A Practical Study and Evaluation of Libraries for Exact Geometric Computing

Diplomarbeit

zur Erlangung des Diplomingenieurgrades an der
Naturwissenschaftlichen Fakultät der Universität Salzburg

eingereicht von

Alexander Schneider

Salzburg, 14. Juni 2002

Acknowledgements

First of all I want to thank my advisor Martin Held for his support during the development of the Core-based version of FIST. Furthermore, his comments helped to improve the technical aspects of this thesis as well as its presentation. I want to thank the developer team of the Core-library, especially Chee Yap and Zilin Du who provided us with the newest versions of the Core-library and tried to solve the problems we had with the library. Thanks also go to Wolfram Stering, who gave me access to the Insure++ memory debugger and who was helpful whenever I had trouble with the license server. I also want to thank my brother Peter Schneider for his offer to use his computer hardware, which is much faster than my current system, as the test platform for all the programs presented in this thesis. Last but not least I want to thank my parents who always supported me during my studies in computer science.

Salzburg, June 2002 Alexander Schneider

Contents

Chapter 1

Preface

Computational Geometry is concerned with the study of geometric algorithms – geometric problems are analyzed, algorithms to solve these problems are designed and their complexities are determined. While researchers concentrated on the theoretical foundations of geometric algorithms, i.e., on finding an algorithm for a geometric problem and analyzing its complexity, the robust implementation of those theoretically correct algorithms has gained more and more attention only in recent years. It turned out that, in general, the straightforward implementation of a geometric algorithm is not feasible. Although the algorithm is correct in theory, it may fail in practice. The reason for this is twofold:

- In theory a geometric algorithm is constructed assuming real numbers in the mathematical sense.
- In order to avoid unnecessary complexity in the algorithm description, special cases are excluded, i.e., only input instances in a so-called "general position" are considered.

Unfortunately, neither of these assumptions hold in practice. In general, geometric algorithms are implemented using floating-point arithmetic. Thus, the majority of numerical quantities cannot be represented exactly and are therefor only approximations. Furthermore, special cases are sure to occur in real world data. As a consequence, a straightforward implementation of a theoretically correct algorithm yields a computer program that crashes or, even worse, computes incorrect results.

The goal of this thesis is to evaluate the Core-library – a library for exact geometric computation. Exact geometric computation is an approach to overcome robustness problems in geometric algorithms and is based on exact arithmetic. Up till now, exact arithmetic is not widely accepted by practitioners in the field of computational geometry. The reason for this is that there is no suitable infrastructure for exact arithmetic, i.e., tools and libraries that support the development of algorithms based on exact arithmetic. Furthermore, the lack of efficiency is another argument against it. The developers of the Core-library claim that their library is easy to use and reasonably fast to be an alternative to floating-point arithmetic. We want to evaluate those claims. Thus, we linked the Core-library with FIST [21] – a triangulation algorithm by Martin Held which is based on floating-point arithmetic, thoroughly tested and incorporated into several industrial graphics packages, such as an implementation for Java 3D by Sun Microsystems. It has to be mentioned, though, that exact arithmetic does not solve any problems that arise due to the assumption of a "general position". Even with exact arithmetic the software developer has to handle special cases or, alternatively, use a general approach based on perturbation theory to eliminate degeneracy.

1.1 Structure of this Thesis

In Chapter 2 we discuss number representation on computer systems. We focus on the floating-point representation which is commonly used in practice and is the reason for the majority of robustness problems in the field of computational geometry. Robustness problems in geometric algorithms and approaches to avoid them are the topic of Chapter 3. The exact geometric computation approach and libraries that implement exact geometric computation techniques are discussed in Chapter 4. The Core-library is the topic of Chapter 5. We discuss basic concepts of the library and provide instructions on how to use the library in own software projects. Finally, Chapter 6 provides a survey of the triangulation algorithm FIST and describes the changes we made to FIST in order to make it compliant with the Core-library. Experimental results are also presented in this chapter.

1.2 Platform Specifications

The specifications of the platform we used for all the tests in thesis are as follows:

CPU: AMD Athlon, 1400 MHz,

RAM: 256MB DDR-RAM,

Operating System: SuSE Linux 7.3,

Compiler: g++ 2.95.3,

Debugger: gdb 5.0

Memory Debugger: Parasoft Insure++ 6.0

Chapter 2

Floating-Point Arithmetic

In this chapter we take a closer look at the way numbers can be represented on a computer system. In the next section we mention basics of fixed-point and floating-point representation of real numbers. Furthermore, we explain important concepts related to these two number representations. The focus of this chapter will lie on the floating-point representation since it is more flexible and therefor unavoidable if we want to implement geometric algorithms. An important step towards applicability of floating-point numbers on computer systems was its standardization by the IEEE. This standard is the topic of Section 2.4 and concludes this chapter. Sources of errors in numerical computations due to the use of floating-point arithmetic and a short introduction to error analysis are the topics of Section 2.2 and 2.3

2.1 Basics of Number Representation

Using computer systems for numerical computations makes it necessary to find a way to represent numbers. For complex computations the use of integer arithmetic does not suffice. This is especially true for many applications in the field of computational geometry. Therefor, a way to represent real numbers is essential. The major problem is that it is impossible to represent infinitely many real numbers with a computer system that only offers a finite number of bits for number representation. The fact that we can only represent a subset of real numbers introduces errors due to rounding and truncation in numerical computations. While those errors may be innocuous in some applications they may be harmful in other applications and even cause those applications to fail. Geometric algorithms are susceptible to numerical errors and software developers have to undertake great efforts to ensure robustness of their applications. As mentioned above there are two important ways real numbers can be represented on a computer system – fixed-point representation and floating-point representation. We will now look at the fixed-point representation in some detail and then focus on the floating-point representation due to its importance for today's computer systems.

2.1.1 Fixed-Point Representation

Suppose we have n bits to represent a real number and we decide to place the radix point in a way such that we use m bits for the fractional part and $n-m$ for the integer part of the number. Since the radix point is fixed in this position[1] we speak of a fixed-point number representation. This means that every number we deal with has to be represented according to this convention even if there is no fractional part. As an example, let us assume that we have 8 bits to represent our number. We decide that we use 3 bits for the fractional part and 5 bits for the integer part of the number. The number we want to represent is 2 in decimal notation. The binary fixed-point representation would be 00010.000. In order to get the decimal value of a binary number in fixed-point representation we multiply each digit of the integer part with a power of 2 according to its position (increasing from the right to the left) and sum up the results. The fractional part is computed by multiplying each digit to the right of the radix point with a negative power of 2 again according to its position (decreasing from the left to the right) and subsequent addition of the results. Recall our example from above. We have 8 bits to represent a real number. We choose to take 3 bits for the fractional part so we have 5 bits left for the integer part of the number. With $d_i \in \{0,1\}$, the decimal value of the number $d_4d_3d_2d_1d_0.d_{-1}d_{-2}d_{-3}$ would be

$$d_4 * 2^4 + d_3 * 2^3 + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0 + d_{-1} * 2^{-1} + d_{-2} * 2^{-2} + d_{-3} * 2^{-3}.$$

One characteristic of fixed-point numbers is that they are equally distributed over the representable range. This means that the absolute distance between two adjacent numbers, called the resolution[2], is constant. The resolution depends on the number of bits used for the fractional part of a real number. According to [38], the resolution r is given by

$$r = 2^{-m},$$

where m is the number of bits used for the fractional part of the number represented. The range of representable fixed-point numbers using n bits for

[1]The position of the radix point is arbitrary and depends on the range and precision we want to achieve.

[2]The resolution of a fixed-point number distribution is often defined as the smallest non-zero representable magnitude [50].

number representation and m bits for the fractional part is, according to [38], given by

$$[0, 2^{n-m} - r),$$

where r is the resolution of the number distribution. There is an interesting connection between the representable range of numbers and the resolution. If no bits are used for the fractional part, meaning $m = 0$, then the representable range has reached its maximum, while the resolution $r = 1$ and is therefor very coarse. As m grows the range of representable numbers decreases while the resolution r increases. An advantage of fixed-point numbers is that the basic arithmetic operations can be carried out using integer arithmetic. Since fixed-point calculations were significantly faster than floating-point calculations on older computer systems this number representation played an important role in computer graphics especially for interactive graphics on low-cost computer systems, e.g., consoles for computer games. This situation has changed dramatically in recent years. Nowadays even conventional personal computers have high-performance hardware floating-point units and modern graphics hardware supports the CPU in a lot of calculations. Therefor, fixed-point arithmetic has lost its importance in modern graphics applications. There is also a major drawback in the use of fixed-point arithmetic. Since the radix point is fixed, problems will arise if the numbers we intend to represent vary a lot in their magnitude. As an example let us use the fixed-point representation from above. The three bits we use for the fractional part are not enough to represent the decimal number $\frac{1}{16}$. The binary representation of $\frac{1}{16}$ equals

$$0.0001,$$

so we need an additional bit for the fractional part of the number. In our 5.3 fixed-point number representation we would represent $\frac{1}{16}$ as

$$00000.000,$$

which is clearly wrong. The same is true for large numbers. The maximum number we can represent with five bits is 31. So we run into similar problems if we wish to represent the number 32. As we can see from these examples we need some information on the bounds of the numbers we expect to represent and then decide how many bits to use for the number representation. Finally, a decision on how to divide those bits for the integer and the fractional part has to be made. If we do not take this decision carefully we would suffer from heavy truncation errors. This ends our insight on fixed number representation. The reader is referred to [38, 50, 25] for more information on the fixed-point representation, for details on how the basic arithmetic operations are carried out, and on the representation of negative numbers using the one's and two's compliment.

2.1.2 Significant Digits

Before we get to the floating-point representation of real numbers we want to explain the notion of significant digits. As we saw in Subsection 2.1.1, we have only a finite number of bits to represent numbers on a computer system. As a consequence we may be forced to throw away digits if the number we want to represent has more digits than we can store. Consider the number 1351245 that has seven digits in decimal notation. Assuming we can only use five digits per number which digits should we throw away? First of all we have to remember that the original number had seven digits instead of just five. In other words the original number is hundred times bigger in magnitude than the number we represent using only five digits. So we need to multiply the five digit number by one hundred[3] if we want to reconstruct the original number. Furthermore, it should be noted that this reconstruction actually leads to an approximation of the original number since we are not able to reconstruct the two digits that have been lost. Clearly this approximation should be as close as possible to the original number. Therefor, we choose to drop the last two digits of 1351245 and store the number[4] 13512. As a second example consider the number 0023478. Trying to store this number using five digits leads to 23478. No information is lost since the two leading zeros are redundant. Apparently the leading zeros are not significant. All in all we have four rules for determining the significant digits of a given number, see also [32]:

1. Non-zero digits are always significant,
2. The digit zero is significant if it lies between other significant digits,
3. The digit zero is significant if it follows an embedded radix point and other significant digits,
4. Zeros that precede all other non-zero digits are not significant.

Zero digits that follow an embedded radix point and other significant digits are significant as stated in Rule 3 because they tell us something about the accuracy of the number [32], e.g., the number 6.20 tells us that it was measured to the nearest hundredth. Once we have determined the significant digits according to these rules, we are able to define the terms most significant digit and least significant digit. The significance of a digit corresponds to the way we write down numbers on a piece of paper: the left most digit of the significant digits determined is called the most significant digit and the

[3]This corresponds to the exponent of a floating-point number.

[4]Remember that we also need to store the multiplication factor 100 to reconstruct an approximation to the original number.

other digits follow in descending order of significance from left to right. The rightmost digit is therefor called the least significant digit.

2.1.3 Floating-Point Representation

A major drawback in the use of fixed-point arithmetic is its lack of flexibility if the numbers we need to represent vary a lot in their magnitude. Remember the problems we had before when we tried to represent the decimal number 32 with our 5.3 binary fixed-point representation. The result was 00000.000, since we did not have enough bits to represent the integer part of the number. On the other hand there are three bits for the fractional part of the fixed-point number that are not needed to represent the integer 32. While a fractional part may be needed for the representation of other numbers it is useless in the case of the integer 32 and the bits used for the fractional part are wasted. Problems like these can be softened but, as we will see later, not completely avoided using the floating-point representation. Nevertheless there is no doubt that the floating-point representation is more flexible than the fixed-point representation in the sense that the radix point is not fixed in one position but is floating around depending on the magnitude of the number represented. A floating-point number x is of the form:

$$x = \pm m \times \beta^e, \text{ where}$$

m is called the significand[5] and represents the significant digits of the real number we want to represent. The significand has a certain number of digits p, called the precision.

β is the base[6] and depends on the number system we use. Commonly used bases are 2 (dual system), 10 (decimal system), 8 (octal system) and 16 (hexadecimal system).

e is called the exponent and corresponds to the power to which the base is to be raised prior to multiplying with the significand. The exponent pinpoints the radix point in its correct position and, therefor, corresponds to the number of digits the radix point has to be shifted to the left or to the right. The value of the exponent ranges from its minimum e_{min} to its maximum e_{max}.

Floating-point numbers are not unique. Consider the decimal fraction 0.5. We can represent 0.5 as 5.0×10^{-1} or 0.05×10^{1}. Another possibility is

[5]The term significand replaced the older term mantissa [20].

[6]The base is sometimes called radix.

	SN	**NEF**
normalized significand	5.0	0.5
unique floating-point representation	5.0×10^{-1}	0.5×10^{0}

Table 2.1: Unique representation of the fraction 0.5 in scientific notation (SN) and normalized exponential form (NEF).

0.000005×10^{5}, and there are many more. In order to overcome this ambiguity the significand of a floating-point number is normalized. There are two common ways to normalize the significand [32]:

1. **Scientific notation (SN):** In scientific notation the radix point is assumed to be located to the right of the most significant digit.

2. **Normalized exponential form (NEF):** If the normalized exponential form is used, then the radix point is placed to the left of the most significant digit.

Sticking to one of these conventions we get a unique floating-point representation. The unique representations for 0.5 in scientific and normalized exponential form are illustrated in Table 2.1.

Floating-point numbers are a subset of the reals. Their range depends on the minimum and maximum values of the exponent while their accuracy depends on the precision of the significand. As a consequence we can only approximate a given real number with a number of the chosen floating-point number system. Due to the fact that the radix point is floating around, a floating-point number system is more flexible than a fixed-point number system. Nevertheless there are similar representation problems as in fixed-point systems if we want to represent numbers that are out of the floating-point number system's range, i.e., e_{min} is too large or e_{max} is too small.

Now that we know what a floating-point number is, let us take a look on how to represent them on a computer system[7]. Basically three components are stored per floating-point number:

The sign: In general, one sign bit is used to determine the sign of the floating-point number. Generally, if the sign bit equals 0 then the corresponding number is positive. A sign bit set to 1 means that we deal with a negative number.

The exponent: A certain number of bits is used to store the exponent. The more bits we use for the exponent, the wider the range of representable

[7]The natural base β used for computers is 2.

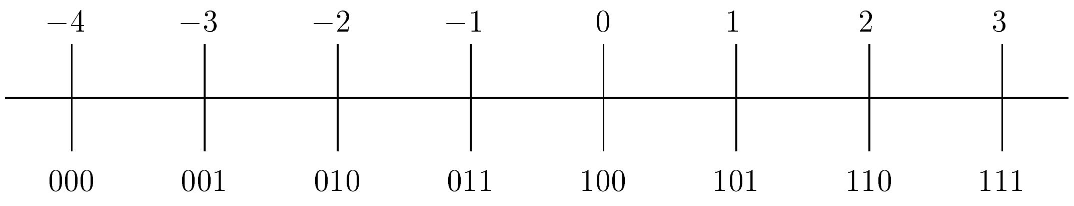

Figure 2.1: Sorted exponents using a three bit biased form.

numbers will be. Since the exponent can be negative, it is stored using either the two's compliment[8] or a biased form. The advantage of the biased form over the two's compliment is that the exponents are sorted from the smallest to the biggest value, see Figure 2.1. Assuming that the floating-point number is stored the usual way with the sign first followed by the exponent and the significand, all the bits of the floating-point format can be treated as a single number that can be sorted without determining its true value. This makes the biased form the representation of choice for exponents in most cases and we will therefor take a closer look at it. For more information on the two's compliment see [50, 38]. Storing the exponent e in biased form using m bits means that a so-called characteristic c is stored instead of e. As outlined in [32], the characteristic is computed by adding a bias b to the exponent:

$$c = e + b, \tag{2.1}$$

with

$$b = 2^{m-1}. \tag{2.2}$$

Given the characteristic of an exponent e one has to subtract the bias from the characteristic to reconstruct the original exponent.

The significand: The third component that has to be stored is the normalized significand[9], i.e., the significant digits. The more bits we use for the significand, the higher the precision of the floating-point number will be. Higher precision results in a better approximation of the desired real number. There is a trick called hidden bit to gain an extra bit of precision for the significand without actually storing it if base two is used. Since the first significant digit in binary form is always

[8]The two's compliment represents negative decimal values with a bigger binary number than positive decimal values.

[9]Note that the radix point is not stored but implied at certain position depending on the normalization.

N bits		
sign	exponent	significand
1 bit	m bits	$N-m-1$ bits

Table 2.2: Floating-point representation on a computer system.

a 1 we can imply the leading 1 and do not have to store it. E.g., if we want to store the significand 110010101 we actually store 10010101. When the stored significand is read back, we know that we did not store the leading 1 and therefor prepend it to get the original significand 110010101. Unfortunately, we are not able to represent 0 using the hidden bit because we always imply a hidden 1. In order to solve this problem a special value to represent 0 has to be defined.

Assuming that we have N bits to represent a single floating-point number and use one bit for the sign and m bits for the exponent that leaves us $N-m-1$ bits for the significand, see Table 2.2. It is easy to see that there is a trade-off of bits between the exponent and the significand. The more bits we use for the exponent the larger the range of representable floating-point numbers and the poorer the precision will be, and vice versa. Unlike fixed-point numbers, floating-point numbers are not distributed equally over their representable range. In fact the density of binary floating-point numbers halves at each power of two as we move farther away from zero, see [32, 23] .

2.2 Sources of Errors

We have seen in the previous section that floating-point representations have some advantages over fixed-point representations. These advantages and the fact that every modern computer system has a hardware floating-point unit, has lead to a widespread use of floating-point numbers to represent real numbers on a computer system. However, there are some drawbacks as mentioned above that may lead to irritating results of floating-point calculations in some cases. As we will see in the next chapter, this is especially true if we want to implement geometric algorithms. A software developer should therefor be aware of these problems to be able to deal with errors introduced by floating-point calculations. This subsection provides an overview on this subject. Excellent resources on this topic are [11, 20].

2.2.1 Representation Problems

As a matter of principle, the problems start at the moment a program reads its input data[10]. Although there are exceptions almost every computer system uses base two to represent floating-point numbers. Unfortunately, humans are used to the decimal system. Consequently, real numbers that are fed to algorithms as input are in decimal notation and have to be converted. This conversion introduces truncation errors since there are a lot of decimal real numbers that have no finite dual representation but an infinite periodical representation. Examples include 0.1 which is represented as $0.00011001100110011\ldots$ or 0.4. Similar problems occur if we wish to compute quantities that have no finite representation and have to be approximated. Depending on the accuracy we wish to achieve, we have to stop this approximation at one point introducing truncation errors again.

2.2.2 Data Uncertainty

Depending on the origin of the data we use as input there exist data uncertainties for different reasons. If the input data we use was measured, i.e, if we use physical quantities as input, errors occur due to measuring. Another possibility is to use input data that was generated by another computer program. Needless to say, computer-generated data suffers from all the possible error sources discussed in this section.

2.2.3 Roundoff Errors

In Subsection 2.1.3 we already mentioned that floating-point numbers cover only a finite subset of the reals. Consequently, results of arithmetic operations have to be rounded to the nearest floating-point number which introduces so-called roundoff errors. Let us take a look at an example that illustrates an roundoff error. To keep things simple we only consider positive numbers, choose base $\beta = 10$ and precision $p = 1$. We set the minimum exponent e_{min} to zero and the maximum exponent e_{max} to one. The numbers representable in this system are illustrated in Table 2.3. Let us see what happens if we calculate the sum $6 + 8$. The returned answer would be 10 and not 14 as one might expect. The reason for this is simple. Without a doubt, the correct result of the sum $6 + 8$ is 14. Unfortunately, we have no way for representing the number 14 in our floating-point number system. Therefor, the result is rounded to the next representable value which is 10. Similar problems arise if we want input the numbers 12 and 24 and want to compute their sum.

[10]Assuming that the input are real numbers.

Exponent	Representable Numbers								
$e = 0$	1	2	3	4	5	6	7	8	9
$e = 1$	10	20	30	40	50	60	70	80	90

Table 2.3: Representable floating-point numbers using $\beta = 10$, $p = 1$, $e_{min} = 0$ and $e_{max} = 1$.

Since we cannot represent the numbers 12 and 24, they are rounded to 10 and 20, respectively. Their sum yields 30 instead of the expected value 36.

2.2.4 Overflows and Underflows

Overflows occur if the exponent of a quantity computed grows too large. Referring to the floating-point number system illustrated in Table 2.3, the computation of $50 + 50$ yields an overflow error. The exact result of $50 + 50$ is 100 which can be rewritten as

$$100 = 1 \times 10^2.$$

Since $e_{max} = 1$ in the corresponding floating-point number system and the exponent required to represent the quantity 100 is 2, an overflow occurs.

While overflows occur if the exponent of a quantity computed grows too large, we get an underflow error if the exponent of a quantity is too small. The division $\frac{1}{2}$ yields 0.5 on a piece of paper. Performing the same calculation in the floating-point number system of Table 2.3 yields an underflow error because

$$0.5 = 5 \times 10^{-1}$$

and the smallest allowable exponent $e_{min} = 0$. In general, underflows are considered to be more harmful than overflows because they are mapped to zero in some floating-point implementations. This behavior could cause divisions by zero.

2.2.5 Problems Due to Floating-Point Errors

There are a number of problems that arise whenever floating-point arithmetic is used. These problems are the subject of this subsection. For more information on this topic see [11, 20, 23].

Insignificant Digits

The following C program is based on the FORTRAN code in [11] and demonstrates the phenomenon of insignificant digits.

```
#include <stdio.h>
int main()
{
        float x = 1000.2;
        float y = 1000.0;
        float z;

        z = x - y;
        printf("%f - %f = %f\n",x, y, z);

        return 0;
}
```

What we intended to do with this program was to compute the difference between the two real numbers 1000.2 and 1000.0. Clearly, the correct result is 0.2. When executed the program produces the following output:

$$1000.200012 - 1000.000000 = 0.200012.$$

If we would have printed the result only, we would have been rather surprised to see 0.200012 instead of the expected 0.2. But since the print statement prints the two operands as well as the result, we get an insight of what really happened. The real number 1000.2 cannot be represented exactly and is rounded to the closest representable floating-point number which is 1000.200012. On the other hand the second operand 1000.0 is represented exactly and therefor the result of the subtraction is 0.200012. It is easy to see in this example that the correct result is 0.20 and that the remaining digits are insignificant but if the calculations get more complex it might not necessarily be so obvious.

Inaccuracy Due to Conversions

Inaccuracies may also occur if a floating-point number is converted into an integer value. The following C code is again based on a FORTRAN program from [11] and demonstrates this conversion error.

```
#include <stdio.h>

int main()
```

```
{
        float x = 21.22;
        int z;
        z = x * 100.0;

        printf("z = %d\n", z);

        return 0;
}
```

The real number 21.22 has no exact representation and is represented as 21.2199.... Multiplying 21.2199... with 100.0 yields 2121.99... and is truncated to 2121 before it is assigned to the integer variable z. The program therefor prints the value 2121 instead of the expected value 2122.

Inaccuracies in Repeated Calculations

If a quantity that cannot be represented is involved in repeated calculations the error can grow and the result might not be the one expected. Take the real number 0.1 as an example. Adding 0.1 ten times yields 1.0. Unfortunately, there is no floating-point number corresponding to the real value 0.1, so it has to be rounded to the closest representable floating-point number. Consequently, adding this approximate value ten times does not result in 1.0. The error is illustrated by the following program.

```
#include <stdio.h>

int main()
{
        double x = 0.0;
        int counter;

        for(counter = 0; counter < 10; counter++)
           x = x + 0.1;

        if (x == 1.0)
           printf("x == 1.0\n");
        else
           printf("x! = 1.0\n");

        return 0;
}
```

When executed the program outputs

$$x! = 1.0$$

since 1.0 is represented exactly while 0.1 is not. The combinatorial part of geometric algorithms is often constructed based on comparisons of computed numerical quantities. Inaccuracies like these lead to errors in the combinatorial structure and are a major problem for software developers.

Cancellation

Catastrophic cancellation occurs whenever we subtract two quantities of similar magnitude or alternatively add such quantities having opposite signs. Suppose we are given two quantities with precision p[11] of similar magnitude and wish to subtract them. What happens is that the majority of significant digits cancel themselves leaving some of the less significant digits that may already have suffered from rounding errors. In the worst case this leaves us with a result where not a single digit is correct and the error is of the magnitude of the quantity itself, if it was exactly computed. While the arithmetic operation, i.e., the subtraction is not the problem[12] in this case, it uncovers errors that have already occurred. The phenomenon described above is known as catastrophic cancellation and can lead to very inaccurate results. Nevertheless there are also other forms of cancellation that are benign or that can even be beneficial whenever they cancel errors that occurred in previous calculations. For more details on catastrophic, benign and beneficial cancellation see [20, 23].

2.3 A Few Words on Error Analysis

We saw in the previous sections that there is no way to avoid errors using standard floating-point arithmetic. This leaves us with the question how accurate the results computed really are? The field of numerical analysis is concerned with this question and has developed a number of methods to analyze and quantify errors. Since a thorough study of error analysis is far out of the scope of this thesis, this section just provides a survey on this topic. More information can be found in [23] and the papers cited therein.

2.3.1 Introduction to Error Analysis

Since the intension of this subsection is to take a glimpse at error analysis we will look at two common methods of error analysis and keep things as simple

[11] That is, we have p significant digits.

[12] In fact this operation could even be exact.

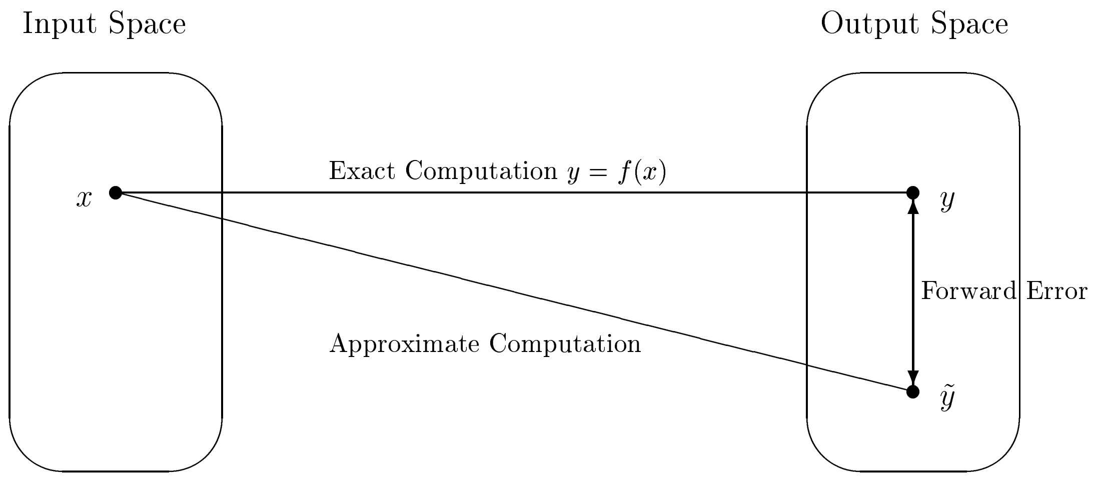

Figure 2.2: Forward error.

as possible. Before we start, let us assume that we compute an approximate value, $\tilde{y}$, for the exact value y. The exact value y is computed by a function $f(x)$, see [23]. Summarizing we have:

$$y = f(x), \text{ where}$$

x is the input value; the input value is a real number and is a member of the so-called input space;

y is the result of the function $f(x)$; the value y is a real number and is a member of the output space;

$f(x)$ is an arbitrary function that computes exact answers.

If we were able to perform exact arithmetic operations we would take a real number from the input space and calculate the exact result which is a real number again. Operating with floating-point arithmetic the operation is performed with floating-point numbers. Therefor, errors can occur in the input space – recall Subsection 2.2.1 – as well as in the output space. Consequently, we have two ways of looking at an error which is reflected in two different methods of error analysis called forward and backward error analysis.

2.3.2 Forward Error Analysis

The method of forward error analysis, illustrated in Figure 2.2, tries to quantify the error in output space. That is, the exact result y of the function $f(x)$

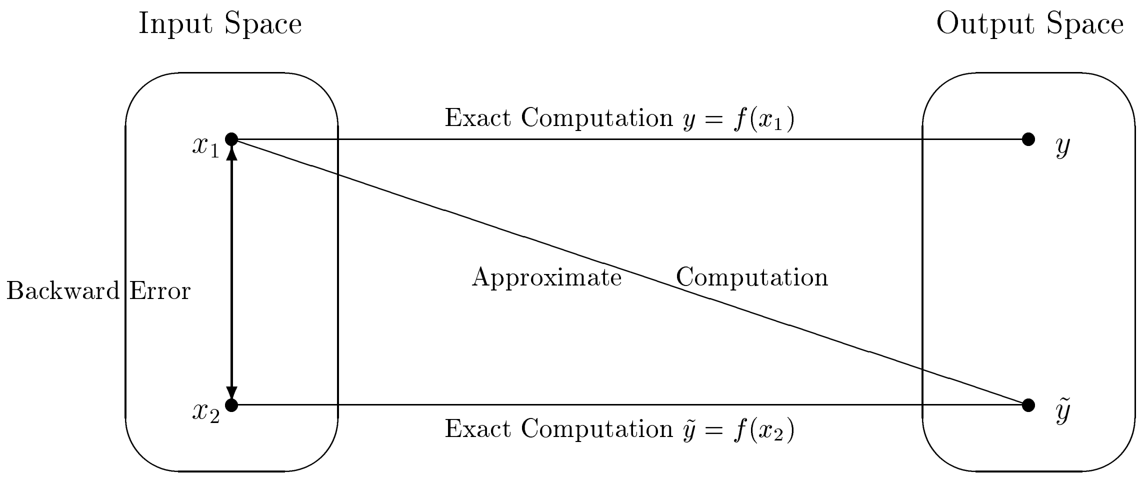

Figure 2.3: Backward error.

is compared to the result $\tilde{y}$ computed in the floating-point number system. Two common ways are used to calculate the forward error. The absolute error $FE_{abs}(\tilde{y})$ is calculated using the following formula

$$FE_{abs}(\tilde{y}) = |y - \tilde{y}|. \tag{2.3}$$

One drawback of absolute errors is that they change if y and $\tilde{y}$ are scaled. Scaling y and $\tilde{y}$ by a factor of α yields an absolute error scaled by α itself. This behavior can be avoided by calculating the relative error $FE_{rel}(\tilde{y})$ which we get by dividing the absolute error $FE_{abs}(\tilde{y})$ by $|y|$ as shown in equation 2.4:

$$FE_{rel}(\tilde{y}) = \frac{|y - \tilde{y}|}{|y|}. \tag{2.4}$$

Scaling y and $\tilde{y}$ does not have any effect on the relative error which is the reason why relative errors are preferred; they make it easy to compare errors of computed quantities that vary a lot in their magnitude.

2.3.3 Backward Error Analysis

Another way of quantifying the error is to check what input data corresponds to the approximated result if the calculation was exact using the function $f(x)$. Recall that $\tilde{y}$ is an approximation to $y = f(x_1)$. What we are looking for is some value x_2 of the input space such that

$$\tilde{y} = f(x_2).$$

Similar to the forward error, the backward error can be expressed in absolute terms as the absolute value of the difference between x_1 and x_2

$$BE_{abs}(\tilde{y}) = |x_1 - x_2|, \tag{2.5}$$

or in relative terms dividing the absolute error by the absolute value of x_1 as expressed by the following equation

$$BE_{rel}(\tilde{y}) = \frac{|x_1 - x_2|}{|x_1|}. \tag{2.6}$$

The backward error, which is illustrated in Figure 2.3, is interesting because errors are interpreted as perturbations in the input data. In general, input data is uncertain due to errors in measuring, storing or previous computations. If the backward error is smaller than the uncertainties in the input data, the computed result is accurate enough. Backward errors are also connected to perturbation theory which is discussed in the next chapter.

2.4 The IEEE-754 Standard

Before the standardization of floating-point formats, the porting of programs from one computer system to another was very cumbersome. Differences between the format of floating-point numbers and their corresponding operations yielded different results on different architectures. When errors occurred in numerical results, it was not clear if the error was due to rounding and truncation or due to a different implementation of floating-point numbers and their corresponding operations. The IEEE standard defines the representation of floating-point numbers as well as the basic operations, making it easy to port programs from one IEEE machine to another. The IEEE standard does not solve the problem of truncation and rounding errors but, at least, they are the same on every architecture that implements the standard. Last but not least the standard simplifies proofs concerning floating-point numbers since one has a reference implementation and does not have to bother with differences in floating-point representations. The IEEE has defined two floating-point representation standards. The IEEE-754 standard [4] defines the representation of floating-point numbers with base $\beta = 2$. The majority of floating-point implementations on computer systems use this standard. A second floating-point standard that was proposed by the IEEE is called IEEE-854 and requires either $\beta = 2$ or $\beta = 10$. We will focus on the IEEE-754 standard. For more information on IEEE-854, see [20].

32 bits

sign	exponent	significand
bit 31	bits 30 . . . 23	bits 22 . . . 0

Table 2.4: IEEE-754 single precision floating-point number.

2.4.1 Storage Format

Four different precisions are proposed in the IEEE-754 standard:

1. Single Precision,
2. Double Precision,
3. Single Extended Precision, and
4. Double Extended Precision.

The normalized exponential form is used to normalize IEEE-754 floating-point numbers, so the radix point is assumed to be left of the first significant digit. Furthermore, the exponent is stored in a biased form, recall Page 10.

Single Precision

Single precision floating-point numbers occupy 32 bits. One bit is used to indicate the sign of the floating-point number. The exponent is stored using 8 bits, leaving 23 bits for the significand. Since IEEE-754 requires $\beta = 2$ the concept of the hidden bit is used to achieve 24 bits of precision. The exact format of a IEEE single precision floating-point number is shown in Table 2.4. The IEEE-754 standard defines several special values for a single precision floating-point number, with exponent in biased form:

Zero: If all the bits of the exponent as well as all the bits of the significand are set to zero, then the value of the floating-point number is defined to be 0[13]. Note that there are actually two representations for zero (+0 and −0) depending on the value of the sign bit.

Infinity: If all the bits of the exponent are set to one and all the bits of the significand are set to zero then the value of the floating-point number is infinite. Depending on the sign bit $+infinity$ and $-infinity$ is represented.

[13]Remember that a special representation for zero is needed since the IEEE-754 numbers use the hidden bit.

64 bits		
sign	exponent	significand
bit 63	bits 62 . . . 52	bits 51 . . . 0

Table 2.5: IEEE-754 double precision floating-point number.

NaN (Not a Number): NaN is represented setting all the exponent's bits to one and not all the bits of the significand to zero. That is, the decimal value of the exponent field is 255 and the decimal value of the significand is non-zero.

Denormalized Number: A value is denormalized if the exponent is zero but the significand is not. The leading one is no longer assumed in this situation. A single precision denormalized floating-point number is therefor represented as $\pm 0.m \times 2^e$.

Double Precision

The standard requires to encode double precision floating-point numbers using 64 bits. Similar to single precision, one bit is used for the sign. The exponent is encoded using 11 bits. The final 52 bits are occupied by the significand that actually has 53 bits of precision due to the hidden bit. The exact format of double precision IEEE-754 floating-point numbers is shown in Table 2.5.

The special values for double precision floating-point numbers are defined as follows:

Zero: The value 0 is represented by setting all of the exponent's bits as well as all the significand's bits to 0, i.e., the decimal value of the exponent and the significand is 0. Depending on the sign bit either $+0$ or -0 is represented.

Infinity: Positive or negative infinity is represented setting the sign bit to 0 or 1, respectively. The exponent field has the decimal value 2047 and therefor all the bits set to 1 while the significand's bits are all set to 0.

NaN (Not a Number): If all the bits of the exponent are set and the decimal value of the significand is non-zero then the value of the corresponding floating-point number is NaN.

Denormalized Number: Similar to single precision a value is denormalized if the exponent is zero but the significand is not. Since the leading one is no longer assumed the value represented by a double precision denormalized floating-point number is $\pm 0.m \times 2^e$.

Single Extended and Double Extended Precision

Single extended and double extended precision is used whenever there is a need for higher precision. This is especially true for intermediate results. Potential problems due to intermediate overflow and underflow or cancellation can be softened if the results are in single or double precision while intermediate results are calculated using extended precision. Nevertheless a phenomenon called double rounding might occur if extended precision is used for intermediate results. Double rounding is permitted in the IEEE-754 standard and means that a result of an operation is first rounded to extended precision with subsequent rounding to the target format, i.e., single or double precision. Since it depends on the IEEE implementation if results are double rounded or rounded directly to the target format, different implementations might yield slightly different results. The size of the single extended and double extended format is not specified exactly but there are lower bounds. Table 2.6 summerizes the four precisions defined in the IEEE-754 standard.

2.4.2 Operations

The results of additions, subtractions, multiplications, divisions and square roots are well defined in the IEEE-754 standard. Floating-point implementations have to guarantee these results to be compliant with the standard. IEEE requires operations to be carried out as if they were computed to infinite-precision and then by default rounded to the next representable floating-point number. Rounding to $\pm$ infinity is also supported which facilitates interval arithmetic. Every operation yields a defined result, exceptional operations like divisions by zero or overflows raise signals. Operations involving infinity are defined according to the well known mathematical conventions:

- $\infty + \infty = \infty$,
- $(-1) \times \infty = -\infty$,
- $\frac{x}{\infty} = 0$, with x being a finite representable floating-point number.

A NaN is generated for invalid operations such as:

- $0/0$,
- $\sqrt{x}$ for $x < 0$,
- $0 \times \infty$,
- $\frac{\infty}{\infty}$,
- $\infty - \infty$.

Operations that involve a NaN result in NaN. NaN can also be used to indicate that a variable has not been initialized yet. Underflows are not flushed to zero but treated as denormalized numbers. The advantage of this behavior called gradual underflow is that divisions by zero due to underflows are prevented. As mentioned before there are two different representations of zero depending on the sign bit. Nevertheless it is reasonable to treat them as a single value in arithmetic operations. Therefor, $-0 = +0$ is required by the standard.

	Single Prec.	Double Prec.	Single Ext. Prec.	Double Ext. Prec.
Format size in bits	32	64	≥ 43	≥ 79
Size of sign in bits	1	1	1	1
Size of exponent field in bits	8 bit	11 bit	≥ 11 bit	≥ 15 bit
Size of significand field in bits	24 bit	53 bit	≥ 32	≥ 64
Hidden bit	yes	yes	no	no
Bias	$+127$	$+1023$	unspecified	unspecified
e_{min}	-126	-1022	≤ -1022	≤ -16382
e_{max}	$+127$	$+1023$	$\geq +1023$	$\geq +16383$

Table 2.6: Summary of the IEEE-754 standard.

Chapter 3

Non-robustness and the Problems That Arise in Geometric Algorithms

This chapter is concerned with robustness issues in the field of computational geometry. In Section 3.1 we will discuss the phenomenon of non-robustness in geometric algorithms. Section 3.2 focuses on robustness problems in connection with floating-point arithmetic. A possible solution to robustness problems due to floating-point arithmetic is provided by the exact geometric computation paradigm, which is discussed in Section 3.3. Concluding this chapter, Section 3.4 is dedicated to the topic of degeneracies.

3.1 Introduction

The field of computational geometry and with it the research on geometric algorithms is relatively young, see the book "Computational Geometry - An Introduction" by Preparata and Shamos [37] for a good introduction. In the beginning researchers concentrated on the theory of geometric algorithms. As research evolved, the need for robust implementations of the excogitated, theoretically correct algorithms grew. It turned out that the robust implementation of an algorithm that is correct in theory is not such an easy task. Robustness became a major issue in the last few years and researchers all over the world are taking great efforts to solve the robustness problem. The reason for the gap between theory and practice is twofold.

1. Theory requires a so-called real RAM. That is, algorithms are developed to run on a conceptional machine that operates with real numbers. A single operation is carried out in constant time.

2. Furthermore, a so-called "general position" assumption is made, excluding all degenerate cases.

It is easy to see that both assumptions do not hold in practice. Practical implementations often use floating-point numbers to represent real numbers, since they are usually supported by the computer system's hardware and are therefor very fast. We already saw in Chapter 2 that there are a lot of potential problems connected with floating-point arithmetic.

The assumption of a "general position" is convenient to keep correctness proofs simple. One can concentrate on the general solution of a geometric problem and does not have to handle every special case. Applications, on the other hand, have to deal with real-world data, computer-generated data or interactive input. Degenerated input is sure to occur from time to time. Take the intersection of two line segments as an example. "General position" reduces this problem to two different cases:

1. The two line segments do not intersect, or

2. the two line segments intersect in one point interior to both segments.

Clearly there might be special cases where the segments intersect at their endpoints or partly overlap, yielding an intersection interval. A software developer has to handle every possible special case to implement a robust algorithm. Needless to say, this is a difficult task, especially if the geometric objects involved get more complex or one moves to higher dimensions.

3.1.1 Why are Geometric Algorithms so Difficult to Implement

Without a doubt, robustness and errors due to floating-point arithmetic are an issue no matter what kind of algorithm a software developer wants to implement. Nevertheless there are a lot of areas in computer science where floating-point arithmetic suffices and thus the implementation of algorithms is straightforward. On the other hand, it seems that the implementation of geometric algorithms is very difficult even if the given geometric problem is quite simple. The reason for this is that geometric objects consist of both combinatorial and numerical data. Combinatorial data like face and boundary descriptions or adjacencies is based on numerical data like vertex coordinates and plane equations. Due to the inexactness of floating-point arithmetic the numerical data is only approximate, which may lead to contradictions with the combinatorial data. A possible situation is illustrated in Figure 3.1. The combinatorial data requires the three line segments to

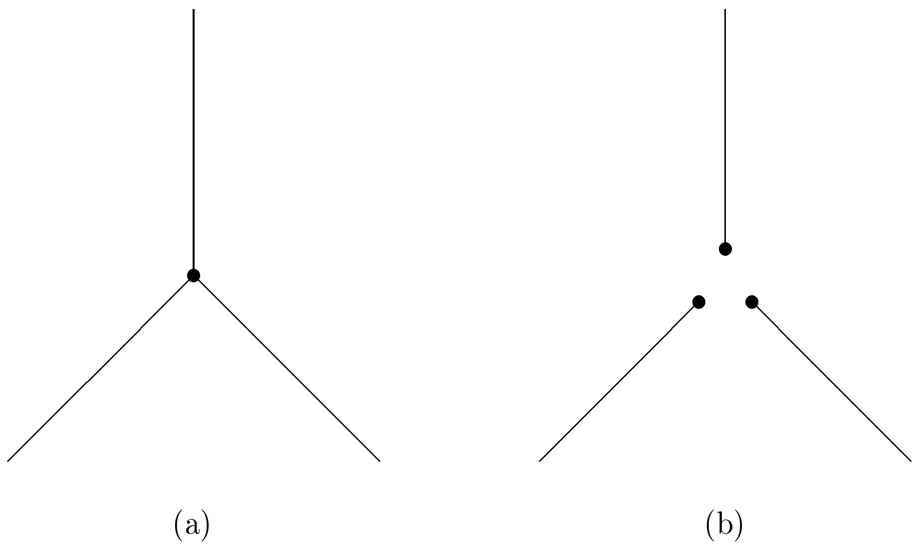

Figure 3.1: The intersection of three line segments. The information we get from the combinatorial data is illustrated in (a), while the situation from the numerical data's point of view is illustrated in (b).

meet in a single point contradicting the information we get from the numerical data, that specifies three nearby points instead of just a single one. Since floating-point arithmetic is inaccurate, the information we get from the numerical data is incomplete. Thus, there is an uncertainty inherent to all decisions that are based upon numerical data, which might lead to combinatorial inconsistency. Redundant decisions made by the algorithm might contradict each other and therefor have to be avoided, see Subsection 3.2.5. The problem gets even worse in cascaded computations where the output from one algorithm is used as input for a second one. The second algorithm is likely to crash if the output produced by the first algorithm is corrupt.

Besides the problems described above, the complexity of solutions of even simple geometric problems is another issue. As already mentioned above, even the intersection of two line segments has a number of special cases that have to be dealt with. It is up to the software developer to handle every single special case that can arise during the program's execution. This is quite a challenging task for complex geometric problems or higher dimensions and not even exact arithmetic can help with that.

In the following sections we will survey some of the techniques developed to cope with the robustness problem in geometric algorithms. We start with the different types of geometric primitives. Excellent introductions on the robustness topic are [41, 17, 24, 27, 40].

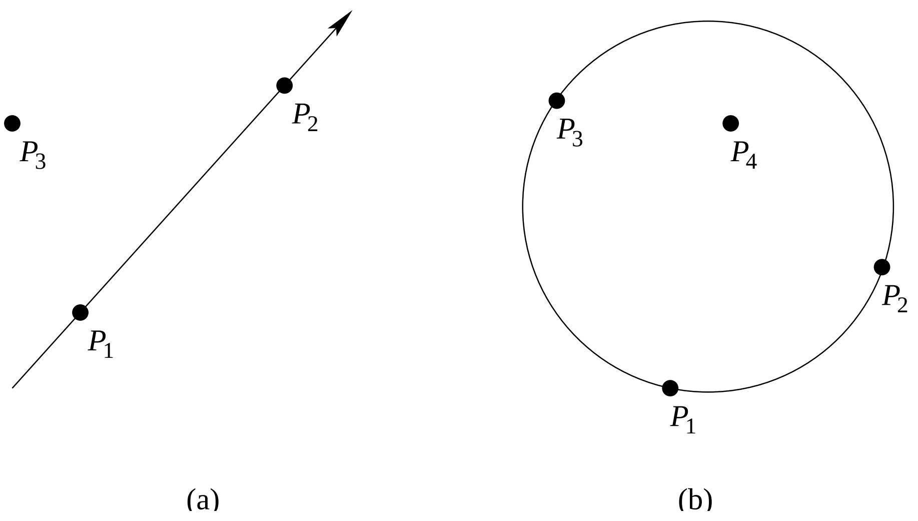

Figure 3.2: The 2D orientation test (a): Is P_3 located on the left, on the right or on the oriented line through P_1 and P_2? And the 2D incircle test (b): Is P_4 inside, outside or on the circle through P_1, P_2 and P_3?

3.1.2 Predicates and Constructors

A basic operation in a geometric algorithm is called a geometric primitive. There are two different types of geometric primitives [18] involved in geometric algorithms. The first type is called a predicate. Predicates are used to make decisions in order to determine the combinatorial structure of a geometric output. Examples of predicates are orientation tests and the incircle test, see Figure 3.2. Decisions made with predicates are usually based on the sign of some arithmetic expression like the determinant of a matrix. If the magnitude of the expression computed is smaller than the rounding error, the sign evaluation might be incorrect. Since the given input data is likely to be inexact itself, for reasons mentioned in Chapter 2, the computed sign is correct for some perturbation of the input data for a single predicate. Problems arise if predicates are repeatedly incorrect because there might not be a global perturbation that satisfies all the incorrect results, thus leading to a corruption of the combinatorial structure. There are some geometric algorithms including the computation of convex hulls and triangulations that solely rely on predicates. Those algorithms can be made robust if one can guarantee that the signs of the predicates computed are correct, or, at the very least consistent.

The second type of calculation involved in geometric algorithms is called a constructor because it is used to construct new geometric objects. Constructors compute the numerical part of a geometric output. An example is the calculation of an intersection point of two line segments. Guaranteeing robustness of geometric algorithms that use both predicates and constructors, e.g., in the case of computing Voronoi diagrams, is a much harder task than for algorithms that rely on predicates only.

3.2 Non-robustness due to Floating-Point Arithmetic

If a software developer decides to implement an algorithm based on floating-point arithmetic, his/her decision was probably guided by the need for fast arithmetic operations. Without a doubt there is no faster way to perform the basic arithmetic operations than with the hardware floating-point unit of the target platform. The drawback of this decision is that he/she has to live with all the error-prone calculations that occur in the world of floating-point arithmetic. Especially if the implemented algorithm relies on constructors as well as on predicates, exact results cannot be expected. Nevertheless a reasonable result has to be computed for correct input data. Algorithms that do not compute the exact result, but a result that is reasonably close to the exact result are called robust algorithms, see [16] and [41].

Ensuring robustness using floating-point arithmetic is a very difficult task. Up till now, no general technique has been introduced to solve this problem. Nevertheless there is a collection of techniques and guidelines that has been developed by researchers in recent years. Since there is no general rule and since the effectiveness of these techniques depends on the type of algorithm one wants to implement, the main challenge is to pick the right technique and adopt it appropriately for the specific needs.

3.2.1 Epsilon Tweaking

One of the most common methods used to increase robustness of an algorithm is referred to as epsilon tweaking. Following the convention in [40] we will assume that the comparison of numerical values in predicates is a comparison of the value of some arithmetic expression with zero. Whenever a floating-point value is used for a comparison there is an uncertainty associated with it. Therefor, it is common practice to code an algorithm according to the rule [40]:

If some numerical value is close to zero it is considered to be zero.

Predicates that are implemented this way do not compare a value computed to zero but to a small constant ϵ instead. A value x is considered to be zero if

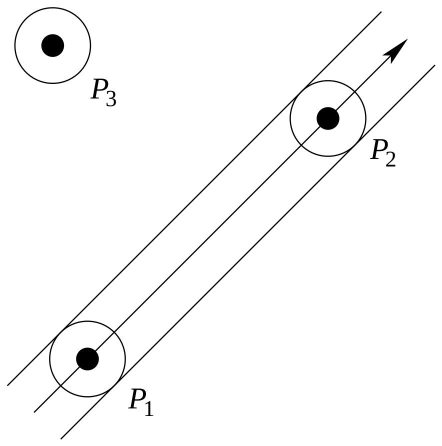

Figure 3.3: Epsilon tweaking in orientation tests.

$$|x| \leq \epsilon, \text{ with } \epsilon > 0.$$

One possible choice for ϵ is the so-called machine precision, which is the smallest number δ such that $(1 + \delta) > \delta$ evaluates to true on the machine's particular floating-point unit. Nevertheless there is no general rule on how to choose an appropriate ϵ. One single ϵ might be used for all predicates in a geometric algorithm or several ϵ's are defined for different predicates. The value of ϵ is arbitrary but should be fairly small and greater than zero. Depending on the application and the input data tested a software developer normally adjusts an appropriate value for ϵ by trial and error.

In [22], Held uses an approach which he called relaxation of epsilon thresholds: The user is asked to specify an upper bound on ϵ. The lower bound is given by the machine precision which is used as the initial value for ϵ. If the computation fails, e.g., because some "sanity check" is not passed, then the value for ϵ is increased and the computation is restarted. If, for some reason, the upper bound on ϵ is reached, then a soundness check of the input data is performed.

Epsilon tweaking is justified by the already mentioned fact that the input data is not exact and we are computing an answer for a perturbed instance of the input data. Consequently a small perturbation of the input suffices for an expression to evaluate to zero. The geometric interpretation is that the objects we are dealing with are fattened due to epsilon tweaking. Figure 3.3 shows the geometric interpretation of epsilon tweaking in an orientation test.

There are some drawbacks if epsilon tweaking is used. First of all we have already mentioned that there is no general rule on how to pick the value for ϵ. Finding an appropriate value can be a time-consuming task.

Since this value is found by trial and error based on the algorithm and the input data tested, there is no guarantee that the algorithm is correct for every input instance it is applied to. Furthermore, the equality relation looses its transitivity property, and the same is true for collinearity, see [40]. From a geometric point of view we have left the Euclidean geometry as soon as we use epsilon geometry and resort to some yet to be determined geometry. Guibas et al. [28] introduced an approach which they called epsilon geometry. They defined some basic properties of fattened geometric objects in the plane and constructed some basic geometric predicates like the 2D collinearity and orientation test, the coincidence test for two points in the plane and point inclusion tests for triangles and convex polygons. They called these predicates epsilon predicates. It appears that this approach has not been taken any further and so only a small set of predicates have been defined this way. Besides there has not been a generalization to three dimensions.

3.2.2 Interval Arithmetic

Another approach to increase robustness of geometric algorithms in the finite-precision world is called interval arithmetic [27, 7]. Using interval arithmetic, each number x is stored as an interval $[x_l, x_u]$[1] containing x and bounded by the floating-point numbers x_l and x_u. Arithmetic operations are defined on the intervals of the corresponding numbers. In [7] the basic arithmetic operations of two numbers $[x]$ and $[y]$ are defined as follows:

$$[x] + [y] = [x_l + y_l, x_u + y_u],$$

$$[x] - [y] = [x_l - y_u, x_u - y_l],$$

$$[x] * [y] = \begin{array}{l} [min\{x_l * y_l, x_l * y_u, x_u * y_l, x_u * y_u\}, \\ max\{x_l * y_l, x_l * y_u, x_u * y_l, x_u * y_u\}], \end{array}$$

$$\frac{[x]}{[y]} = \begin{cases} [x] * [\frac{1}{y_u}, \frac{1}{y_l}] & \text{if } 0 \notin [y], \\ ]-\infty, +\infty[& \text{otherwise}, \end{cases}$$

$$\sqrt{[x]} = \begin{cases} [\sqrt{x_l}, \sqrt{x_u}] & \text{if } 0 \notin [x], \\ ]-\infty, +\infty[& \text{otherwise}. \end{cases}$$

A drawback of interval arithmetic is that the resulting intervals can grow very large in the course of computation. Furthermore, correct rounding of the interval bounds is necessary in order to ensure that the final interval still contains the correct answer. For more information on interval arithmetic see [27, 7] and the papers cited therein.

[1] We denote a number x stored as an interval by $[x]$.

3.2.3 Careful Programming

No matter which geometric algorithm a software developer intends to implement, he/she has to do this very carefully. There are some basic guidelines [35] one has to keep in mind in order to implement a robust algorithm:

- The sequence of numerical calculations does have an impact on the magnitude of the error. If two different sequences of operations are used to compute a single value, the results might differ, although they should be mathematically equal. Furthermore, it is often possible to compute a single quantity using different conceptional methods or formulas. Although logically and mathematically equal, the results might be different. It is therefor essential to always compute a numerical quantity the same way, i.e., with the same formula.

- Computed or derived quantities have already suffered from roundoff errors, so do use the original input data instead, wherever this is possible.

- Interchanging the input values in a formula may yield different results. Assigning each input quantity a unique index and using them in the same order, e.g., increasing order, each time the formula is used will eliminate this problem.

- Handle cases that might cause problems during computation as special cases. E.g., if the intersection of a vertical line and an oblique one has to be computed, assign the abscissa with the vertical line and do not compute it.

Held designed a triangulation algorithm [21] based on some of the techniques described above. In addition, he implemented a so-called multilevel-recovery system, which we will discuss in a subsequent chapter. The algorithm is fast and to our knowledge has not crashed yet.

3.2.4 Designing Robust Geometric Primitives

One important prerequisite for implementing robust geometric algorithms is the robust implementation of the underlying geometric primitives. Spending some time thinking of tricks to increase the robustness of geometric primitives is unavoidable. As an example we will now take a closer look at the benefits of translating geometric entities.

As we already know from Chapter 2, precision decreases if the magnitude of the input quantities grows larger, because more bits of the significand are used for the integer part of the quantity. In [41] Shewchuk points out that, in general,

1. the absolute coordinates, i.e., the distance to the origin of a geometric entity are much bigger than their relative coordinates, i.e., the distances of the geometric entity's defining elements from each other, and

2. that the result of many geometric calculations is independent of the geometric entity's position in the coordinate system. That is, the result is translation-invariant.

Together, both observations can be used to gain some extra precision for numerical calculations. If a calculation satisfies the observations stated above, then accuracy can be increased by translating the geometric primitive such that one of its defining elements is located at the origin. Shewchuk illustrates this with the well known formula for the area of a polygon. Assuming that we are given a polygon with n points[2] $p_1 \ldots p_n$ in the plane, where $p_i = (x_i, y_i)$, the standard formula

$$\frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - y_i x_{i+1})\right| \tag{3.1}$$

can yield quite inaccurate results if the dimension of the polygon is small compared to its distance from the origin. Translating p_n to the origin by replacing each p_i with $p_i - p_n$ results in the improved formula

$$\frac{1}{2}\left|\sum_{i=1}^{n}[(x_i - x_n)(y_{i+1} - y_n) - (y_i - y_n)(x_{i+1} - x_n)]\right|. \tag{3.2}$$

In general, robustness does not come for free. There is always a trade-off between robustness and speed. The loss of speed is not that dramatic in the example above, nevertheless, it is apparent that the calculation of Formula 3.2 requires more floating-point operations than the calculation of Formula 3.1. One has to keep in mind that the translation of a geometric entity to the origin might also introduce roundoff errors. Furthermore, there are geometric calculations that are not translation-invariant. For more information see [41].

3.2.5 The Topology-Oriented Approach

A common problem that causes a corrupt combinatorial structure is inconsistency of the predicates used. Even if errors in numerical calculations occur that may lead to incorrect judgments, the algorithm could still produce a reasonable output. As stated before this output may not be the exact solution but should be close to it. So it is not a matter of correct or incorrect judgments. The important thing is that the judgments are consistent. Predicates that contradict each other are likely to fail to produce a reasonable

[2]All indices taken modulo n.

output. If, on the other hand, both of them take the same decision, the algorithm's output should still be consistent. One way to achieve this goal is to emphasize the combinatorial data of geometric entities and use the numerical data as secondary information only. This technique is called the topology-oriented approach and was introduced by Sugihara and Iri [43] in 1988. The advantage of the topology-oriented approach is that inconsistencies in the geometric output are prevented, degeneracies do not have to be treated explicitly and that the solution computed converges[3] to the exact solution if the precision of the numerical operations is increased. Thus, the topology-oriented approach seems to be very successful in designing robust algorithms especially if it is combined with careful implemented geometric primitives. Since its introduction, the topology-oriented approach has been applied to a number of geometric problems like the three-dimensional Delaunay triangulation [12, 42] and Voronoi diagrams [44]. Held designed an algorithm for computing the Voronoi diagram [22] of a set of points and line segments in the plane based on [44].

We will now take a closer look on how to design topology-oriented algorithms. The intension of the topology-oriented approach is to design an algorithm that has the following properties:

Robust: The algorithm should never end in an endless loop, terminate abnormally or crash. Furthermore, it should always compute an output.

Consistent: The topological part of the output should satisfy some predefined topological properties. Note that this does not mean that the produced output is equal to the exact solution. If the precision of the arithmetic operations is increased, the output computed converges to the exact solution for non-degenerate input.

Easy to design: The design of the algorithm should be "easy" in the sense that the design is independent from error analysis, i.e., error bounds, thus separating the topological consistency issue from the numerical error issue.

Easy to implement: There is no need to handle degeneracies in a special way.

The topology-oriented approach is based on the following assumptions, see [45]:

1. Logical and combinatorial computations can be done correctly.

2. Numerical computations are, in general, erroneous.

[3]Assuming that the input is not degenerate.

3. No a-priori error bound is available.

Assumption 1 ensures the correctness of the implementation of a topological algorithm. Assumptions 2 and 3 allow an algorithm design that is independent from error analysis and free from handling any degenerate situation. Based on these assumptions a topology-oriented algorithm can be designed in three steps, see [45].

Step 1: Identify a set of purely combinatorial properties that should be satisfied by the solutions of the geometric problem. Make sure that these properties can be checked efficiently.

It is important to stress that this step is crucial for the correctness of the algorithm. In a subsequent step the so-called topological skeleton is designed in a way such that the topological properties specified in Step 1 are guaranteed in the algorithm's output. If, however, these combinatorial properties are not chosen carefully, the algorithm might not compute a reasonable solution to the geometric problem. Furthermore, it should be noted that the set of combinatorial properties is, in general, only a necessary condition to insure topological consistency. Since a purely topological characterization is not known for every set of solutions to a geometric problem, a sufficient condition is often hard to find. Another limitation is the computational cost. Note that Step 1 requires the combinatorial properties to be checked efficiently. Thus, it is not possible to take a topological property into account, if it cannot be checked with acceptable computational cost. Once the set of topological properties is chosen, the actual algorithm can be designed:

Step 2: Construct the topological skeleton by describing the basic part of the algorithm in purely combinatorial and topological terms such that the combinatorial properties chosen in Step 1 are guaranteed.

Note that there is no need to handle degeneracy, since the design of the topological skeleton is not based on any numerical operation. According to Assumption 2, numerical computations are inaccurate anyway, thus we cannot even detect reliably whether a degeneracy occurs. The topological skeleton specifies every possible behavior of the algorithm, regardless of the precision that is used for numerical operations. The algorithm is therefor robust in the sense that it always terminates and produces an output. A carefully chosen set of combinatorial properties and Assumption 1 ensure topological consistency of the output computed. Unfortunately, there are non-deterministic branches in the topological skeleton, see also [45]. This situation is shown in Figure 3.4. The tree structure describes all the possible branches of the topological algorithm. The algorithm starts at the root and takes a non-deterministic branch at every node until it ends up in one of the

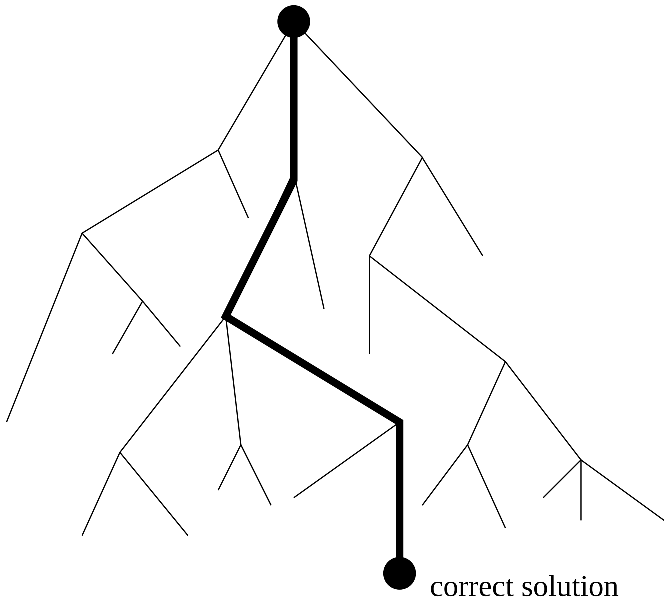

Figure 3.4: Tree structure illustrating the non-deterministic branches of a topological algorithm.

leaves. The combinatorial properties chosen in Step 1 are guaranteed at each node of the tree but only one path corresponds to the true solution of the geometric problem we wish to solve. In order to find the desired path, the final step in the design of a topology-oriented algorithm relies on numerical computations:

Step 3: Perform numerical computations at each node of the tree to choose the branch that will likely lead to the correct solution of the geometric problem.

This last step ensures that the designed algorithm is deterministic. If all the judgments based on numerical computations are correct and there is no degeneracy, the algorithm computes the exact answer to the geometric problem. Without degeneracy the solution computed converges to the exact solution with increasing precision of the numerical operations. For degenerate input, the algorithm computes an output that converges to an infinitesimally perturbed version of the correct answer.

We have already mentioned that robustness does not come for free. Generally, there is a trade-off between robustness and speed. Since the precision of numerical operations does not affect the robustness[4] of topology-oriented

[4]The precision of numerical operations does, of course, affect the quality of the output computed.

algorithms they can be implemented using standard floating-point arithmetic. Nevertheless there is a slow-down due to checking the combinatorial properties. Degenerate input data often introduce complicated microstructures, see [45], thus there might be additional computational cost. It seems though, that carefully designed topological algorithms, especially if they are combined with tuned primitive operations are both reliable and fast and yield good results in practice, see [22].

3.3 Exact Arithmetic

As we saw in the previous section there are a couple of drawbacks and pitfalls if one decides to implement a geometric algorithm using floating-point arithmetic or any other finite-precision arithmetic. We have discussed methods to cope with these drawbacks and guidelines on how to implement a robust algorithm. Unfortunately, we also saw that there is no general rule to achieve robustness. Different geometric problems require different methods and it is difficult to choose and implement the right techniques for a specific algorithm. These problems motivated researchers to look for an alternative number representation that would allow a straightforward implementation of the theoretically developed algorithms. In order to achieve this goal it is necessary to mimic the real RAM model. The arithmetic model where every numerical quantity is computed exactly and that is closest to the real RAM model is called exact arithmetic. It has to be noted, though, that exact arithmetic is defined on a subset of the reals, e.g., integers or rationals only. Furthermore, it is based on the assumption that the input data can be represented exactly.

Using exact arithmetic, it is possible to compute every numerical quantity to arbitrary precision. Naturally, there is a trade-off between precision and speed. Higher precision results in higher computational cost. Thus, there is a significant slow-down compared to finite-precision arithmetic.

3.3.1 Big Number Packages

A prerequisite to exact arithmetic is an arbitrary precision representation of numerical quantities. Usually multi-precision integers, implemented in so-called big number packages, are used achieve this goal. Overflows in integer operations are prevented if big number packages are used, since they support the representation of arbitrary integers. Due to the fact that the numerator and denominator of rationals as well as the significand and the exponent of floating-point numbers are integers, it is easy to extend multi-precision integers to multi-precision rationals and multi-precision floating-point numbers.

Multi-precision integers are represented by a sequence of fixed-size integers and can be implemented using one of the following data structures:

Linked Lists: Implementing multi-precision integers as linked lists is very flexible but imposes additional overhead due to pointer management. Since linked lists can be extended at will, the only restriction on the size of the represented integers is the available memory of the computer system. Due to the trade-off between precision and computational cost, the slow-down can be significant for operations involving very large numerical quantities.

Arrays: Big number packages that are implemented using arrays usually impose a restriction on the size of the representable integers. The advantage of this method is that it is usually faster than the previous method because the size of the representable numbers is limited and there is no overhead due to use of pointers.

A variety of big number packages are available, see [49]. Each of them can be used as a basis for exact arithmetic. The choice is up to the user.

3.3.2 Exact Arithmetic in Geometric Algorithms

The notion of exact arithmetic usually implies that every numerical quantity is computed exactly. This may be necessary in some fields of computer science like computer algebra, but can be relaxed in other fields like computational geometry. Thus, problems like the high computational cost of exact arithmetic can be softened if certain properties of algorithms in computational geometry are exploited. Exact arithmetic in the context of computational geometry therefor implies that a representation is found that guarantees a certain accuracy.

There is also no uniform definition what an exact representation of a real number really is. While Schirra [40] suggests to call the representation of a real number x exact if an arbitrary approximation of whatever precision to x can be computed, Yap [46] calls the representation of a subset of real numbers exact if exact comparisons between any two numbers of this subset are guaranteed.

Depending on the type of algorithm one wants to implement, different requirements arise on exact arithmetic. We have already mentioned that correct comparisons imply the correctness of the combinatorial structure of an geometric output if the algorithm relies on predicates only. Thus, if comparisons are computed exactly, i.e., the control flow of the implemented algorithm is identical to the theoretical one, the correct combinatorial structure is guaranteed. Yap's definition of an exact representation reflects this

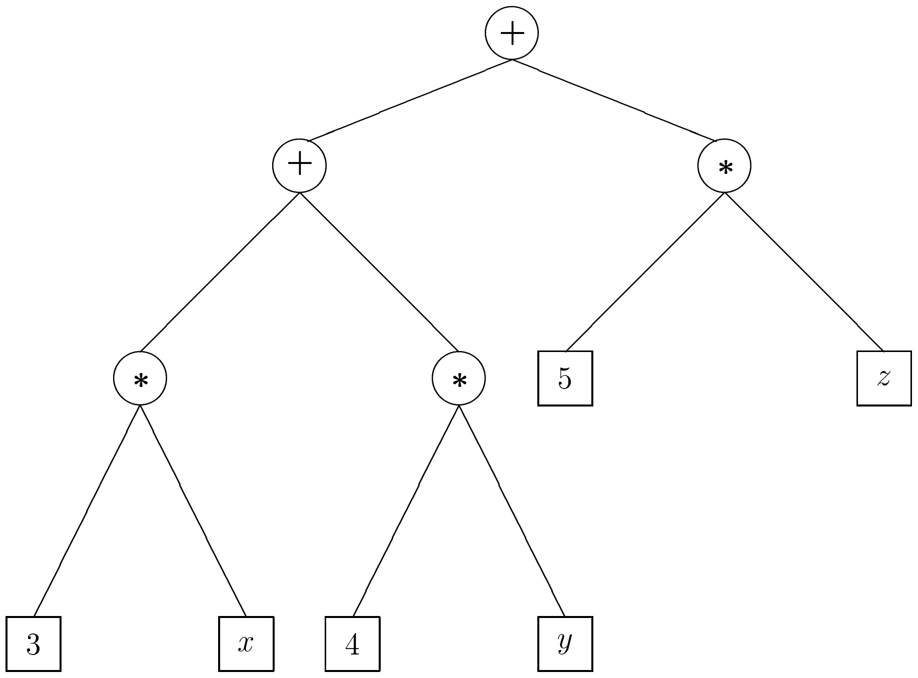

Figure 3.5: The implicit representation of the expression $3 * x + 4 * y + 5 * z$ with an expression dag.

requirement of exact arithmetic. It should also be noted that the output of an algorithm, that uses exact arithmetic is the correct result for the specified input data and not just for some perturbation of it. Schirra's definition of exact arithmetic reflects the requirements of geometric algorithms that also depend on constructors and thus have to create new geometric objects. There are a number of different representation schemes for exact numbers like:

Rationals: One possibility to represent exact values is to use rationals of the form $\frac{numerator}{denominator}$, where the numerator as well as the denominator are integers of arbitrary precision, see Chapter 4.

Symbolic Representation: Combinatorial relationships are stored rather than numerical values. E.g., if two line segments intersect we do not store the intersection point but the two line segments that intersect.

Implicit Representation: Numerical quantities can be stored by remembering their computation history. This can be done by storing a so-called expression dag[5]. An expression dag is an acyclic directed graph that stores the basic arithmetic operations in its interior nodes and the

[5]Dag is used as a short form for directed acyclic graph.

original operands, i.e., the original input data in its leaves, see Figure 3.5.

Despite the fact that there are ways to represent numerical data exactly it is still an open problem how to round them back into a finite representation without introducing errors in the combinatorial structure again. Furthermore, problems arise in cascaded computations because the algorithms that use the output of an algorithm that has been run previously need to operate with the same representation as the previous algorithm. The growth of the operands and the high computational cost are further problems. Thus, an algorithm that depends on exact arithmetic tends to be significantly slower than an algorithm that is based on floating-point arithmetic.

This ends our insight on exact arithmetic in the field of computational geometry for now. More information can be found in Chapter 4, which is dedicated to this topic.

3.4 Degeneracies

We have already mentioned that theoretical algorithms are developed assuming general position thereby excluding all possible special cases. This assumption makes it easier to present an algorithm and prove its correctness. A "general position" cannot be assumed if an algorithm is implemented because special cases are sure to arise in real-world data for several reasons:

- Degeneracies might be introduced due to finite-precision arithmetic. E.g., if floating-point arithmetic is used, two nearby but distinct input points might have the same floating-point representation and are therefor mapped to a single point.

- What is called degeneracy by an algorithm designer can be correct in the real world. E.g., a CAD designer might need to position four points such that they are cocircular. Degeneracy in the input data is therefor meant to be that way.

The issues of precision and degeneracy are closely related. Degeneracy might be introduced because the precision for representing numerical quantities is not sufficient. On the other hand degeneracies might be removed for the same reason. Degeneracies cause precision problems and robustness problems. We already saw that branching in geometric algorithms is done by evaluating a predicate. In general, evaluating a predicate means that the sign of some polynomial is determined. Depending on whether the sign is positive or negative a corresponding branch is chosen. If two geometric objects are positioned very close to each other, a predicate might evaluate to zero, thus making the decision which branch to take unclear. There are two different ways of dealing with degenerate data which we will discuss in the following subsections.

3.4.1 Handling Degeneracy Manually

One common method for coping with degeneracy is to treat a degenerate case manually as a special case. Whenever a predicate evaluates to zero a code fragment is activated that detects the type of degeneracy and handles it appropriately. Although the treatment of degenerate cases as special cases is common practice there are some drawbacks inherent to this method. First of all, a software developer has to consider every possible degenerate case that may arise in his application. There are a number of special cases for even simple geometric input like point sets, e.g., two points coincide, three points collinear or four points cocircular. The number of special cases increases tremendously if more complex geometric objects or higher dimensions need to be considered. Detecting the type of degeneracy is not such an easy task either. And even if all possible degeneracies can be detected there might be no straightforward method to handle them. Needless to say, that the resulting code grows larger and is harder to maintain.

3.4.2 A General Method for Handling Degeneracy

Perturbation methods can be used to eliminate degeneracy as a whole. If a perturbation method is used a software developer can implement an algorithm assuming "general position" and thus does not have to worry about degeneracy at all. This makes the implementation easier and more stable. The basic idea behind perturbation is to manipulate the input data such that degenerate cases vanish. In general, it is important that the perturbation is small such that the relative position of non-degenerate objects is not changed.

A commonly known perturbation method is called Simulation of Simplicity [13], or SoS for short. SoS perturbs the coordinates of geometric objects symbolically, i.e., every coordinate is replaced by a polynomial in ϵ with ϵ sufficiently small. The polynomial is chosen such that the perturbed set of objects converges towards the original set as ϵ goes to zero. Furthermore, the polynomials should satisfy the following requirements:

1. The resulting perturbed set of objects has to be non-degenerate if $\epsilon > 0$ is sufficiently small.

2. The resulting perturbed set of objects has to retain all non-degenerate properties of the original set.

3. The computational overhead caused by the simulation should be small.

Assuming we are given a set of n geometric objects $O = \{O_0, O_1, \ldots, O_{n-1}\}$. Each object has d coordinates:

$$O_i = \{\pi_{i,1}, \pi_{i,2}, \ldots, \pi_{i,d}\} \text{ for } 0 \leq i \leq n-1.$$

As already mentioned above, the set O is perturbed by replacing each coordinate $\pi_{i,j}$, $0 \leq i \leq n-1$ and $1 \leq j \leq d$, with a polynomial in ϵ. In [13] the perturbed set $O(\epsilon)$ is called the ϵ-expansion of the original set O and is defined as follows:

$$O(\epsilon) = \{O_i(\epsilon) = (\pi_{i,1} + \epsilon(i,1), \pi_{i,2} + \epsilon(i,2), \ldots, \pi_{i,d} + \epsilon(i,d)) \mid 0 \leq i \leq n-1\},$$

Assuming that each geometric object O_i has a unique index between 0 and $n-1$, the polynomials $\epsilon(i,j)$ are chosen in different orders of magnitude corresponding to the index pairs (i,j). E.g., in [13], $\epsilon(i,j)$ is chosen such that

$$\epsilon(i,j) = \epsilon^{2^{i \cdot \delta - j}} \text{ for } 0 \leq i \leq n-1,\ 1 \leq j \leq d,\ 0 < \epsilon < 1 \text{ and } \delta \geq d.$$

Thus, an expression involving several factors of the form $\epsilon(i,j)$ can be compared solely on the basis of the index pairs (i,j) involved. Since the perturbation in SoS is not directly computed but carried out symbolically by replacing each coordinate by a symbolic expression instead, geometric predicates based on adapted operations that operate on symbolic expressions have to be implemented. As an example, the function `Smaller` [13] which is passed two coordinates as its arguments and returns `true` if the first argument is smaller than the second one and returns `false` otherwise, can implemented with the following pseudocode fragment:

```
boolean Smaller($\pi_{i,j}$, $\pi_{k,l}$)
{
    if ($\pi_{i,j} \neq \pi_{k,l}$)
        return ($\pi_{i,j} < \pi_{k,l}$);
    else if ($i \neq k$)
        return ($i > k$);
    else
        return ($j < l$);
}
```

If $\pi_{i,j} \neq \pi_{k,l}$ then we have a non-degenerate situation and the coordinates can be compared directly, else the index pairs (i,j) and (k,l) are used to determine if $\pi_{i,j}$ is smaller than $\pi_{k,l}$.

A limitation of SoS is that the polynomials in ϵ used as a replacement for the actual coordinates can become very complicated if deep algebraic computations are involved. Furthermore, SoS cannot be used if square root operations are involved in the computation. It is also important to note that the solution computed by an algorithm that utilizes a perturbation method is the solution of a perturbed version of the input instance. If the result for the original input instance is needed, some form of post processing is necessary, which might also be non-trivial. The computational cost of algorithms that depend on a perturbation technique is generally higher because perturbation techniques usually resort to exact arithmetic for specific tasks. SoS, for example, relies on exact arithmetic to detect degenerate cases. As a last remark, it is also important to note that a perturbation method removes every degeneracy, even if it was intensional. This ends our discussion of symbolic perturbation, for more information see [13].

Chapter 4

Exact Geometric Computation

This chapter is concerned with Exact Geometric Computation (EGC), a variant of exact arithmetic that exploits properties of geometric algorithms in order to improve their efficiency. Following an introduction to EGC, we provide an overview of basic EGC concepts followed by a survey of techniques that can be used to accelerate EGC. A discussion of geometric libraries that implement or utilize EGC techniques concludes this chapter.

4.1 What is Exact Geometric Computation

As we saw in the previous chapter, exact arithmetic can be used to avoid numerical inacurracy. Exact arithmetic in conjunction with a perturbation method simulates the real RAM model. Algorithms can therefor be implemented in a straightforward manner. A major drawback of exact arithmetic is its high computational cost. EGC relaxes the requirement to compute every single numerical quantity exactly. EGC requires exact comparisons only to ensure the correctness of the combinatorial structure of an geometric output. According to [29], this approach has the following advantages over naive exact arithmetic:

1. No exact values are computed where they are not feasible. Since full numerical accuracy is not always needed it is a waste of CPU time to compute every numerical quantity exactly.

2. The solution to a geometric problem is computed with the precision that is actually needed. This is called the precision-driven approach and facilitates the use of techniques like lazy evaluation, adaptive computation or floating-point filters to achieve an additional speed-up. Furthermore, the precision-driven approach allows user control of the precision needed for a geometric output.

Assuming that the input is numerically accurate and consistent with the combinatorial structure, EGC ensures the correctness of a geometric algorithm that solely relies on predicates. For algorithms that also depend on constructors, a numerical output that is consistent with its combinatorial output to some absolute or relative precision requirements can be computed. It has to be noted, though, that the numerical data of an geometric output can grow very large which might cause problems in cascaded computations. Rounding numerical quantities back to a finite-precision representation introduces problems similar to those in standard floating-point computations and is still an open problem.

As we have mentioned before, the goal of the EGC approach is to compute numerical quantities sufficiently high such that exact comparisons are guaranteed. The problem of computing exact comparisons can be reduced to the problem of determining the correct sign of an arithmetic expression. In EGC the class of radical expressions is considered. These are expressions involving the basic arithmetic operations $(+, -, *, /)$ and $\sqrt[k]{\ }$. Expressions with transcendental functions or π are still an open problem [29]. In order to guarantee correct comparisons, an expression[1] is computed to a precision where the sign can be determined exactly while root separation bounds are used to test whether the expression evaluates to zero.

4.2 Basic Concepts of EGC

Let us recall that exact computation in the context of EGC does not imply to compute every numerical quantity exactly. Instead, it suffices to insure correct comparisons. There are three basic building blocks that EGC relies on:

Root separation bounds: Root separation bounds are used to determine if an expression evaluates to zero. Thus, they are a justification for the use of approximate numbers in EGC.

Expressions: Expressions are used to represent a numerical quantity exactly. This is achieved by remembering its whole computational history.

Big number package: Some big number package is used to represent numerical quantities directly. This is necessary in the case of input data

[1]We will use the term expression as a synonym to radical expression throughout the rest of this thesis, since radical expressions are the type of expressions that are commonly used in EGC.

as well as for evaluated expressions. Due to the existence of root separation bounds some form of approximate representation based on a big number package suffices to determine the sign of an expression correctly.

We will now take a closer look on these basic building blocks of EGC and sketch the sign determination process. Concepts to accelerate EGC and software libraries that implement EGC techniques will be discussed in the subsequent sections.

4.2.1 Root Separation Bounds and the Sign Determination Process

Root separation bounds are used to determine whether an expression evaluates to zero or not. Without the theory of root separation bounds there would be no way to determine the correct sign of an expression using approximate values. Therefor, root separation bounds are the justification for EGC. In [29] a positive number b is called a root separation bound of an algebraic expression E if the following holds:

$$\text{if } E \neq 0 \text{ then } |E| \geq b.$$

There are a number of different root separation bounds with different properties. In the context of EGC, it is important that the bound is reasonably tight and easy to compute. It has to be mentioned, though, that there is not a single bound that is always better than all the other known bounds. Therefor, it is important to find a bound that is best suited for those classes of expressions that frequently occur in geometric algorithms. The computation of root separation bounds is out of the scope of this thesis; see [29, 31, 9] for more information on this topic.

We will now sketch the sign determination process and presume that we have already found a suitable root separation bound b. In order to determine the correct sign of an expression it is necessary to check whether the expression evaluates to zero or not. The sign determination of an algebraic expression can then be executed in two steps, see [29, 31].

Step 1: Compute an numerical approximation $\tilde{E}$ to the algebraic expression E progressively until $\tilde{E}$ satisfies either

$$|E - \tilde{E}| < \tfrac{b}{2} \text{ or } |\tilde{E}| > |E - \tilde{E}|.$$

Step 2: If $|\tilde{E}| > |E - \tilde{E}|$ is reached first, then the sign of the approximation $\tilde{E}$ is the same as the sign of E. Thus, no root separation bound is needed to determine the sign safely. Note, that this condition is usually reached first if $|E|$ is large. If the condition $|E - \tilde{E}| < \frac{b}{2}$ is reached first, then the root separation bound is necessary and the sign of E is determined according to the following rule:

$$sign(E) = \begin{cases} sign(\tilde{E}) & \text{if } |\tilde{E}| \geq \frac{b}{2} \\ 0 & \text{otherwise.} \end{cases}$$

The approximation $\tilde{E}$ can be computed using double precision floating-point arithmetic in a first step. If this is not sufficient, the precision is doubled and the approximation is computed again. According to this procedure, precision is increased until one of the conditions in Step 1 is reached. This ends our discussion of this topic; see [29, 10] for more information and implementational details.

4.2.2 Expressions

The use of expressions in EGC is motivated by the observation that arithmetic operations do not occur arbitrarily and unpredictably in geometric algorithms [49]. Furthermore, they do not change dynamically [19]. Analyzing geometric problems, one can observe that there is a known set of primitive operations that can be used to solve such problems. Determinant evaluation is an example of such a primitive operation that suffices to solve a variety of problems in computational geometry, such as triangulations or convex hulls.

As already outlined in Chapter 3, an expression is represented as a labeled directed acyclic graph (dag). The root as well as all the internal nodes of the expression dag are labeled by an operator. The number of successors of a non-leaf node corresponds to the number of operands of the arithmetic operation that is represented by that particular node. E.g., if a node is labeled by a binary operation, it has two successors. While the root represents the value of the whole expression, each internal node represents the value of a subexpression. Finally, the leaves are labeled by numerical variables that represent the input values. If the expression is instantiated with some input data, the expression is evaluated bottom-up by simply propagating values to all the internal nodes until the root of the dag is reached. As already mentioned, the class of expressions considered in EGC is generally the class of radical expressions involving the basic arithmetic operations $+$, $-$, $*$, $/$ and the $\sqrt[k]{\ }$. A new operation is simply inserted into the expression dag. Thus, an arithmetic operation takes constant time. Besides the exact representation of numerical values, expressions have the additional advantage of having a lot of room for optimizations, see [19, 46, 29, 49, 19]. Similar to a compiler that parses and optimizes the code of a programming language, the expression can be compiled and optimized prior to their evaluation. As an example, Yap suggests in [46] to exploit the associativity property of the addition operation. If the signs of the operands can be estimated at compile-time, then the operands can be grouped into positive and negative values which are added among themselves first. This approach helps to avoid unnecessary precision.

4.2.3 BigFloat as an Example for an Approximate Representation of Numerical Values

We have already mentioned that the EGC approach is a relaxation of the standard exact arithmetic. Therefor, EGC is based on exact arithmetic which implies that a big number package is necessary to implement EGC techniques. As we outlined in the previous chapter, several different big number packages are available. It has to be noted though, that the majority of big number packages have been developed for computer algebra. While the precision needed in computer algebra programs is usually unpredictable, the precision needed in geometric algorithms usually is, see [46]. Thus, there is room for optimizations if such a package is redesigned to address the needs in computational geometry. For optimization reasons mentioned above the Core-library[2] uses a big number package called BigFloat [48, 46, 49, 29]. We will now take a closer look at it. Remember that in the context of EGC, exactness is defined by guaranteeing error-free decisions. Therefor, there is no need to compute every numerical quantity exactly. The existence of root separation bounds, see Subsection 4.2.1, justifies the use of approximate numbers in EGC. Thus, a proper representation for approximate numbers is needed. Furthermore, this representation should satisfy the following requirements:

1. In contrast to fixed-precision arithmetic, where every numerical quantity is computed with the same precision, a more flexible approach that facilitates the specification of different precisions for different variables is needed. The precision specified should be arbitrarily large.

2. The number representation should decouple the magnitude of a number from its precision. Similar to the precision, the magnitude of the number should be arbitrarily large.

3. Each approximate number should carry an error bound that is automatically computed.

[2]The Core-library is an EGC library that is implemented in C++. We will discuss it in the next chapter.

The designers of the Core-library opted for a floating-point representation. Since the exponent determines the size of the number and the significand specifies its precision, floating-point numbers are best suited to decouple precision from magnitude. By representing both the exponent as well as the significand with big integers, the size and precision can be arbitrary large. In order to satisfy requirement Number 3, an error bound is associated with every floating-point number. An arbitrary BigFloat number x is of the following form:

$$x = (m \pm \delta) \times \beta^e, \text{ where}$$

$\beta > 1$ is the base, m and e are the significand and the exponent, both represented by big integers and δ is an error bound that is automatically computed for each arithmetic operation. If $\delta = 0$, the represented number is exact. Since a BigFloat number is actually an interval, care has to be taken that the interval, i.e. δ, does not grow too large. Thus, a BigFloat number is normalized after each arithmetic operation. The reader is referred to [36, 49, 48] for more information on how the arithmetic operations are carried out, and for the details of the normalization process.

4.3 Accelerating EGC

The utilization of expression dags in EGC facilitates the use of techniques like lazy evaluation [6] and the precision-driven approach [48, 49] to tune the process of sign determination. Another technique to accelerate EGC is a floating-point filter. We will now take a look at these concepts.

4.3.1 Floating-Point Filter

A common method to accelerate the exact computation of an expression is to use a so-called floating-point filter [19, 7]. The purpose of a floating-point filter is to filter out those computations that yield a correct result with standard floating-point arithmetic, and to use exact arithmetic only in the remaining cases. In a first step, the expression E is evaluated using standard floating-point arithmetic yielding an approximation $\tilde{E}$. Depending on the type of filter, an upper bound ξ on the error of $\tilde{E}$ is computed before or during runtime. If

$$|\tilde{E}| > \xi,$$

then the sign is known safely [7], otherwise exact arithmetic is used to determine the sign correctly. There are three types of floating-point filter:

Fully static filter: A fully static filter can be used with expressions that are built from the basic arithmetic operations $(+, -, *, /)$ and $\sqrt{\ }$. First, an upper bound on the error of all the coefficients of the expression E is computed. Then, based on these upper bounds, an upper bound ξ which is valid for all possible input instances is computed. Therefor, the upper bound ξ is computed before runtime.

Semi-static filter: If it is not possible to find an upper bound on the error of the coefficients of an expression E, a semi-static filter can be used assuming that it is possible to find a formula similar to E that yields an upper bound for a particular input instance. The upper bound ξ is computed according to this formula at runtime but prior to the evaluation of the expression E.

Dynamic filter: A dynamic filter computes ξ at runtime during the evaluation of E. For each operation in E, an error bound is computed resulting in the upper bound ξ at the root of the expression.

In general, floating-point filter yield a significant speed-up in geometric algorithms since the use of exact arithmetic can be reduced significantly in the majority of input instances.

4.3.2 Lazy Evaluation

Lazy evaluation [6] utilizes interval arithmetic, see Page 31, and is based on the following paradigm:

> Why should a numerical quantity be computed exactly if it is not involved in conflicting issues in subsequent computations.

The basic idea of lazy evaluation is to store enough information to compute the exact value of a numerical quantity if this is necessary. Therefor, a lazy number is basically an interval that contains the exact value of the represented number. The interval is bounded by two floating-point numbers. Such an interval is automatically assigned to each input quantity. The input values form the leaves of an expression dag. As already mentioned above an operation can be performed in constant time using expression dags. If necessary, the intervals of a node can be refined or the expression is evaluated using exact arithmetic. There are only three different cases [6] where this is necessary:

1. The numerical quantity represented by an expression dag is compared to another numerical quantity and their intervals intersect.

2. The reciprocal of a numerical quantity is required and its interval includes zero.

3. The evaluation of an predecessor in the expression dag is required.

The evaluation process is carried out in a bottom-up fashion. Starting with the leaves of the expression dag, tighter intervals are propagated to the internal nodes until, finally, the root is reached. This procedure is carried out until the interval at the root is tight enough to make a correct decision. If this is not possible, the expression is evaluated with exact arithmetic.

The benefit of the lazy approach is that the evaluation of an expression is delayed as long as possible. Furthermore, it is only necessary to reevaluate an expression in the cases mentioned above. Thus, there can be a significant speed-up compared to the naive use of exact arithmetic.

4.3.3 The Precision-Driven Approach

Another technique to accelerate EGC is the precision-driven approach of Yap and Dubé [48, 49]. Similar to the lazy approach, an expression dag is used to represent numerical quantities. The main difference between the lazy approach and the precision-driven approach is the evaluation process of an expression. The lazy approach specifies a precision at the leaves of the expression dag. This precision is increased until the precision at the root is acceptable. In the precision-driven approach, on the other hand, the precision we want at the root is specified first. This precision is propagated in a top-down fashion. At the leaves, a sufficient approximation is computed such that the precision specified at the root holds. Finally, starting from the leaves, the expression is evaluated bottom-up, similar to the lazy approach.

Yap and Dubé introduce a composite precision bound to specify the required precision at the root of an expression dag. In [48, 49] they define the approximation $\tilde{x}$ of a real number x to a composite precision $[a, r]$, written $\tilde{x} \cong x[a, r]$, as follows:

$$|x - \tilde{x}| \leq max\{2^{-a}, 2^{-r}|x|\},$$

where a is the absolute precision and r the relative precision. Thus, the user can decide if he wants to specify the precision in relative terms only by specifying $[\infty, r]$, in absolute terms only by specifying $[a, \infty]$ or in relative and absolute terms by specifying $[a, r]$ at the root.

The precision-driven approach is a more active approach than lazy evaluation. By propagating the precision specified at the root of the expression dag downwards, an approximation that satisfies this precision bound can be computed at the leaves thus avoiding unnecessary iterations in the evaluation process.

4.4 EGC Libraries

A number of different libraries that implement EGC techniques have been developed since the first proposal of EGC:

LEDA: LEDA [34, 33, 3], which is distributed under a commercial license, provides low-level data types, predicates and algorithms for EGC. More information on the availability of LEDA and the LEDA academic program can be found on the LEDA homepage [3].

CGAL: CGAL [14, 15] provides a set of data structures and algorithms. It uses the standard machine data types or low-level data types provided by other libraries. CGAL licenses are available free of charge for academic use; see the CGAL homepage [1] for more information on the license model.

Core-library: The Core-library [26] provides low-level number types for EGC and the corresponding operations. It is available free of charge and can be downloaded at [2].

There are a number of design goals [39, 14] that have to be considered by the developers of a geometric library such as:

Robustness and Correctness: Since the goal of geometric libraries is to facilitate the development of robust geometric algorithms, it is necessary that the implementation of the library itself is robust and correct. If the library provides geometric predicates and algorithms in addition to some low-level data types and operations, care has to be taken that those algorithms and predicates are correct. As pointed out in [39, 14], an algorithm or predicate is correct if it behaves according to its specification. For example, an algorithm that is restricted to input in "general position" only is correct if it computes the correct answer for the specified input instances.

Efficiency: Another important requirement for a geometric library is efficiency. The practical value of a geometric library is questionable if the computational cost is too high. Industrial-strength algorithms have to cope with large input data sets and since "time is money", a correct output has to be produced quickly.

Ease of use: Implementing an algorithm based on a library with a complicated interface can be very time-consuming. Furthermore, it is an additional source of errors. It is therefor essential to hide implementational details from the user and provide him with a simple interface instead. Ease of use is an important property for a library to become widely accepted.

Generality, Modularity and Openness: The algorithms developed in computational geometry have many potential applications in other areas of computer science like computer graphics, virtual reality, computer aided design (CAD), computer vision, solid modeling, robotics or geographical information systems (GIS). Different needs may arise in these areas. A geometric library should therefor be designed in a rather general way in order to provide a sound basis for all kinds of applications that arise in computer science. Besides generality, modularity and openness are necessary to facilitate the adaption and extension of the library according to a user's needs.

The success and the acceptance of a library in computational geometry depends heavily on the consideration of the design goals mentioned above. In the following, we will survey the most important libraries for computational geometry – LEDA, CGAL and the Core-library.

4.4.1 Computational Geometry Algorithms Library – CGAL

CGAL [14, 15], the short form for Computational Geometry Algorithms Library, is the result of a cooperation between eight European institutions: Utrecht University (Netherlands), ETH Zürich (Switzerland), Free University Berlin (Germany), Martin-Luther University Halle (Germany), INRIA Sophia-Antipolis (France), Max-Planck Institute of Computer Science and University Saarbrücken (Germany), RISC Linz (Austria), and finally, Tel-Aviv University (Israel). The goal of the CGAL project is to bring the variety of efficient data structures and algorithms that have been developed in the field of computational geometry to practice and make them available for industrial application. The CGAL library is implemented in C++, since C++ is widely accepted and it can easily be interfaced with existing C and FORTRAN programs. Furthermore, it facilitates library design and implementation and the resulting code is usually faster than, for example, Java code.

In order to achieve a maximum of flexibility and modularity, the CGAL-library design is based on a lot of C++ concepts. Virtual base classes with virtual functions are used to provide a uniform interface to the CGAL functionality. Based on a virtual class, different classes can be derived, providing the same functionality but with different implementations. For example, geometric objects can be represented using standard Cartesian coordinates or alternatively homogeneous coordinates, a line can be represented by the coefficients of its equation or by its two endpoints, and so on. Another application of this concept is to provide algorithms with different functionality, e.g., an algorithm that can cope with degenerate input and is naturally slower or, alternatively, an algorithm that is faster but does not handle degenerate input. Depending on his/her needs, the user is free to choose the functionality and representation he/she desires. It has to be mentioned, though, that virtual classes and virtual functions impose additional overhead due to the virtual function table pointer that has to be stored in each object that is derived from a virtual base class and the indirection through the virtual function table. Furthermore, templates are used to implement, for example, container classes like lists and trees for different data types. Finally, so-called circulators, a concept closely related to the concept of iterators used in the C++ Standard Template Library (STL), are introduced to iterate through circular data structures which frequently arise in geometric algorithms. In the following, we will focus on the structure and functionality of CGAL. For more information on implementational details, see [15].

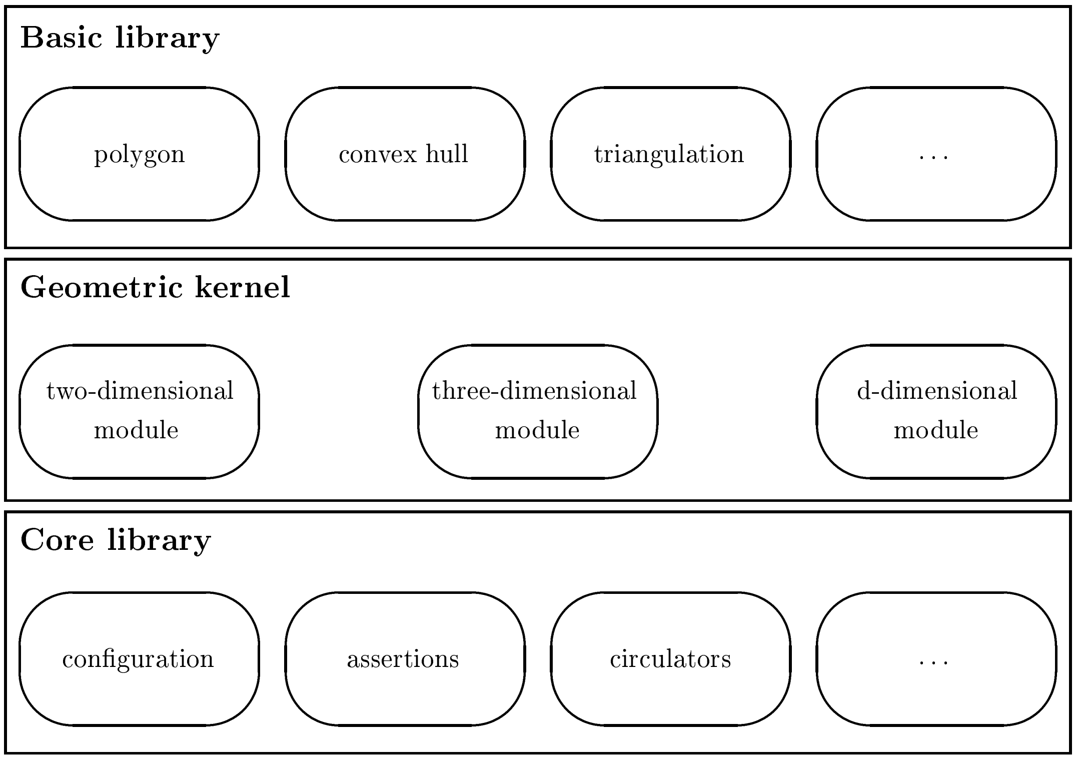

Figure 4.1: The three CGAL layers.

As illustrated in Figure 4.1, CGAL is comprised of three different layers built on top of each other. The two lowest layers, the core library[3] and the geometric kernel form the so-called CGAL kernel. In addition to the the three layers of CGAL there is a support library that stands apart from the rest and provides additional functionality like visualization of geometric objects. We will now describe the functionality provided by each layer:

Core library: The core library is the lowest CGAL layer and provides functionality that is needed by the upper layers but is not purely geometric. It offers support for circulators, dealing with assertions, compatibility issues of different C++ compilers, and random numbers.

Geometric kernel: The geometric kernel provides simple geometric objects like points, lines, line segments, triangles, and tetrahedra, plus geometric predicates, basic operations like intersection and distance, and transformations on these objects. In order to utilize certain properties of different dimensions, the geometric kernel is divided into three parts for two-dimensional, three-dimensional and general-dimensional objects. Furthermore, Cartesian and homogeneous representations are supported in every dimension. It is interesting to note that no basic number type for the representation of numerical quantities is provided by CGAL itself. Due to the use of templates, the data structures for the basic geometric objects are parameterized. Thus, the user is free to pick the number type that suits his/her needs. If, for example, speed is more important than robustness, standard machine data types like `float` or `double` can be chosen. If, on the other hand, robustness is more important than computational cost, then a data type like `leda_real` from LEDA or `Expr` provided by the Core-library (NYU) can be chosen to support EGC. Since divisions can be prevented using homogeneous coordinates, it is an other interesting possibility to use arbitrary precision integer arithmetic in conjunction with the homogeneous representation of geometric objects.

Basic library: The basic library provides high-level data structures and algorithms. Examples include polygons, polyhedrons, triangulations, kd-trees and algorithms for computing convex hulls, the smallest enclosing circle, and triangulations. Care is taken that all the functionality provided by the basic library is independent from each other. Thus, changes made in one component do not affect the other components. Furthermore, this makes it easier to test the components independently. The communication between components in a layer and between different layers is carried out using well-defined interfaces, see [15].

[3]Core library in the context of CGAL refers to the lowest layer of CGAL and must not be confused with the Core-library developed at NYU.

We have already mentioned above that there is a so-called support library that is not part of a layer but stands apart from the rest of the library. The support library adds functionality that is not vital for the rest of the library but is quite useful for developing a geometric application. This includes support for visualization of CGAL objects by providing interfaces to languages like VRML and PostScript, or to programs like GeomView and LEDA windows.

Summarizing, CGAL is a high-level geometric library that provides a set of geometric data structures and geometric algorithms. The library itself is divided into three layers which communicate through a well-defined interface. Components within a layer are designed to be independent from each other. Communication between those components is again established through a well-defined interface. CGAL does not provide a data type that supports EGC. Instead, all geometric data structures are parameterized such that a number type to represent numerical quantities can be chosen at will. If robustness is the major issue then data types like `leda_real` or `Expr` can be used to achieve EGC functionality. For ease of use, CGAL provides a uniform interface with a functionality that is closely related to the interface provided by the C++ STL. For more information on CGAL see [15, 14, 1].

4.4.2 Library of Efficient Data Types and Algorithms – LEDA

The LEDA project [34, 33, 3] started 1989 prior to CGAL but with similar intentions. Just like CGAL, the main goal of LEDA is to transfer technology from theory to practice. Advanced knowledge gained in the field of computational geometry and combinatorial computing should be provided to the user. Thus another similarity of LEDA and CGAL is that both are high-level libraries that provide a collection of data structures and algorithms that operate on these data structures. LEDA is written in C++ for reasons similar to CGAL. Therefor, all high-level data structures are parameterized using templates. The user decides which data type he wants to use to represent numerical quantities. Unlike CGAL, LEDA provides a number of low-level data types for this task. Among the supported data types are the standard machine data types like `int`, `float` and `double` as well as arbitrary precision versions called `Int` and `Float`. `Int` is an arbitrary precision integer type in the mathematical sense and `Float` is a floating-point data type with arbitrary precision significand and exponent. The data type `leda_real` [8, 10] is used to support EGC.

We will now survey the features of LEDA and take a closer look on the data type `leda_real` because it has a lot of similarities to the type `Expr` provided by the Core-library. LEDA is organized into six logical units, see [34]:

Basic data types: The basic data types are strings, lists, queues, stacks, arrays, partitions and trees. All of them are parameterized such that any of the supported low-level number types can be chosen to represent numerical quantities.

Numbers, vectors and matrices: The supported low-level number types are the machine data types `int`, `float`, and `double`. In addition the multi-precision versions `Int` and `Float` are implemented to support exact computation. EGC is provided by the type `leda_real`. Vectors and matrices are available for all these data types.

Dictionaries and priority queues: Dictionaries, priority queues, dictionary and hashing arrays, sorted sequences and persistent dictionaries are provided by this logical unit of LEDA.

Graphs: The graph unit of LEDA implements data structures for directed, undirected and planar graphs like arrays indexed by nodes and edges, priority queues on nodes and node partitions. Algorithms that operate on graphs like shortest paths, biconnected and strongly connected components, transitive closure, topological sorting, unweighted and weighted bipartite matching, network flow, planarity testing, planar embedding, etc. are also provided by this unit.

Windows and panels: This LEDA unit provides an X11 interface and supports the output of geometric objects and interactive mouse input.

Geometry: Finally, the geometry unit provides points, lines and line segments plus some higher-level data structures on these objects like planar subdivisions, range trees, segment trees, and interval trees. Algorithms like line-segment intersection, Voronoi diagrams, Delaunay triangulations and convex hulls are also implemented in this unit.

The algorithms in LEDA are robust in the sense that exact rational arithmetic can be used for geometric objects. Furthermore, the algorithms are designed to cope with degeneracies. If the user prefers to use EGC techniques instead of exact arithmetic he can resort to the LEDA data type `leda_real`. We therefor take a closer look at it below. For flexibility and performance reasons, different implementations of data structures and algorithms are provided and if computational speed is more important than robustness one can use the machine types `int`, `float` and `double` to represent numerical quantities.

There seems to have been a cooperation between the CGAL and the LEDA project. Some of the developers are even contributing to both projects. Thus, there are a lot of similarities in both LEDA and CGAL – both provide high-level data structures and algorithms for geometric computing. The high-level data structures are parameterized in both libraries, thus providing support for different low-level data types to represent and operate on numerical quantities. While CGAL does not implement any arbitrary precision number types, LEDA implements the data type `leda_real`, that supports EGC and that is also used by CGAL as well as arbitrary precision integer and floating-point arithmetic.

The data type `leda_real` is basically a C++ class implemented within the LEDA framework. It it is closed under the basic operations $+, -, *, /$ and the $\sqrt[k]{\ }$ operation. Comparisons of numerical quantities are reduced to sign determinations, see Subsection 4.2.1, and can be done exactly. Thus, `leda_real` provides EGC functionality. Similar to the Core-library, see Chapter 5, `leda_real` represents a numerical quantity by remembering its whole computational history in an expression dag. The nodes of the expression dag are labeled by arithmetic operations. As already mentioned in Subsection 4.2.2, there is a correspondence between the operands of an operation in a node and its successors. The leaves hold the input values which are arbitrary length integers. If an arithmetic operation occurs, it is inserted into the expression dag and an initial approximate value using standard double precision floating-point arithmetic is computed. Thus, an arithmetic operation takes constant time. Both `leda_real` and the Core-library use the precision-driven approach from Subsection 4.3.3 to evaluate expressions but with different precision bounds. For more information and implementational details, see [8, 10].

4.4.3 Core-Library

The Core-library is an EGC library that has been developed at the Department of Computer Science at New York University. In contrast to LEDA and CGAL, the Core-library does not provide any high-level geometric data types and algorithms, except for the data structures and corresponding operations provided by so-called Core-extensions. Instead, it provides an expression data type together with the supported operations $(+, -, *, /, \sqrt{\ })$ that allows error-free comparisons and the computation of numerical quantities to arbitrary precision. The Core-library is based on the Real/Expr-library [36] and implements the EGC techniques outlined in [48] and [49]. Being a low-level library, Core is very flexible in the sense that the user has to implement the whole geometric algorithm by himself based on the expression data type and assuming exact arithmetic. Furthermore, the Core-library is easy to use because of the promotion of data types. While the Core-library itself is based on a small number core, additional functionality can be added by Core-extensions. For more information on the Core-library see the following chapter.

Chapter 5

Core-Library

This chapter is dedicated to the Core-library. After a short introduction, we discuss basic concepts and implementational issues of the library in Section 5.2. Then, we discuss how to use the Core-library in existing C/C++ programs in Section 5.3. Finally, we present the Core-based version of the program from Page 15. It is important to note that some of the concepts we mention in this chapter, e.g., Level II, Level IV and the optimizing compiler are not available in the current version 1.4 of the Core-library. Nevertheless we do not exclude these concepts from our discussion in order to take a glimpse at future improvements.

5.1 Introduction to the Core-Library

The Core-library [47, 26, 30] is a C++ library that implements EGC techniques. Besides stability, two major issues for EGC-libraries are ease of use and efficiency. The developers of the Core-library tried to address both issues. A simple interface is provided through the overloading of existing data types. This technique should support software developers to use the library in existing programs as well as in new ones. Compiler optimizations[1] and the use of the newest knowledge of EGC techniques should ensure reasonable performance. The Core-library provides four different accuracy levels:

Level I - Machine Accuracy: This level represents the conventional IEEE standard, see Section 2.4. Thus, Level I is fast but does not provide any EGC functionality, i.e., there are no gains in robustness compared to programs using standard floating-point arithmetic.

Level II - Arbitrary Accuracy: Level II provides the functionality of big number packages. The user is allowed to choose an accuracy that suits

[1]The compiler is still under development.

his/her needs. Specifying 100 bits of accuracy means that there will be no overflows or underflows in the numerical operation until 100 bits are exceeded. Thus, numerical quantities can be computed to an arbitrary, user-specified precision. Level II is useful if the user is aware of the precision needed in his/her application in advance. Naturally, there is a trade-off between precision and performance. The higher the precision, the poorer the performance will be. Therefor, it is important to find a balance between these two factors in order to achieve an robust algorithm with reasonable performance. Level II is not fully implemented yet.

Level III - Guaranteed Accuracy: Finally, Level III provides EGC functionality and is therefor the key innovation of the Core-library. It is slower than Level II but guarantees the correct results up to the specified number of bits. If 100 bits are specified, then the user is guaranteed that 100 bits of a quantity computed are correct. In Level III, numerical quantities are represented using expression dags, see Page 47. The precision-driven approach, described on Page 51, is implemented to evaluate expressions. Error-free comparisons are guaranteed in this level only. In order to assure the correct sign of a numerical quantity, it is sufficient to specify one bit of accuracy.

Level IV - Mixed Accuracy: This level is intended for finer accuracy control since the levels are intermixed and localized to individual variables. Unfortunately, this feature has not been implemented in Version 1.4 of the Core-library yet.

Since only Level I and Level III are fully supported by the current release of the Core-library, we mean Level III functionality whenever we speak of the Core-library unless we explicitly refer to a specific level.

The basic operations for EGC computations are provided by a small numerical core of the library which currently supports the mathematical operations $+$, $-$, $*$, $/$ and $\sqrt{\ }$. High-level data structures and algorithms or functionality for specific applications is provided through so-called Core-extensions. In the current release, two Core-extensions are available:

LinearAlgebra: The linear algebra Core-extension provides the class Matrix for general $n \times m$ matrices and the class Vector for general n-dimensional vectors. The classes also implement basic operations for matrices and vectors, e.g., the Gaussian elimination algorithm.

Geometry: The geometry Core-extension is comprised of a $2D$ package called geometry2D and a $3D$ package called geometry3D. Geometry2D provides basic two-dimensional objects like points and lines. Geometry3D offers points, lines and planes in the three-dimensional space. Both packages depend on the LinearAlgebra Core-extension.

The functionality of both Core-extensions mentioned above is very rudimentary in the current release. Nevertheless a software developer is free to implement new Core-extensions or customize the existing ones to suit his/her needs. Since a separate Core-extension can be built for every accuracy level, high-level data structures and algorithms can be provided for different requirements regarding robustness and computational cost.

In the next section we describe the internals of the Core-library. Section 3 will explain how to use the library in own programs and what has to be considered when using the Core-library in existing programs.

5.2 Internals of the Core-Library

This section provides an overview of the promotion and demotion mechanism which makes it easy to use the Core-library in existing C/C++ programs. Furthermore, we will provide an overview of the classes that implement the functionality of the numerical core and the data types provided by the Core-library. Subsection 5.2.4 and Subsection 5.2.5 explain how Level II and Level III work. A drawback in the use of C++ techniques like virtual functions and parameterization is that there is additional overhead due to virtual function tables and runtime type checking. These sources of overhead and ways to avoid them in future releases of the library are discussed in Subsection 5.2.6.

5.2.1 Supported Data Types

In the Core-library every accuracy level, except of Level IV, operates on its own set of data types. Level I uses the machine data types `int`, `long`, `float` and `double`. Level II implements its own data types called `Real`, `BigFloat`, `BigInt` and `BigRat`. These are all big number data types which are based on a custom big number package[2]. The data type `Real` is not a particular representation of numbers but a superclass of all the Level I and Level II data types and provides a uniform interface to access them. The data type `Expr` offers EGC functionality and is only available in Level III. Level IV does not need an own data type since it is used to intermix the previous three accuracy levels in order to provide finer accuracy control.

[2]Version 1.4 of the Core-library uses the big number package gmp.

5.2.2 Promotion and Demotion of Data Types

There is a natural partial ordering between the Level II data types which is defined as follows:

$$\texttt{float} < \texttt{double} < \texttt{BigFloat} < \texttt{BigRat},$$
$$\texttt{int} < \texttt{long} < \texttt{BigInt} < \texttt{BigRat} < \texttt{Real}.$$

Promotion and demotion of data types is carried out automatically when certain operations are performed or when the accuracy level is changed. If, for example, a `BigFloat` value is assigned to an `int` variable, then the `BigFloat` value has to be demoted to an `int` before it is assigned. Similarly, whenever the accuracy level is changed from Level I to Level III, then the data types `double` and `long` promote to the Level III data type `Expr`, while `int` and `float` remain unchanged. Even in Level III it is desirable to have Level I data types for efficiency reasons, e.g., for values that do not need to be computed exactly. The reason why `int` and `float` are not promoted is that they are low-precision data types. If `int` and `float` are used in a program then high precision, overflows and underflows do not seem to be an issue, otherwise the software developer would have used `long` and `double` instead. Therefor, it is reasonable to let `int` and `float` unchanged, even if the accuracy level is changed from Level I to Level III. The Level III data type `Expr` is demoted to `Real` if the accuracy level changes from Level III to Level II[3]. According to [26], the general principles for promotion and demotion are:

1. A program is called a Level l (l = 1,2,3) program if it explicitly declares data types of Level l but no data types of a level above l.
2. The functionality of lower levels is also available at higher levels.
3. Variables and features are demoted if a program changes from a higher level to a lower level.
4. In Level IV promotions and demotions are only performed whenever assignments are carried out.

The concept of promotion and demotion makes it easy to use the Core-library in existing C/C++ programs as well as in new ones. A software developer does not have to use the data types provided by the Core-library explictly. Instead, the well-known standard C/C++ data types can be used. If a software developer wants EGC functionality, he/she simply has to set the accuracy level, include the Core-library's header file and has to link the

[3]Since Level II is not fully implemented, this is currently of no relevance in practical applications.

program with the Core-library. This is especially true if the program was designed to use the Core-library from the beginning. In existing programs, however, there are a number of additional issues that have to be considered. We will discuss them in Subsection 5.3.2 and in the next chapter.

5.2.3 Classes Provided by the Core-Library

The basic functionality of the Core-library is provided by five classes which we describe briefly in this section.

The class `Real`: The class `Real` is the superclass of all Level II data types. It defines common operations for the derived classes and manages initialization.

The class `BigFloat`: The class `BigFloat` depends on the class `BigInt`, since the significand is a big integer. It is used to approximate values of the type `Expr`. A `BigFloat` number has three components:

1. A significand m which is of type `BigInt`.
2. An error $\delta \in \{0, \ldots, B-1\}$, where B is the base of the floating-point number. The type of the error value is `unsigned long`.
3. The exponent e which is of type `long`.

As we already mentioned in the previous chapter, instances of the class `BigFloat` carry an error bound δ that is automatically computed. Therefor, a `BigFloat` number actually represents an interval:

$$[(m-\delta) * B^e, (m+\delta) * B^e].$$

The error bound is automatically adapted using interval arithmetic when operations are performed. Due to interval arithmetic the error can grow very quickly, see [26, 48, 49], and thus the interval might grow too large. Since arbitrarily large intervals are not desirable, efforts are made to ensure an interval that is as tight as possible. Thus, the error is normalized such that:

$$0 \leq \delta \leq B-1.$$

For a detailed description on how the supported arithmetic operations are implemented; see [36, 29].

The class `Expr`: EGC functionality is provided by the class `Expr`. Thus, in Level III, numerical quantities are represented using expression dags. The leaves hold the operands and the inner nodes represent the operations. Assuming that the values at the leaves are error-free, expression dags represent the exact value of numerical operations. The class `Expr` encapsulates three basic components:

1. an expression dag T,
2. a precision p,
3. a number $\tilde{E}$ of the data type `Real`[4].

The `Real` value $\tilde{E}$ is used to approximate the value E of the expression represented by T to the precision p. Internally, operations and operands are represented using the class `ExprRep`. Therefor, the expression dag is built from instances of the class `ExprRep` while an instance of the class `Expr` points to the root of the expression dag. This situation is illustrated in Figure 5.1.

The class `ExprRep`: As mentioned above, instances of the class `ExprRep` hold unary operators, binary operators or an operand if they correspond to a leaf of the expression dag. Currently, operands in an expression are always `BigFloat` values. Instances of the class `ExprRep` hold two pointers if they represent a binary operator, one pointer if they represent a unary operator, or no pointer if they hold an operand to other `ExprRep` instances. Level III expressions are modeled as hierarchies of `ExprRep` instances, see Figure 5.1.

The class `extLong`: The class `extLong` is a wrapper for the standard data type `long` with three additional values:

1. `CORE_negInfty`,
2. `CORE_posInfty` and
3. `CORE_NaN`.

The value `CORE_negInfty` and `CORE_posInfty` represent negative and positive infinity respectively, while `CORE_NaN` is the value for not a number and indicates an illegal operation. The class `extLong` is designed in a way that no overflows and underflows can occur; its main purpose is to set the precision variable *defInputDigits*, which controls the accuracy of the input data.

[4]Recall that `Real` is only a superclass of all Level II data types. Thus, $\tilde{E}$ is actually of the type `BigFloat`.

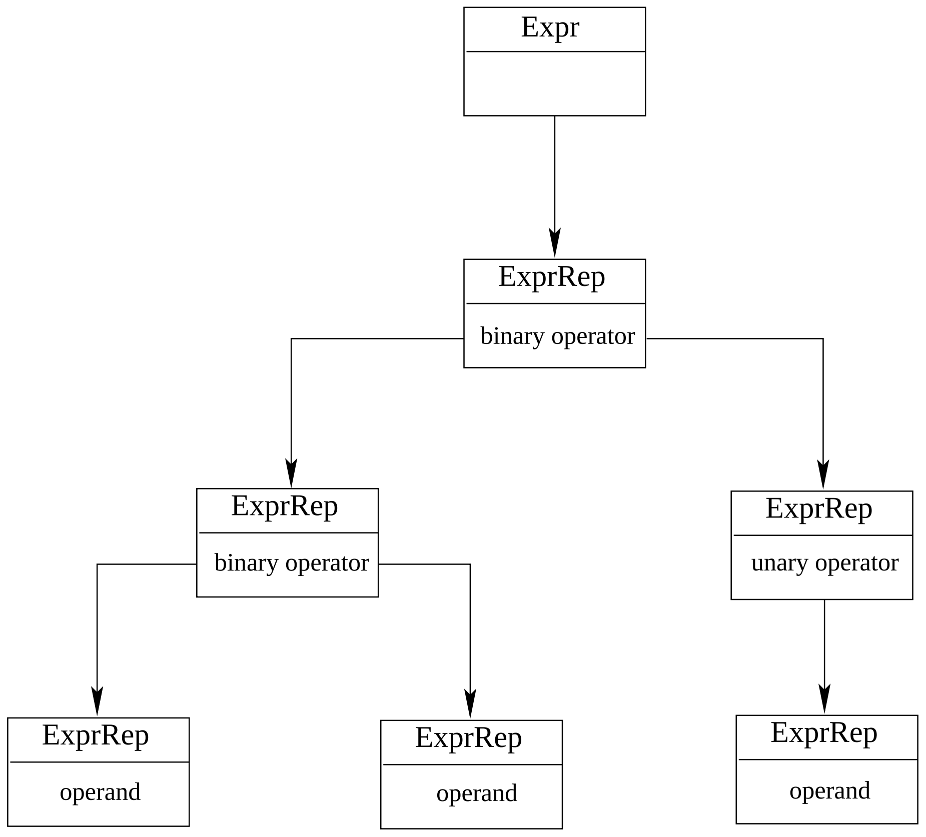

Figure 5.1: Internally, the expression dag is constructed from instances of the class `ExprRep`. The whole Expression is represented by an instance of the class `Expr`, which points to the root of the expression dag.

Now that we know how an expression is represented internally, we are going to see what happens to the variables used in a program when Level II or Level III of the Core-library are used.

5.2.4 How Level II Works

Level II works just like a big number package. Variables of the data types `long` and `double` are promoted to `Real`. Actually, `long` is converted to `BigInt` and `double` is converted to `BigFloat` internally, since `Real` is just a superclass of all the Level I and Level II data types. Variables of the type `int` and `float` remain unchanged for reasons mentioned in Section 5.2.2. In order to achieve a maximum of efficiency, the Core-library tries to leave `long` and `double` variables unchanged as long as possible and converts them only if overflows or underflows occur. Therefor, Level II has a built-in mechanism to check for overflows and underflows at runtime. To illustrate this process let us take a look at the following example:

```
double $w$, $x$, $y$, $z$;
int $i$;

$x = y * i$;
$w = x/z$;
```

If this code fragment is processed in Level II, the variables w, x, y, or z are promoted to `BigFloat` as soon as an overflow or underflow occurs involving that particular variable. The integer i is unchanged during the execution of the program.

5.2.5 How Level III Works

Variables of the data types `long` and `double` are converted to the data type `Expr` if Level III is used. Again, `int` and `float` variables remain unchanged. Whenever a program is executed in Level III, expression dags are built that remember the dependency of values from other values. Following the precision-driven approach from Page 51, a certain precision is specified at the root of an expression dag, and is propagated downwards to the leaves. At the leaves an error bound is computed by the system and propagated upwards to the root. This process is iterated until the error is smaller than or equal to the requested precision. E.g., if E is an expression, $\tilde{E}$ an approximation to E and $[r, a]$ is a composite precision bound then an error

$$Err_{\tilde{E}} \leq max\{|E| * 2^{-r}, 2^{-a}\}$$

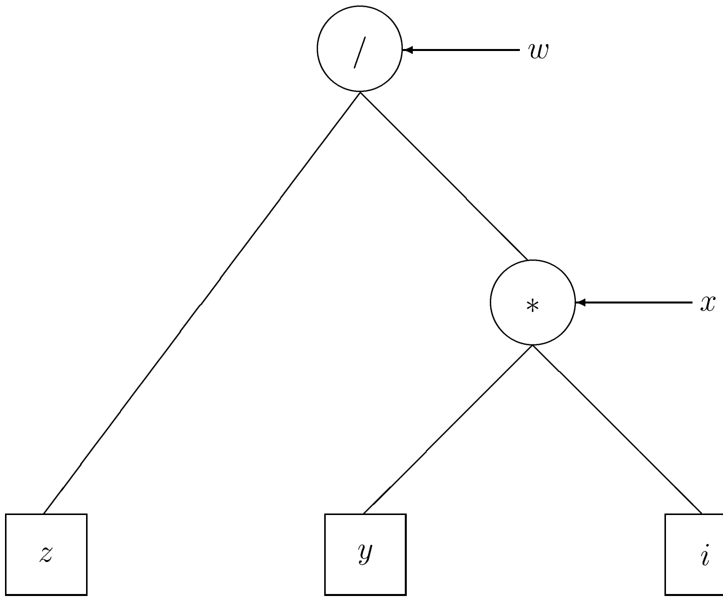

Figure 5.2: The expression dag that is constructed if the sample code from Subsection 5.2.4 is executed in Level III.

is guaranteed by the system. The expression tree constructed for the example in Subsection 5.2.4 is illustrated in Figure 5.2. The dependence of w from x is remembered in a natural way due to the construction of the expression dag.

5.2.6 Sources of Overhead and Optimization

The practical value of a library for EGC depends primarily on its performance. Besides the use of fast algorithms and their careful implementation, optimizations at the compiler level may help to gain additional performance. The developers of the Core-library are therefor working on an optimizing compiler that analyzes the expressions used in a program and tries to optimize them. Furthermore, additional overhead introduced by the object-oriented programming style is reduced. According to [26], Level III evaluation is suffering from the following sources of overhead that are not present if Level I is used:

Function Call Overhead: Overhead is introduced through dynamic dispatching and binding due to the use of virtual functions.

Memory Management Overhead: Object hierarchies are created to implement Level III expression dags. This introduces cache line fragmentation and poor spatial locality – a general problem that comes with the use of pointers for dynamic memory allocation and deallocation.

Operation Overhead: Several iterations of the downward propagation of precision bounds and upward propagation of error bounds may be necessary to evaluate an expression in Level III. For reasons of encapsulation these steps are performed at the granularity of individual expression nodes (operations). Reduction of overhead by exploiting the global structure of an expression is possible. E.g., similar operations like + could be grouped in one node in order to reduce the depth of the expression dag.

General Overhead: General overhead is introduced by the object-oriented programming style. E.g., global objects are constructed using smaller component objects.

A compiler that optimizes memory management and brakes some of the limitations imposed by the object-oriented programming style might yield a significant speed-up. The developers of the Core-library therefor plan to implement such an optimizing compiler. Some of the optimization ideas and the results achieved are outlined in [26].

5.3 Using The Core-Library

We will now see how to use the Core-library in own software projects. First, we take a look at the statements necessary to prepare, compile and link a program with the Core-library. Then we mention basic guidelines which are necessary to make a program compliant with the Core-library. Finally, we present the Core-based version of the program from Page 15. We assume the GNU g++ compiler throughout this section.

5.3.1 Building Programs That Use the Core-Library

Besides performance the developers of the Core-library also emphasized ease of use. Thanks to the promotion and demotion mechanism that substitutes the standard data types with the Core data types and vice versa, and operator overloading, only minimal changes are required to convert existing C/C++ programs.

In the simplest case the software developer just has to do the following:

Set the accuracy level: The accuracy level is set by inserting the following #define statement before the code starts:

#define <Level_number>,

where Level_number = 1,2,3 or 4. Since Level I and Level III are the only levels that are fully implemented in Version 1.4 of the Core-library, Level II and Level IV cannot be used at present.

Include the header file: To include the header file for the Core-library one has to place the

#include <CORE.h>

statement after all the standard #include statements but before the code starts.

Build the program: In order to build Core-based programs, the include path and the library path need to be specified appropriately with the -I and -L compiler flags. Then the program is linked with the Core-library and the big number package gmp by specifying the -lcore and -lgmp flags. Assuming one wants to link the Core-library with the program foo.c, the command for building foo.c has to look like this:

```
g++ -I$(CORE_PATH)/inc foo.c -o foo -L$(CORE_PATH)/lib -lcore
                              -lgmp -lm,
```

where $(CORE_PATH) is the directory where the Core-library is installed.

Unfortunately, the situation is not so simple in practice. There are a number of additional issues, see [30], that have to be considered if one wants to link the Core-library with existing programs. We will describe these issues in the next subsection.

5.3.2 Converting Existing Programs

We have already mentioned that in the simplest case it is sufficient to set the accuracy level and include the Core-library's header file appropriately. In general, this is only true for very simple programs. Thus, there are some basic guidelines that have to be considered if an existing program is linked with the Core-library. Furthermore, a new program has to be implemented according to these principles in order to be compliant with the Core-library. We take a look at these guidelines in this subsection.

Assume error-free results and comparisons: Since existing programs often take precautions to prevent inconsistency and non-robustness due to numerical errors this fundamental rule is violated and can cause problems. A possible solution to solve problems with the commonly used ϵ-tweaking technique is to set $\epsilon = 0$.

Preventing promotion: All the variables of the machine data types `long` and `double` that occur after the preamble are promoted to `Expr`[5] in Level III. If one still wants to use the machine data types instead, one has to resort to the data types `machine_long` and `machine_double` as a replacement of `long` and `double`.

Initialization: All the objects that are explicitly declared to be of type `Expr` or automatically promoted to `Expr` have to be initialized. For dynamically allocated `Expr` objects, correct initialization is only performed if the `new` operator is used. That is, `malloc` does not work in conjunction with Level III objects.

Using library operations: The machine data types `int` and `float` are never promoted to `Expr`. This means that the normal `sqrt` operation of math.h is used in the following example:

int $i = 2$;
double $x =$ sqrt(2);
double $y =$ sqrt(i);

Since i is of type `int`, y is computed using the standard `sqrt` operation. The situation for x is similar. A possible solution is given in the following code fragment:

int $i = 2$;
double $z =$ i;
double $x =$ sqrt(Expr(2));
double $y =$ sqrt(z);

Since z is of type `double` it is promoted to `Expr` and y is computed with the `sqrt` function of the Core-library. The variable x is also computed exactly because of the explicit promotion of the constant 2.

Literals and constant arithmetic expressions: Literals and constant arithmetic expressions are not promoted. E.g.:

[5]A similar promotion occurs in Level II.

double $x = 2/3$;
double $y = 2.0/3$;
double $z = 1.3$;

In this example the value of x is 0 because the standard integer division operator is used here. The value of y is an approximation to 2.0/3 since the standard division operator for floating-point numbers is used to compute this fraction. A possible solution would be:

double $x =$ Rational(2/3);
double $y =$ Rational(2/3);
double $z =$ "1.3";

In general, constant literals have to be placed in quotation marks and the global variable *defInputDigits* needs to be set to ∞ in order to represent numerical values exactly.

5.3.3 An Example of a Simple Core-Based Program

We will now present the Core-based version of the program from Page 15. The program is shown in Figure 5.3. Note, that the Core-based version looks almost similar to the original program. The only differences are:

1. The inserted preamble (accuracy level and header file), and
2. the constant literals that have to be placed in quotation marks in order to enable exact comparisons. Furthermore, we use the function `setDefaultInputDigits` to set the global variable *defInputDigits* to infinity [6].

Since no Core objects are written, the standard output function `printf` can still be used here. Fortunately, there is a major difference to the floating-point version of the program in the result computed. Due to the use of exact arithmetic, the program prints

x == 1.0

which is the correct result while the floating-point version of the program prints

x != 1.0.

[6]If *defInputDigits* is not set to infinity then this Core-based version shows the same incorrect behavior as its floating-point sibling: it also outputs the string "x != 1.0"!

```
#define Level 3
#include <stdio.h>
#include <CORE.h>
int main()
{
        setDefaultInputDigits(CORE_posInfty);
        double x = "0.0";
        int counter;

        for(counter = 0; counter < 10; counter ++)
          x = x + "0.1";

        if (x == "1.0")
          printf("x == 1.0\n");
        else
          printf("x ! = 1.0\n");

        return 0;
}
```

Figure 5.3: The Core-based version of the program from Page 15.

This ends our discussion of the Core-library. For more information on how to use the library in own projects, see the Core-library tutorial [30] which is part of the full distribution of the Core-library and can be downloaded together with the library from [2].

Chapter 6

Linking FIST with the Core-Library

This chapter provides an introduction to the triangulation algorithm FIST in Section 6.1. Steps for making FIST compliant with the Core-library are presented in Section 6.2. Section 6.3 presents experimental results. We conclude in Section 6.4.

6.1 A Survey of FIST

FIST [21], an acronym for fast industrial-strength triangulation, is a triangulation algorithm for polygons including polygons with islands and polyhedral faces based on ear clipping. FIST uses standard floating-point arithmetic and was designed not to crash or loop. Furthermore, FIST also handles degenerate input data. Thus, FIST always produces a topologically valid triangulation of a polygon P, i.e., the vertices of P form the vertices of the triangulation, every edge of every triangle is shared with one other triangle or is on the border of P, and every edge of P belongs to exactly one triangle. We explain the basics of the ear clipping algorithm and discuss the heuristics needed to guarantee the robustness of FIST in this section.

6.1.1 The Ear Clipping Algorithm

We now explain the ear clipping algorithm used in FIST. Similar to [21], in this discussion we assume that the polygon P is simple and oriented counterclockwise (CCW). With FIST a polygon is triangulated by clipping ears. Three vertices v_{i-1}, v_i and v_{i+1} form an ear of the polygon P if v_i is a convex vertex and the line segment $[v_{i-1}, v_{i+1}]$ is a diagonal of P, i.e., the open line segment (v_{i-1}, v_{i+1}) is completely contained in the interior of P, see Figure

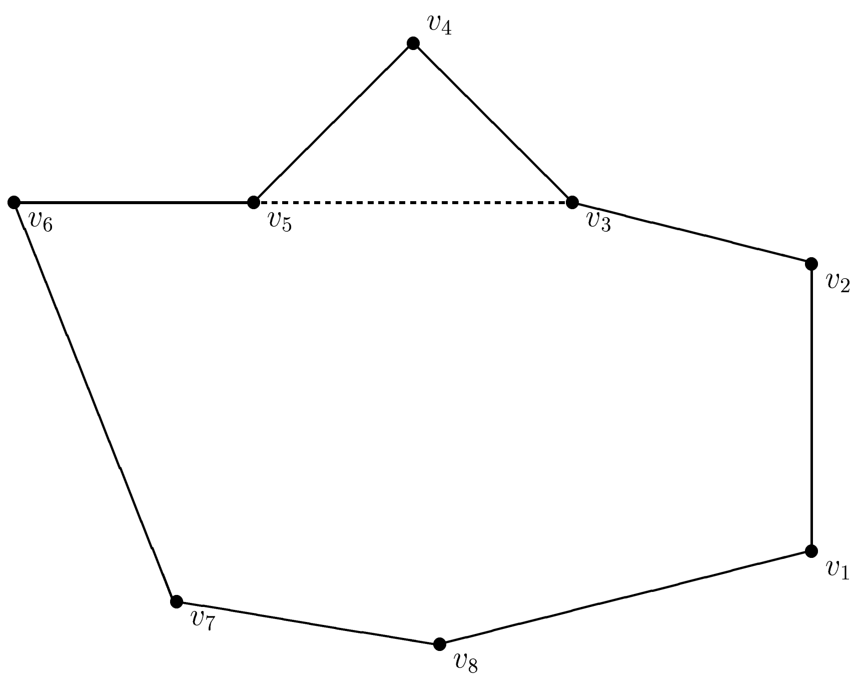

Figure 6.1: The vertices v_3, v_4 and v_5 form an ear of the polygon.

6.1. By clipping an ear formed by v_{i-1}, v_i and v_{i+1}, the ear is replaced by the line segment $[v_{i-1}, v_{i+1}]$. Thus, the number of vertices of P is reduced by one every time an ear is clipped. This process is terminated when the polygon is reduced to a single triangle, which forms the last triangle of the triangulation.

The implementation of the ear clipping algorithm in FIST is executed in five steps:

Step 1: First, in a preprocessing step, the vertices of the polygon are sorted lexicographically according to their x and y-coordinates. Then every vertex is assigned a unique index according to its position in the sorted array. Vertices with identical coordinates have identical indices. Due to those unique indices, comparisons of vertices are reduced to comparing their corresponding indices which are integers. In this sense, comparisons between vertices can be done error-free[1] once the vertices have been sorted.

Step 2: Determine the orientation of all polygonal loops. In case of multiply-connected polygonal areas, also determine the outer contour.

[1]Of course, misjudgments can be made in the course of the sorting process since floating-point values are compared. Nevertheless these misjudgments are used consistently throughout the program.

Step 3: In Step 3 all the vertices of the polygon P are classified as convex or reflex.

Step 4: In the fourth step, every consecutive triple of vertices is checked for earity, i.e., if they form an ear. Detected ears are marked and stored.

Step 5: In the final step one of the previously stored ears is clipped and stored in the triangulation. As already mentioned above, clipping an ear of a polygon with n vertices yields a polygon with $n-1$ vertices. Furthermore, every previously detected ear is still valid except for up to two ears that involve the clipped vertex. Thus, if the ear v_{i-1}, v_i, v_{i+1} is clipped, the ears involving the vertex v_i are not valid any more. Instead, the vertex triples v_{i-2}, v_{i-1}, v_{i+1} and v_{i-1}, v_{i+1}, v_{i+2} have to be checked for earity. Step 5 is carried out until the original polygon has been transformed into a triangle.

In [21] Held formulates two different sets of conditions that are necessary and sufficient to detect an ear. Ear detection is carried out according to both sets, which yield two different worst-case time complexities. One of those sets yields an ear detection process that is sensitive to the shape of the polygon: for a polygon with r reflex vertices its worst-case time complexity is $O(n \cdot (r+1))$. The worst-case complexity of the ear detection process according to the second set depends on the vertices of the polygon only, which results in $O(n^2)$ for a polygon with n vertices. For details on the ear detection process and the condition sets mentioned above see [21].

6.1.2 Extending the Basic Ear Clipping Algorithm

The basic ear clipping algorithm is extended to handle multiply-connected polygonal areas. This is achieved by linking the inner loops with the outer contour by so-called contour bridges. Basically, a contour bridge is a doubled diagonal which links two different boundary loops. Contour bridges are found by determining the left-most vertex of each polygonal loop. Next, the inner loops are sorted according to their left-most vertex. Recall that this can be done error-free, since it is sufficient to compare the indices of the vertices – see Step 1 of the ear clipping algorithm. Finally, starting from the left-most inner loop, all inner loops are connected to the outer boundary. For details on this process see [21].

6.1.3 Ensuring Robustness

A number of heuristics are implemented to support the basic ear clipping algorithm. There are heuristics to soften the problems that arise whenever floating-point arithmetic is used. Furthermore, heuristics are needed to cope with degeneracies. In order to handle degenerate input data, the ear clipping process has to be adopted appropriately. E.g., an ear v_{i-1}, v_i, v_{i+1} where v_i coincides either with v_{i-1} or v_{i+1} is still considered a valid ear. In general, degenerate ears are clipped first. Thus, for input that exhibits only little degeneracies, the algorithm eliminates those degeneracies quickly and continues with the standard ear clipping process. Of course, there are a number of additional degeneracies that have to be treated as special cases. We will omit a thorough discussion at this point and concentrate on general concepts implemented in FIST in order to improve robustness of the triangulation algorithm. The reader is referred to [21] for more information on the treatment of special cases.

Consistent Primitive Operation

As pointed out in [21], FIST is based on a single predicate namely the orientation test, which is evaluated by computing the sign of a 3×3 determinant. As we have pointed out on Page 32 the sequence of the input values passed to the determinant evaluation function does matter, i.e., interchanging the order of operands might yield different results. Since all the vertices are assigned a unique index in Step 1 of the ear clipping algorithm, the determinant is evaluated by ensuring that the index of the first argument is smaller then the index of the second argument which is again smaller then the index of the third argument, i.e., if the vertices v_i, v_j and v_k are passed to the determinant evaluation function, care is taken that $i < j < k$ holds. This reordering of the vertices might change their cyclic order. If so, the sign of the determinant computed is inverted. Furthermore, epsilon tweaking is used to increase robustness of the determinant evaluation function.

Determining the Orientation and the Outer Boundary of Polygonal Loops

Since real-world data is likely to have all kinds of deficiencies including incorrectly specified orientations, FIST does not rely on the orientation specified in the input file. Instead, the orientation of a polygon is determined by computing its signed area, denoted by $A(P)$. The signed area is computed by determining the signed areas of each triangle $\Delta(v_0, v_i, v_{i+1})$ and subsequent summation of the results. There are three possibilities:

$A(P) < 0$: The polygon is oriented clockwise (CW),

$A(P) = 0$: The polygon is twisted or collapsed to a chain and a random orientation is chosen,

$A(P) > 0:$ The polygon is oriented counterclockwise (CCW).

In case of multiply-connected polygonal areas, the outer contour has to be determined. According to [21] this is achieved with the following heuristic: The polygonal loop, which has the biggest absolute area is considered the outer contour. After the outer contour is determined it is oriented CCW, while all the inner loops are oriented clockwise.

Classifying Internal Angles

The classification of internal angles formed by v_{i-1}, v_i, and v_{i+1} is again reduced to the orientation test, i.e., the evaluation of the sign of a 3×3 determinant. Depending on the sign computed there are three different cases:

Positive Sign: If the sign computed is positive, then the internal angle at v_i is convex.

Negative Sign: A negative sign indicates that the internal angle at v_i is reflex.

Determinant equals zero: If the determinant equals zero, then the internal angle is either $0°$, $180°$ or $360°$. The angle is $180°$ if the dot product of the vectors determined by $v_i v_{i-1}$ and $v_i v_{i-1}$ is less than zero. The remaining cases cannot be determined locally. Thus, one has to move away from v_i in both CW and CCW direction until two non-overlapping segments are found. The angle at v_i is then determined by a lengthy case study.

As usual the determinant is evaluated by passing it the vertices in increasing order of their indices as arguments.

The Multi-Level Recovery Process

Despite all the efforts taken to find a valid ear, there are self-intersecting polygons, where the algorithm runs out of ears before the polygon is completely triangulated. A multi-level recovery process together with a so-called desperate mode is implemented to handle such situations. Simply speaking, the triangulation process is restarted but with more aggressive methods for the ear finding process. There are four different levels:

1. In the first level, there is a reclassification of ears. Since some of the reflex vertices may already have disappeared due to previous ear clipping, this may reveal new valid ears that have not been valid before.

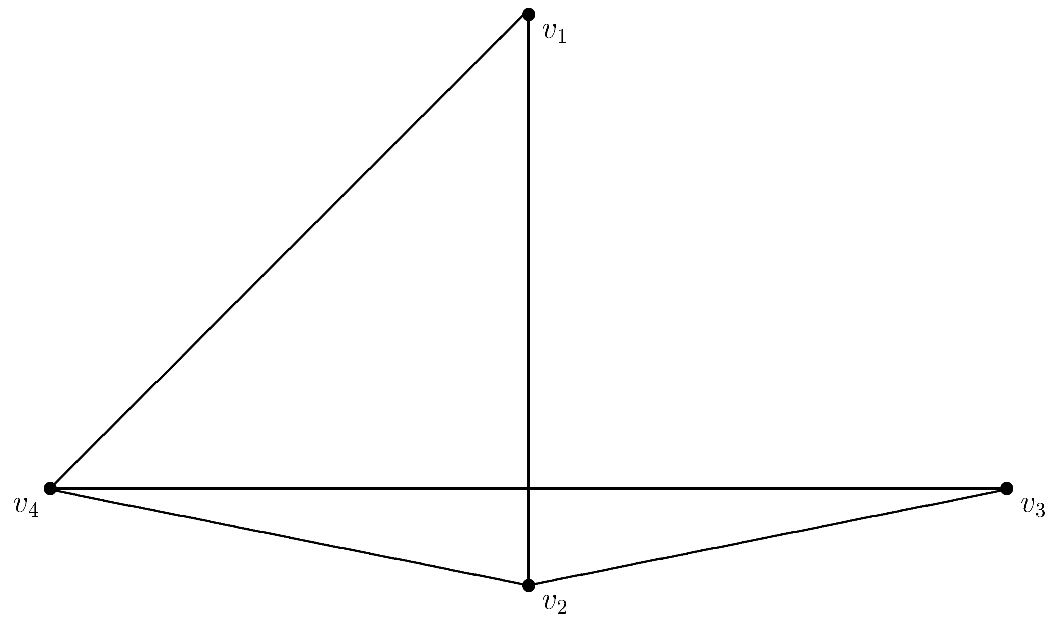

Figure 6.2: A self-intersecting polygon.

2. In the second level FIST checks the polygon for self-intersections. E.g., the open line segments of two edges (v_{i-1}, v_i) and (v_{i+1}, v_{i+2}) intersect. If so, then the triangles $\Delta(v_i, v_{i+1}, v_{i+2})$ and $\Delta(v_{i-1}, v_i, v_{i+2})$ are clipped. For the example polygon in Figure 6.2, the triangles $\Delta(v_2, v_3, v_4)$ and $\Delta(v_1, v_2, v_4)$ are clipped. As pointed out in [21], computing the point of intersection of the line segment $[v_1, v_3]$ with the line segment $[v_2, v_4]$ would result in a gap for 3D polyhedral input. Thus, clipping the triangles as described above is a reasonable solution to this problem.

3. In the third level, the polygon is split into sub-polygons by inserting a diagonal. Then, the resulting polygons are triangulated. It has to be noted, though, that not every valid diagonal helps to improve the situation. Thus, care has to be taken to choose a suitable one, see [21] for more information.

4. If all of the above levels fail to find new ears, then FIST enters a so-called desperate mode. In desperate mode, a random convex vertex or, alternatively, if no convex vertex exists, a random reflex vertex is chosen and the corresponding ear is clipped.

It is important to stress that FIST tries to resume the standard triangulation process as soon as possible, e.g., if new ears are found in Level 1, then they are clipped and the code resumes the standard ear clipping algorithm and does not remain any of the subsequent levels. Similarly, FIST tries to leave desperate mode as soon as possible.

Speeding Things Up

Depending on the ear detection process, see Page 76, used for the triangulation, the worst-case time complexity for FIST is $O(n^2)$ or $O(n\cdot(r+1))$, where n is the number of vertices and r is the number of reflex vertices of the input polygon. Therefor, the practical CPU time consumption would increase tremendously with increasing number of vertices if no countermeasures are taken.

FIST uses geometric hashing to overcome this deficiency. In [21], Held evaluates bv-trees and grids to improve the practical running time. Experiments showed that grids are faster than bv-trees and yield a slightly super-linear time consumption in practice. For more information on geometric hashing in FIST and experimental results see [21].

6.2 Making FIST Compliant with the Core-Library

The goal of our work is to modify FIST in a way such that it is able to work with exact arithmetic provided by the Core-library. Furthermore, we wanted to evaluate if the Core-library is easy to use in existing programs and reasonably fast as claimed by the developers of the library. Thus, we tried to

1. make as minimal changes to FIST as possible, and
2. to achieve maximal speed.

Besides the evaluation of ease of use, the motivation for Design Goal 1 is to keep FIST consistent, i.e., a user should be able to build both the floating-point and the exact version of FIST. Thus, we used the C preprocessor and included all the Core-related changes in the following preprocessor statement:

```
#ifdef LIB_CORE
    ⋮
#endif
```

Furthermore it is important not to introduce any errors by the changes necessary to use the Core-library. In order to achieve Design Goal 2, we tried to use the Core-library in those parts of FIST only where it is necessary. Of course, there were situations where a trade-off between Design Goal 1 and Design Goal 2 was necessary.

The developers of the Core-library claim that is easy to use in both new and existing programs and fast enough to be a reasonable alternative to floating-point arithmetic. As stated above, we wanted to evaluate those claims. As we have already mentioned in the previous chapter, one requirement for Core-based applications is to design them assuming exact arithmetic and error-free decisions. Clearly, this is not the case with the standard version of FIST since it was designed to cope with all kinds of problems that arise in floating-point arithmetic. Thus, it is interesting to see if the measures that ensure robustness in the floating-point version of FIST cause any problems in the exact version of FIST.

In the following subsection we explain the changes we made in order to create a Core-based version of FIST. Since we use the Core-library in Level 3 only, all `double` and `long` variables are promoted to the Core data type `Expr`.

6.2.1 Adapting the I/O-Routines of FIST

Since the standard C input and output routines cannot be used with Core objects, they have to be adapted appropriately. Clearly, all the input values need to be represented by Core objects. On the other hand there was no need to produce an exact output since the graphical output is based on integer coordinates and since FIST does not create new geometric objects. In addition, rounding exact values back to a finite representation without producing errors similar to floating-point arithmetic is still an open problem. Thus, we decided to use the `Expr` class member `doubleValue`[2] to extract the machine double value from an exact `Expr` value and used the standard output routines. As we have mentioned above, the situation is different in the case of input values. All the input values need to be represented by Core objects in order to guarantee exact comparisons. Basically, there are two possible solutions to this problem:

1. One way to solve this problem is to rewrite every input routine from the scratch using the C++ input stream `cin`.

2. A second possibility is to use the fact that Core objects can be initialized using string literals. Thus, all input values can be read in as

[2]Note that the use of the member `doubleValue` can cause silent overflows and underflows.

```
void ReadPolygon(...)
{
    char Str_xc1[STR_LENGTH];
    char Str_yc1[STR_LENGTH];
    char Str_xc2[STR_LENGTH];
    char Str_yc2[STR_LENGTH];
    double xc1, yc1, xc2, yc2;

            ⋮

    if ( EOF == fscanf(inputfile, "%s %s %s %s", Str_xc1, Str_yc1, Str_xc2, Str_yc2) )
        /* print an error message */
    else
    {
        xc1 = Str_xc1;
        yc1 = Str_yc1;
        xc2 = Str_xc2;
        yc2 = Str_yc2;
    }

            ⋮

}
```

Figure 6.3: An example of an input function in the Core-based version of FIST.

strings using the standard input routines.

We opted for the second possibility since this allowed us to read the exact input values without rewriting every input routine. The code fragment in Figure 6.3 demonstrates this approach. Recall, that the `double` variables $xc1$, $yc1$, $xc2$ and $yc2$ are promoted to `Expr` when the Core-library is used. Instead of reading `double` values, we read strings which are assigned to the corresponding `Expr` objects in a subsequent step. Thus, the `Expr` objects are properly initialized and hold the exact values of the input data if the global variable *defInputDigits* is set to infinity.

Allocation method	floating-point arithmetic	exact arithmetic
STL Vector	12 ms	294 ms
Copy (memcpy)	24 ms	segmentation fault
Copy (for loop)	36 ms	1401 ms

Table 6.1: Dynamic memory allocation for 65000 elements using block size 8125.

6.2.2 Dynamic Memory Allocation

Another major issue is dynamic memory allocation. Core objects have to be allocated using the C++ operator `new` in order to be initialized properly. The data structures of the floating-point version of FIST are built on top of arrays which are allocated by calling the function `ReallocateArray`. If `ReallocateArray` is called for the first time, a memory block is allocated using the C function `calloc`. If necessary the memory block is extended using the C function `realloc` in subsequent calls to `ReallocateArray`. Again, we had two possibilities to adapt dynamic memory allocation:

1. The STL data type `vector` could be used instead of arrays. Basically, the data type `vector` implements a dynamic array. It supports random access, new elements can be inserted, and old elements can be deleted efficiently at the end of the `vector`. Memory is automatically reallocated if this is necessary.

2. An alternative method is to allocate a new and bigger array. Then, the elements of the old array are copied to the new array and the old array is deleted.

The second method can be implemented using the C function `memcpy` or, alternatively, by iterating through the old array with a `for` loop. In order to detect any performance differences between these methods, we implemented a test program that allocates memory for 65000 elements with different block sizes according to the methods described above and measured the CPU time consumption. The results for the block sizes 8125, 16250 and 32500 are summarized in Table 6.1, Table 6.2, and Table 6.3 respectively. Note that the CPU time consumption for the solution with the STL `vector` data structure is independent from the block size used for memory allocation, since the allocation process is managed automatically.

It turned out that using the STL `vector` data structure for memory allocation is the fastest method. However, it does not fit well into the framework of FIST. Thus, there are some major changes that have to be made to the

Allocation method	floating-point arithmetic	exact arithmetic
STL vector	12 ms	294 ms
Copy (memcpy)	12 ms	segmentation fault
Copy (for loop)	13 ms	837 ms

Table 6.2: Dynamic memory allocation for 65000 elements using block size 16250.

Allocation method	floating-point arithmetic	exact arithmetic
STL vector	12 ms	294 ms
Copy (memcpy)	8 ms	540 ms
Copy (for loop)	6 ms	544 ms

Table 6.3: Dynamic memory allocation for 65000 elements using block size 32500.

code in order to use the STL `vector` data structure. Furthermore, the performance difference is not that dramatic for large block sizes. We therefor opted for the second method and chose to copy array elements in a `for` loop, since the variant with `memcpy` suffered from occasional segmentation faults, see Table 6.1 and Table 6.2.

There is another interesting fact that has to be mentioned. Looking at Table 6.1, Table 6.2 and Table 6.3 one notices a dramatic slow-down in the memory allocation of `Expr` objects compared to the allocation of `double` values in the floating-point version of FIST. Therefor, we can expect a significant slow-down in the overall running time when the Core-library is used.

In order to implement a generic function for memory allocation, we utilized C++ templates. Therefor, we were able to implement a single function called `ReallocateArray_Copy` which can be used in the floating-point version as well as in the Core-based version of FIST. Thus, a performance comparison can be done using the same type of memory allocation. Summarizing, the code for the function `ReallocateArray_Copy` is shown in Figure 6.4.

Of course, we need an additional function as a replacement for the function called `FreeMemory` which is used in the floating-point version of FIST to free the memory that was previously allocated using `ReallocateArray`. Therefor, we implemented another generic function called `FreeMemory_Copy` which uses the C++ operator `delete`, see Figure 6.5.

It is important to stress that for performance reasons we use the function `ReallocateArray_Copy` only where it is necessary, i.e., whenever memory is allocated for Core objects.

```
template <class DATATYPE>
DATATYPE* ReallocateArray_Copy
(DATATYPE *old_mem, int old_size, int new_size, size_t size, char var_name[])
{
        DATATYPE *new_mem = NULL;

        if (old_mem! = NULL)
        {
                new_mem = new DATATYPE[new_size];
                if (new_mem != NULL)
                {
                      for (int i = 0; i < old_size; i + +)
                            new_mem[i] = old_mem[i];
                      delete []old_mem;
                }
                else
                {
                      /* Print an error message */
                      exit(1);
                }
        }
        else
        {
                new_mem = new DATATYPE[new_size];
                if (new_mem == NULL)
                {
                      /* Print an error message */
                      exit(1);
                }
        }

        return new_mem;
}
```

Figure 6.4: The function `ReallocateArray_Copy` is used for dynamic memory allocation in the Core-based version of FIST.

```
template <class DATATYPE>
voidFreeMemory_Copy(DATATYPE** ptr)
{
    if (*ptr == NULL) return;

    delete []*ptr;
    *ptr = NULL;
}
```

Figure 6.5: The function `FreeMemory_Copy` is used to free dynamically allocated memory in the Core-based version of FIST.

6.2.3 Constant Literals

Constant literals are not promoted. If, for example, the statement

$$\text{double } x = 0.1;$$

is used to assign the value 0.1 to the variable x, then x holds only an approximation of the assigned value. This can be changed by using the statement

$$\text{double } x = \text{"0.1"};$$

instead. Therefor, we had to change all constant literals in FIST appropriately. Note, that enclosing every constant literal in quotation marks would not be a feasible solution, since this prevents the successful compilation of the floating-point version of FIST. We therefor labeled all the constant literals that involved Core objects and placed them in a header file as follows:

```
#ifdef LIB_CORE

#define C_0_0     "0.0"
#define C_0_01    "0.01"
#define C_0_1     "0.1"
          ⋮

#else

#define C_0_0     0.0
#define C_0_01    0.01
#define C_0_1     0.1
          ⋮
#endif
```

Depending on whether the Core-library is used or not, the labels are replaced by the corresponding exact or approximate values. Furthermore, the function `setDefaultInputDigits` is used to set the global variable *defInputDigits* to infinity.

6.2.4 Setting all Epsilons to Zero

One requirement for a program to be compliant with the Core-library is that it is implemented assuming exact arithmetic. This requirement conflicts with the epsilon tweaking technique used to increase robustness in the floating-point version of FIST. Therefor, all epsilons have to be set to zero while care has to be taken that this measure does not cause divisions by zero. Similar to the solution used for constant literals, we set all epsilons to zero if the compiler option THRESHOLD is not specified. Thus, the compilation of the Core-based version of FIST requires the compiler flags LIB_CORE and MEM_COPY while it is necessary to specify the compiler flag THRESHOLD for the floating-point version of FIST in order to enable epsilon tweaking. Note that setting all epsilons to zero would change the semantics of the program if the comparison of a numerical value x with epsilon is performed such that

$$|x| < \epsilon.$$

Thus, care has to be taken that all the epsilon-related comparisons are carried out with $\leq$ instead of $<$, i.e,

$$|x| \leq \epsilon.$$

6.2.5 Miscellaneous Adaptions

There are a number of minor changes that had to be made in order to make FIST compliant with the Core-library. First of all there are some variables of the types `double` and `long` that should not be promoted because they are only used to model floating-point heuristics that do not affect the correctness of the program. In order to prevent their promotion, we replaced them with the Core-library's data types `machine_long` and `machine_double` which represent the corresponding machine data types. The preprocessor statements

```
#ifndef LIB_CORE
    #define machine_double double
    #define machine_long long
#endif
```

make sure that this adaption is still compatible with the floating-point version of FIST. Furthermore, statements like

$$i = (\text{int})\ x;$$

where i is an integer variable and x is a `double` variable had to be replaced by

$$i = x.\text{intValue}();$$

where `intValue` is a member function of the Core data type `Expr` that returns the integer value of the `Expr` variable. As we have already mentioned before, the use of the `Expr` member functions `intValue`, `longValue`, `floatValue` and `doubleValue` might cause silent overflows and underflows and have to be used with care as stated in the Core-library tutorial [30]. Nevertheless there were situations where we had to use them. If, for example, the output computed needs to be displayed on the screen, it has to be converted to an integer and there are many other situations where we cannot get around this problem.

Finally, the statements

```
#define Level 3
#include <CORE.h>
```

had to be included after the include statements of the standard header files in every source file that calls functions of the Core-library. We included these statements in a separate include file which is then included in the FIST source files. Thus, there is one central file where we can change the behavior of the Core-library, e.g., change the accuracy level.

6.3 Experimental Results

We now present experimental results. Basically, we tested different versions of FIST with different classes of input data and measured the CPU time consumption. Unfortunately, the Core-based version of FIST misbehaved badly[3] on some of our input data. Therefor, we tried to get to the bottom of this and used a standard and a memory debugger to see what causes these problems. Based on the results from those debuggers, we expected memory problems within the Core-library, which we reported to the developers. We then got an improved version of the library which did not solve all the problems. Thus, we repeatedly sent bug reports, got new versions of the library, and finally provided the source code of our Core-based version of FIST to

[3]We explain the details in the following subsections.

Input class	Range of segments
"random"	8 – 32768
"smooth"	16 – 32768
"smoother"	64 – 32768
"thinned"	8 – 8192

Table 6.4: Segment ranges for the four different input classes.

the Core-library developers to see if we made any obvious mistakes. Despite all the efforts made by the developers of the Core-library and on our end, we were not able to solve all the problems with the Core-based version of FIST. In the following we report the CPU time consumptions for the data sets that could be processed correctly and the problems we had in the remaining cases.

6.3.1 Two-Dimensional Test Data

We set up our experiment similar to [21]. That is, we have four different classes of input data which were generated by the RANDOM POLYGON GENERATOR (RPG), see [5] for more information on RPG. The first class of our input data is called "random" and is generated by RPG by distributing points uniformly in the unit sphere. The second class, called "smooth", is generated by applying the Smooth algorithm of RPG twice to the polygons of the "random" class. The Smooth algorithm doubles the input polygon's number of vertices by replacing each vertex v_i with two new vertices $\frac{v_{i-1}+3v_i}{4}$ and $\frac{v_{i+1}+3v_i}{4}$. The third class of our test data, called "smoother", is generated by applying RPG's Smooth algorithm four times to the polygons of the "random" class. Finally, the fourth class which is called "thinned" is computed by randomly clipping three quarters of the ears of the polygons in the "random" class. For each class, one sample polygon with 64 vertices (together with the triangulation computed by FIST) is shown in Appendix A. As pointed out in [21], the characteristics of the four classes are as follows: While the vertices of the "random" polygons are uniformly distributed, this is not true for the "smooth", "smoother" and "thinned" polygons. In addition, the "smoother" polygons tend to have very short edges and a number of long diagonals. The "thinned" polygons, on the other hand, cover, in general, less space than the polygons in the "random" class.

The "random" input class consists of polygons with the number of vertices ranging from 8 to 32768 and there are ten different polygons for each number of vertices. Thus, we get the segment ranges shown in Table 6.4 for the different input classes. We tested all four input classes with four different versions of FIST and measured the CPU time consumption. The four versions of FIST are as follows:

FIST_fp_std: This is the standard floating-point version of FIST.

FIST_fp_mod: Same as FIST_fp_std but modified to use the same memory management routines as the Core-based version of FIST.

FIST_exact_def: This is the Core-based version of FIST with the default value of the global variable *defInputDigits* which is 16.

FIST_exact_infty: Same as FIST_exact_def but with the global variable *defInputDigits* set to infinity.

As we pointed out in Subsection 6.2.2, we had to adopt the memory management routines in order to link FIST with the Core-library. As can be seen in Table 6.1 – Table 6.3, the new routines are slower than the ones used in the original version of FIST. Thus, for reasons of fairness, we wanted to compare the original floating-point version and the modified floating-point version of FIST to the exact version of FIST. The reason why we used two different exact versions of FIST is that we had some trouble, which we discuss later, running our test data with FIST_exact_infty, i.e., when the global variable *defInputDigits* is set to infinity. Thus, we also conducted tests with FIST_exact_def, where the default value of *defInputDigits* is used. It is important to stress that only FIST_exact_infty represents all input data exactly while input data is represented with a maximum absolute error of 10^{-16} if FIST_exact_def is used.

Test Results for FIST_exact_def

The tests for FIST_exact_def went quite smoothly. We measured the CPU time consumption t_{fp_std} for FIST_fp_std , t_{fp_mod} for FIST_fp_mod and t_{exact_def} for FIST_exact_def in milliseconds, and computed the factors

$$F_{std} = \frac{t_{exact\_def}}{t_{fp\_std}} \text{ and } F_{mod} = \frac{t_{exact\_def}}{t_{fp\_mod}}.$$

The results for the different input classes are shown in Table 6.5 – Table 6.8. Since the CPU time consumption of the floating-point versions of FIST is less than 10 ms for polygons with 8 to 1024 segments and we measure the CPU time consumption accurate to 10 ms, we only present results for polygons with at least 2048 segments. Finally, Table 6.9 shows the CPU time consumption for FIST_exact_def for all input classes.

As expected, the difference between the CPU time consumption of the standard and the modified floating-point version of FIST is negligible and within the inaccuracy in measurement, because the block sizes used in FIST

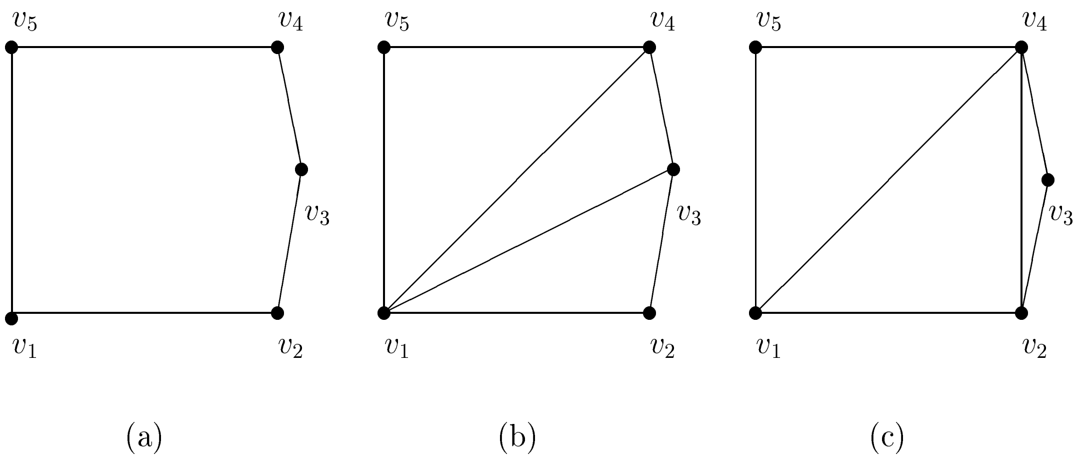

Figure 6.6: The input polygon (a); the triangulation computed with the floating-point version of FIST (b) and the triangulation computed with the exact version of FIST (c).

are large enough such that there is no need to reallocate memory repeatedly. Looking at Table 6.5 – Table 6.8, one can observe a slightly super-linear CPU time consumption for the floating-point version of FIST; see also [21]. The exact version of FIST, on the other hand, exhibits an almost linear running time. A possible explanation for this behavior is that FIST uses preprocessing that involves sorting, thus yielding a worst-case time complexity of $O(n \log n)$ for preprocessing. Due to geometric hashing, a linear average-case time complexity can be expected for the ear clipping process. Since the majority of exact calculations is executed in the course of the ear clipping process, most of the CPU time of the exact version of FIST is spent there and the time consumption for preprocessing is negligible. Thus, the CPU time consumption for the exact version of FIST is dominated by the ear clipping process and since it is growing almost linear, the ear clipping process seems to have linear time consumption in practice.

Finally, we wanted to see if there is a difference in the triangulations computed by the floating-point version and the exact version of FIST. We used a convex polygon with 5 vertices which was created by Martin Held and is depicted in Figure 6.6 (a). The x-coordinate of vertex v_3 is chosen such that the vertices v_2, v_3 and v_4 are considered collinear by the floating-point version of FIST. For clarity, we shifted v_3 to the right in Figure 6.6. The triangulations computed by FIST_fp_std and FIST_exact_def are shown in Figure 6.6 (b) and Figure 6.6 (c) respectively. While the floating-point version of FIST cannot distinguish the x-coordinates of the vertices v_2, v_3 and v_4 and considers them as collinear, the exact version of FIST is able

Number of segments	t_{fp_std}	F_{std}	t_{fp_mod}	F_{mod}
2048	8	321	10	257
4096	17	302	19	271
8192	45	231	46	226
16384	119	178	118	180
32768	274	156	272	157

Table 6.5: Random: CPU time consumption in milliseconds of FIST_fp_std and FIST_fp_mod and the corresponding rounded factors F_{std} and F_{mod}.

Number of segments	t_{fp_std}	F_{std}	t_{fp_mod}	F_{mod}
2048	10	302	12	252
4096	22	277	20	304
8192	53	232	54	227
16384	136	182	135	183
32768	316	159	306	164

Table 6.6: Smooth: CPU time consumption in milliseconds of FIST_fp_std and FIST_fp_mod and the corresponding rounded factors F_{std} and F_{mod}.

to distinguish the x-coordinates of v_2 and v_4 from v_3 and therefor clips the triangle $\Delta(v_2, v_3, v_4)$.

Test Results for FIST_exact_infty

As already mentioned above, we experienced some major problems when we tried to process 2D data sets if the Core-library represents numerical values exactly, i.e., if the global variable *defInputDigits* is set to infinity. We observed that FIST has to resort to its multi-level recovery process and

Number of segments	t_{fp_std}	F_{std}	t_{fp_mod}	F_{mod}
2048	11	309	11	309
4096	22	315	23	302
8192	53	269	56	254
16384	138	208	138	208
32768	330	176	331	175

Table 6.7: Smoother: CPU time consumption in milliseconds of FIST_fp_std and FIST_fp_mod and the corresponding rounded factors F_{std} and F_{mod}.

Number of segments	t_{fp_std}	F_{std}	t_{fp_mod}	F_{mod}
2048	7	363	8	318
4096	18	285	18	285
8192	41	253	40	259

Table 6.8: Thinned: CPU time consumption in milliseconds of FIST_fp_std and FIST_fp_mod and the corresponding rounded factors F_{std} and F_{mod}.

Number of segments	random	smooth	smoother	thinned
8	26	–	–	9
16	19	63	–	19
32	37	72	–	39
64	73	88	93	76
128	152	177	213	148
256	312	365	402	308
512	619	730	821	623
1024	1286	1482	1667	1253
2048	2567	3021	3396	2542
4096	5140	6084	6939	5125
8192	10412	12284	14239	10376
16384	21219	24720	28692	–
32768	42649	50159	57951	–

Table 6.9: The CPU time consumption in milliseconds of FIST_exact_def for all input classes.

x	y
-588.00000000000000	-576.02699999999999820
-1.3000000000000000	-1.122000000000000000
-26.875000000000000	11.799545454545454545
-30.625000000000000	0.0000000000000000000
-30.625000000000000	-23.599090909090909091
-48.966666666666675	-37.190561224489783065
-49.966666666666675	-41.067602040816311862

Table 6.10: Some values for x and y for which the comparison $x > y$ goes wrong.

even enters desperate mode although the polygons are simple. Furthermore, the triangulations computed are incorrect. Finally, we found some test data where FIST terminates abnormally since an assertion fails. We tried to understand the cause of these failures and found out that FIST enters an `if` statement of the form

$$\text{if } (x > y),$$

where x is a value of the original input data and y is a value computed, that it should not have. We printed the numerical values for x and y and found out that the comparison $x > y$ goes wrong. This could be observed for a variety of different values like the ones in Table 6.10. It has to be mentioned, though, that not every comparison involving a negative number goes wrong. Thus, the Core-library does not simply ignore negative signs as it might appear at first glance.

6.3.2 Three-Dimensional Test Data

Unfortunately, we were not able to process any but the simplest 3D data sets with any of the Core-based versions of FIST. We tried some of the small 3D test samples with about 8 – 50 vertices and FIST did not terminate although we ran some of the tests over night.

Looking at the problem with a debugger, we found out that FIST does not return from a `qsort` call. This is a really strange phenomenon since `qsort` worked perfectly for the 2D data sets. Furthermore, the processing of 3D data is reduced to the 2D case by projecting and triangulating each face of a polyhedron. Thus, the only difference between the 2D and the 3D case is an additional `for` loop for iterating through all the faces of the polyhedron and the projection of each face in order to get 2D coordinates. We also wrote the projected points to a file, which we used as input for a stand-alone program which sorted the data without any problem. Then we replaced `qsort` with a self-coded bubble sort algorithm. Unfortunately, it did not make a difference, since the bubble sort did not terminate. We used a debugger again and stepped through the program. The debugger finally stopped at a source file of the Core-library. Thus, we were not able to solve the problem.

6.4 Conclusion

We wanted to evaluate the practical value of exact arithmetic in computational geometry by linking the Core-library with the triangulation algorithm FIST. We have had some trouble with the Core-library from the beginning. Thus, we sent several bug reports to the developers of the Core-library and received a number of new releases of the library. We ended up with Version 1.4, release date 15.03.2002. The experience with this release of the Core-library can be summarized as follows:

- We did not manage to compile the Core-library and the example programs on Sun workstations with the g++ or SunPro cc compiler. Previous versions of the Core-library could be successfully compiled on Sun workstations. Building the library on Linux boxes works but there are a lot of warnings.

- Operating with infinite precision, the exact version of FIST computes incorrect triangulations, resorts to the multi-level recovery process and desperate mode although the polygons are simple, and does not pass an assertion due to the comparison bug described on Page 94 for several 2D input data sets.

- 2D data sets seem to be processed correctly if we use the exact version of FIST with default precision.

- As described in Subsection 6.3.2, the exact version of FIST hangs whenever 3D data sets have to be processed with both default and infinite precision.

We spent a long time debugging FIST in order to find any bugs on our end but without success. The floating-point version of FIST works perfectly even if we use the modified memory management routines which are also used in the exact version of FIST. As a last resort, we checked the floating-point versions of FIST as well as the exact versions of FIST with a memory debugger. While the memory debugger did not report any problems for both floating-point versions, it reported tons of errors, e.g., wild pointers and memory leaks within the Core-library for the exact versions of FIST.

Based on this experience and the problems we observed with 3D and 2D data, especially the comparison bug mentioned above, we assume that there are memory problems within the Core-library. Since we experienced problems whenever values computed are involved, e.g., the problem with `qsort` whenever original input values are projected, and the comparison of an input value with a value computed, while everything seems to be fine if input values are used, it seems that something goes wrong when arithmetic operations are performed.

Due to all the problems we have come across using the Core-library it is difficult to rate ease of use, performance or the practical value of the library. Nevertheless we assume that the bugs we expect within the Core-library will be removed some day, and thus we also want to say a few words on our experience regarding ease of use and performance. From our point of view, the Core-library is easy to use. Of course, it is a substantial amount of work to link an existing program with the Core-library, especially if it is written in C. For C++ programs, there is no need to change memory management and the I/O-routines which were two major issues in our work to link FIST with the Core-library. Due to the promotion and demotion mechanism, a software developer is free to use the standard C/C++ data types he is used to. Therefor, it is easy to implement a new Core-based program from scratch if it is implemented according to the guidelines described in Section 6.2.

Regarding the performance, we can only present results for the Core-based version of FIST with default precision due to the problems with infinite precision described above. Naturally, there is a price to pay if the calculations in a program are executed with exact arithmetic. Thus, the exact version of FIST is 150 – 320 times slower than the floating-point version of FIST for our input data sets; see Table 6.5 – Table 6.8. Furthermore, an additional slow-down can be expected if a Core-based version of FIST with infinite precision is used. Thus, the practical value for industrial-strength applications is questionable at present. Nevertheless, faster computer systems, better algorithms or even hardware support for exact calculations might improve the situation in the future and exact arithmetic might be an alternative to floating-point arithmetic in future applications. Until that day, a software developer still has to struggle with all the limitations of floating-point arithmetic if he/she wants to develop industrial-strength applications.

Originally we also wanted to link the Core-library with VRONI [22]: an algorithm for computing the Voronoi diagram of a sets of points and line-segments designed by Martin Held. Since the computation of a Voronoi diagram is based on predicates and constructors, rounding the results computed back to a finite representation without introducing errors is an additional issue with VRONI that does not have to be considered with FIST. Unfortunately, we were forced to abandon those plans because of the problems we experienced with the Core-based version of FIST.

Appendix A

Sample Polygons for the Four Input Classes

The polygons shown in Figure A.1 – Figure A.4 depict samples of the polygons of our four input classes of test polygons. The figures were provided by Martin Held.

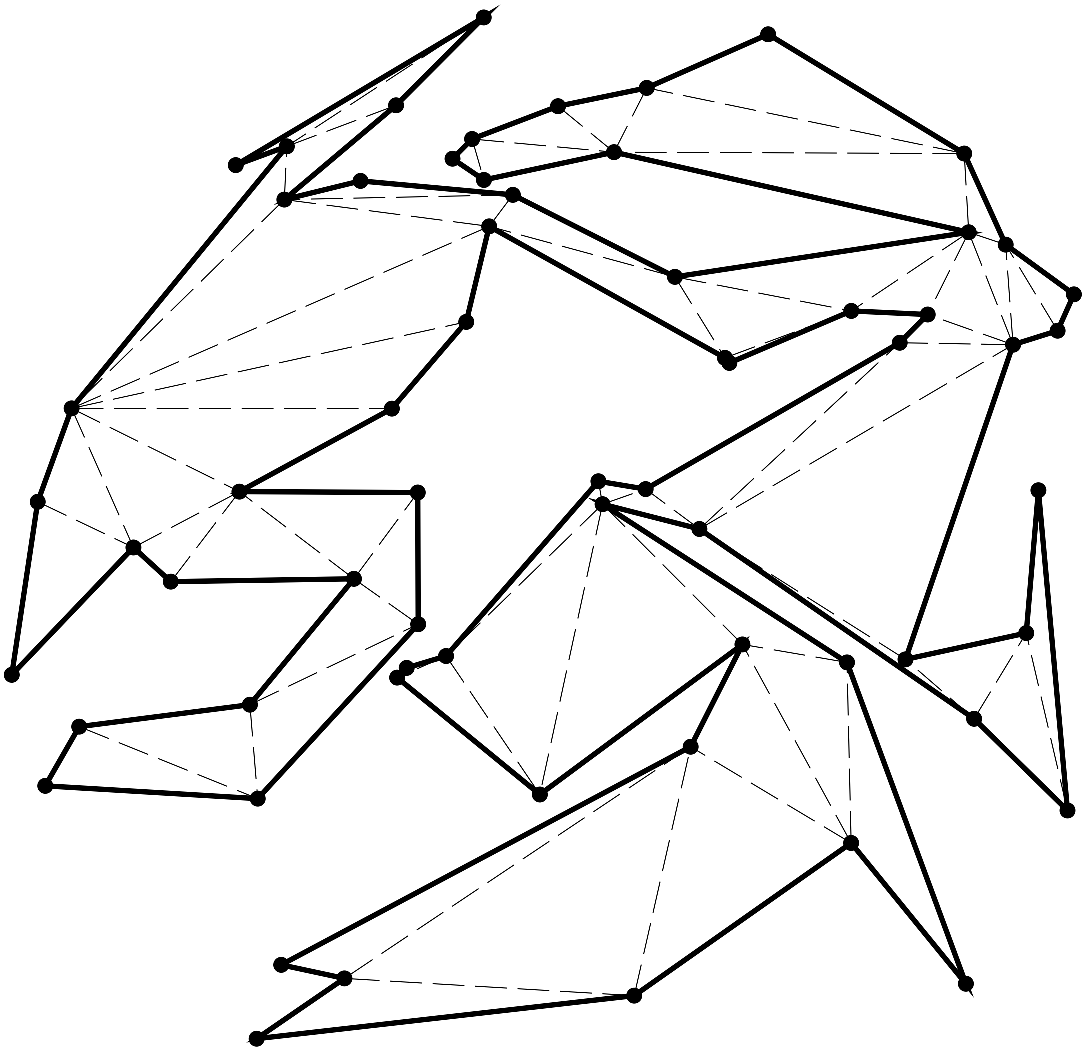

Figure A.1: Sample 64-gon for the "random" class.

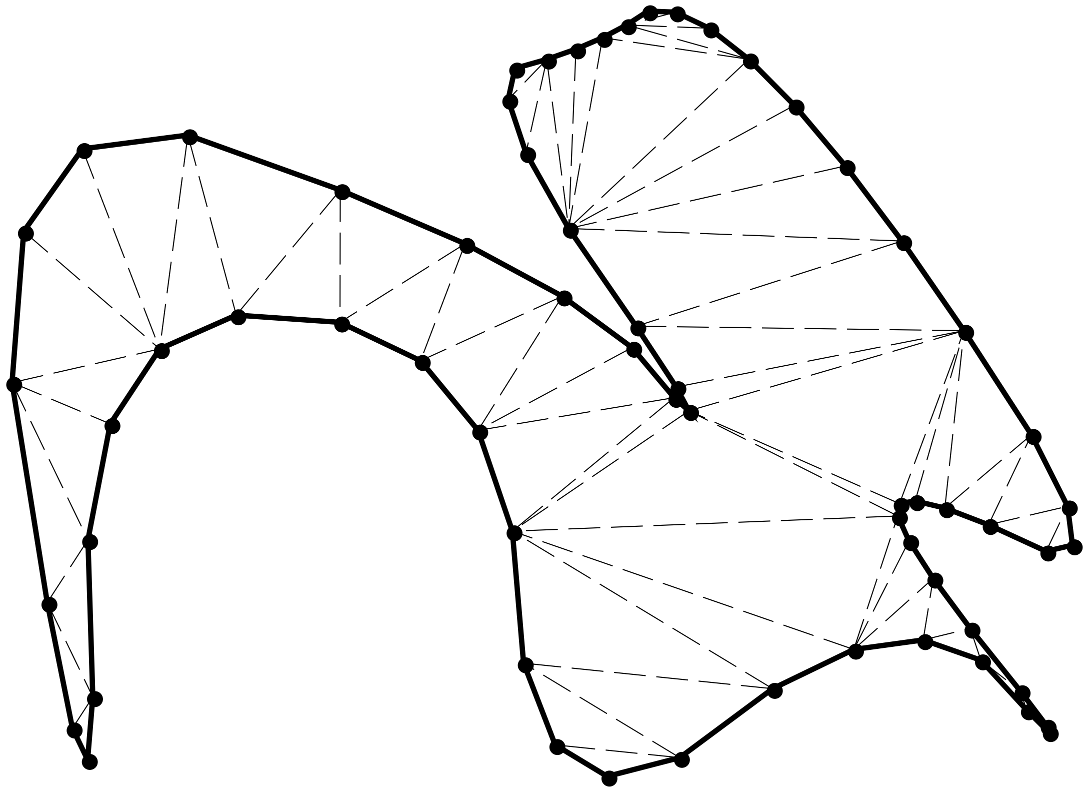

Figure A.2: Sample 64-gon for the "smooth" class.

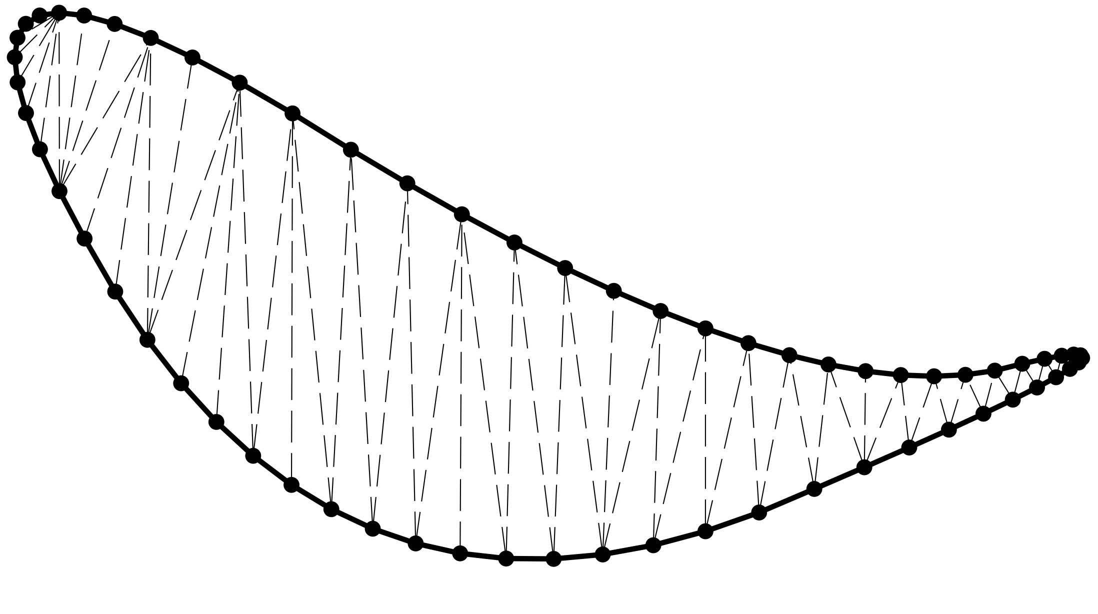

Figure A.3: Sample 64-gon for the "smoother" class.

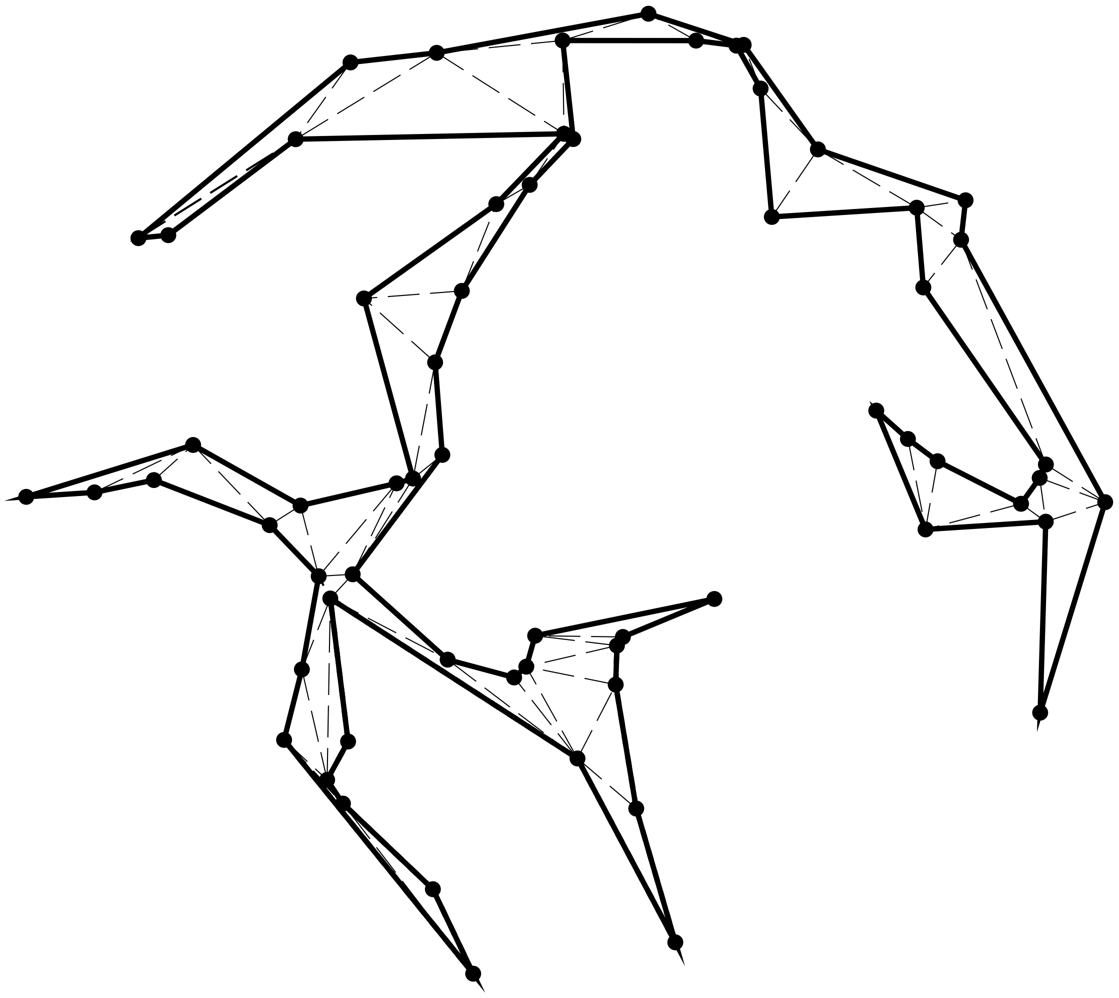

Figure A.4: Sample 64-gon for the "thinned" class.

Bibliography

[1] Homepage of the CGAL project. http://www.cgal.org.

[2] Homepage of the Core-library. http://www.cs.nyu.edu/exact/core.

[3] Homepage of the LEDA project. http://www.algorithmic-solutions.com.

[4] IEEE 1997. IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. *SIGPLAN Notices*, 2(22):9 – 25, 1987.

[5] T. Auer and M. Held. Heuristics for the Generation of Random Polygons. In *Proc. 8th Canadian Conference on Computational Geometry*, pages 38 – 44, Ottawa, Canada, 1996.

[6] M. O. Benouramer, P. Jaillon, D. Michelucci, and J. M. Moreau. A "Lazy" Solution to Imprecision in Computational Geometry. In *Proc. 5th Canadian Conference on Computational Geometry*, pages 73 – 78, Waterloo, Canada, 1993.

[7] H. Bronnimann, C. Burnikel, and S. Pion. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Proc. 14th Annual ACM Symposium on Computational Geometry*, pages 165–174, 1998.

[8] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient Exact Geometric Computation Made Easy. In *Proc. 15th ACM Symposium on Computational Geometry*, pages 341 – 450, Miami Beach, Florida, 1999.

[9] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A Strong and Easily Computable Seperation Bound for Arithmetic Expressions Involving Radicals. *Algorithmica*, 27:87 – 99, 2000.

[10] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA Class Real Number. Technical Report MPI-I-96-1-001, Max-Planck Institute for Computer Science, 1996.

[11] B. M. Bush. The Perils of Floating-Point. Lahey Computer Systems Inc., Incline Village, NV 89450, USA.
http://www.lahey.com/float.htm.

[12] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay Triangulations in Three Dimensions with Finite Precision Arithmetic. *Computer Aided Design*, 9:457 – 470, 1992.

[13] H. Edelsbrunner and E. P. Mücke. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[14] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL Kernel: A Basis for Geometric Computation. In *Proceedings Workshop on Applied Computational Geometry*, Philadelphia, Pennsylvania, May 1996.

[15] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the Design of CGAL, the Computational Geometry Algorithms Library. Technical Report MPI-I-98-1-007, Max-Planck Institute for Computer Science, 1998.

[16] S. Fortune. Stable Maintenance of Point Set Triangulations in Two Dimensions. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 494 – 505, 1989.

[17] S. Fortune. Robustness Issues in Geometric Algorithms. In *Proc. 1st ACM Workshop on Applied Computational Geometry*, pages 20 – 23, Philadelphia, PA, USA, 1996.

[18] S. Fortune. Introduction. *Algorithmica*, 27:1 – 4, 2000.

[19] S. Fortune and C. J. Van Wyk. Efficient Exact Arithmetic for Computational Geometry. In *Proc. 9th Annual ACM Symposium on Computational Geometry*, pages 163–172, May 1993.

[20] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5 – 48, 1991.

[21] M. Held. FIST: Fast Industrial-Strength Triangulation of Polygons. *Algorithmica*, 30(4):563 – 596, 2001.

[22] M. Held. VRONI: An Engineering Approach to the Reliable and Efficient Computation of Voronoi Diagrams of Points and Line Segments. *Comput. Geom. Theory Appl.*, 18:95–123, 2001.

[23] N. J. Highham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996. ISBN 0-89871-335-2.

[24] C. M. Hoffmann. The Problems of Accuracy and Robustness in Geometric Computation. *IEEE Computer*, 22(3):31 – 41, 1989.

[25] O. Hommes. MathFP: The Basics.
http://home.rochester.rr.com/ohommes/MathFP/mathfp_bg.html.

[26] V. Karamcheti, C. Li, I. Pechtchanski, and C. K. Yap. A Core Library for Robust Numeric and Geometric Computation. In *Proc. 15th Annual Symposium on Computational Geometry*, volume 15, 1999.

[27] J. Keyser. Robustness Issues in Computational Geometry. Comp 234 Final Paper, A&M University, Texas, USA, Spring 1997. http://citeseer.nj.nec.com/247595.html.

[28] D. Salesin L. Guibas and J. Stolfi. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. *In Proc. 5th ACM Conference on Computational Geometry*, pages 208 – 217, 1989.

[29] C. Li. *Exact Geometric Computation: Theory and Applications.* PhD thesis, Dept. Comp. Sci, NYU, NYU, New York, NY 10012, USA, January 2001.

[30] C. Li and C. K. Yap. *Core Library Tutorial.* NYU, New York, NY 10012, USA, January 1999. This tutorial is contained in the Core-library distribution version 1.4.

[31] C. Li and C. K. Yap. A New Constructive Root Bound for Algebraic Expressions. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*, pages 496–505, 2001.

[32] J. C. Lowery. CSC 110 – Computer Mathematics. Mississippi College, Clinton, Mississippi, USA.
http://sandbox.mc.edu/~bennet/cs110/textbook.

[33] K. Mehlhorn and S. Näher. Algorithm Design and Software Libraries: Recent Developments in the LEDA Project. In *Algorithms, Software, Architectures, Information Processing 92*, volume 1, pages 493–505, Amsterdam, 1992. Elsevier Science Publishers B.V. North-Holland.

[34] K. Mehlhorn and S. Näher. LEDA: A Platform for Combinatorial and Geometric Computing. *Commun. ACM*, 38(1):96–102, 1995.

[35] D. Michelucci. The Robustness Issue.
Internal report, Laboratoire d'Image de Synthése de St. Etienne, France.
http://www.emse.fr/~micheluc/english/michelucci.html.

[36] K. Ouchi. Real/Expr: Implementation of an Exact Computation Package. Master's thesis, Dept. Comp. Sci, NYU, NYU, New York, NY 10012, USA, 1997.

[37] F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction*. Springer-Verlag, 1990. ISBN 3-540-96131-3.

[38] W. Schiffmann and R. Schmitz. *Technische Informatik 2: Grundlagen der Computertechnik*. Springer-Verlag, 1994. ISBN 3-540-57432-8.

[39] S. Schirra. Designing a Computational Geometry Algorithms Library. In *Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems*, pages 1 – 9. CISM, Udine, Italy, September 1996.

[40] S. Schirra. Robustness and Precision Issues in Geometric Computation. In *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[41] J. R. Shewchuk. Lecture Notes on Geometric Robustness. University of California, Berkeley, USA.
http://www.cs.berkeley.edu/~jrs/meshpapers/robnotes.ps.gz.

[42] K. Sugihara and H. Inagaki. Why is the 3D Delaunay Triangulation Difficult to Construct? *Information Processing Letters*, 54:275 – 280, 1995.

[43] K. Sugihara and M. Iri. Geometric Algorithms in Finite-Precision Arithmetic. Technical Report 88-10, Math. Eng. and Physics Dept, U. of Tokyo, Japan, Sept 1988.

[44] K. Sugihara and M. Iri. Construction of the Voronoi Diagram for "One Million" Generators in Single-Precision Arithmetic. *Proceedings of the IEEE*, 80(9):1471 – 1484, 1992.

[45] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-Oriented Implementation - An Approach to Robust Geometric Algorithms. *Algorithmica*, 27:5 – 20, 2000.

[46] C. K. Yap. Towards Exact Geometric Computation. In *Fifth Canadian Conference on Computational Geometry*, pages 405 – 419, August 1993.

[47] C. K. Yap. A New Number Core for Robust Numerical and Geometric Libraries, October 1998. Abstract of Invited Talk at 3rd CGC Workshop on Computational Geometry, Brown University, October 11-12, 1998.

[48] C. K. Yap and T. Dubé. A Basis for Implementing Exact Geometric Algorithms, September 1993. Extended Abstract.

[49] C. K. Yap and T. Dubé. The Exact Computation Paradigm. In *Computing in Euclidean Geometry*. World Scientific Press, 1994.

[50] C. R. Yates. Fixed-Point Arithmetic: An Introduction. http://personal.mia.bellsouth.net/lig/y/a/yatesc/fp.pdf.