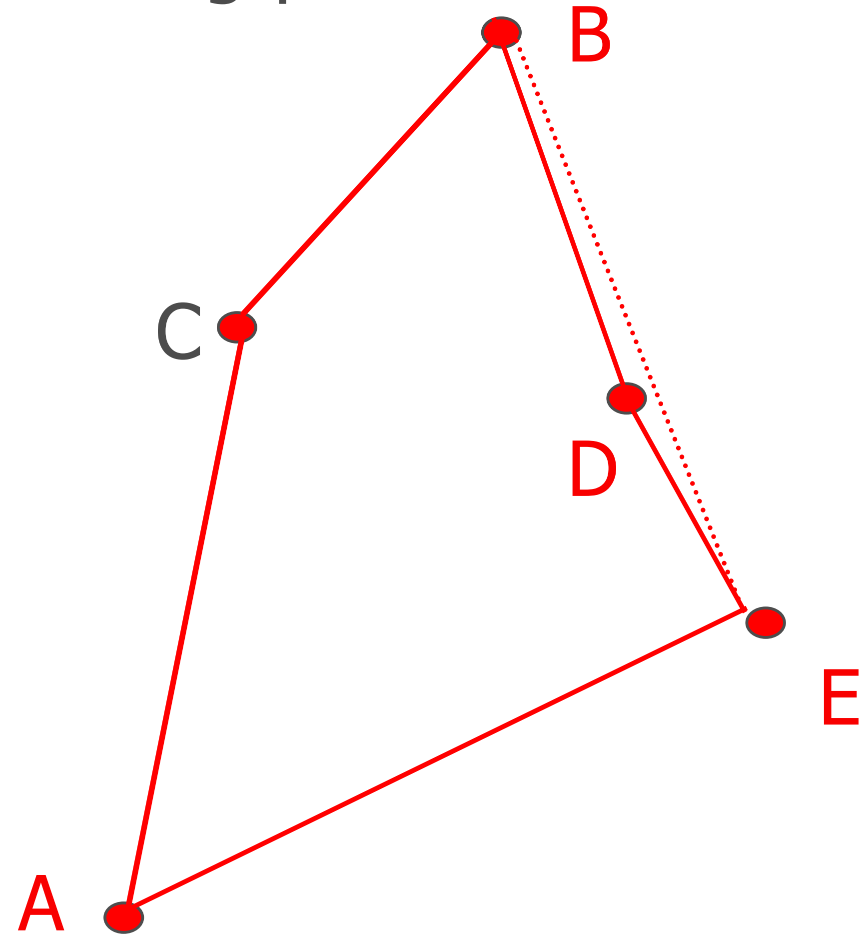
Image from [**Kettner et al. 2008**].

Convex Hull Miscomputed

Using Orientation Predicate in exact arithmetic



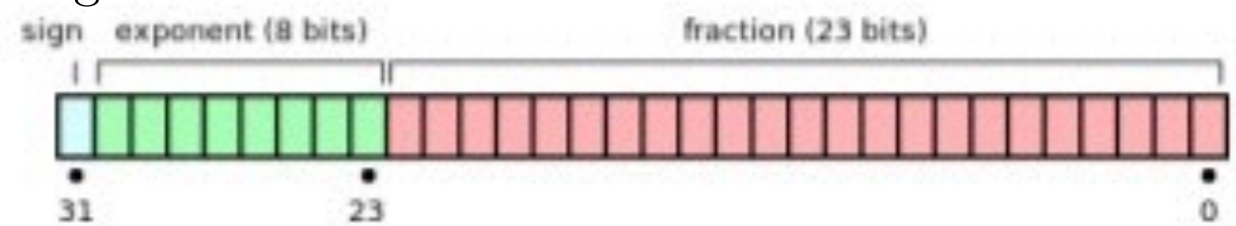
Using Orientation Predicate in floating-point arithmetic



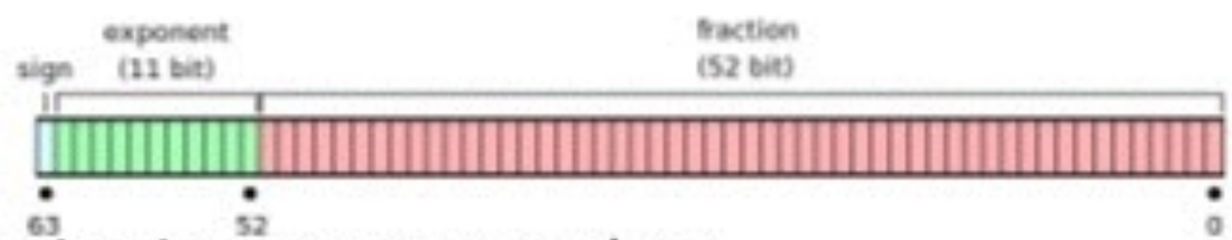
Floating Point

- **Single-precision**
  - 32 bits
    - 1 bit - sign
    - 8 bits - exponent
    - 23 bits - fraction
- **Double-precision**
  - 64 bits
    - 1 bit - sign
    - 11 bits - exponent
    - 52 bits - fraction
- Note how the size of the fraction increases more than the exponent between single-precision and double-precision
  - Accuracy (precision) is more important than range

range from 1.175494×10^{-38} to $3.40282346 \times 10^{38}$



range from $2.22507385 \times 10^{-308}$ to $1.79769313 \times 10^{308}$

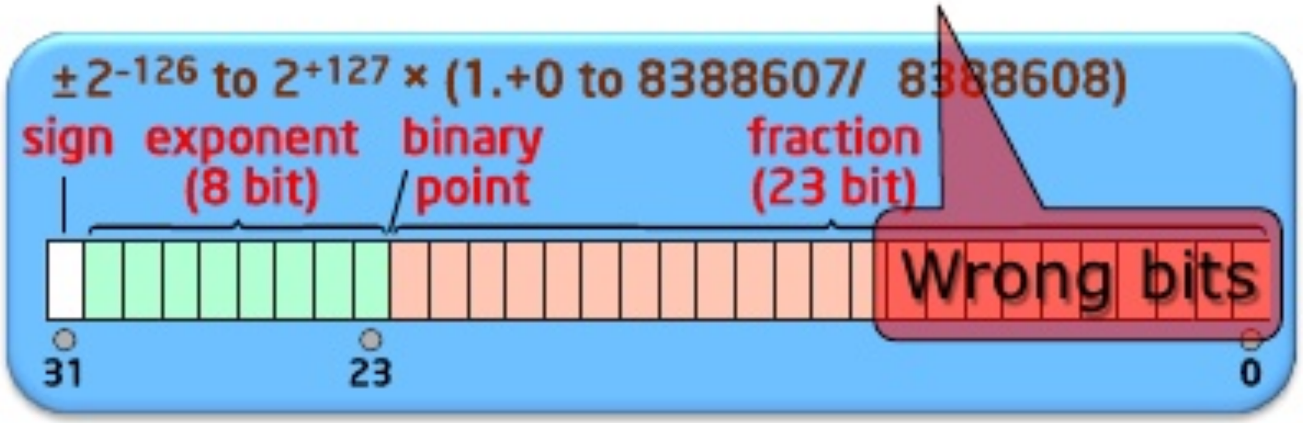


Rounding Error

"0.1" in binary is not exact. It's 0.100000002384185791015625, rounded. Programmers and disk archives use decimal, creating rounding error

Clock example: accumulating seconds, 0.1 at a time, for 100 hours, will be off by at least *three minutes*!

$(a + b) + c$
is NOT the same as
$a + (b + c)$
in floating-point math.



$a = 1.0$
$b = 100000000.$
$c = -100000000.$

$(a + b)$ rounds down to = 100000000. Add c, get 0.0.
$(b + c)$ = 0 exactly, with no rounding. Add a, get 1.0

```
float a = 0.15 + 0.15
float b = 0.1 + 0.2

if(a == b) // can be false!
if(a >= b) // can also be false!
```

Sources of Computational Errors

FLP approximates exact computation with real numbers

Two sources of errors to understand and counteract:

1. Representation errors

   e.g., no machine representation for 1/3, 2 , or π

2. Arithmetic errors

   e.g., $(1 + 2^{-12})^2 = 1 + 2^{-11} + 2^{-24}$

   not representable in IEEE format

Errors due to finite precision can lead to disasters in life-critical applications

- **Exact geometric computation**. The predicates are computed exactly. It is able to make the geometric algorithm robust [**Yap 1997**]. There are various exact computation techniques, such as the *multiple precision arithmetic* (available in the `GNU GMP library`, `http://gmplib.org`), and the *arbitrary precision floating-point arithmetics* based on the IEEE standard [**Priest 1991**]. However, performing exact arithmetics are expensive and will result a poor performance in practice.

```
{
  mpz_t integ;
  mpz_init (integ);
  ...
  mpz_add (integ, ...);
  ...
  mpz_sub (integ, ...);

  /* Unless the program is about to exit, do ... */
  mpz_clear (integ);
}
```

```
#include <gmp.h>

void
foo (void)
{
  mpz_t  n;
  int    i;
  mpz_init (n);
  for (i = 1; i < 100; i++)
    {
      mpz_mul (n, ...);
      mpz_fdiv_q (n, ...);
      ...
    }
  mpz_clear (n);
}
```

GNU MP

The GNU Multiple Precision Arithmetic Library
Edition 6.1.2
16 December 2016

line-line intersection

```c
int line_line_intersection(double X0, double Y0,
                           double X1, double Y1,
                           double X2, double Y2,
                           double X3, double Y3,
                           double *t1, double *t2)
{
  double Ux = X1 - X0;
  double Uy = Y1 - Y0;
  double Vx = X3 - X2;
  double Vy = Y3 - Y2;
  double Wx = X2 - X0;
  double Wy = Y2 - Y0;
  double det = Ux*Vy - Uy*Vx;

  double absolut = fabs(Ux*Vy) + fabs(Uy*Vx);
  if (fabs(det) / absolut < 1e-6) {
    printf("Warning: Two lines are nearly parallel.\n");
    *t1 = *t2 = 0.0;
    return 0;
  }

  *t1 = (Wx*Vy - Wy*Vx) / det;
  *t2 = (Ux*Wy - Uy*Wx) / det;
  return 1;
}
```

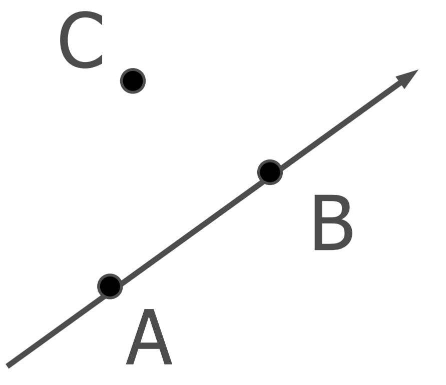
```cpp
#include <gmpxx.h>
#include <mpfr.h>

int line_line_intersection(double X0, double Y0,
                           double X1, double Y1,
                           double X2, double Y2,
                           double X3, double Y3,
                           double *t1, double *t2)
{
    mpf_set_default_prec(PRECISION);
    mpf_class x0,x1,x2,x3,y0,y1,y2,y3;
    x0=X0;x1=X1;x2=X2;x3=X3;
    y0=Y0;y1=Y1;y2=Y2;y3=Y3;

    mpf_class Ux = x1 - x0;
    mpf_class Uy = y1 - y0;
    mpf_class Vx = x3 - x2;
    mpf_class Vy = y3 - y2;
    mpf_class Wx = x2 - x0;
    mpf_class Wy = y2 - y0;
    mpf_class det = Ux*Vy - Uy*Vx;

    if(det==0.0)
    {
        cout<<""<<endl;
        *t1=0;
        *t2=0;
        return 0;
    }

    mpf_class r1 = (Wx*Vy - Wy*Vx) / det;
    mpf_class r2 = (Ux*Wy - Uy*Wx) / det;
    *t1=r1.get_d();
    *t2=r2.get_d();

    cout<<"PRECISION: "<<r1.get_prec()<<endl;

    return 1;
}
```

Geometric Predicates

- Programs need to test relative positions of points based on their coordinates.
- Simple examples (in 2D):

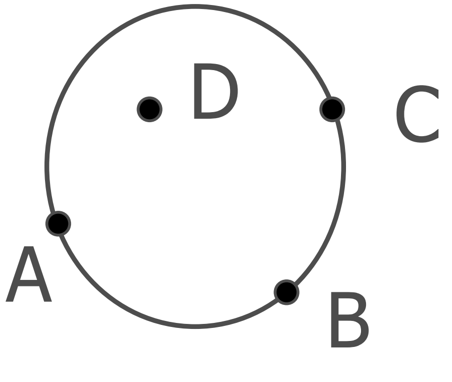
Orientation test (in convex hull)

does C lie left/right/on the line AB?



Incircle test (in Delaunay triangulation)

does D lie in/out/on the circle ABC?



Floating-point Filters

- Get **correct sign** (-1, 0 or 1) of an exact expression E using floating-point!

"filters out" the easy cases

let F = E (X) in **floating point**
    if F > **error bound** then 1 else
        if –F > error bound then –1 else
            increase precision and repeat
            or switch to exact arithmetic

- If the correct result is 0, must go to exact phase

Static filters

Static analysis of error propagation on evaluation of a polynomial expression, assuming **bounds on the input data**.

x being a positive floating point value,
and y the smallest floating point value greater than x

$$\mathtt{ulp}(x) = y - x$$

(Unit in the Last Place).

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

Application: *orientation* predicate

Approximate non guaranteed version

```
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 filtering (for entries 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
}
```

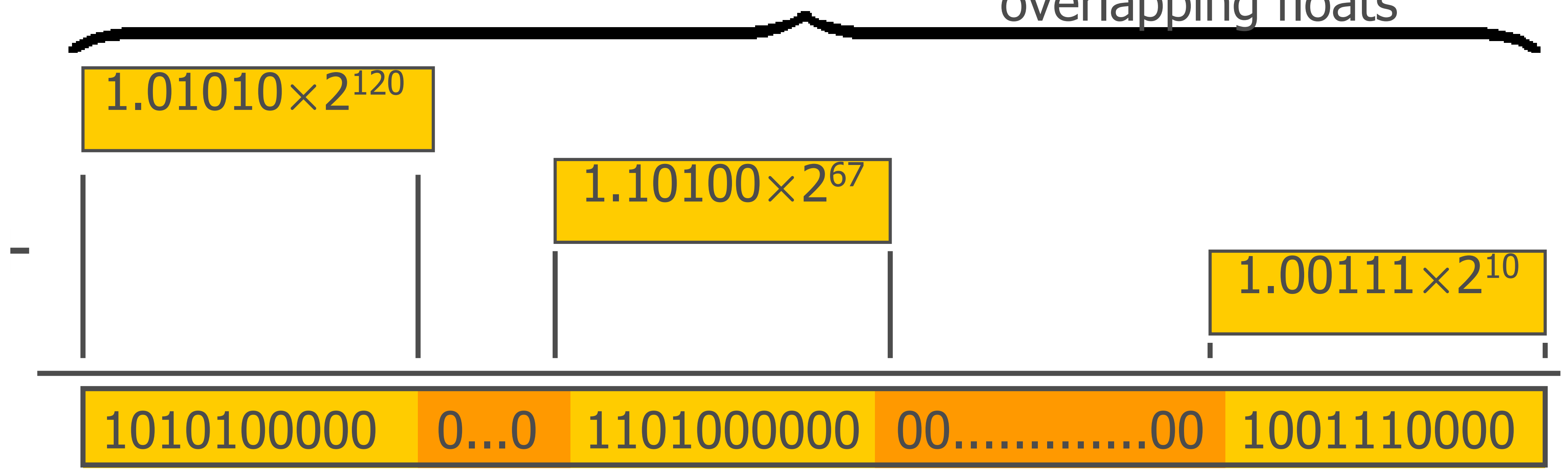
Variants - Ex : compute the bound at running time

```
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
}
```

How to Increase Precision?

- Knuth's theorem: Pair (x, Δ) → twice the precision.
- (D. Priest) Sorted list of fp-numbers → arbitrary precision.

Expansion = list of magnitude decreasing non-overlapping floats



Phases and re-use

- (J. Shewchuk) Arbitrary precision arithmetic $\Rightarrow$ phases can reuse the results of their predecessors.

Example:

$$E = (a_1 - b_1)^2 - (a_2 - b_2)^2$$

Let $a_1 - b_1 = x_1 + \Delta_1$ and $a_2 - b_2 = x_2 + \Delta_2$. Expand E as

$$E = \underbrace{(x_1^2 - x_2^2)}_{O(1)} + \underbrace{(2x_1\Delta_1 - 2x_2\Delta_2)}_{O(\epsilon)} + \underbrace{(\Delta_1^2 - \Delta_2^2)}_{O(\epsilon^2)}$$

Phases and re-use (cont'd)

- Strategy for finding the **sign** of

$$\boldsymbol{E} = \underbrace{(x_1^2 - x_2^2)}_{O(1)} + \underbrace{(2x_1\Delta_1 - 2x_2\Delta_2)}_{O(\epsilon)} + \underbrace{(\Delta_1^2 - \Delta_2^2)}_{O(\epsilon^2)}$$

- Evaluate E in phases, increasing precision on demand.
- Example:

$$\left.\begin{aligned} A &= (x_1 \otimes x_1) \ominus (x_2 \otimes x_2) \end{aligned}\right\} \textbf{floating point phase}$$

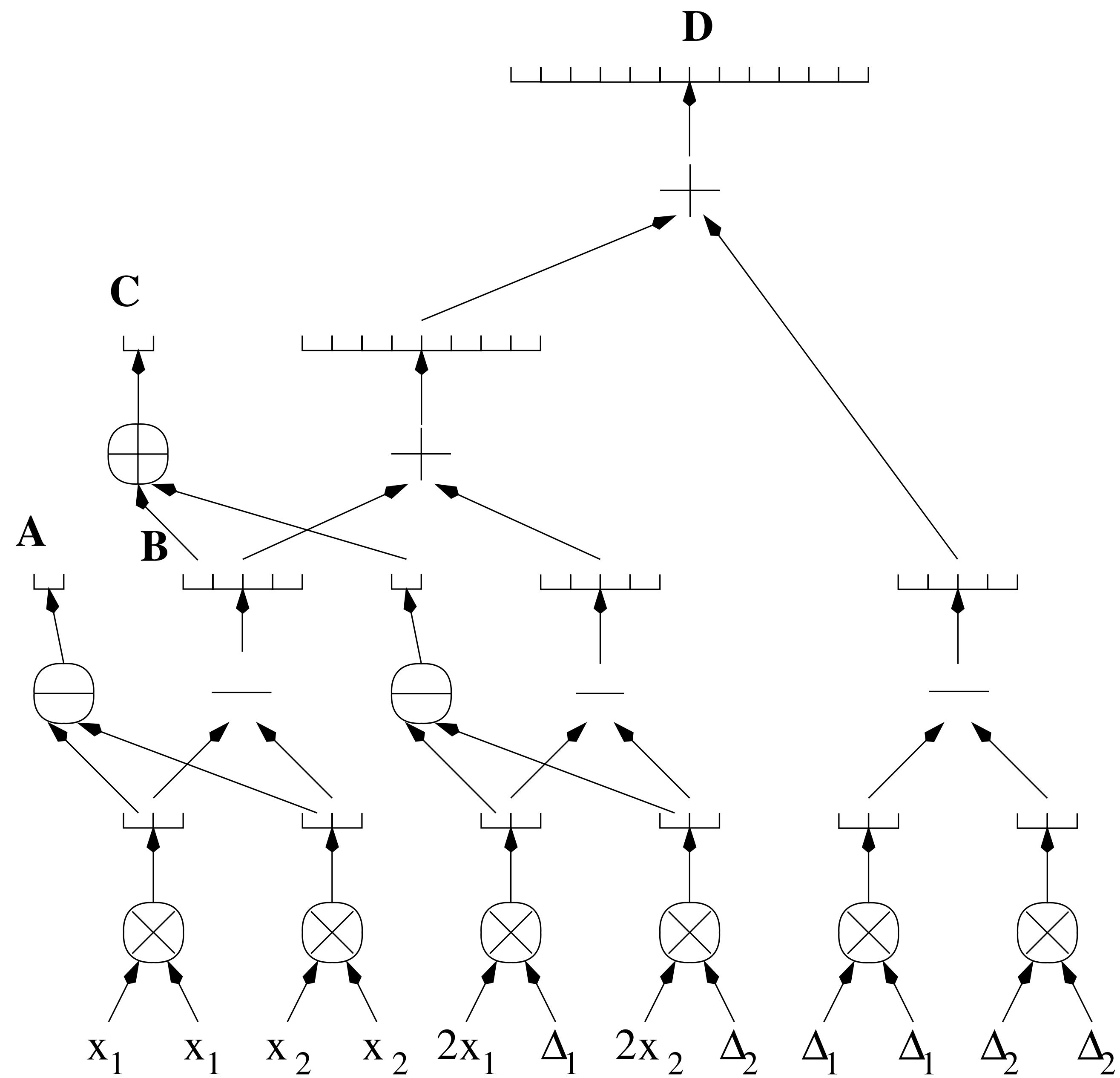
$$\begin{aligned} B &= x_1^2 - x_2^2 \\ C &= \text{round}(B) \oplus ((2x_1 \otimes \Delta_1) \ominus (2x_2 \otimes \Delta_2)) \end{aligned}$$

$$\left.\begin{aligned} D &= B + (2x_1\Delta_1 - 2x_2\Delta_2) + (\Delta_1^2 - \Delta_2^2) \end{aligned}\right\} \textbf{exact phase}$$

Reusing results

$$
\begin{aligned}
E &= (a_1 - b_1)^2 - (a_2 - b_2)^2 \\
&= (x_1^2 - x_2^2) \\
&+ (2x_1\Delta_1 - 2x_2\Delta_2) \\
&+ (\Delta_1^2 - \Delta_2^2)
\end{aligned}
$$

$$
\begin{aligned}
A &= (x_1 \otimes x_1) \ominus (x_2 \otimes x_2) \\
B &= x_1^2 - x_2^2 \\
C &= \text{round}(B) \oplus \\
&\quad ((2x_1 \otimes \Delta_1) \ominus (2x_2 \otimes \Delta_2)) \\
D &= B + \\
&\quad (2x_1\Delta_1 - 2x_2\Delta_2) + \\
&\quad (\Delta_1^2 - \Delta_2^2)
\end{aligned}
$$



- Shewchuk implemented the filtered exact predicates for the `orient3d` and `insphere` tests by an adaptive version of [**Preist 1991**]. They are freely available at `http://www.cs.cmu.edu/~quake/robust.html`, and are used by `TetGen`.

Version history

- 2018-12, ZJU
- 2019-07, UCAS, Beihang University