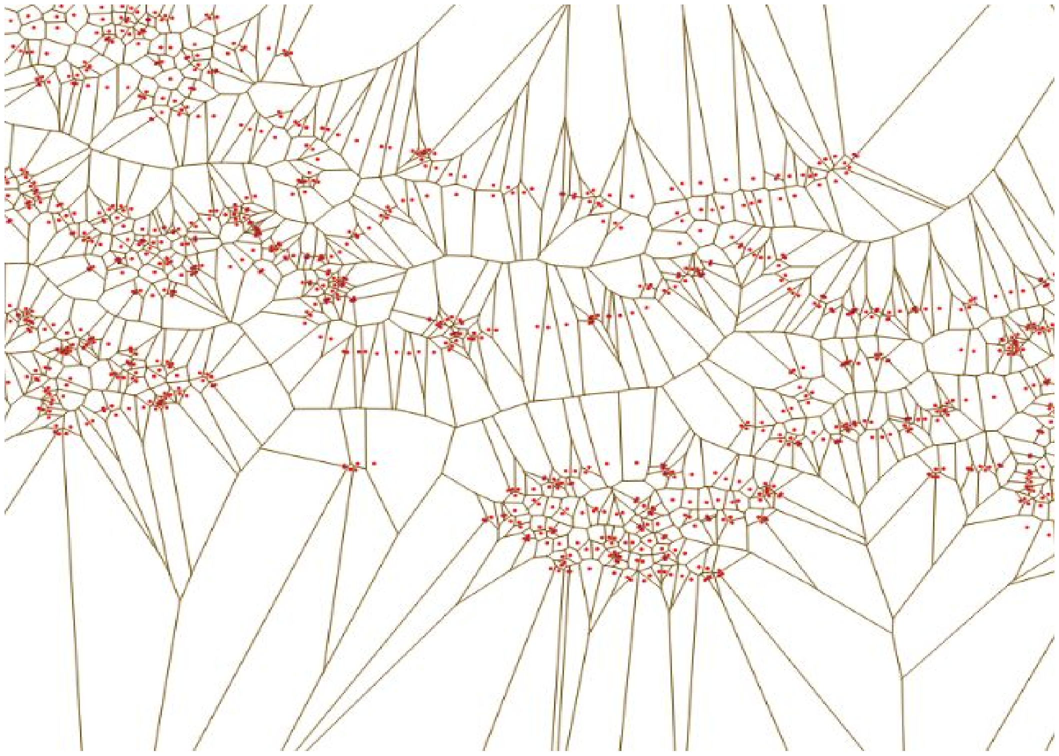
EXACT COMPUTING FOR GEOMETRIC PROBLEMS USING CGAL



B. Tech Project Report

G.Prithvi Raj Reddy
14CS10016,

Under the guidance of

Prof. Sudebkumar Prasant Pal

INTRODUCTION

In the world of computational geometry exact computation is assumed in most algorithms. In practice, if implementors perform computation in some fixed-precision model(usually the machine floating-point arithmetic), such implementations have many well-known problems, informally called “robustness issues”.

In practical terms, an algorithm is termed non-robust if it can lead to unpredictable failures during execution. It is clear that such failures occur with a sufficiently high probability to cause widespread concem. The unexamined premise in many of these solutions is the commitment to fixed-precision computation.

To reconcile theory and practice, redesigning theoretical algorithms to become robust under fixed-precision arithmetic is not viable. Hence to make robustness a non-issue *exact* computation is required.

FLOATING POINT COMPUTATIONS AND ITS LIMITATIONS

When designing geometric algorithms in the real world, the availability of exact arithmetic on real numbers is usually assumed. No computer directly provides exact arithmetic on real numbers(default operations). Using floating-point arithmetic would result in programs that may not behave as expected as it is hardware-supported approximate arithmetic.

In crude terms, if the two numbers differ by less than some small tolerance, treat them as equal. This often works, but you can never be sure about it.

Most modern processors support floating-point numbers of the form:

$$+\text{-}\textit{significand} \times 2^{\textit{exponent}}$$

Significand: (b.bbb) (Each b is a bit)
Exponent: x
Number: b.bbb * (2^x)

The approximation:
Floating-point values are generally normalized, which means that if a value is not zero, then its most significant bit is set to one, and the exponent adjusted accordingly.
For example,
floating-point representations have a base β (generally 2) and a precision *p*. If β = 2 and *p* = 24, then the decimal number 0.1 cannot be represented exactly, but is approximately $1.10011001100110011001101 \times 2^{-4}$.

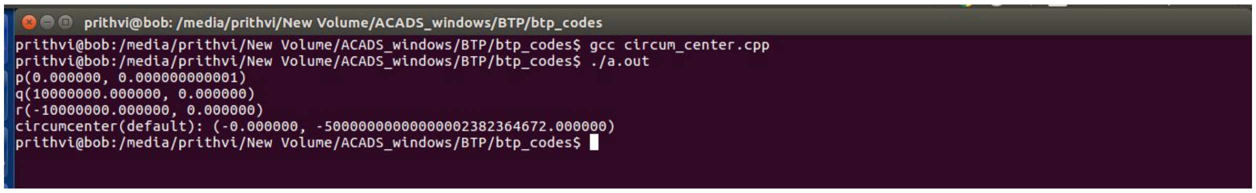
These roundoff errors due to limitations on precision, that originate at a certain point in a computation propagates to subsequent steps. They are amplified in ill-conditioned problems up to the point of making these problems intractable. There are several ways a geometric algorithm may behave when floating-point arithmetic is used instead of exact arithmetic.

Some of the geometric computations have accuracy requirements that vary with their input. For instance, consider the problem of finding the center of a circle, given three points that lie on the circle. Normally, hardware precision arithmetic will suffice, but if the input points are nearly collinear, the problem is ill-conditioned and the approximate calculation may yield a wildly inaccurate result or a division by zero

ILLUSTRATION OF FLOATING POINT LIMITATIONS BY FINDING CIRCUMCENTER

Circumcenter Of Circle:
Three nearly collinear points are taken as input:
p: -1e+7, 0
q: 1e+7, 0
r: 0, 1e-12

Regular Solution:
```c
#include <stdio.h>
#include <math.h>

void equation_perpendicular_bisector(double x1, double y1, double x2, double y2, double *a,
double *b, double *c){
	*a = 2 * (x1 - x2);
	*b = 2 * (y1 - y2);
	*c = (x2 * x2 - x1 * x1) + (y2 * y2 - y1 * y1);
}

void sol_of_equations(double a1, double b1, double c1, double a2, double b2, double c2, double
*x, double *y){
	if(a1/b1 == a2/b2) {
		printf("lines are parallel\n");
		return;
	}
	*y = -(a1 * c2 + a2 * c1) / (a1 * b2  + a2 * b1);
	*x = -(b1 * c2 + b2 * c1) / (b1 * a2  + b2 * a1);
}

void center_of_circle(double x1, double y1, double x2, double y2, double x3, double y3, double
*x, double *y ){
	double a1, a2, b1, b2, c1, c2;
		/* line 1 - perpendicular bisector of p1, p2
		    line 2 - perpendicular bisector of p3, p2 */

	equation_perpendicular_bisector(x1, y1, x2, y2, &a1, &b1, &c1);
	equation_perpendicular_bisector(x2, y2, x3, y3, &a2, &b2, &c2);
	sol_of_equations(a1, b1, c1, a2, b2, c2, x, y);
}

int main(){
        double x1, y1, x2, y2, x3, y3, x, y;
        x1 = 0;
        y1 = pow(10,-12);
        x2 = pow(10,7);
        y2 = 0;
        x3 = -pow(10,7);
        y3 = 0;
        center_of_circle( x1,  y1,  x2,  y2,  x3,  y3,  &x, &y);
        printf("p(%lf, %.12lf)\nq(%lf, %lf)\nr(%lf, %lf)\n", x1, y1, x2, y2, x3, y3);
        printf("circumcenter(default): (%lf, %lf)\n", x, y);
}
```



```
prithvi@bob: /media/prithvi/New Volume/ACADS_windows/BTP/btp_codes
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ gcc circum_center.cpp
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ ./a.out
p(0.000000, 0.000000000001)
q(10000000.000000, 0.000000)
r(-10000000.000000, 0.000000)
circumcenter(default): (-0.000000, -50000000000000002382364672.000000)
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ 
```

The y-coordinate -5e25 has been approximated to -50000000000000002382364672 due to floating point approximations

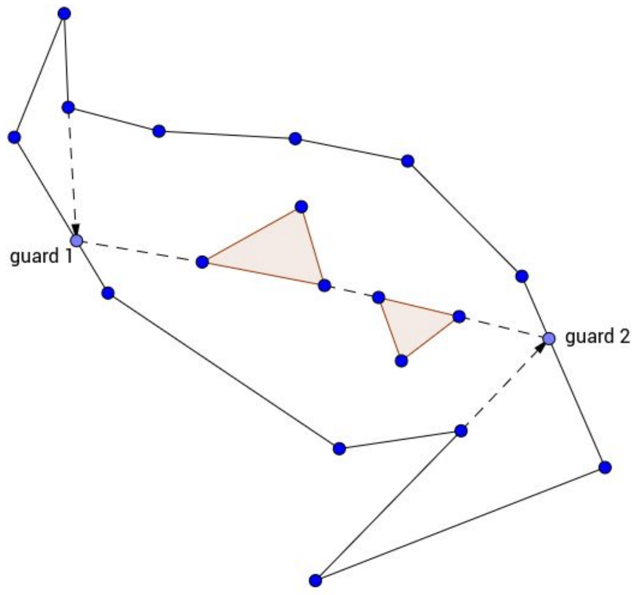
COMBINATORIAL PROBLEM, NOT A NUMERICAL ONE

The default numerical computations are less effective in geometric computing than they are in other fields.In geometry, rather than numbers, structures are computed: convex hulls, triangulations, etc.

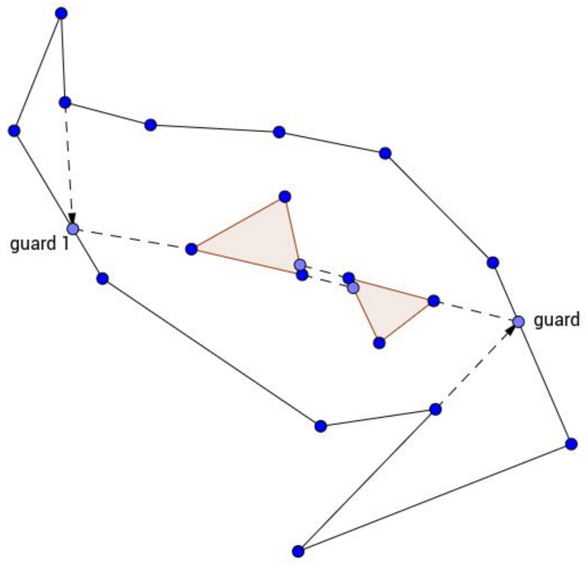
In building these structures, the underlying algorithms ask questions like "is a point to the left, to the right, or on the line through two other points?" Such questions have no answers that are "slightly off". Either you get it right, or you don't. And if you don't, the algorithm might go completely astray. Even the most fancy round off control techniques don't help here: it's primarily a *combinatorial* problem, not a numerical one.

THE ART-GALLERY PROBLEM (A COMBINATORIAL ERROR)

The number of guards required to cover the following polygon will be computed to be 2



In the following problem the holes have been slightly moved resulting in a small area(not upto scale) formed in between the two holes.
This would be neglected in the floating point computations and hence the solution would be same as the above problem. So instead of requiring 3 guards, the regular computation method for the following problem would result in the answer 2.



The CGAL Approach

In CGAL, the high-level algorithms is written in terms of a well-chosen set of basic questions (where is a point with respect to a line?) and basic objects (like a circle through three points). Doing this in the right way may not be easy, but once done, all the numerical issues will be outsourced, and the part of CGAL concerned with the basics returns correct results. Given this, the algorithms on top of it just work. Not in most cases, but always!

What essentially happens is that the numerical precision of the computations is increased, if necessary, by using numbers that in principle allow arbitrary precision.

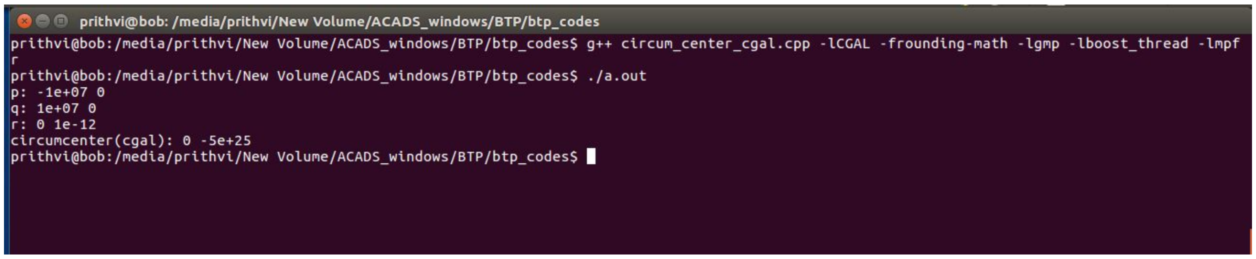
The numerical robustness handling does not happen in the algorithm itself, but in its basic questions and objects as said above.CGAL offers many different ways of getting these basics right, and which one is the best depends on the application. That's why you have to choose.

Circumcenter Of Circle using CGAL

Three nearly collinear points are taken as input
p: -1e+7, 0
q: 1e+7, 0
r: 0, 1e-12

CGAL Solution:
```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <CGAL/Kernel/global_functions.h>

// g++  -lCGAL -frounding-math -lgmp -lboost_thread -lmpfr

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2 Point_2;

int main(){
	 Point_2 p(-pow(10,7),0), q(pow(10,7),0), r(0,pow(10,-12)), x;

	x = CGAL::circumcenter(p,q,r);

	std::cout<<"p: "<<p;
	printf("\n");
	std::cout<<"q: "<<q;
	printf("\n");
	std::cout<<"r: "<<r;
```

```
        printf("\n");
        std::cout<<"circumcenter(cgal): "<<x;
        printf("\n");

        return 0;
}
```



```
prithvi@bob: /media/prithvi/New Volume/ACADS_windows/BTP/btp_codes
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ g++ circum_center_cgal.cpp -lCGAL -frounding-math -lgmp -lboost_thread -lmpf
r
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ ./a.out
p: -1e+07 0
q: 1e+07 0
r: 0 1e-12
circumcenter(cgal): 0 -5e+25
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$
```

The CGAL approach has given the exact y coordinate as -5e+25

CGAL produces correct results. If three lines meet in one point, they will do so in CGAL as well, and if a fourth line misses this point by 1.0e-380, then it also misses it in CGAL. it ultimately relies on computing with numbers of arbitrary precision.

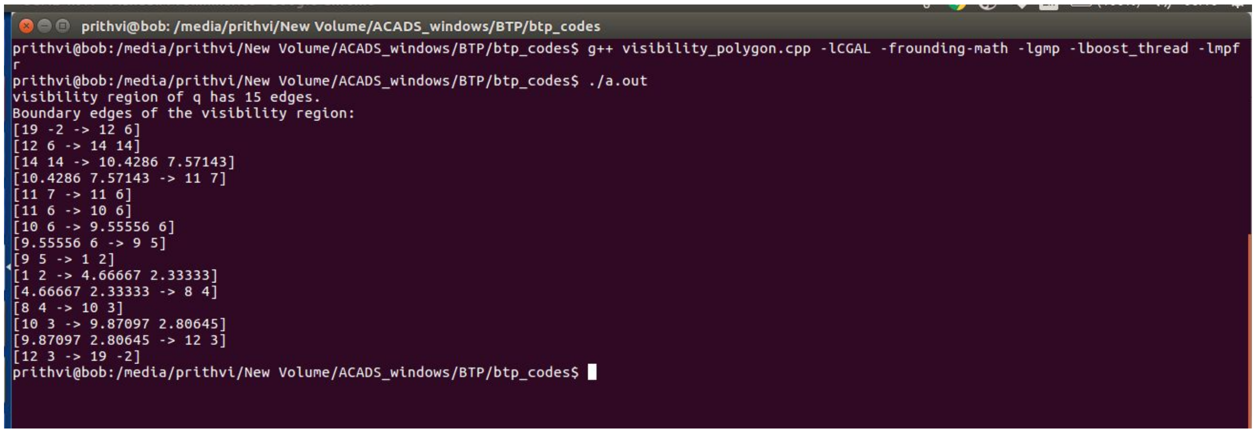
Such guaranteed correctness requires that CGAL is properly used, but it comes at a price: compared to algorithms that use fixed-precision numbers only, the performance is lower. This is possible because CGAL tracks *error bounds* and resorts to extended precision only when this is really needed.

VISIBILITY POLYGON USING CGAL

Implementing the CGAL method for computing the visibility region of a polygon with holes from a given point:

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Triangular_expansion_visibility_2.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <iostream>
#include <vector>

// Define the used kernel and arrangement
typedef CGAL::Exact_predicates_exact_constructions_kernel       Kernel;
typedef Kernel::Point_2                                         Point_2;
typedef Kernel::Segment_2                                       Segment_2;
typedef CGAL::Arr_segment_traits_2<Kernel>                      Traits_2;
typedef CGAL::Arrangement_2<Traits_2>                           Arrangement_2;
typedef Arrangement_2::Halfedge_const_handle                    Halfedge_const_handle;
typedef Arrangement_2::Face_handle                              Face_handle;

// Define the used visibility class
typedef CGAL::Triangular_expansion_visibility_2<Arrangement_2>  TEV;

using namespace std;

int main() {
  // Defining the input geometry
  Point_2 p1(1,2), p2(12, 3), p3(19,-2), p4(12,6), p5(14,14), p6( 9,5);
  Point_2 h1(8,3), h2(10, 3), h3( 8, 4), h4(10,6), h5(11, 6), h6(11,7);
  vector<Segment_2> segments;

  segments.push_back(Segment_2(p1,p2));
  segments.push_back(Segment_2(p2,p3));
  segments.push_back(Segment_2(p3,p4));
  segments.push_back(Segment_2(p4,p5));
  segments.push_back(Segment_2(p5,p6));
  segments.push_back(Segment_2(p6,p1));

  segments.push_back(Segment_2(h1,h2));
  segments.push_back(Segment_2(h2,h3));
  segments.push_back(Segment_2(h3,h1));
  segments.push_back(Segment_2(h4,h5));
  segments.push_back(Segment_2(h5,h6));
  segments.push_back(Segment_2(h6,h4));

  // insert geometry into the arrangement
  Arrangement_2 env;
  CGAL::insert_non_intersecting_curves(env,segments.begin(),segments.end());

  //Find the halfedge whose target is the query point.
  //(usually you may know that already by other means)
  Point_2 query_point = p4;
  Halfedge_const_handle he = env.halfedges_begin();
  while (he->source()->point() != p3 || he->target()->point() != p4)
    he++;

  //visibility query
  Arrangement_2 output_arr;
  TEV tev(env);
  Face_handle fh = tev.compute_visibility(query_point, he, output_arr);

  //print out the visibility region.
  cout << "visibility region of q has "
            << output_arr.number_of_edges()
            << " edges." << endl;

  cout << "Boundary edges of the visibility region:" << endl;
  Arrangement_2::Ccb_halfedge_circulator curr = fh->outer_ccb();
  cout << "[" << curr->source()->point() << " -> " << curr->target()->point() << "]" << endl;
  while (++curr != fh->outer_ccb())
    cout << "[" << curr->source()->point() << " -> " << curr->target()->point() << "]"<< endl;
  return 0;
}
```

Source: http://www.cgal.org



```
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ g++ visibility_polygon.cpp -lCGAL -frounding-math -lgmp -lboost_thread -lmpfr
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$ ./a.out
visibility region of q has 15 edges.
Boundary edges of the visibility region:
[19 -2 -> 12 6]
[12 6 -> 14 14]
[14 14 -> 10.4286 7.57143]
[10.4286 7.57143 -> 11 7]
[11 7 -> 11 6]
[11 6 -> 10 6]
[10 6 -> 9.55556 6]
[9.55556 6 -> 9 5]
[9 5 -> 1 2]
[1 2 -> 4.66667 2.33333]
[4.66667 2.33333 -> 8 4]
[8 4 -> 10 3]
[10 3 -> 9.87097 2.80645]
[9.87097 2.80645 -> 12 3]
[12 3 -> 19 -2]
prithvi@bob:/media/prithvi/New Volume/ACADS_windows/BTP/btp_codes$
```

References and Bibliographies:

- http://www.cgal.org/
- https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- Classroom examples of robustness problems in geometric computations-Lutz Kettner *et al*
- Towards exact geometric computation" -Chee-Keng Yap