Robust Geometric Computation (RGC), State of the Art

Francisco J. Santisteve*

fjsantis@lsi.upc.es

This work represents an historical introduction of the *Robust Geometric Computation* problem (*RGC*) and a state of the art report of the different methods to avoid the inaccuracies that usually may lead to inconsistent decisions, causing unexpected failures for some correct input data.

We define the *RGC* problem as the trouble of developing programs that do not crash and that provide the expected outcomes. It does not necessary implies an exact computation of the numerical values. This target could be reached in several scopes with different methods, therefore we also propose a taxonomy.

*Software Department, Universitat Politècnica de Catalunya (U.P.C.).

Contents

1 Introduction

This work represents an historically introduction of the *Robust Geometric Computation* problem (*RGC*) and a state of the art report of the different methods to avoid the inaccuracies that usually may lead to inconsistent decisions, causing unexpected failures for some correct input data.

We define the RGC problem such as the trouble of developing programs that do not crash and that provide the expected outcomes and, as we remark in chapter 6, it not necessary implies an exact computation. This target could be reached in several scopes with different methods that we have classified following the taxonomy:

- Heuristic computation approaches, also subdivided in:
  - Careful programming.
  - ε's and bounds.
- Fixed precision computation approaches.
- Exact computation approaches.

That is just a proposal but clearly it is not the only one. Other works in this field have done similar taxonomies, yet with some differences. The root of these differences is usually in the semantic of some words like: exact, approximate or rounded.

Clearly, no method could always have an exact value of all computations due to the discrete machine's nature. Therefore, exact are the methods that calculate good enough approximations, that at last, necessary have been rounded. Unlike that, some approaches do not try to get always an exact value. They discretize the world and just try to represent this rounded versions of the real world in the machine. Despite this discretization there are calculations that have to been rounded. We have called the methods that work with a discrete world: *Fixed precision* computation approaches. Opposite them we have called *Exact* computation approaches the methods that try to get a continuous computer. Lastly, the programming tricks and the methods that work with bounds (but don't justify these bounds) have been encapsulated in *Heuristic* computation approaches.

2 Robustness geometric computations

We live in a world that we see as a continuous one and that means a high level of precision but we only have discrete computers. That is a problem that makes it hard to program robust algorithms.

2.1 Brief history

Looking at our early history it is so easy to find computational disasters caused by computer arithmetic errors. These two have been excerpted from [12] :

- On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqui Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. A published Internet report of the General Accounting Office (GAO) [16] reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors.
- On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $ 7 billion. The destroyed rocket and its cargo were valued at $ 500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report [33]. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was large than 32,768 (the largest integer store-able in a 16 bit signed integer) and thus the conversion failed.

These two that we have taken out are more famous than others but, a lot of software over any kind of computers, crashes every day due to arithmetic errors. That happens because a discrete computer can't achieve a total representation of real numbers.

How can we avoid inaccuracies? Mathematics and computer scientists have been working for several years looking for a way to develop robust algorithms. In the beginning (60's and 70's) each line of computers supported its own range and precision for its floating point numbers, and rounded off arithmetic operations in its own way. These differences made porting software between machines really hard. As W. Kahan explained in IEEE Computer [53] [28] interview:

> *Some of one fast computer's numbers behaved as non-zeros during comparison and addition but as zeros during multiplication*

and division; before a variable could be used safely as a divisor it had to be multiplied by 1.0 and then compared with zero. But another fast computer would trap on overflow if certain of its numbers were multiplied by 1.0 although they were not yet so big that they could not grow bigger by addition. (This computer also had nonzero numbers so tiny that dividing them by themselves would overflow.) On another computer, multiplying a number by 1.0 could loop off its last four bits. Most computers could get zero from $X - Y$ although X and Y were different; a few computers could get zero even though X and Y were huge and different.

Arithmetic aberrations like these were not derided as bugs; they were "features" of computers too important commercially for programmers to ignore. Programmers coped by inventing bizarre tricks like inserting an assignment $X := (X + X) - X$ into critical spots in a program that would otherwise have delivered grossly inaccurate results on a few aberrant computers. And then there were aberrant compilers ...

2.2 The binary floating-point arithmetic standard: IEEE 754

In the middle 80s, IEEE standard 754 for binary floating-point arithmetic [3] became official. This standard specifies how the implementation of floating-point numbers, formats (Single, Double and Double-Extended), rounding strategies, representations for NaNs (Not-a-Number), $\pm\infty$, etc. must be. The history of how this standard was adopted is given by W. Kahan in [53] [28]. There is another unofficial status report [27] subject to changes that explains the features and the current status of the IEEE 754. Another great classical survey on floating-point arithmetic is the work of D. Goldberg [17], there's also an addendum to this work in Sun's Numerical Guide [18]. One of the open problems that the work of Goldberg left is the *Table Maker's Dilemma (TMD).*

TMD is the problem of always getting correctly rounded results when computing elementary functions (sine, cosine, exponential, natural logarithm, etc.). The IEEE-754 standard for floating point arithmetic requires that the results of the arithmetic operations (addition, subtraction, multiplication, etc.) should always be correctly rounded. That is, once a rounding mode is chosen among the four possible ones (round towards $-\infty$, $+\infty$, 0 or the nearest number), the system must behave as if the result were first computed exactly, with infinite precision, then rounded. In other words, it is not practical to specify that the precision of elementary functions be the same as if the function were computed to infinite precision then rounded.

There are different techniques used for computing these functions in different machines (Rational approximation, CORDIC [1], large tables), yet no one works acceptably over the wide range of current hardware. In 1998 Lefèvre et al. [57] have presented a new solution that solves this problem for the double-precision exponential function in small domain. His paper also surveys in the last advances of this field.

Requiring correctly rounded results not only improves the accuracy of computations, it is also the best way to make numerical software portable. Moreover, as noticed by Agarval et al. [14], correct rounding facilitates the preservation of useful mathematical properties such as monotonicity, some symmetries and some identities. These are specially important in geometric computations.

Unfortunately, the state of the art, as Kahan explains [29], is that the faster software libraries of elementary transcendental functions (like exp, log, cos, sin, tan, arctan, etc.) are substantially more accurate on some computers than on others. For example, this functions on Motorola 68040, Cyrix'87 or Intel Pentium and P6 are generally 3 decimals more accurate than Sun SPARCs and the public-domain library distributed with 4.3 BSD UNIX or IBM RS/6000 and Power PCs. Finally, the library that comes with H.P. workstations is substantially less accurate.

2.3 Motivations: Robustness over IEEE 754

But IEEE-754 just provides us with a standard representation of $\mathbb{R}$(and in fact just a subset of $\mathbb{Q}$) in different architectures. That only means that the user/programmer knows when the computer is going to crash, and he knows also that this is going to happen in the same source point, no matter what platform it is. That is very useful but it is not enough. We can show in figure 1 what happens if we try to generate a sequence of real numbers starting with a SEED=1 and multiplying by a GROWING factor 10 by a simple program over a Sparc SUN Ultra-1.

In fact, when we do a calculation over a computer, we could expect three different kinds of errors:

- Round errors due to work with subset of $\mathbb{Q}$. That makes that some numbers will never be exactly represented. This set could be detailed in three subset:

[1] CO-odinate R-otation DI-gital C-omputer is the method used on the Intel 8087 and the Motorola 68881 that uses mostly shifts and adds and very few multiplications and divisions for computing elementary functions.

  - Conversion errors. Input numbers are usually decimal and they may be converted to binary numbers, we cannot always represent exactly the outcomes of this conversion. For example, the decimal 0.6 is, in binary, the periodic 0.100110011001....
  - Roundoff errors, due to the finited mantissa length. It happens when the outcomes of a calculation requires higher precision and then must be rounded.
  - Digit-cancellation errors, also due to the finited mantissa length, when the difference of two nearly equal numbers has fewer significant digits and seem to be the same number.

- Discretization errors, because the number of iterations that we can do is bounded. Therefore we have to turn continuous in discrete problems.

- Convergence errors. Some of the methods commonly used in numerical analysis are iteratives and they tend to the exact solution. They may be stopped after some iterations and the calculated number has to be understood as a approximated solution.

2.4 Overview of methods

The great goal would be to avoid all the imprecision (and the crashes that they produce) in computers in order to get robust computations. Doing a great simplification there are three sets of approaches:

- Heuristic approaches, a careful programming can avoid a lot of unlikely inaccuracies. In other words, working with ε's and bounds is one of the most extended solution. Lastly, the most usual option is a graceful combination of this approach with one of the others.

- Getting a continuous computer, or in other words, trying to get the most continuous computer, developing big floats libraries that can work with higher precision.

- Getting a discrete world, or in other words, work in the computer only with discrete representations of it. These approaches must be careful with the growing or changing of the numbers that can go farther than was expected.

```
The generated sequence is:

 se[1]=10.000000
 se[2]=100.000000
 se[3]=1000.000000
 se[4]=10000.000000
 se[5]=100000.000000
 se[6]=1000000.000000
 se[7]=10000000.000000
 se[8]=100000000.000000
 se[9]=1000000000.000000
 se[10]=10000000000.000000
 se[11]=99999997952.000000
 se[12]=999999995904.000000
 se[13]=9999999827968.000000
 se[14]=100000000376832.000000
 se[15]=999999986991104.000000
 se[16]=10000000272564224.000000
 se[17]=99999998430674944.000000
 se[18]=999999984306749440.000000
 se[19]=9999999980506447872.000000
 se[20]=100000002004087734272.000000
 se[21]=1000000020040877342720.000000
 se[22]=9999999778196308361216.000000
 se[23]=99999997781963083612160.000000
 se[24]=1000000013848427855085568.000000
 se[25]=10000000714945030854279168.000000
 se[26]=100000002537764290115403776.000000
 se[27]=1000000062271131048573140992.000000
 se[28]=10000000622711310485731409920.000000
 se[29]=100000001504746621987668885504.000000
 se[30]=1000000015047466219876688855040.000000
 se[31]=9999999848243207295109594873856.000000
 se[32]=100000003318135351409612647563264.000000
 se[33]=1000000071866979741764260066230272.000000
 se[34]=10000000409184787596297531937521664.000000
 se[35]=100000004091847875962975319375216640.000000
 se[36]=1000000040918478759629753193752166400.000000
 se[37]=10000000567641112624826207124609564672.000000
 se[38]=100000006944061726476491472742798852096.000000
 se[39]=Inf
```

```c
#include <stdio.h>

#define LONG_SEQ 39
#define SEED 1
#define GROWING 10

float se[LONG_SEQ];

void create_se(void)
{
  int k;

  se[0]=SEED;
  for(k=1; k<=LONG_SEQ; k++)
  {
    se[k]=se[k-1]*GROWING;
    printf("se[%d]=%f \n", k, se[k]);
  }
}

main()
{
  create_se();
}
```

Figure 1: Corrupted generation sequence.

In some kinds of applications, applying solutions from one of these sets can be easy. That does not mean that the problems are easier in some fields but, for instance, additional information gives us a better representation of a discrete world. That happens in computational geometry where a lot of three set techniques have been developed. All the techniques to achieve accuracy in numerical computations can be used in geometric computations yet in geometric computations we also have some combinatorial structures and some consistency conditions between the numerical and the combinatorial data that give us some additional knowledge. C. K. Yap elucidates in one of his works [62] the meaning of *geometric computation*.

Another kind of robustness problems that sometimes can crash the programs are the contradictions between several pieces of software written by different programmers using different conventions, notations and formulas. Some small error can be accepted in one method but can be critical if it is mixed with another algorithm.

2.5 Decision Making (DM)

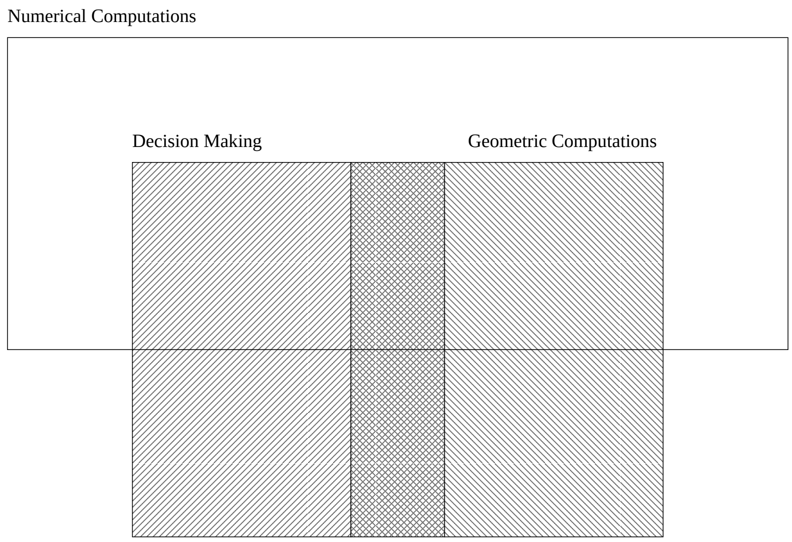


Figure 2: *Decision Making* and *Geometric Computations* not always connote Numerical calculations.

One special case, that can be found, is the role of *Decision Making*, classified in figure 2. That is when the outcomes of a calculation are used to take a decision. If the computed value is inaccurate the decision that we make could be wrong, specially near the decision border. Unlike that, there are the calculations of values where a wrong one just involves incorrect positioning of a point, line, etc. This could be critical but probably in a local scope. A numerical calculation that involves making a decision must be carefully controlled.

3 Heuristic computation approaches.

As we've said before there are two basically heuristic approaches:

- A careful programming can avoid a lot of inaccuracies.
- To work with ε's and bounds.

These two approaches, especially the first one, can be easily used along with others.

It's important to see that not all in this section is heuristics, some tools like *condition numbers* are a rigorous ones yet they are used to detect the possibility of an imprecision, not to solve it. So, these tools can also be used in the next sections.

3.1 Careful programming.

3.1.1 Tricks.

When a programmer has an error is very usual that he tries to develop tricks to avoid it. This is very common with numerical errors when the order between the iterations can be critical. Sometimes a good arrangement of this iterations is all that we need to prevent an overflow.

When we do some computation it is very important to know what are we going to do with the outcomes. An isolated computation could produce a small error that can be critical but it will be easy to set apart. If the computation is chained with others the errors will be also chained and their importance will also increase. We must pay special attention to control the growing chained errors that can be produced.

In addition, is very important not to mix great and small numbers in floating point arithmetic representation. For instance, if we have a finite growing sequence of numbers S with a big magnitude difference between the numbers (figure 3, 4), we will probably get different outcomes if we add these numbers from the first to the last or from the last to the first.

```
#include <stdio.h>

#define LONG_SEQ 100
#define SEED 10e-20
#define GROWING 2.5

float se[LONG_SEQ];

void create_se(void)
{
  int k;

  se[0]=SEED;
  printf(" se[0]=%44.22f \n", se[0]);
  for(k=1; k<LONG_SEQ; k++)
  {
    se[k]=se[k-1]*GROWING;
    printf(" se[%d]=%44.22f \n", k, se[k]);
  }
}
```

```
main()
{
  float res;
  int i,j;

  create_se();

  res=se[0];
  for(i=1; i<LONG_SEQ; i++)
  {
    res=res+se[i];
  }
  printf("\n First way: %f \n", res);

  res=se[LONG_SEQ-1];
  for(j=LONG_SEQ-2; j>=0; j--)
  {
    res=res+se[j];
  }
  printf("\n Second way: %f \n", res);
}
```

Figure 3: The source code that implements a finite growing sequence of numbers.

```
The generated sequence is:

 se[0]=  0.0000000000000000001000
 se[1]=  0.0000000000000000002500
 se[2]=  0.0000000000000000006250
 se[3]=  0.0000000000000000015625
 se[4]=  0.0000000000000000039062
 se[5]=  0.0000000000000000097656
 se[6]=  0.0000000000000000244141
 se[7]=  0.0000000000000000610351
 se[8]=  0.0000000000000001525879
 se[9]=  0.0000000000000003814697
 se[10]= 0.0000000000000009536743
 se[11]= 0.0000000000000023841858
 se[12]= 0.0000000000000059604645
 se[13]= 0.0000000000000149011612
 se[14]= 0.0000000000000372529020
 se[15]= 0.0000000000000931322551
 se[16]= 0.0000000000002328306276
 se[17]= 0.0000000000005820765961
 se[18]= 0.0000000000014551914361
 se[19]= 0.0000000000036379785903
 se[20]= 0.0000000000090949461504
 se[21]= 0.0000000000227373658096
 se[22]= 0.0000000000568434153914
 se[23]= 0.0000000001421085332742
 se[24]= 0.0000000003552713401245
 se[25]= 0.0000000008881783641890
 se[26]= 0.0000000022204458272057
 se[27]= 0.0000000055511146790366
 se[28]= 0.0000000138777869196360
 se[29]= 0.0000000346944659668225
 se[30]= 0.0000000867361649170562
 se[31]= 0.0000002168404193980678
 se[32]= 0.0000005421010200734599
 se[33]= 0.0000013552526070270687
 se[34]= 0.0000033881315175676718
 se[35]= 0.0000084703287939191796
 se[36]= 0.0000211758215300505981
 se[37]= 0.0000529395547346211970
 se[38]= 0.0001323488831985741854
 se[39]= 0.0003308721934445202351
 se[40]= 0.0008271804545074701309
 se[41]= 0.0020679510198533535004
 se[42]= 0.0051698777824640274048
 se[43]= 0.0129246944561600685120
 se[44]= 0.0323117375373840332031
 se[45]= 0.0807793438434600830078
 se[46]= 0.2019483596086502075195
 se[47]= 0.5048708915710449218750
 se[48]= 1.2621772289276123046875
 se[49]= 3.1554431915283203125000

se[50]=                    7.8886079788208007812500
se[51]=                   19.7215194702148437500000
se[52]=                   49.3037986755371093750000
se[53]=                  123.2594985961914062500000
se[54]=                  308.1487426757812500000000
se[55]=                  770.3718261718750000000000
se[56]=                 1925.9295654296875000000000
se[57]=                 4814.8237304687500000000000
se[58]=                12037.0595703125000000000000
se[59]=                30092.6484375000000000000000
se[60]=                75231.6250000000000000000000
se[61]=               188079.0625000000000000000000
se[62]=               470197.6562500000000000000000
se[63]=              1175494.1250000000000000000000
se[64]=              2938735.2500000000000000000000
se[65]=              7346838.0000000000000000000000
se[66]=             18367096.0000000000000000000000
se[67]=             45917740.0000000000000000000000
se[68]=            114794352.0000000000000000000000
se[69]=            286985888.0000000000000000000000
se[70]=            717464704.0000000000000000000000
se[71]=           1793661696.0000000000000000000000
se[72]=           4484154368.0000000000000000000000
se[73]=          11210385408.0000000000000000000000
se[74]=          28025962496.0000000000000000000000
se[75]=          70064906240.0000000000000000000000
se[76]=         175162261504.0000000000000000000000
se[77]=         437905653760.0000000000000000000000
se[78]=        1094764134400.0000000000000000000000
se[79]=        2736910237696.0000000000000000000000
se[80]=        6842275594240.0000000000000000000000
se[81]=       17105688723456.0000000000000000000000
se[82]=       42764221808640.0000000000000000000000
se[83]=      106910552424448.0000000000000000000000
se[84]=      267276385255424.0000000000000000000000
se[85]=      668190946361344.0000000000000000000000
se[86]=     1670477332348928.0000000000000000000000
se[87]=     4176193196654592.0000000000000000000000
se[88]=    10440483528507392.0000000000000000000000
se[89]=    26101207747526656.0000000000000000000000
se[90]=    65253017221332992.0000000000000000000000
se[91]=   163132547348299776.0000000000000000000000
se[92]=   407831368370749440.0000000000000000000000
se[93]=  1019578438106742784.0000000000000000000000
se[94]=  2548946129626595328.0000000000000000000000
se[95]=  6372365324066488320.0000000000000000000000
se[96]= 15930913035288313856.0000000000000000000000
se[97]= 39827284237488226304.0000000000000000000000
se[98]= 99568210593720565760.0000000000000000000000
se[99]=248920530882347925504.0000000000000000000000

First way: 414867551470579875840.000000

Second way: 414867586654951964672.000000
```

Figure 4: A finite growing sequence of numbers S with a big magnitude difference between the numbers.

We will get a more accurate answer adding from the first to the last because in that way the numbers involved in the addition have always similar magnitude. In fact, neither of the two answers are correct because the sequence (as we show in the figure 4) is not correctly computed due to floating point errors. If we develop the sequence in a symbolical way, we have:

$$S = \sum_{0 \leq i \leq 99} 10^{-20} \cdot 2.5^i = 10^{-20} \cdot \frac{2.5^{100}-1}{1.5}$$

And that, computed with a *big float library*, is: $0.4148676852 \cdot 10^{20}$

Some other tricks are:

- Use numerical input data rather than derived one that can be corrupted.
- Some inconsistencies are due to sort numerical operations between geometric structures that must be first checked. For instance, as Michelucci said [37]: *before computing the power of a vertex relatively to some line, verify first if the line is topologically incident to the vertex from the data structures at hand.*
- When a mathematical operation has more than one solution (an easy sample could be the two homomorphic images of $\sqrt{4} = \pm 2$) some extra-tools are needed to decide which one of them is the desired one.
- Etc.

In the same work [37] Michelucci also suggests that it is better to use non redundant structures, to limit the probability of contradictions and not to use several distinct formulas for the same value. We do completely agree with this. It is true that sometimes the non careful use of redundant structures could produce contradictions yet in some cases could avoid similar inconsistencies helping to decide which one is the true answer. That happens in the second proposed trick.

It also happens if we use different formulas to compute the same value. If we suppose that these expressions are identities and don't control the equality between the answers we can get inconsistencies. For instance, to compute the inverse of a matrix A, we could choose between different methods. Some of them are:

1. The most common way, working with the determinant.
2. Using the *LU descomposition* [19]. We know that $PA = LU$, where P is a permutations matrix that swap the k-row with the r-row of A when

is multiplied by P. Then we can solve the equations $LUa_k = Pe_k$, where a_k is the k-column of A^{-1} and e_k is the k-vector of the canonical base.

3. Modifying the Gauss method, in order to get zeros in both sides of the diagonal of the matrix and solving simultaneously all the equations. In that manner, we change the matrix equation $AX = I$, where I is the identity matrix and $X = A^{-1}$ by $DX = B$ and therefore $A^{-1} = D^{-1}B$ (Gauss-Jordan method, also in [19]).

3.1.2 Condition numbers.

But this could be one of the ways to detect that some computation is not consistent. The important thing is that we need to control the similarity between these different computations of the same values. In other way, these parallel-solution computation could be done just when we know that a algorithm is not stable and then they could be a help. More precisely, the stability or the sensitivity of the different floating points computations had been studied since long time ago into the numeric analysis field. Different books could survey about this, a wide view is developed in [4] and a very practical approach in [13]. The main idea is to know, before programming, if a geometric computation is *well conditioned.*

We have the intuition that a geometric operation is like a black-box with an input and an output and thereupon we can assume that the outcomes of the calculation are always good. They are not good for the actual inputs, but they are correctly for other inputs that represent a small perturbation of the actual ones. If we produce a small perturbation in the input, we expect to get a small perturbation in the output. Then we can say that this black-box (this geometric operation) has a good quality and a small error. If for some small perturbations of the input we get a big perturbation in the output we should say that this geometric operation is ill conditioned so it will be convenient to control it carefully.

Being formal, if the input has an error ε, the output will be a function $\delta(\varepsilon)$ of this error . For small values of ε, these should be a constant k that:

$$\delta(\varepsilon) \approx k \cdot \varepsilon$$

A small k means that $\delta(\varepsilon)$ is small for small values of ε. Then we can say that the operation is *well conditioned.* A large k means that $\delta(\varepsilon)$ is large for small values of ε. Then we can detect that the operation is *ill conditioned.* This k is called *condition number* and, as we said before, has been deeply studied into the numerical analysis field.

Sometimes we can detect that an operation is not *well conditioned* but it is not possible for us to develop a good algorithm because the input perturbations sensitivity depends in part on the nature of the problem. That happens, for instance, in the intersection of two lines. If the angle between the lines is very sharp, any small perturbation of the two lines will produce a big change in the coordinates of the intersection point. Hence, if the angle between the lines is nearly a right angle the computation will be more accurate. That is reflected in the condition number of the two lines intersecting problem:

$$k \leq \frac{4}{sin(\beta)}$$

where β is the angle between the lines. In the best case we know that we get a perturbation of order 2^{-t} where t is the mantissa length of the outcomes, because the last representable digit will be rounded. For small angles β, we could approximate $sin(\beta) \approx \beta$ so for an intersection angle of $\frac{1}{2^m}$, we expect to lose about $m + 2$ binary digits. These cases will have to be specially controlled adding topology information and so on.

How to calculate this condition number is explained in [23]. A similar example with a pentagon was proposed by Dobkin and Silver in [10], who advocate to extend the precision of the mantissas dynamically during the computation as required.

If we know that an operation is ill-conditioned we could try to develop another, well-conditioned, algorithm, put bounds around the outcomes or compute the answer in different manners. In some way, the last, is like a *second* opinion and is very useful into the role of *Decision Making* (section 2.5), also called *DM* problem. Hoffman, in [23], concludes that the independence between logical decisions is very important, in order to avoid the growing chained problem, and to establish consistency of the decisions by symbolic reasoning.

3.1.3 Mixed representations.

A usual way in graphic computations is to use symbolic information added to the numerical one that we have. This added information, that gives us a large range of careful programming possibilities, could be:

- The adjacency graph (that is, topology).
- An embedding of some nodes of the graph (that is, geometry).

- Relations between graphical entities like co-planarity of some points or parallelism of planes, etc. (that's also geometry because it quests about the position of the entities).

In his book [23] Hoffmann presents the notion of representation and model. Using these two concepts he also presents a way to compare the theoretical model with the practical representation of them. Then he introduces symbolical versus numerical representations and different levels of hybrid ones. In this work Hoffmann also explores the possibility of use purely symbolic representations, then the model exists if and only if we can assign real numbers to the symbolic coordinates that satisfies the constraint equations but he concludes that answering this existence question is not trivial. For instance in [24] Fudos and Hoffmann demonstrate that the particular case of finding a real solution of a system of geometric constraints solvable by a constructive method is NP-hard. That doesn't means that all this set of problems are NP-hard, we can add some information that help us to decide the problem and then this can turn in P-hard. Regrettably, deciding what information we need can be again NP-hard.

Anyway, if you get a mixed numerical and symbolic representation, you should decide what of two informations are suitable to change and in what way. There are two possibilities:

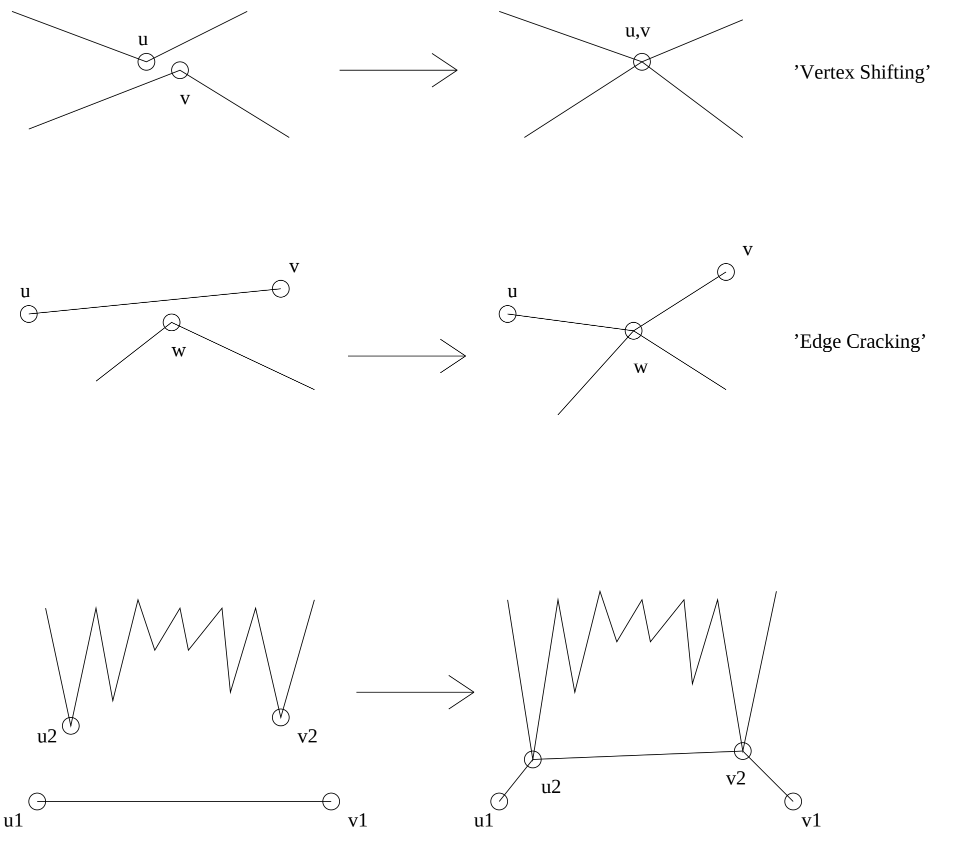
1. To preserve the symbolic representation changing, if needed, the numerical one. That's a bit dangerous option because after some iterations the explicit geometry could have so many changes. One way is to accept a boundary around the entities (see section 3.2) and remember the original values. Then one change could subtract another but no change will be recognized if it moves any original value far away the established boundary. Hoffmann [23] show us that preserve the symbolic representation could have two pains:

   (a) Every time that one entity is embedded it is possible that all the others must be re-fixed.

   (b) The shape of the visualized object could experiment positional perturbations, between different frames, shaking like a jelly.

2. To preserve the numerical representation and modify the symbolic data in an attempt to minimize positional perturbations or in order to protect the calculated geometry. For instance, we could have a cube definition detailing the position of six vertex (18 numerical values) and the topology (some asserts). In this case we could decide to preserve this 18 numerical values paying no attention to the co-planarity of some vertex. Changing symbolical data implies subdividing faces and

edges, merge vertexes and so on. One way, exposed by Milenkovic in [40], for polygonal regions in the plane is *normalizing* the input data (see section 4). He postulates that no entities are closer than a tolerance ε. This was called by Milenkovic *data-normalization* and it is also called *coordinates discretization*. It is very popular for different purposes. Rossignac, in his work [50], presents a useful application of this approach in the polygonal simplification field, calling this technique *vertex clustering*. Nowadays, Rossignac is still working on how to perform this clustering without loss of information. The way that Milenkovic [40] laid out needs two operations: *vertex shifting* and *edge cracking*. This operations, as the figure 5 shows, can produce positional perturbations. Milenkovic also proposes a second paradigm for altering the symbolic data, called *hidden-variable* method. It allow us to classify each point/line pair in the sets: above, on or below.



Positional perturbation due to 'edge cracking'

Figure 5: Entities merging.

3.2 ε's and bounds.

Work with ε's and bounds has always been the classical solution for a lot of engineer problems. That is a very usual trick due to allow us to get an answer with some expected precision (or in other words, with some accepted error). In a general scope, work with ε's and bounds has been classified as an heuristic computation approaches because the process of taking the decision of the what ε must be used has usually been heuristic and founded on experience. After that decision, the approaches are theoretical so-so. Some people prefer not to classify some of this approaches as heuristics ones. They call their *Approximate Computation Paradigm*.

3.2.1 Overview.

We can detect to different sets of approaches:

- Bound entities (*interval geometry*).
- Bound predicates of the entities (*interval predicates*).

Yap, in [62], call these sets of approaches *interval geometry* because it is analogue of *interval arithmetic* [48] [30]. We prefer to call the second one *interval predicates* because the bounds are on the predicates on the geometry.

Another approach is to put the ε in the arithmetic that operates with the entities, creating that Michelucci called an ε-arithmetic [38]. The Michelucci's work applies the Yap' perturbation technique [59] [60] in order to create *black box* schemes that automatically remove degenerations in geometric algorithms. His work proposes an ε-arithmetic to represent explicitly small numbers and redefine the basic operations so that the arithmetic itself will remove degeneracies.

3.2.2 Bounding entities.

The main idea is to use a threshold, traditionally called ε, in the comparations of the floating point numbers. If the distance between the numbers is less than ε, the numbers will be accepted as the same number. This *distance* test could be done in different manners:

- Absolute test: $|e_1 - e_2| \leq \varepsilon$.
- Relative test, for instance: $|e_1 - e_2| \leq \varepsilon \times max(|a|, |b|)$.

Where the entities e_1 and e_2 can be numbers (therefore $|...|$ means absolute value) or points (therefore the $|...|$ means a norm: manhattan, frobenius, infinity, the nth. norm). It is also very usual to have different ε's for lengths, areas or angles.

In general these approaches are very indicate if we work with *fuzzy data.* Otherwise, they are fast and intuitive but all of them present some common problems:

- Changing the sequence of computations, we could get different outcomes. That means that some mathematical properties are not respected. Hoffmann reports two of this inconsistencies in [23]:

  Incidence Asymmetry: It happens if $\exists u, v \in$ 2D points and $(u = v) \not\Rightarrow (v = u)$. We show that in figure 6 when the lines L_1, L_2, L_3 and L_4 define, with its two-two intersections, the 2D points u and v. The ε error accepted has been the same in both intersections yet the regions in the plane defined by they are pretty different.

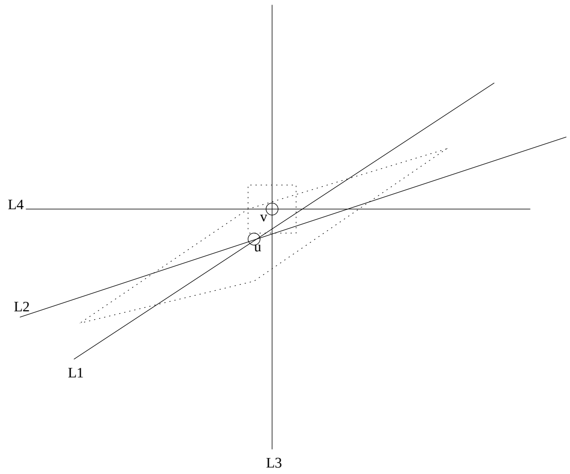


Figure 6: Incidence Asymmetry.

  Incidence Intransitivity: It happens if $\exists u, v, w \in$ 2D points and $(u = v) \wedge (v = w) \not\Rightarrow (u = w)$. We show that in figure 7 when we always use the same ε but the regions are slightly shifted. The geometric inconsistence that has been produced could derive in inconsistencies of the topology.

- It works quite good if the outcomes of the computations are not chained. If this outcomes are chained the methods are usually conservative with the bounds, in order to avoid the first reported problem,

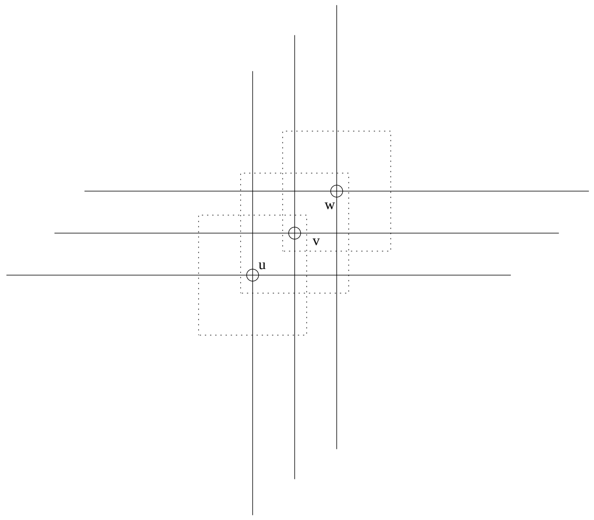


Figure 7: Incidence Intransitivity.

and the zone inside the limits grows quickly. Segal [52] reports that a sequence of duplicate-rotate-union operations repeated eleven times to a cube eventually collapsed it to a single vertex with a large bounded zone around.

- The computation of a bound and its combination with other bounds implies numerical computations that are also sensitive to floating point robustness problems. So, at the end, the inconsistencies remain possible.

As has been said in section 3.1.3, a knows way to avoid the problems reported by Hoffmann, is not to work with fixed ε but to use a fuzzy one. It means that any entity has its own ε and when two of them are combined, their ε's are also combined following an interval arithmetic. As soon as two entities overlap they are merged in a third larger entity that contains both previous ones (see figure8). Due to the fact that the ε change its value, this kind of entities are called by Michelucci [37] *fuzzy entities* and the ε's are *fuzzy* ε.

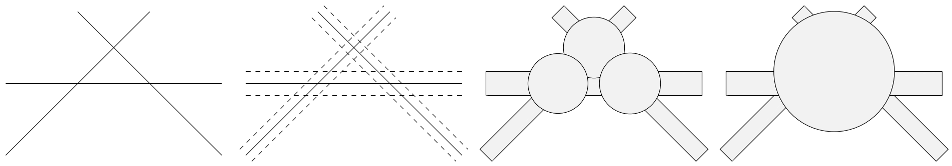


Figure 8: Fuzzy entities merged. Pay attention to the growing ε.

They *fuzzy entities* were presented by Segal and Sequin in [35] [52]. They present a line like a strip between two parallel lines. They called these *tolerance objects*. After that, Jackson in [25] studied how to apply these techniques on 3D solid modeling and he implements an algorithm to compute boolean operations between 3D geometric objects with curved surfaces. Petrikalakis's team has also used this method [8] to create his *interval splines*, which differ from classical splines in that the control point coordinates are represented as intervals and combined with a rounded interval arithmetic.

In all of this approaches the target is to have a *minimum feature separations* and to improve this the features that are too close must be merged, with the risk of loosing information. This merging could be done in some different ways, as we reported in 3.1.3 Rossignac is still working on how to perform this clustering without loss of information.

The Canny's gap theorem [9] give us some theoretical background of the ε heuristic. It gives a way to prove numerically that a number is zero computing a interval containing it. If the interval does not contain zero, the number is not zero and its sign is known. The arithmetics based on that are known as *Gap arithmetics* and they are also studied in section 5. In Yap's book [61] some of the gap theorems are presented.

Unluckily the Canny's gap theorem doesn't have an easy application due to the size of the ε_c that it computes. It implies the use of a big-floats package (see section 5). In other hand, to prove that a number is zero needs a exponential number of calculated digits.

3.2.3 Bounding predicates of the entities.

Here, the idea is also to use a threshold around the graphical entities but it differs of the section 3.2.2 approaches because now the bounds are on the predicate on the geometry. That is why we call this approaches *interval predicates*.

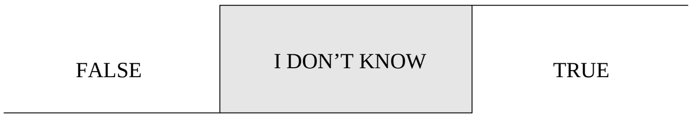


Figure 9: Guibas et al. call this outcomes ε-boxes.

Guibas et al. presented in [32] the ε-predicates that differs of classical ones because instead of insisting on *minimum feature separations* they are allowed to return the *I DON'T KNOW* value (see figure 9). In fact, they return a interval on $\mathbb{R}$ that divides it in three sets: the predicate is TRUE (set T), undeterminate (set U) and FALSE (set F). With this predicates they construct an ε-geometry. Their method combines interval arithmetic techniques, backward analysis error and geometric reasoning.

4 Fixed precision computation approaches.

If we do not want to work with heuristics and if we accept that our computer is naturally discrete, we can try to work in our computer only with discrete representations of the continuous world.

4.1 Overview.

There are two possibilities (figure 10), that represent two different levels in how to apply this technique:

Topological Consistent Distortion: Focused on the lines that define the entities. This lines are distorted into polylines or pseudo-lines. The segments of this polylines or their vertices are constrained to a grid.

Geometric Rounding: Focused on the equations that define lines of the entities. The coefficients of these equations are rounded with some precision, therefore only could be representable a grid of lines.

The two approaches are similar, and the techniques presented in one of them are usually useful in the other. In fact, the authors of the works are not completely agree with the names. It is usually to find works of the first scope, where the writers call their methods as *rounded geometry.* The *Topological Consistent Distortion* approaches can be viewed as a *Geometric Rounding* ones, where the points of the real lines are rounded to the closest grid point without introducing self-intersections. Here we accepted the taxonomy proposed by Yap [62].

We can easily detect some pitfalls in these approaches:

- We can extend our discretized world (representable points or coefficients of the equations) in order to represent all the input entities, yet it is impossible for us to know, a priori, the points that will be necessary when these entities will be combined. Therefore the entities will be necessary distort. As we see in the figure 11 the intersection of

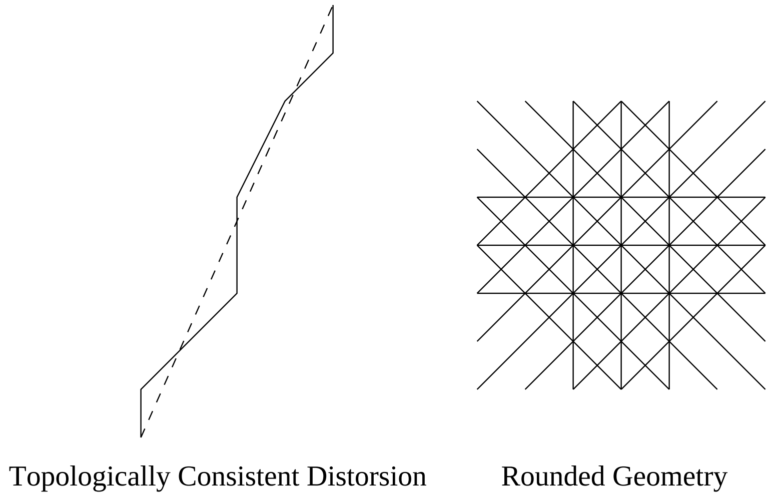


Figure 10: Topological Consistent Distortion vs. Rounded Geometry.

two accepted lines could be in an unrepresentable point. It creates an accumulated *distort error* proportionally of the number of segments that we intersect and that result critical because the distort could introduces new intersections. To avoid it is essential to detect carefully the consequences of the changes we make.

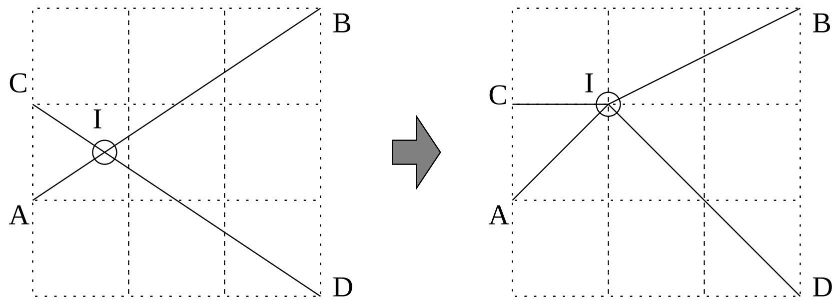


Figure 11: The incomplete representation introduces a growing error.

- Some combinations of the entities, like intersecting segments at a small angle, needs two or three times the input precision in order to get an outcome with the same accuracy as the input entity definitions. That occurs because they are *ill conditioned* (see section3.1.2). If we do not provide this required precision, we will be introducing an *roundoff errors* (see section 2.3). Due to the overlapping edges generated by rounds, very small angle intersections between perturbed segments are usual. If we combine three entities in an intersection we must use more than six times the input precision to decide which of three possible sketch is produced.

- We can find what Milenkovic called, in [43], *unexpected incidences* . We could see the discretization of the world as to work with a grid and that usually introduces in the model *overlapping edges*. If we have this, some modeling rules can not be naively used. For instance, as Milenkovic remarks, we can not attempt to take a regular boolean operation like the union using a *covered edge rule*[2]. Like we see in figure 12, it could work not properly. This could also happen if we use an exact representation of the model, but then we have a general technique called *simulation of simplicity* [20][58] for removing these special cases.

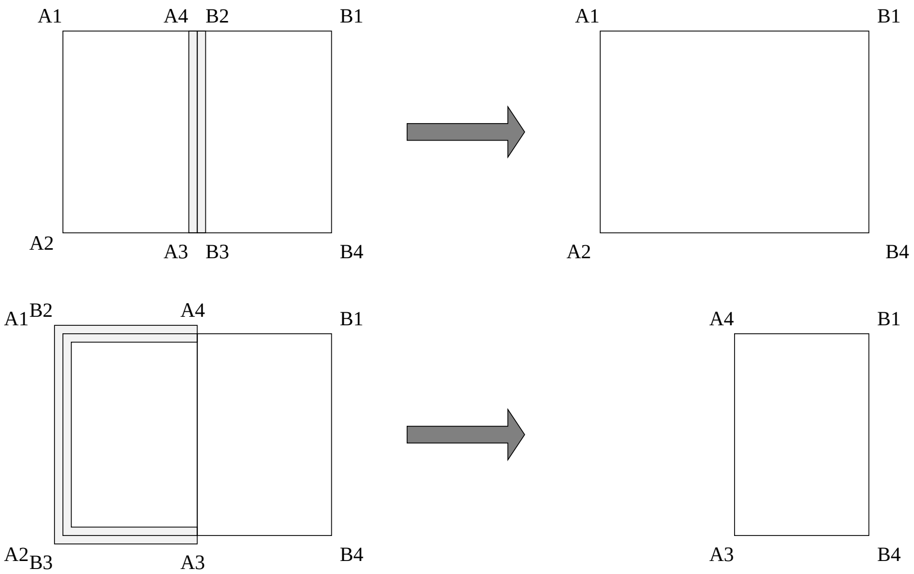


Figure 12: Some modeling rules, as *covered edge rule*[2], can not be naively applied if we have overlapping edges. In this figure we can see the union of the polygons $A_1A_2A_3A_4$ and $B_1B_2B_3B_4$ defined in two different manners

A discussion of the more general problem of improving the reliability geometric computations using these methods could be found in [6] [22] [23]. In [6] Hoffmann et al. address the problem of intersecting polygons and introduce the *reasoning paradigm*: The problem of verifying consistency of combinatorial structures.

[2]remove any edge covered by the other polygon

4.2 Topological Consistent Distortion.

The first algorithm for rounding an arrangement of lines segments to a grid was presented by Greene and Yao in [11]. This method distorted lines into polylines, where the vertices of these polylines are constrained to be a grid points. To do this, they introduce the concept of a *hook* between intersection points and the closest grid point in order to pull the new points to a known ones (the perturbation error of every point is $\leq \sqrt{2}$). Unlikely, this could generate new intersections called *derived hooks* as we introduce in section 4.1.

Another problem of this method is that introduce $\Theta(m)$ extra vertices per edge, being m the number of bits of precision of the grid. To cope with it, Milenkovic presented an algorithm for rounding a polygon [41] that introduced no new vertices. He also present how to round both sets of line segments in 2D [44] (with no new vertex locations) and sets of triangles in 3D [45] (that introduces new vertices) to a nonuniform grid with a controlled error. Further developments include the numerically stable algorithms in [15] and bounded-error rounding algorithm for nonuniform grids [47] that uses an evolved *shortest path rounding algorithm* from his previous work [40] [42] [46].

Hobby, also present a method [21] for rounding an arrangement of line segments to the uniform integer grid. This method hooks each edge to every vertex whose *rounding cell*[3] will be intersect by the edge. The pros are that runs in linear time,the no introduction of new vertices and the introduction of an error at most equal to the grid spacing. The con is that it does not minimize the combinatorial or geometric changes.

4.3 Geometric Rounding.

Sugihara presents in a lecture given in 1987 at the IMA Summer Program on Robotics at the University of Minnesota the idea of exact rational arithmetic, where the equation's coefficients that define the entities are rounded with some precision. We call this approach *Geometric Rounding.* In [55] shows that the problem of rounding free equations to rationals ones can be

[3]A *rounding cell* of a grid point is defined as the space covered with the *26-adjacent* points of the grid. Two points in 3D-space are said to be *26-adjacent* if they are distinct and each coordinate of one differs from the corresponding coordinate of the other by at most 1. In the same manner, they are *18-adjacent* if they are 26-adjacent and differ in at most two of their coordinates. They are *6-adjacent* if they are 26-adjacent and differ in at most one coordinate. A complete introduction on Digital Topology could be found at [31].

reduced to the classical problem of *simultaneous approximation by rational* [4]. It is very important to find tighter bounds on the internal precision needed to decide incidence. For sharp lower bounds on the minimum separation distance see [63]. For plane rounding there are methods that usually give better results than others. Sugihara considers *The Continued-Fraction method* (see [23]) and *The Basis-Reduction method* (see [34]) but both of them ignore the interaction between coefficient perturbation and the locality of the final face on the plane.

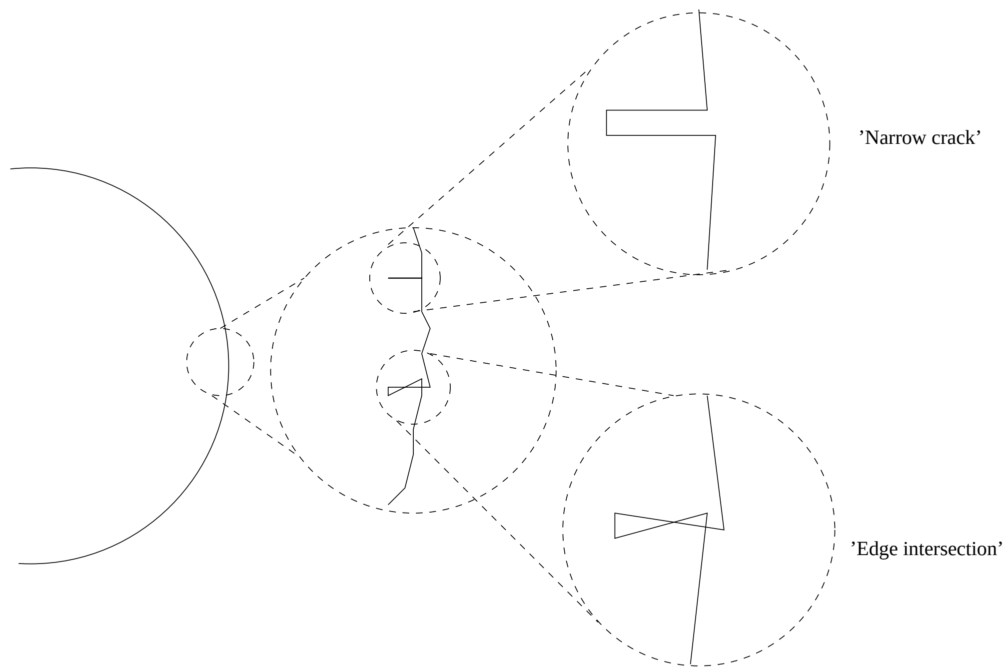


Figure 13: To round the geometry could introduce topological inconsistencies.

To round the geometry could introduce topological inconsistencies such as *narrow crack* or *edge intersection* (see figure 13) specially if the geometry has been translated or rotated. There are several ways to reduce the lacks of consistence:

- Building the polyhedrons with boolean operations of simple entities (cone, cylinder, sphere and so forth). Therefore, when a polyhedron will be translated, each simple entities will be also translated and the polyhedron could be rebuild again with boolean operations. Of course the correct rotation or translation of the simple entities is required.

[4] Given $a_1, ..., a_n \in \mathbb{R}$ find integers $p_1, ..., p_n$ and q such that $max_{1 \leq i \leq n} |a_i q - pi|$ is minimized.

That is easy if the topology of this simple entities are also simple for instance trihedral. Knowing that all polyhedron could be constructed in principle by a suitable sequence of regularized Boolean operations, Hoffmann presents a method [23] to control the topological integrity of the rebuilt polyhedron. The most important con of these options is the time needful to do the computations.

- In modeling time, must be guaranteed a minimum separation between edges in function of the representable grid of planes. That improves the possibilities of to skip the perturbations.
- A rotation center closest to the rotation faces will introduces less perturbations than a distant one.

Milenkovic et al. [56] show that rounding a collection of disjoint simple polygons while preserving their combinatorial structure in NP-complete.

5 Exact computation approaches.

A different approach is that Michelucci called *Exact Computation Paradigm*, where the target is to compute every geometric object exactly. To do that we need to assure an exact input. We can get our input data in two different manners:

- Introduced by the user:
  - clicking with the mouse, usually on a grid.
  - as a numerical values. It can have as precision as the user decided. Usually, if it is introduced directly by the user, it does not have a lot of decimals but if it comes from a device, it could have higher precision.
  - as an expressions that can involve implicit inexact values such as $\sqrt{2}$ that must clearly preserved.

  In all these cases the introduced data could be free or constrained. If the data are free, the numerical quantities are representable and do not involve implicit inexact values there are not any problem. If it not happens we have transform the data into values that can be managed in an exact way. To do that we have two options:

  1. Change the data in order to clean up the inexactness. It is not a trivial process and has to be carefully controlled.

2. Rewrite the problem in order to treat the data as exact. As Yap suggest [62] *inexact input points can be viewed as center of small balls.*

- As an outcome of another computation. If it has not been an exact one, the values can be treated in the ways previously exposed. When we will have to make an exact computation we will always choose error-less branches[5].

5.1 Overview

Inside of the *exact computation approaches* we could find two sets:

Big Number Packages (BNP): They increase the precision in the calculations, usually in a dynamical way, in order to avoid inaccuracies but do not take profit of all simplify capabilities of the expressions and usually, they have excessive numerical precision. Like a naive approach, it is easy to implement and could be seen such as *carefully computation approach.* Some works, like [7], have tested the improvement that these approaches introduce.

Big Expression Packages (BEP): These approaches, like *BNP* increase the precision in the calculations but they also take profit of the fact that numerical part in geometric computations can usually be reduced to repeated evaluations of a few expressions. In fact, these approaches differ from *BNP* in that they work with the expressions in addition to the numbers. Some of this usually repeated expressions can be computed in different manners if target is to get a exactly numerical outcome or just a boolean answer (distance between two points, lines arrangement, tests that involves determinants, etc.)

Some works, like [54], are between this two sets because they are more than a *BNP* but less than a *BEP*. For instance, this that has been reported, implements a floating-point arithmetic with adaptative-precision evaluation of geometric predicates techniques.

5.2 Big Expression Packages (BEP)

Michelucci [37] surveys on the different methods beside these packages dividing them in *Rational Arithmetics* (eg. *LN*) and *Algebraic Arithmetics* (eg.

[5]We could see the different manners to do a computation as a different branches in a computation tree as Yap explains in [62]

Real/Expr). In this lastly set the gap arithmetics are presented. As we have introduced in 3.2.2, the Canny's gap theorem [9] give us some theoretical background of the ε heuristic. It gives a way to prove numerically that a number is zero computing a interval containing it. If the interval does not contain zero, the number is not zero and its sign is known. The arithmetics based on that are known as *Gap arithmetics.* In Yap's book [61] some of the gap theorems are presented.

Unluckily the Canny's gap theorem doesn't have an easy application due to the size of the ε_c that it computes. It implies the use of a big-floats package. In other hand, to prove that a number is zero needs a exponential number of calculated digits.

The most known packages are:

LN (Little Numbers) [51]: That is a system for efficient exact arithmetic based on the principle that most of the time, the floating-point expression has the correct sign and there is no need to use exact arithmetic. The target is to determine when we need an exact arithmetic. To do that, *LN* proceed in three steps:

- First, in a pre-compilation, the minimum numbers of digits needed for the exact arithmetic is determined and outputs C++ code which computes the tests in standard floating-point arithmetic, derives error bounds for each expression and if needed include the calls to the exact library in order to evaluate the expression.
- Second, in a compilation, the C++ code is linked with the exact library. The tests are made with references to the original data and a floating-point filter based on static error analysis. The outcomes are the maximum possible error of each expression.
- At run-time, the expressions are first evaluated in floating-point. If the magnitude of the resulting values are greater then the error bound, it returns the resulting sign. Otherwise, the full precision required for an exact computation is used.

The pros of this library are to remove all overhead with function calls or dynamic allocation of space for numbers with unknown sizes and a very efficient timing. The cons are that it forbids some on-line and reentrant algorithms and introduces a lot of new (if not, unnecessary) types.

LEA (Lazy ExAct numbers) [36]: implements a *lazy arithmetic* that works with *lazy rational numbers* which are represented by:

- An interval of two floating-point numbers that contains the exact value of the number, no matter if it is evaluated or not.

- A symbolic definition of the number that will be use to recover the exact value, if need be. This definition could be:
  - A dynamic structure that completely represent the exact value of the number.
  - An operation (sum or product) with two other *lazy rational numbers*.
  - An operation (inverse or opposite) with one *lazy rational numbers*.

  As we see, this introduces a tree structure where each *lazy rational numbers* is the root of a tree of other *lazy rational numbers*. More exactly it is not a tree because its leafs are shared between these different trees. It creates a acyclic graph structure.

The target is to delay evaluating an expression until forced to, and maintain intervals of uncertainty for values. The intervals become insufficient and thus the symbolic definition has to be evaluated when:

- A comparison of two *lazy rational numbers* is needed and their intervals overlap.
- Is necessary to know the sign or the inverse of a *lazy rational number* and its interval contains 0.

Upon a complete evaluation, the expression is discarded. The cons of this library, opposite to *LN*, are the overhead due to function calls or dynamic allocation of space for numbers. The pros are that it allow on-line and reentrant algorithms. If *LN* is usable, it would be faster than *LEA*.

LEDA (Library of Efficient Data structures and Algorithms) [5][6]: That is a complete library of data structures for combinatorial (and thus geometric) computing inside what the exact computation is just a part. It started in fall of 1988 such as a project to build a small but growing library of data types and algorithms in a form which allows them to be used by non-experts.

In order to provide accurate numerical computations, *LEDA* implements:

- *Integers* with arbitrary length as vectors of unsigned long numbers. The sign and the size are stored in extra variables and some critical functions are implemented in assembler code in order to get good timing. These *Integers* are also used to implement *Rationals* and *Big Floating-point* numbers with arbitrary length. In

[6] *LEDA* is available via WWW from: www.mpi-sb.mpg.de/LEDA

other way, they also implement *Vectors with Integer* entries and with them *Matrices with Integer* entries. Finally, the *Rationals* are used to implement *Rational* vectors which represent vectors in homogeneous coordinates.

- *Algebraic Real* numbers represented by the expression that defines it, an approximation x' computed with double precision and a relative error bound ε_x. When a sign of a *real* number needs to be determined *LEDA* computes a number q, using [39], such that $|x| \leq 2^{-q}$ implies $x = 0$. These *Algebraic Real* numbers are also used to implement *Real-Valued* vectors and with them *Real-Valued* matrices.
- *Floating Point filters* that provides an efficient way to approximately compute with large integers. They are represented by a double (the value) and an error bound. An operation with *Floating Point filters* performs the corresponding operation on the values and also compute the error bound, using [26], for the result.

These implementations include exact comparations for constructible expressions by maintaining root bounds.

It also includes a comfortable data type graph that offers standard iterations such as:

1. *for all nodes v of the graph G do ...*
2. *for all neighbors w of v do ...*
3. etc.

It could help geometric programmers to skip numerical computations working with symbolical information as we introduce in chapter 3.1.3.

Real/Exp [7]: Is a set of C++ libraries which supports the precision-driven approach to implementation of exact algorithms in computational geometry. For each expression, the user must demand a precision in absolute or relative number bits or even using a combination of both. This initial precision can then be propagated to other parts of the computation, at run time, so that precision is locally adaptative. This approach work specially good when the interests of a computation are focused on the sign and not on the numerical value.

This approach implement the *Expr* package for the class of constructible real numbers where this class is built on top of two classes of independent interests *BigFloat* and *Real*. Therefore in the package are three major classes:

Expr, that captures a set of algebraic expressions.

BigFloat, a class with automatic error-bound propagation.

Real, an extendible class that encompasses a variety of real number representations.

CGAL (Computational Geometry Algorithms Library) [1] [2] [49]: It is a collaborative project that effort to develop a robust, easy to use, and efficient C++ software library of geometric data structures and algorithms. *CGAL* is partially based on *LEDA*. In CGAL you can choose the underlying number types and arithmetic. So you can choose between implementations with fast but occasionally inexact arithmetic and implementations guaranteeing exact computation and exact results. Of course you have to pay for the exactness in terms of execution time and storage space.

6 Conclusion.

A very naive, yet also important, conclusion that we must remark is that robust computation does not necessary means a high precision computation. Robust means sufficiently accurate for our purposes and that necessary is a fuzzy definition. If we want to compute a control point of a spline like Petrikalakis do in [8] we can permit us some lacks of precision but if we want to intersect some lines in the space could be necessary to know exactly the intersection point.

More generally, some mathematical problems usually need exactness due to that they work with ideally tools, but computer graphics applications in CAD/CAM, Medicine, Chemistry or Physics only need accurate enough representations of the world to work with it.

Therefore a robust computation must be what guarantees the stability of the algorithm. That is, given a metric of the distortion on the outcomes of a computation, there is a threshold which bounds the maximum accepted error before the crash of the algorithm.

7 Acknowledgments.

This research has been done in the scope of my Phd. Thesis. The author would like to thank Dr. Robert Joan-Arinyo for his work as my Phd. advisor. I would also like to thank Toni Soto and Carlos Andujar for their helpful information and suggestions. Finally, thanks are due to Xavier Ponce for reading drafts of this paper and providing many useful comments.

References

[1] American National Standards Institute (ANSI), Institute of Electrical, and Electronic Engineers (IEEE). IEEE standard for binary floating-point arithmetic. Technical Report Std 754-1985, ANSI/IEEE Standard, New York, 1985.

[2] M. O. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy arithmetic library. In *11th. Annu. IEEE Sympos. Comput. Arith.*, pages 242–269, 1993.

[3] R. L. Burden. *Numerical analysis (5th ed).* Boston : PWS Pub. Co. ISBN:0-534-93219-3, 1993.

[4] C. Burnikel, J. Konnemann, and K. Mehlhorn et al. Exact geometric computation in leda. In *11th. Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[5] J. Canny. *The complexity of robot motion planning.* M.I.T. Press, Cambridge, Mass., 1988.

[6] D.N.Arnold. Two disasters caused by computer arithmetic errors. http://www.math.psu.edu/dna/455.f96/disasters.html, 1996.

[7] D. Dobkin and D. Silver. Recipes for geometry and numerical analysis, part 1:an empirical study. *4th ACM Symp. on Comput. Geom.*, pages 93–105, June 1988.

[8] H. Edelsbrunner and E. P. Mucke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. Technical Report UIUCDCS-R-87-1393, Dept. of Comput. Sci., Univ. of Illinois, Urbana-Champaign, Urbana, Illinois, Dec 1987.

[9] G. Engeln-Müllges. *Numerical algorithms with C.* Berlin [etc.] : Springer, ISBN:3-540-60530-4, 1996.

[10] R. C. Agarval et al. New scalar and vector elementary functions for the ibm system/370. *IBM Journal of Research and Development*, 30(2):126–144, March 1986.

[11] A. Fabri, G.-J. Giezemann, and L. Kettner et al. The cgal kernel: A basis for geometric computation. In *ACM Workshop on Applied Computational Geometry*, Lecture Notes in Computer Science 1148, pages 191–202, 1996.

[12] A. Fabri, G.-J. Giezemann, and L. Kettner et al. On the design of cgal, the computational geometry algorithms library. In Dorothea Wagner ed., editor, *Algorithm Engineering.* New York, 1997.

[13] S. Fortune. Numerical stability of algorithms for line arrangements. In *7th Annu. Symp. on Comput. Geom. (SCG)*, North Conway, N. H., June 10-12 1991.

[14] S. J. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. In *ACM Trans. Graph.*, pages 15:223–248. 1996.

[15] I. Fudos and C. M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, April 1997.

[16] General Accounting Office (G.A.O.). (IMTEC-92-26) patriot missile defense: Software problem led to system failure at dharham, saudi arabia, 1992.

[17] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[18] D. Goldberg. *SUN's Numerical Computation Guide: Differences Among IEEE 754 Implementations*, chapter Appendix D. Sun Soft, Inc., 1997.

[19] M. Grau and M. Noguera. *Càlcul numèric.* Ed. UPC, ISBN 84-7653-256-3., 1993.

[20] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *27th. Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

[21] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proceedings of the fifth annual Symposium on Computational Geometry: Saarbrucken, West Germany, June 5–7, 1989*, pages 208–217, New York, NY 10036, USA, 1989. ACM Press.

[22] J. Hobby. Practical segment intersection with finite precision arithmetic. Technical report, AT&T Bell Labs., October 1993.

[23] C. Hoffmann, J. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. In *4th. Annu. ACM Sympos. Comput. Geom.*, pages 106–117, 1988.

[24] C. M. Hoffmann. The problem of accuracy and robustness in geometric computation. Technical Report CSD-TR-771, Comp. Sci. Dept. Purdue Univ., April 1988.

[25] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann, San Mateo CA, 1989; 335 pages; ISBN 1-55860-067-1, 1989.

[26] C.-Y. Hu, N. Patrikalakis, and X. Ye. Robust interval solid modelling. part 1: Representations. part 2: Boundary evaluation. In *CAD*, pages 28(10):807–817,819–830. 1996.

[27] D. Jackson. Boundary representation modelling with local tolerances. In *Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 247–253, 1995.

[28] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Technical report, Elect. Eng & Computer Science, Univ. California, Berkeley CA 94720-1776, May 31, 1996.

[29] William Kahan. An interview with the old man of floating-point. http://HTTP.CS.Berkeley.EDU/˜ wkahan/ieee754status/754story.html, 20 Feb.1998.

[30] William Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics and chemistry. Univ. California, Berkeley CA 94720-1776, June 11, 1996.

[31] R. B. Kearfott. *Rigorous Global Search: Continuous Problems.* Number 13 in Nonconvex optimization and its applications. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1996. Includes a module on interval arithmetic, as well as Fortran 90 code for automatic differentiation, nonlinear systems code, and constrained and unconstrained optimization.

[32] T. Y. Kong. Digital topologu: Introduction and survey. In *Computer Vision, Graphics and Image processing*, pages 48:357–393. 1989.

[33] Vincent Lefère, Jean-Michel Muller, and Arnaud Tisserand. The table maker's dilemma. Technical report, Laboratoire de l'Informatique du Parallélisme, École Normale Supŕieure de Lyon., Feb. 1998. Research Report N^o 98-12.

[34] J. L. Lions. ARIANE 5 flight 501 failure, report by the inquiry board. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, Paris, 19 July 1996.

[35] L. Lovász. An algorithmic theory of numbers, graphs, and convexity. In *SIAM Publications.* Philadelphia, 1986.

[36] K. Mehlhorn and S. Naher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical report, Max-Planck-Institut fur Informatik, Saarbrucken, 1994.

[37] D. Michelucci. The robustness issue. Found at http://www.emse.fr/micheluc/english/michelucci.html.

[38] D. Michelucci. An epsilon-arithmetic for removing degeneracies. In *IEEE Symposium on Computer Arithmetic*, Bath, July 1995.

[39] M. Mignotte. *Mathematics for Computer Algebra.* Springer Verlag, 1992.

[40] V. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic.* PhD thesis, Comp. Sci., Carnegie-Mellon Univ., 1988.

[41] V. Milenkovic and L. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *6th. Annu. ACM Sympos. Comput. Geom.*, pages 244–252, 1990.

[42] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. In *Artificial Intelligence*, pages 37:377–401. 1988.

[43] V. J. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *30th. Annu. Symp. on the Found. of Comput. Sci.*, pages 500–506. IEEE, October 1989.

[44] V. J. Milenkovic. Robust geometric computations for vision and robotics. In *DARPA Image Understanding Workshop*, pages 764–773, May 1989.

[45] V. J. Milenkovic. Rounding face lattices in the plane (extended abstratc). In *1st. Canadian Conference on Computational Geometry*, Montreal, Quebec, Canada, August 21-25 1989.

[46] V. J. Milenkovic. Rounding face lattices in d dimensions. In Ed. Jorge Urrutia, editor, *2nd. Canadian Conf. on Comput. Geom.*, pages 40–45, Ontario, August 6-10 1990. Univ. of Ottawa.

[47] V. J. Milenkovic. Robust polygon modelling. In *Camputer-Aided Design*, pages 25(9):546–566. September 1993.

[48] V. J. Milenkovic. Practical methods for set operations on polygons using exact arithmetic. In *7th Canadian Conf. on Comput. Geom.*, pages 55–60, Quebec, August 1995. Laval University.

[49] R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM Philadelphia, 1979.

[50] M. H. Overmars. Designing the computational geometry algorithms library cgal. In *ACM Workshop on Applied Computational Geometry*, Lecture Notes in Computer Science 1148, 1996.

[51] J. Rossignac and P. Borrel. Multiresolution 3D aproximations for rendering complex scenes. In *Modeling in Computer Graphics*. Springer-Verlag, 1993.

[52] M. G. Segal. Using tolerancesto guarantee valid polyhedral modeling results. In *Comput. Graph.*, pages 24:105–114. SIGGRAPH'90, 1990.

[53] M. G. Segal and C. H. Sequin. Consistent calculations for solids modelling. In *1st Annu. ACM Sympos. Comput. Geom.*, pages 29–38, 1985.

[54] Charles Severance. IEEE 754: An interview with william kahan. *IEEE Computer*, 31(3):114,115, March 1998.

[55] J. R. Shewchuk. Robust adaptative floating-point geometric predicates. *12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[56] K. Sugihara. On finite-precision representations of geometric objects. *Comput. Syst. Sci.*, pages 39:236–246, 1989.

[57] C. K. Yap. A geometric consistency theorem for a symbolic perturbation theorem. In *Symposium on Computational Geometry, ACM*, 1988.

[58] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, pages 40:2–18, 1990.

[59] C. K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, pages 10:349–370, 1990.

[60] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1996.

[61] C. K. Yap. *Handbook of discrete and computational geometry*, chapter Chapter 35: Robust geometric computation, pages 653–668. CRC Press, 1997.

[62] C. K. Yap and T. Dubé. *Computing in Euclidean Geometry, 2nd edition.*, chapter The exact computation paradigm., pages 452–486. In D.Z. Du, F.K. Hwang Ed. World Scientific, Singapore., 1995.

[63] J. Yu. *Exact Solid Modeling.* PhD thesis, Dept. Comp. Sci., Purdue Univ., 1991.