"Round-off errors arise because it is impossible to represent all real numbers exactly on a finite-state machine (which is what all practical digital computers are)."

— Numerical Analysis entry in Wikipedia

"The discrepancy between the true answer and the answer obtained in a practical calculation is called the truncation error*. Truncation error would persist even on a hypothetical, "perfect" computer that had an infinitely accurate representation and n roundoff error. As a general rule, there is not much that a programmer can do about roundouff error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of "stability" below). Truncation error, on the other hand, is entirely under the programmer's control. In fact, it is only a slight exaggeration to say that clever minimization of truncation error is practically the entire content of the field of numerical analysis!"*

— Numerical Recipes in C: The Art of Scientific Computing (Chapter 1, p. 30)

Lecture 1
ON NUMERICAL NONROBUSTNESS

This chapter gives an initial orientation to some key issues that concern us. What is the nonrobustness phenomenon? Why does it appear so intractable? Of course, the prima facie *reason for nonrobustness is numerical errors. But this is not the full story: the key lies in understanding the underlying geometry.*

§1. What is Numerical Nonrobustness?

Nonrobustness of computer systems means different things to different people. Sometimes a computer solution is described as "nonrobust" if it is non-scalable (i.e., as the problem size gets larger, the solution becomes no longer practical) or non-extensible (i.e., when we make simple variations in the problem, the solution could not be similarly tweaked). In this book, we are only interested in nonrobustness that can be traced to errors in numerical approximations. For brevity, we simply say "(non)robustness" instead of "numerical (non)robustness".

This nonrobustness phenomenon is well-known and widespread. The fact is that most numerical quantities in computers are approximations. Hence there is **error**, which may be defined as the absolute difference $|x - \widetilde{x}|$ between the exact[1] value x and the computed value $\widetilde{x}$. Most of the time, such errors are **benign**, provided we understand that they are there and we account for them appropriately. The phenomenon that we call nonrobustness arises when benign errors leads a computation to commit **catastrophic errors**. There is no graceful recovery from such errors, and often the program crashes as a result. Instead of crashing dramatically, the program could also enter an infinite loop or silently produce an erroneous result. Note that crashing may be preferred over a silent error that may eventually lead to more serious consequences. We now illustrate how benign errors can turn catastrophic:

(A) Here is a robustness test that most current geometric libraries will fail: assume the library has primitives to intersect lines and planes in 3-dimensional Euclidean space, and predicates to test if a point lies on a plane. First, we construct a plane H and a line L. Then we form their intersection $H \cap L$ which is a point $q \in \mathbb{R}^3$. Finally, let us call the predicate to test if q lies on the plane H. The predicate ought to return "YES" meaning that q lies on H. But because of numerical errors, our predicate may return

[1] We initially assume that each quantity has a theoretically "exact" value. A more general setting allows values to be inexact but quantifiably so. We will return to this later. The nonrobustness issue is already non-trivial in the exact setting.

"NO", which is what we call a catastrophic error. Alternatively, if errors such as $|x - \widetilde{x}|$ in computing a number x are called **quantitative errors**, then catastrophic errors may be called **qualitative errors**.

For random choices of H and L, there is a high likelihood that the answer comes back as "NO". For instance, suppose H_0 is the plane $x + y + z = 1$ and $L_{i,j}$ is the line through the origin and through $(i, j, 1)$. Thus the parametric equation of the line is $L_{i,j}(t) = (it, jt, t)$. We can implement all these primitives using a straightforward implementation in IEEE arithmetic (see Exercise). For instance, H_0 will intersect $L_{1,1}$ in the point $p = \frac{1}{i+j+1}(i, j, 1)$. An experiment in [23] performed a variant of this test in 2-D for pairs of intersecting lines. With 2500 pairs of intersecting lines, this tests yields a failure rate of 37.5% when implemented in the straightforward way in IEEE arithmetic.

The related problem of "line segment intersection" is a well-known in the literature on robustness, and was first posed as a challenge by Forrest [14]. Although this challenge must have been taken up in every geometric library implementation, no universally accepted solution is known. See Knott and Jou [24] for a proposed solution.

(B) An important class of physical simulations involve tracking a "front", which is just a triangulated surface representing material boundary or the interface between two fluids such as oil and water. Such a simulation might be useful to understand how an oil spill evolves under different conditions. This front evolves over time, and a qualitative error arises when the surface becomes "tangled" (self-intersects), and thus no longer divides space into two parts.

(C) Trimmed algebraic patches are basic building blocks in nonlinear geometric modeling. Because of the high algebraic complexity, the bounding curves in such patches are usually approximated. This approximation leads to errors in queries such as whether a ray intersects a patch, resulting in topological inconsistencies and finally crashing the modeler. Visually, such errors are often seen as "cracks" in a supposedly continuous surface.

(D) In mesh generation, an important primitive operation is to classify a point as inside or outside of a cell. Typically a cell is a triangle in 2-D and a tetrahedron in 3-D. If we mis-classify a point, meshes can become inconsistent, ambiguous or deficient. Current CAD systems will routinely produce such "dirty meshes". This is a serious problem for all applications downstream.

In recent years, several research communities have been interested in numerical robustness and reliability in software: books, surveys, software, special journal issues, and special workshops have been devoted to this area. Some of this information is collected at the end of this chapter, under the Additional Notes Section. Connection between the viewpoints of different communities will be discussed. For instance, numerical analysts have know for a long time the various "pitfalls" in numerical computations, but the critical missing element in such discussions is the role of geometry as brought out by the above examples.

¶1. Book Overview. The purpose of this book is develop an approach to robust algorithms this is now dominant in the Computational Geometry community. This approach is called **Exact Geometric Computation** (EGC). It is now encoded in software such as `CGAL`, `LEDA` and `Core Library`. Most of these results are obtained within the last decade, and we feel it is timely to organize them into a book form. The book is written at an introductory level, assuming only a basic knowledge of algorithms. In brief, this is what our book cover:

- We begin by introducing and analyzing the fundamental problem of nonrobustness in geometric software.
- We briefly survey various approaches that have been proposed, including EGC.
- EGC can be seen as a new mode of numerical computation which need to be supported by critical computational techniques: filters, root bounds, root finding, etc. The necessary algebraic background will be developed.
- We will study in detail the application of EGC principles in several basic algorithms and geometric primitives. These simple examples come from linear geometry (i.e., geometry that does not go beyond basic linear algebra).

- We also consider the application of EGC principles in non-linear geometry, such as in algebraic curves and surfaces.
- We address three important topics related to EGC and robustness: data degeneracy, perturbation techniques, and geometric rounding.
- Applications of EGC in areas such as theorem proving and software certification will be illustrated.
- We touch on the subject of non-algebraic computation, which lay at the frontier of current knowledge.
- Finally we describe a new theory of real approximation which is motivated by the practical development in EGC. It points to an interesting complexity theory of real computation.

§1.1. Primitives for Points and Lines

We will now flesh out the 2-D version of example (A) from the introduction. Consider a simple geometry program which uses a library with primitives for defining points and lines in the Euclidean plane. The library also has primitives to intersect two lines, and to test if a point is on a line. Mathematically, a point p is a pair (x, y) of real numbers, and a line ℓ is given by an equation

$$\ell : ax + by + c = 0$$

where (a, b, c) are the real parameters defining ℓ. We write $\ell := \mathsf{Line}(a, b, c)$ to indicated that the line ℓ is defined by (a, b, c). Note that $\mathsf{Line}(a, b, c)$ is well-defined only if $ab \neq 0$; moreover, $\mathsf{Line}(a, b, c) = \mathsf{Line}(da, db, dc)$ for all $d \neq 0$. Thus each line can be viewed[2] as a point in a peculiar homogeneous space. Similar, $p := \mathsf{Point}(x, y)$ is a definition of the point p. If $\ell' = \mathsf{Line}(a', b', c')$ is another line, then the intersection of ℓ and ℓ' is the $\mathsf{Point}(x, y)$ that satisfies the linear system

$$\begin{bmatrix} a & b \\ a' & b' \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = -\begin{pmatrix} c \\ c' \end{pmatrix}$$

which yields

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{\Delta} \begin{bmatrix} -b' & b \\ a' & -a \end{bmatrix} \begin{pmatrix} c \\ c' \end{pmatrix} \tag{1}$$

where $\Delta = ab' - a'b$ is the determinant. In two exceptional cases, the intersection is not a point: (a) when $\ell = \ell'$ (so there are infinitely many points of intersection) or, (b) when $\ell \neq \ell'$ and $\ell \| \ell'$ (the lines are distinct and parallel, so there are no intersection points). In these cases, we assume that the intersection is an "undefined value" (denoted $\uparrow$). So we define the **operation** Intersect(ℓ, ℓ') to return the intersection point (possibly $\uparrow$) of lines ℓ and ℓ'. This operation is easily implemented in any conventional programming language using the formula (1). Similarly, it is easy to program the **predicate** OnLine(p, ℓ) which returns True iff p lies on line ℓ, otherwise returns False. Note that $p = \mathsf{Point}(x_0, y_0)$ lies on $\ell = \mathsf{Line}(a, b, c)$ iff $ax_0 + by_0 + c_0 = 0$.

> *CONVENTION: In general, geometric primitives (operations or predicates or constructors) are partial functions, capable of returning the undefined value, $\uparrow$. In particular, if any input argument is undefined, then the result is undefined.*

Consider the following expression:

$$\text{OnLine}(\text{Intersect}(\ell, \ell'), \ell) \tag{2}$$

[2] A homogeneous point (a, b, c) is defined up to multiplication by a non-zero constant, and is often written as a proportionality $(a : b : c)$. In standard homogeneous space, we exclude the point $(0 : 0 : 0)$. But here, we exclude an additional point, $(0 : 0 : c)$, $c \neq 0$.

Mathematically, the value of this expression is True unless Intersect(ℓ, ℓ') is undefined. Yet, any simple implementation of these primitives on a computer will fail this property. That is to say, for "most choices" of ℓ and ℓ', the expression will not evaluate to True. The reason for this is clear – computer arithmetic are approximate. For most inputs, the expression (2) is sensitive to the slightest error in its evaluation.

An exercise below will attempt to quantify "most choices" in the above remark.

Operations such as line intersection form the basis of more complex operations in geometry libraries. If such basic operations are nonrobust, applications built on top of the libraries will face the same nonrobustness problem but with unpredictable consequences. So it is no surprise that applications such as geometric modelers are highly susceptible to catastrophic errors.

The property of failing to return True when it should have, is a qualitative error, and this is a result of some quantitative error. Numerical errors are, by nature, a quantitative phenomenon and they are normally benign. But under certain conditions, they become a qualitative phenomenon. We need to understand when such transitions take place, and to find measures either to detect or to avoid them. This is the essence of the approach called Exact Geometric Computation.

§1.2. Epsilon Tweaking

A common response of programmers to this example is to say that the comparison "is $ax+by+c = 0$?" is too much to ask of approximate quantities. Instead, we should modify the predicate OnLine(Point(x, y), Line(a, b, c)) to the following

$$\text{"Is } |ax + bx + c| < \varepsilon\text{?"} \tag{3}$$

where $\varepsilon > 0$ is a small constant. This simple idea is general and widely used: every time we need to compare something to 0, we will compare it to some small "epsilon" constant. There could be many different epsilons in different parts of a large program. It is somewhat of an art to pick these epsilons constants, perhaps chosen to avoid errors in a battery of test cases. In the programming world, the empirical task of choosing such constants is called "epsilon tweaking". We now try to analyze what epsilon tweaking amounts to.

Note that in our expression (2) above, if we make ε large enough, the predicate will surely return True for all pairs of intersecting lines. This avoids the problem of false negatives, clearly it introduces many false positives. We should understand that the introduction of ε changes the underlying geometry – we may call this **epsilon geometry**. But what is the nature of this geometry? One interpretation is that we are now dealing with lines or points that are "fat". A point $p = \mathsf{Point}(x, y)$ is really a disk centered about (x, y) with radius $\varepsilon_1 \geq 0$. Similar, a line $\ell = \mathsf{Line}(a, b, c)$ really represents a strip S of region with the mathematical line as the axis of S. The strip has some width $\varepsilon_2 \geq 0$. For simplicity, suppose $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$. Then the OnLine$(p, \ell)$ predicate is really asking if the fat point p intersects the fat line ℓ. This is illustrated in figure 1.

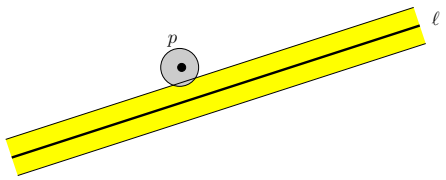


Figure 1: A fat point p intersecting a fat line ℓ.

This interpretation of equation (3) is justifiable after we normalize the equation of the line $ax+by+c = 0$ so that $a^2 + b^2 = 1$. To normalize the coefficients (a, b, c), we simply divide[3] each coefficient by $\sqrt{a^2 + b^2}$. The distance of any point (x, y) from the normalized $Line(a, b, c)$ is given by $|ax + by + c|$. Thus the epsilon test (3) amounts to checking if p is within distance ε from the line ℓ.

More important, we observe that by resorting to epsilon-tweaking, we have exchanged "plain old Euclidean geometry" (POEG) for some yet-to-be-determined geometry. In a later chapter, we will look at some attempts

[3]

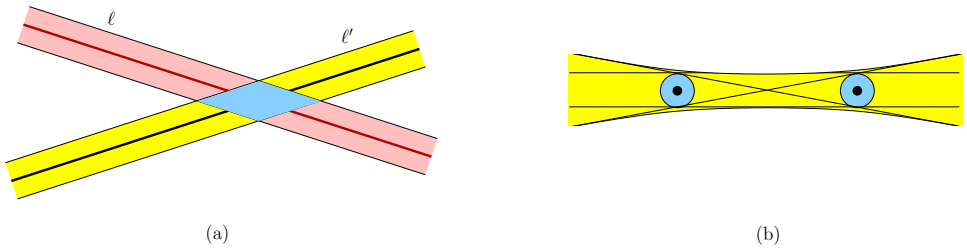


Figure 2: (a) Intersecting two fat lines, (b) Fat line through two fat points.

to capture such kinds of novel geometries. Even if a consistent interpretation can be obtained, it is hard to reason correctly with such geometries. For instance, what is the intersection of two fat lines? As we see from Figure 2(a), it is not obvious how to interpret the diamond-shaped intersection as a fat point, unless we further generalize the concept of a point to include diamond shapes. Again, we expect to be able to define a fat line that passes through two fat points. In Figure 2(b), we see that fat lines might have to be generalized to a double-wedge with non-linear boundaries.

Most of our intuitions and algorithms are guided by Euclidean geometry, and we will find all kinds of surprises in trying to design algorithms for such geometries.

¶2. Interval Geometry. There is an alternative interpretation of the test (3). Here, we assume that each point is only known up to some ball of radius ε. Then the test (3) is only useful for telling us if the point definitely does not lie on the line. But the conditions for knowing if a point is definitely on the line can only be determined with additional assumptions. We then proceed to a three-valued logic in our programs, in which "partial predicates" gives answers of yes/no/maybe. This is basically the geometric analog of the well-known idea of **interval arithmetic**. This is not unreasonable in some applications, but a common problem is that the "maybe" answers might rapidly proliferate throughout a computation and lead to no useful results. Alternatively, we might think of the "geometric intervals" (unknown region about a point or a line) growing rapidly in an iterated computation.

Exercises

Exercise 1.1: (a) Write a simple planar geometry program which handles points and lines. An object oriented programming language such as C++ would be ideal, but not essential. Provide the operations to define points and lines, to intersect two lines and to test if a point is on a line. You must provide for undefined values, and must not attempt any epsilon-tweaking.

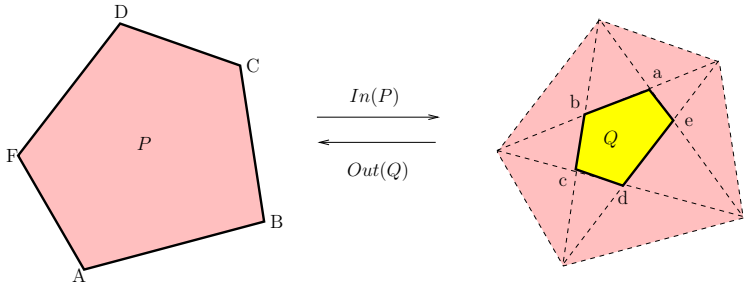
(b) Conduct a series of experiments to quantify our assertion that the expression (2) will not evaluate to True in a "large number" of cases. Assume that we implement the primitives in the straightforward way, as outlined in the text, using IEEE double precision arithmetic. Assume that $\ell = \mathsf{Line}(1, 1, -1)$ and $\ell' = \mathsf{Line}(i, j, 0)$ where $i, j = 1, \ldots, 20$. Estimate the percentage of times when the expression (2) fails (i.e., the expression did not evaluate to the True). ♢

Exercise 1.2: (a) Extend the previous exercise to 3-D, so that you can also define planes.

(b) Repeat the experiment explained in Example (A) of §1: let H_0 be the plane $x + y + z = 1$ and $L_{i,j}$ denote the parametrized line $L_{i,j}(t) = (it, jt, t)$. Compute the intersection p_{ij} of $L_{i,j}$ with H_0, and then check of p_{ij} lies on the plane H_0. Assume $i, j = 1, \ldots, 50$. How many percent of the 2500 tests fail? Does your experiment achieve the reported 37.5% failure rate as noted in the text? ♢

Exercise 1.3: Verify that for a normalized line $\ell = \mathsf{Line}(a, b, c)$, the *signed distance* of $p = \mathsf{Point}(x, y)$ from is $ax + by + c$. Interpret this sign. ♢

Exercise 1.4: We want to define the $In()$ and $Out()$ operations on convex pentagons. Let P be the convex pentagon in Figure 3(a). The 5 diagonals of P intersect pairwise in 5 points. These points determine



(a) Pentagon $P = Out(Q)$ (b) Pentagon $Q = In(P)$

Figure 3: Convex pentagon P and Q and their connection via In, Out

a convex polygon Q inside P, as in Figure 3(b). We define $In(P) := Q$. Similarly, let $Out(Q)$ denote the convex pentagon P obtained by extending the edges of Q until they intersect; there are exactly five intersection points, and these define P. It is easy to see that the following identity holds

$$P = In^m(Out^m(P)) = Out^m(In^m(P)) \tag{4}$$

for any $m \geq 0$, where $In^m(\cdots)$ and $Out^m(\cdots)$ means applying the indicated operations m times. But due to numerical errors, the identity Equation (4) is unlikely to hold. In this exercise, we ask you to implement $In(\cdots)$ and $Out(\cdots)$, and check the identity (4) (for various values of m). You could try more exotic identities like $In^m(Out^{2m}(In^m(P))) = P$ for $m \geq 0$. ♢

Exercise 1.5: If P is non-convex, and perhaps even self-intersecting, is there an interpretation of $In(P)$ and $Out(P)$? ♢

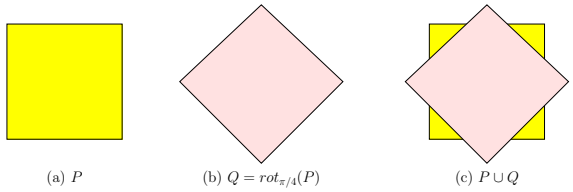
End Exercises

§2. Nonrobustness In Action

The previous section gives a hint of the potential problems with non-robust geometric primitives such as line intersection. We now illustrate consequences of such problems at a macroscopic level.

§2.1. Nonrobustness of CAD Software

Most commercial CAD software can perform Boolean operation on planar polygons, and apply rigid transformations on polygons. Boolean operations such as union, intersection and complement are defined on polygons viewed as sets. By definition, rigid transformations preserve distance; examples are translations and rotations. See Figure 4 where we rotate a square P about its center and then union this with P.



(a) P (b) $Q = rot_{\pi/4}(P)$ (c) $P \cup Q$

Figure 4: Rigid transformation and Boolean operations on polygons.

SYSTEM	n	α	TIME	OUTPUT
ACIS	1000	1.0e-4	5 min	correct
ACIS	1000	1.0e-5	4.5 min	correct
ACIS	1000	1.0e-6	30 min	too difficult!
Microstation95	100	1.0e-2	2 sec	correct
Microstation95	100	0.5e-2	3 sec	incorrect!
Rhino3D	200	1.0e-2	15 sec	correct
Rhino3D	400	1.0e-2	–	crash!
CGAL/LEDA	5000	6.175e-6	30 sec	correct
CGAL/LEDA	5000	1.581e-9	34 sec	correct
CGAL/LEDA	20000	9.88e-7	141 sec	correct

Table 1: Boolean Operations on CAD Software

Consider the following test[4] which begins by constructing a regular n-gon P, and rotating P by α degrees about its center to form Q, and finally forming the union R of P and Q. For a general α, R is now a $4n$-gon. The following table shows the results from several commercial CAD systems: ACIS, Microstation95 and Rhino3D.

The last three rows are results from the CGAL/LEDA software. In contrast to the other nonrobust software, the CGAL/LEDA output is always correct. We shall describe CGAL/LEDA and similar software later, as they embody the kind of solution developed in this book.

This last column in this table illustrates three forms of failures: program crashing, program looping (or taking excessive time, interpreted as too difficult) or silent error. The last kind of failure is actually the most insidious, as it may ultimately cost most in terms of programmer effort.

This table indicates just the tip of the iceberg. CAD software is widely used in automotive, aerospace and manufacturing industry. The cost of software unreliability to the US automobile industry (alone) is estimated at one billion dollars per year, according to a 2002 report [36] from National Institute of Standards and Technology (NIST). Of course, this estimate combines all sources of software errors; it is a difficult task to assign such costs to any particular source. But from a technical viewpoint, there is a general agreement among researchers. In a 1999 MSRI Workshop on Mathematical Foundations of CAD, the consensus opinion of the CAD researchers stated that "the single greatest cause of poor reliability of CAD systems is lack of topologically consistent surface intersection algorithms".

Nonrobust software has a recurring cost in terms of programmer productivity, or may lead to defective products which have less predictable future cost.

§2.2. Nonrobustness of Meshing Software

A mesh is a data structure which represents some physical volume or the surface of such a volume using a collection of interconnected "elements". We call it a **volume mesh** or a **surface mesh**, respectively. The mesh elements may be connected together regularly or irregularly. An example of a regular mesh is a square grid. See Figure 5 for an example of irregular surface meshes.

Mesh generation is a fundamental operation in all of computational sciences and engineering, and in many physical simulation areas. A typical operation is to create a surface meshes from, say, an implicit description of a surface as an algebraic equation. For instance, the mesher might[5] break down when one tries to create surface mesh for the cylinder of unit radius centered about the z-axis.

[4] Mehlhorn and Hart, Invited Talk at ACM Symp. of Solid Modeling, Ann Arbor, June 2001.

[5] This is an actual experience with a meshing software that is part of a fairly widely distributed CFD system. The fix, it turns out, was to perturb the axis slightly away from the z-axis.

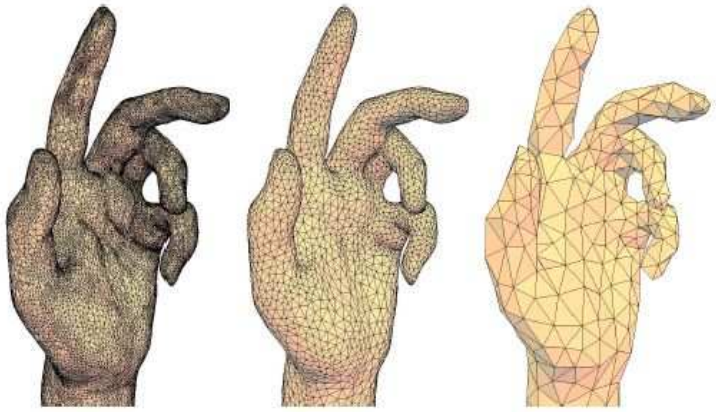


Figure 5: Surface Meshes from Dyer (SGP'07)

Quoting[6] industry experts in the field of computational fluid dynamics (CFD) on a typical scenario: *in a CFD aircraft analysis with 50 million elements, we spend 10-20 minutes for surface mesh generation, 3-4 hours for volume meshing, 1 hour for actual flow analysis, and finally 2-4 weeks for debugging and geometry repair.*

Some explanation of this scenario is in order. The first phase (surface mesh) typically involves constructing a triangulated closed manifold representing, say an aircraft wing. The second phase (volume meshing) involves partitioning the space exterior to the triangulated closed manifold. This volume is of interest for the study of air flow around the body. In other applications such as heat flows, we might be more interested in the volume internal to closed manifold. The third phase (simulation) amounts to solving partial differential equations in the volume. The final phase (debugging) arises when the simulation breaks down due to inconsistencies in the data or model. Note that geometry repair logically belongs to the surface and volume meshing phases. But in practice, errors are not immediately detected and these are caught when problems develop further downstream, and are traced back to meshing errors.

The proportion of time spent for three phases of two such applications are shown in the next table, also taken from Boeing Company:

	Fly Back Vehicle	Target for Computational Electromagnetics
Geometry Fixup	66%	60%
Gridding	17%	25%
Solving	17%	15%

Meshing ought to be a routine step for many applications. But lacking robust meshing, an engineer be in the loop to manually fix any problems that arise. In other words, this step cannot be fully automated.

Nonrobustness is a barrier to full automation in many industrial processes. This has a negative impact on productivity.

§2.3. Testing Numerical Accuracy of Software

[6]Tom Peters (University of Connecticutt) and Dave Ferguson (The Boeing Company), talk at the DARPA/NSF CARGO Kickoff Workshop, Newport RI, May 20-23, 2002.

Mehlhorn [27] pose the "Reliable Algorithmic Software Challenge" (RASC). A critical component in the solution to RASC is a new mode of numerical computation in which we can demand any *á priori* accuracy of a numerical variable. We call this **guaranteed accuracy computation** [47]. Here we explore one class of applications for this mode. In many areas of numerical computation, there are many commercial software available. To evaluate their individual performance, as well as to compare these software, standardized test suites have been collected. Very often test suites only contain input data (called benchmarks), and software are compared in terms of their speed on these benchmarks (under a variety of platforms). But in numerical software, the correctness of these software cannot be taken for granted – we need to also know the accuracy of the software. To do this, each benchmark problem ought to come with model answers to some precision. This would allow us to measure deviation from the model answers. But which software do we trust to produce such model answers? This is one clear application for guaranteed accuracy computation. We look at two such examples.

For our first example, consider linear programming, a major contender for the shortlist of most useful algorithms. Linear programming can be defined as follows. Given vectors $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ $(m > n)$, and a matrix $A \in \mathbb{R}^{m \times n}$, to find a point $x \in \mathbb{R}^n$ which maximizes the dot product $c^T x$ subject to $Ax \leq b$ Geometrically, the point x can be taken to be a vertex of the convex polyhedron defined by $Ax \leq b$ whose projection onto the direction c is maximum. Because of its importance in the industry, many commercial software standardized test suites are available. One of the most well-known software is called `CPLEX`, and `netlib` is a popular collection of benchmark problems. The following table taken from [27] shows the performance of `CPLEX` on 5 benchmark problems.

PROBLEM				CPLEX Solution		CPLEX Solution		
Name	m	n	# non-zeros	RelObjErr	ComputeTime	Violations	Opt?	VerifyTime
degen3	1504	1818	26230	$6.91e-16$	8.08s	0	opt	8.79s
etamacro	401	688	2489	$1.50e-16$	0.13s	10	feas	1.11s
fffff800	525	854	6235	$0.00e+00$	0.09s	0	opt	4.41
pilot.we	737	2789	9218	$2.93e-11$	3.8s	0	opt	1654.64
scsd6	148	1350	5666	$0.00e+00$	0.1s	13	feas	0.52s

For each problem, in this table, we not only give the dimension $m \times n$ of the matrix A, but column 4 also gives the number of non-zeros in the matrix A, as a better indication of the input size. The solution from `CPLEX` is a basis of A. The relative error in the objective value of this basis is indicated. Using guaranteed precision methods, the solution returned by `CPLEX` is checked. Thus, we find that two of these solutions are actually non-optimal (etamacro and scsd6). The column labeled 'Violations' counts the number of constraints violated by the basis returned by `CPLEX`. The time to check (last column) is usually modest compared to that of `CPLEX`, but the benchmark problem 'pilot.we' is an obvious exception. does not always produce the correct optimal answer. Such errors are qualitative, not merely quantitative.

Let us return to the problem of generating model answers for benchmark problems. As in linear programming, the primary goal here is qualitative correctness. Next, we want to ensure certain bounds on the quantitative error. We need software that can guarantee their accuracies in both these senses. In this book, we will see techniques which produce such software. See [10] for the issues of certifying and repairing approximate solutions in linear programming.

Another example is from numerical statistical computation [25]. Here, McCullough [9] address the problem of evaluating the accuracy of statistical packages. At the National Institute of Standards and Technology (NIST), such model answers must have 15 digits of accuracy, and these are generated by running the program at 500 bits of accuracy. It is unclear how 500 bits is sufficient; but it is clear that 500 bits will often be unnecessary. We would like software like `LEDA Real` [7] or `Core Library` [23] that can automatically generate the necessary accuracy needed.

Essentially the same problem has been addressed by Lefévre et al [26], who posed the **Table Maker's Dilemma**, namely how to compute correctly rounded results from evaluation of transcendental functions.

The correct rounding problem is part of a larger, industry-wide concern about the reliability of computer hardware arithmetic. Prior to the 1980's there was widespread concern about the non-portability and unpredictability of computer arithmetic across hardware vendors. The upshot of this is the IEEE standard

on arithmetic which is now widely accepted. The paper [49] provides the foundations of exact rounding.

> *Software testing is a meta application where guaranteed quality of numerical output is not an option.*

§2.4. Accuracy as a Contractual Issue

Geometric tolerancing is one cornerstone of the manufacturing sciences. It provides the language of specifying geometric shapes for the purposes of manufacture. As all manufacturing processes are inherently inexact, part and parcel of this specification includes the acceptable deviation from the ideal shape.

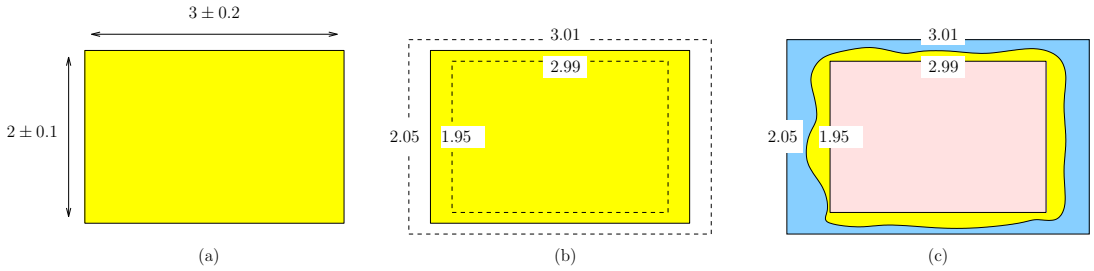


Figure 6: Tolerance Geometry: (a) ideal, (b) zone, (c) actual.

For instance, suppose we wish to manufacture a $3'' \times 2''$ piece of metal plate (see Figure 6). We might tolerance this as $3 \pm 0.2'' \times 2 \pm 0.1''$. The naive and highly intuitive understanding of this tolerance is that any rectangular plate with dimension $L'' \times W''$ is acceptable as long as $|L - 3| \leq 0.2$ and $|W - 2| \leq 0.1$. The problem is that no manufactured object is really a rectangle – it may not even have any straight edge at all. The modern understanding is that the specification describes a **tolerance zone** (Figure 6(b)) and an object is within tolerance if there is some placement of the object whose boundary lies inside this zone (Figure 6(c)). Note that by "placement", we mean a position of the object obtained by rotation and/or translation.

The language and semantics of dimensioning and tolerance is encoded into industry standards (e.g., ANSI standard Y14.5M-1982 [45, 31]); its mathematical basis is similarly encoded [43]. There is a parallel development at the International Standards Organization (ISO).

Beyond having a language to specify shape tolerances, we must have some attendant "assessment methodology" to measure the deviation of physically manufactured shapes, and to verify conformance. Such methodology can be regarded as a subarea of the much larger subject of *metrology*. We use the term *dimensional metrology* to refer to its application in geometric tolerancing. Dimensional metrology is much less understood than tolerancing itself [2]. In fact, this has precipitated a minor crisis [42]. The background of this crisis comes from the proliferation of the **coordinate measurement machine** (CMM) technology in the 1980's. A CMM is a physical device that can be programmed to take measurements of a physical object *and* to compute tolerance characteristics of the object using these measurements. Such machines represent a major advancement over, say, traditional hard gauges (or calipers). While hard gauges give only a Boolean value (go or no-go), the CMM computation returns "characteristic values" that may be used in other applications of metrology such as process monitoring. Thus CMM is regarded as state-of-the-art in dimensional metrology. Note that CMM's are inherently coupled to computational power via embedded software. The problem is that that there is no standard governing this software – different vendor software can give widely divergent answers. This uncertainty could in principle force all government procurement services to come a grinding halt! Imagine a scenario where a critical component of a jet engine is to be toleranced to within 0.001mm but because of inaccurate software, it is really 0.01mm. This could give rise to contractual disputes, premature mechanical failures, or worse. This was the situation uncovered by Walker [42] in 1988.

A basic shape for dimensional tolerancing is the circular disc. We might tolerance such a disc to have radius 2 ± 0.05 cm. The "tolerance zone" semantics in this case is clear: we want the manufactured disc to

have its boundary within an annular region whose inner and outer radii are $1.95 - -2.05$ cm. But what kind of assessment method is available for this? The standard practice is to uniformly measure $5 - 10$ points on the boundary of the manufactured disc, and algorithmically decide if these points lie within the tolerance zone. The justification of such a procedure is far from obvious (see [28]).

In summary, a major component of dimensional metrology revolves around the computational issues, or what Hopp has termed *computational metrology* [21]. To be sure, the uncertainty in CMM technology is grounded in the measurement hardware as well as the software. Modeling and compensating for hardware biases is a major research area. But often, software errors can be orders of magnitude worse than hardware uncertainty.

Dimensional metrology have many economic consequences that may not be obvious. Imagine that we want to tolerance the roundness of a cylinder for a jet engine. If we over-tolerance, this may lead to premature wear-and-tear, and possibly disastrous breakdown. If we under-tolerance, this can greatly add to the cost and number of machining hours, without producing any appreciable benefits. There are limits to how tightly we can tolerance parts, because of the inherent uncertainty in various processes, measurements and computations. If we tolerance too close to this limit, the results are probably unreliable. Moreover, manufacturing costs can increase dramatically as our tolerance approach this limit: the number of machining hours will greatly increase, and The slightest mistake[7] can lead to a rejection of expensive parts made of special alloys.

We describe an anecdote from Hopp: in the aluminum can industry, the thickness of the top of the pop-top cans has some physical limitations. It must be (say) not be thicker than 0.4mm, or else the pop-top mechanism may not work (many of us knows this from first hand experience). It must not be thinner than 0.3mm, or else the internal pressure may cause spontaneous explosion. Hence, we might tolerance this thickness as 0.35 ± 0.05mm. Now, if manufacturing processes were more reliable, and our ability to assess tolerance were more accurate, we might be able to tolerance this thickness as 0.32 ± 0.02mm. Such a change would save the aluminum can industry multi-millions of dollars in material cost each year.

For a survey of the computational problems in dimensional metrology, see [13] or [1]; for a perspective on current dimensional theory, see Voelcker [41]. An NSF-sponsored workshop on manufacturing and computational geometry addresses some of these issues [46].

§2.5. For Lack of a Bit

On 4 June 1996, the maiden flight of the[8] Ariane 5 launcher in French Guiana ended in self-destruction. About 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. A report from the Inquiry Board located the critical events: "At approximately 30 seconds after lift-off, the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer. Approximately 0.05 seconds later the active inertial reference system, identical to the back-up system in hardware and software, failed for the same reason. Since the back-up inertial system was already inoperative, correct guidance and attitude information could no longer be obtained and loss of the mission was inevitable."

The software error began with an overflow problem: "The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected."

There is a similar episode involving the Patriot Missiles in the 1990 Gulf War. We refer to Higham [18, p. ???] for details.

Naturally, any major incident could be attributed to failures at several levels in the system. For instance, the Inquiry Board in the Ariane incident also pointed at failure at the organizational level: "The extensive

[7] Such parts end up as $10,000 paper weights in some offices.

[8] Ariane is the French version of the Greek name Ariadne (in English!). In Greek mythology, Ariadne was the daughter of King Minos of Crete.

reviews and tests ... did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure." Kahan and Darcy [22] responded: "What software failure could not be blamed upon inadequate testing? The disaster can be blamed just as well upon a programming language (Ada) that disregarded the default exception-handling specifications in IEEE Standard 754." The Ariane failure lost a satellite payload worth over half a billion dollars. Other costs, including the potential loss of lives, are harder to quantify.

In mission-critical applications, less than fully robust software is not an option.

§2.6. Human Failures

There are many other examples of the drastic consequences of unreliability of numerical computation:

- Intel Pentium chip floating point error (1994-5)
- North Sea Sleipner Class oil rig collapse (1991) due inaccurate finite element analysis,
- Among major engineering structures, bridges seem particularly prone to unpredictable errors. Three modern examples are Tacoma Bridge, Washington, Auckland Bridge, and London Millennium Bridge. Better computer simulations with improved accuracy in numerical simulation may have been able to predict the problems.
- An incidence at the Vancouver Stock Exchange index, Jan 1982 to Nov 1983.

It should be stressed that although numerical errors are at the root of these examples, one could also point to various human errors in these episodes. For instance, in the Ariane 5 failure, one can also point a finger at imperfect software testing, at management lapses, etc. On the other hand, these failings of human institutions would be irrelevant if the source of these problems (numerical errors) could be eliminated.

§3. Responses to Numerical Nonrobustness

The ubiquitous nature of numerical nonrobustness, would seem to cry out for a proper resolution. As computers become more integrated into the fabric of modern society, this problem is expected[9] to worsen. The fact that we do not see a general declaration of war on nonrobustness (in the same way that computer security has marshaled support in recent years) calls for explanation. Here are some common (non)responses to nonrobustness:

¶3. Nonrobustness is inevitable. It is often stated in textbooks and popular science articles that computer arithmetic is inherently inexact, and so exact computation is impossible. Whatever such a statement means, it is wrong to conclude that numerical nonrobustness is unavoidable in software.

¶4. Nonrobustness is unimportant because it is a rare event. It is true that, for any given software, the inputs that causes nonrobust behavior are in a certain sense "rare". The problem is that these rare events occur at a non-negligible frequency.

¶5. Nonrobust software is a mere inconvenience. When the productivity of scientists and engineers is negatively impacted, this "inconvenience" becomes a major issue. A study from NIST [36] estimates the cost of software unreliability in several industries including transportation (aircraft and automobile manufacture) and financial services.

[9] Aided by the relentless progress of Moore's law, we may experience nonrobustness more frequently in the following sense: what used to fail once a month may now fail every day, simply because the same code is executed faster, perhaps on larger data sets.

¶6. Avoid ill-conditioned inputs. It is well-known that we can lose significance very quickly with ill-conditioned inputs. One response of numerical analysts is to suggest that we should avoid ill-conditioned inputs. For instance, if the input is ill-conditioned, we should first perform some random perturbation. Unfortunately, we note that ill-conditioned inputs are often a deliberate feature of the input data. For example, in engineering and architectural designs, multiple collinear points is probably a design feature rather than an accident. So a random perturbation destroys important symmetries and features in the data.

¶7. Use stable algorithms. Stability is an important concept in numerical computation. The definition[10] of what this means is usually informal, but see a definition in [40]. Higham [18, p.30] gives guidelines for designing numerically stable algorithm. The problem is that stable algorithms only serves to reduce the incidence of non-robustness, not eliminate it.

¶8. No one else can solve it either. This might be the response of software vendor, but it seems that many users also buy into this argument. We believe that this response[11] is increasingly weak in many domains, including meshing software.

§4. Additional Notes

An early effort to address nonrobustness geometric algorithms resulted in a 2-volume special issue of *ACM Trans. on Graphics*, Vol.3, 1984, edited by R. Forrest, L.Guibas and J.Nievergelt. The editorial deplores the disconnect between the graphics community and computational geometry, and points out the difficulties in implementing robust algorithms: "A Bugbear ... is the plethora of special cases ... glossed over in theoretical algorithms". "In practice, even the simplest operations such as line segment intersection... are difficult, if not impossible, to implement reliably and robustly". "All geometric modeling systems run into numerical problems and are, on occasion, inaccurate, inconsistent, or simply break...". This was in the early days of computational geometry where the field has a bibliography with just over 300 papers at that time, and the book of Shamos and Preparata was just out. See also [14].

In the literature, what we call "robustness" also goes under the name of **reliability**. The interest in robust or reliable software appears in different forms in several communities.

- The area called **validated computing** or **certified computing** aims to achieve robustness by providing guaranteed error bounds. The techniques comes from interval arithmetic. There is active software and hardware development, aimed to make interval (or enclosure) methods easily accessible to programmers. There are several key books in this area; in particular, [35] addresses problems of computational geometry. The journal[12] **Reliable Computing** from Kluwer Academic Publishers devoted to this field; see also the special issue [34].

- The geometric design community is acutely aware of nonrobustness in CAD software. Hoffman [20, 19] has been an advocate of efforts to address this problem. Farouki [12] looks at numerical stability issues, especially in connection to the favorable properties of the Bezier representation. Patrikalakis [32] surveys robustness issues in geometric modelling software. One of the key open problems in this field is the development of fully reliable numerical algorithms for the surface-surface intersection (SSI) problem.

- Nonrobustness in computer graphics might appear to be a non-issue if we think the main computational issue is deciding how to turn on a pixel on the screen. But in reality, there is much geometric processing before making this final decision. The book of Christer Ericson [11, Chap. 11] is devoted to numerical robustness, presenting many of the issues that implementor of computer games face.

- There has been long-standing interest in theory of real computation. We mention two schools of thought: one is the "Type Two Theory of Effectivity" (TTE Theory) [44], with roots in the classical

[10] We are indebted to Professor Michael Overton for this remark.

[11] We think CAD software vendors ought to be extremely interested in the current research development when confronted with the data in Table 1.

[12] From 1991 to 1994, it was known as "J. of Interval Computations".

constructive analysis. The other school (BSS or Algebraic Theory) [6] is based on the uniform algebraic models. The annual workshop *Computability and Complexity in Analysis* is representative of the TTE approach. The Algebraic School is initiated by the work of Smale and his co-workers. Many interesting connections with standard complexity theory has been discovered.

- Over the years, there has been several software developed to support real computation: these are usually based on the lazy-evaluation paradigm, which provides more and more bits of accuracy on demand. A recent system along this line is Mueller's `iRRAM` [30]. The annual workshop "Real Numbers and Computers" is representative of this community.

- Numerical analysts have long recognized the pitfalls associated with numerical computing [15, 38, 17]. Key topics here are numerical stability, accuracy and conditioning. A modern treatment of these issues is Higham [18]. The book of Chaitin-Chatelin and Valérie Frayssé [8] addresses stability of finite-precision computation. The key role of condition numbers have also been studied in the Smale school.

- The meteorologist Edward N. Lorenz (1917-2008) accidentally discovered computational chaos in the winter of 1961. He was running a weather simulation for a second time, but for a longer time period. He started the second simulation in the middle, by typing in numbers that were printed from the first run. He was surprised to see the second weather trajectory quickly diverge from the first. He eventually realized that the computer stored numbers to an accuracy of 6 decimal places (e.g., 0.123456), but the printout was shorted to 3 decimal places (e.g., 0.123). This discrepancy led to the diverging simulation results. He published this finding in 1963. In a 1964 paper, he described how a small change in parameters in a weather model can transform a regular periodic behavior into chaotic pattern. But it was the title of his 1972 talk at the American Association for the Advancement of Science that entered the popular imagination: "Predictability: Does the Flap of a Butterfly's Wings in Brazil Set Off a Tornado in Texas?" This story in recounted in James Gleick's book "Chaos".

- Various workshops been devoted to robust computation, including: SIGGRAPH Panel on Robust Geometry (1998), NSF/SIAM Workshop on Integration of CAD and CFD (1999), MSRI Workshop on Foundations of CAD (1999), Minisymposium Robust Geometric Computation at Geometric Design and Computing (2001), DIMACS Workshop on Implementation of Geometric Algorithms (2002).

- Reports in computing fields emphasize the challenges of robust computation: NSF Report on Emerging Challenges in Computational Topology [3], Computational Geometry Task Force Report [5, 4], ACM Strategic Directions Report [39].

- Special journal issues have address this topic. E.g., [16], [29], [33]. Two surveys on nonrobustness are [37, 48]. Major software such as `LEDA` and `CGAL` have been highly successful in solving nonrobustness in many basic problems of geometry.

§5. APPENDIX: General Notations

We gather some common notations that will be used in this book.

The set of natural numbers is denoted by $\mathbb{N} = \{0, 1, 2, \ldots\}$. The other systems of numbers are integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$, rational numbers $\mathbb{Q} = \{n/m : n, m \in \mathbb{Z}, n \neq 0\}$, real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$.

For any positive real number x, we write $\lg x$ and $\ln x$ for their logarithms to base 2 and the natural logarithm, $\log_2 x$ and $\log_e x$.

The set of $m \times n$ matrices with entries over a ring R is denoted $R^{m\times n}$. Let $M \in R^{m\times n}$. If the (i, j)th entry of M is x_{ij}, we may write $M = [x_{ij}]_{i,j=1}^{m,n}$ (or simply, $M = [x_{ij}]_{i,j}$). The (i, j)th entry of M is denoted $M(i; j)$. More generally, if $i_1, i_2, \ldots, i_k$ are indices of rows and $j_1, \ldots, j_\ell$ are indices of columns,

$$M(i_1..i_k; j_1..j_\ell) \tag{5}$$

denotes the submatrix obtained by intersecting the indicated rows and columns. In case $k = \ell = 1$, we often prefer to write $(M)_{i,j}$ or $(M)_{ij}$ instead of $M(i; j)$. If we delete the ith row and jth column of M,

the resulting matrix is denoted $M[i;j]$. Again, this notation can be generalized to deleting more rows and columns. E.g., $M[i_1, i_2; j_1, j_2, j_3]$ or $[M]_{i_1,i_2;j_1,j_2,j_3}$. The **transpose** of M is the $n \times m$ matrix, denoted M^T, such that $(M^T)_{i,j} = (M)_{j,i}$.

References

[1] G. Anthony, H. Anthony, B. Bittner, B. Butler, M. Cox, R. Drieschner, R. Elligsen, A. B. Forbes, H. Groß, S. Hannaby, P. Harris, and J. Kok. Chebyshev best-fit geometric elements. NPL Technical Report DITC 221/93, National Physical Laboratory, Division of Information Technology and Computing, NPL, Teddington, Middlesex, U.K. TW11 0LW, October, 1992.

[2] W. Beckwith and F. G. Parsons. Measurement methods and the new standard B89.3.2. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 31–36, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[3] M. Bern, D. Eppstein, P. K. Agarwal, N. Amenta, P. Chew, T. Dey, D. P. Dobkin, H. Edelsbrunner, C. Grimm, L. J. Guibas, J. Harer, J. Hass, A. Hicks, C. K. Johnson, G. Lerman, D. Letscher, P. Plassmann, E. Sedgwick, J. Snoeyink, J. Weeks, C. Yap, and D. Zorin. Emerging challenges in computational topology, 1999. Invited NSF Workshop on Computational Topology, (organizers: M.Bern and D.Eppstein), Miami Beach, Florida, June 11–12, 1999. Report available from Computing Research Repository (CoRR), http://xxx.lanl.gov/abs/cs.CG/9909001.

[4] Bernard Chazelle, et al. The computational geometry impact task force report: an executive summary. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 59–66. Springer, 1996. Lecture Notes in Computer Science No. 1148.

[5] Bernard Chazelle, et al. Application challenges to computational geometry: CG impact task force report. Tr-521-96, Princeton Univ., April, 1996. See also URL www.cs.princeton.edu/~chazelle/.

[6] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.

[7] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact efficient geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.

[8] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[9] B. M. Cullough. Assessing the reliability of statistical software: Part II. *The American Statistician*, 53:149–159, 1999.

[10] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, , and D. Weber. Certifying and repairing solutions to large LPs, how good are LP-solvers? In *SODA*, pages 255–56, 2003.

[11] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[12] R. T. Farouki. Numerical stability in geometric algorithms and representation. In D. C. Handscomb, editor, *The Mathematics of Surfaces III*, pages 83–114. Clarendon Press, Oxford, 1989.

[13] S. C. Feng and T. H. Hopp. A review of current geometric tolerancing theories and inspection data analysis algorithms. Technical Report NISTIR-4509, National Institute of Standards and Technology, U.S. Department of Commerce. Factory Automation Systems Division, Gaithersburg, MD 20899, February 1991.

[14] A. R. Forrest. Computational geometry and software engineering: Towards a geometric computing environment. In D. F. Rogers and R. A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 23–37. Springer-Verlag, 1987.

[15] G. E. Forsythe. Pitfalls in computation, or why a math book isn't enough. *Amer. Math. Monthly*, 77:931–956, 1970.

[16] S. Fortune. Editorial: Special issue on implementation of geometric algorithms, 2000.

[17] L. Fox. How to get meaningless answers in scientific computation (and what to do about it). *IMA Bulletin*, 7(10):296–302, 1971.

[18] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[19] C. M. Hoffmann. *Geometric and Solid Modeling: an Introduction*. Morgan Kaufmann Publishers, Inc., San Mateo, California 94403, 1989.

[20] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–42, March 1989.

[21] T. H. Hopp. Computational metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 207–217, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[22] W. Kahan and J. D. Darcy. How Javas floating-point hurts everyone everywhere, March 1998. Paper presented at ACM 1998 Workshop on Java for High Performance Network Computing, Stanford University.

[23] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[24] G. Knott and E. Jou. A program to determine whether two line segments intersect. Technical Report CAR-TR-306, CS-TR-1884, Computer Science Department, University of Maryland, College Park, August 1987.

[25] K. Lange. *Numerical Analysis for Statisticians*. Springer, New York, 1999.

[26] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Trans. Computers*, 47(11):1235–1243, 1998.

[27] K. Mehlhorn. The reliable algorithmic software challenge (RASC). In *Computer Science in Perspective*, volume 2598 of *LNCS*, pages 255–263, 2003.

[28] K. Mehlhorn, T. Shermer, and C. Yap. A complete roundness classification procedure. In *13th ACM Symp. on Comp. Geometry*, pages 129–138, 1997.

[29] N. Mueller, M. Escardo, and P. Zimmermann. Guest editor's introduction: Practical development of exact real number computation. *J. of Logic and Algebraic Programming*, 64(1), 2004. Special Issue.

[30] N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[31] A. G. Neumann. The new ASME Y14.5M standard on dimensioning and tolerancing. *Manufacturing Review*, 7(1):16–23, March 1994.

[32] N. M. Patrikalakis, W. Cho, C.-Y. Hu, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Towards robust geometric modelers, 1994 progress report. In *Proc. 1995 NSF Design and Manufacturing Grantees Conference*, pages 139–140, 1995.

[33] S. Pion and C. Y. G. Editors). Reliable, 2004.

[34] H. Ratschek and J. G. R. G. Editors). Editorial: What can one learn from box-plane intersections?, 2000. Special Issue on Reliable Geometric Computations.

[35] H. Ratschek and J. Rokne. *Geometric Computations with Interval and New Robust Methods: With Applications in Computer Graphics, GIS and Computational Geometry*. Horwood Publishing Limited, UK, 2003.

[36] Research Triangle Park (RTI). Planning Report 02-3: The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), U.S. Department of Commerce, May 2002.

[37] S. Schirra. Robustness and precision issues in geometric computation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1999.

[38] I. A. Stegun and M. Abramowitz. Pitfalls in computation. *J. Soc. Indust. Appl. Math.*, 4(4):207–219, 1956.

[39] R. Tamassia, P. Agarwal, N. Amato, D. Chen, D. Dobkin, R. Drysdal, S. Fortune, M. Doorich, J. Hershberger, J. O'Rourke, F. Preparata, J.-R. Sack, S. Suri, I. Tollis, J. Vitter, and S. Whitesides. Strategic directions in computational geometry working group report. *ACM Computing Surveys*, 28(4), Dec. 1996.

[40] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[41] H. B. Voelcker. A current perspective on tolerancing and metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 49–60, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[42] R. K. Walker. CMM form tolerance algorithm testing. GIDEP Alert X1-A1-88-01 and X1-A1-88-01a, Government-Industry Data Exchange Program, Pomona, CA, 1988.

[43] R. K. Walker and V. Srinivasan. Creation and evolution of the ASME Y14.5.1M standard. *Manufacturing Review*, 7(1):16–23, March 1994.

[44] K. Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.

[45] A. Y14.5M-1982. *Dimensioning and tolerancing*. American Society of Mechanical Engineers, New York, NY, 1982.

[46] C. K. Yap. Report on NSF Workshop on Manufacturing and Computational Geometry. *IEEE Computational Science & Engineering*, 2(2):82–84, 1995. Video Proceedings. Workshop at the Courant Institute, New York University, April 1–2, 1994.

[47] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[48] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.

[49] C. K. Yap and J. Yu. Foundations of exact rounding. In *Proc. WALCOM 2009*, volume 5431 of *Lecture Notes in Computer Science*, 2009. To appear. Invited talk, 3rd Workshop on Algorithms and Computation, Kolkata, India.

It is better to solve the right problem the wrong way than to solve the wrong problem the right way.

The purpose of computing is insight, not numbers.

– Richard Wesley Hamming (1915–1998)

Lecture 2
MODES OF NUMERICAL COMPUTATION

To understand numerical nonrobustness, we need to understand computer arithmetic. But there are several distinctive modes of numerical computation: symbolic mode, floating point (FP) mode, arbitrary precision mode, etc. Numbers are remarkably complex objects in computing, embodying two distinct sets of properties: quantitative properties and algebraic properties. Each mode has its distinctive number representations which reflect the properties needed for computing in that mode. Our main focus is on the FP mode that is dominant in scientific and engineering computing, and in the corresponding representation popularly known as the IEEE Standard.

§1. Diversity of Numbers

Numerical computing involves numbers. For our purposes, numbers are elements of the set $\mathbb{C}$ of complex numbers. But in each area of application, we are only interested in some subset of $\mathbb{C}$. This subset may be $\mathbb{N}$ as in number theory and cryptographic applications. The subset may be $\mathbb{R}$ as in most scientific and engineering computations. In algebraic computations, the subset might be $\mathbb{Q}$ or its algebraic closure $\overline{\mathbb{Q}}$.

These examples show that "numbers", despite their simple unity as elements of $\mathbb{C}$, can be very diverse in manifestation. Numbers have two complementary sets of properties: **quantitative** ("numerical") properties and **algebraic properties**. Quantitative properties of numbers include ordering and magnitudes, while algebraic properties include the algebraic axioms of a ring or field. It is not practical to provide a single representation of numbers to cover the full range of these properties. Otherwise, computer systems might as well provide a single number type, *the* complex number type. Depending on the application, different aspects of quantitative properties or algebraic properties would be emphasized or supported. Over the years, different computing fields have developed suitable number representation to provide just the needed properties. Corresponding to these number representations, there also evolved corresponding **modes of numerical computation**. We briefly review a few of these modes:

- The **symbolic mode** is best represented by computer algebra systems such as `Macsyma`, `Maple` or `Mathematica`. In the present context of numerical computing, perhaps the most important subclass of $\mathbb{C}$ in the symbolic mode is the algebraic numbers $\overline{\mathbb{Q}}$. A simple example of an algebraic number is $\sqrt{2}$. Here, there are two common representations. A number $\alpha \in \mathbb{Q}[\beta] \subseteq \overline{\mathbb{Q}}$ can be represented by a polynomial $A(X) \in \mathbb{Q}[X]$ modulo $B(X)$ where $B(X)$ is the minimal polynomial of β. This representation is useful if we are only interested in the algebraic properties of numbers. It is sufficient to model the field operations and to check for equality. When α is real, and we are interested in the quantitative properties of numbers, then the polynomial $A(X)$ is inadequate. Instead, we can use the **isolated interval representation**, comprising of a polynomial-interval pair $(A(X), I)$ where α is the only root of $A(X)$ inside the interval I. For instance, if $\alpha = \sqrt{2}$ then we could choose $A(X) = X^2 - 2$ and $I = [1, 2]$. We can perform the arithmetic operations on such isolated interval representations. The quantitative properties of numbers can be captured by the interval I, which can of course be arbitrarily narrowed. Both representations are exact.

- The unquestioned form of numerical computing in most scientific and engineering applications involves machine floating point numbers. We refer to this as the **FP mode**, standing for "floating point" or

"fixed precision", both of which are characteristic of this mode. Thus, $\sqrt{2}$ is typically represented by 64 bits in some floating point format, and converts to the printed decimal value of 1.4142135623731. Algebraic properties are no longer universally true (e.g., $x(y+z) \neq xy + xz$, and $xy = 0$ even though $x \neq 0$ and $y \neq 0$). In modern hardware, this format invariably conforms to the IEEE Standard [16]. The FP mode is very fast because of such hardware support. It is the "gold standard" whereby other numerical modes are measured against. One goal of numerical algorithm design in the FP mode is to achieve the highest numerical accuracy possible using machine arithmetic directly on the number representation alone (perhaps after some re-arrangement of computation steps). Over the last 50 years, numerical analysts have developed great insights into the FP mode of computation.

The characterization "fixed precision" needs clarification since all FP algorithms can be regarded as parametrized by a precision number θ ($0 \leq \theta < 1$). Most algorithms will produce answers that converge to the exact answer as $\theta \to 0$ (see Chaitin-Chatelin and Frayssé [7, p. 9]). In practice, FP algorithms are "precision oblivious" in the sense that their operations do not adapt to the θ parameter.

- The **arbitrary precision mode** is characterized by its use of Big Number types. The precision of such number types is not fixed. Because of applications such as cryptography, such number types are now fairly common. Thus the `Java` language comes with standard Big Number libraries. Other well-known libraries include the GNU Multiprecision Package `gmp`, the `MP` Multiprecision Package of Brent [5], the `MPFun` Library of Bailey [3], and `NTL` from Shoup. Surveys of Big Numbers may be found in [11, 42].

  The capabilities of Big Number packages can be extended in various ways. Algebraic roots are not normally found in Big Number packages, but the PRECISE Library [21] provides such an extension. Arbitrary precision arithmetic need not be viewed as monolithic operations, but can be performed incrementally. This gives rise to the **lazy evaluation mode** [25, 4]. The `iRRAM` Package of Müller [28] has the interesting ability to compute limits of its functions. The ability to reiterate an arbitrary precision computation can be codified into programming constructs, such as the precision begin-end blocks of the `Numerical Turing` language [15].

- The **validated mode** refers to a computational mode in which computed values are represented by intervals which contain the "true value". Properties of the exact answer can often be inferred from such intervals. For instance, if the interval does not contain 0, then we can infer the exact sign of the true value. This mode often goes by the name of **interval arithmetic**. It is orthogonal to the FP mode and arbitrary precision mode, and thus can be combined with either one. For instance, the Big Float numbers in `Real/Expr` [42] and also in PRECISE are really intervals. This amounts to automatic error tracking, or significance arithmetic.

- The **guaranteed precision mode** is increasingly used by computational geometers working on robust computation [41]. It is encoded in software libraries such as `LEDA`, `CGAL` and `Core Library`. In this mode, the user can freely specify an *á priori* precision bound for each computational variable; the associated library will then compute a numerical value that is guaranteed to to this precision. In its simplest form, one simply requires the correct sign – this amounts to specifying one relative-bit of precision [39]. Guaranteed sign computation is enough to achieve robustness for most basic geometric problems. This mode is stronger than the validated mode because the precision delivered by a validated computation is an *á posteriori* one, obtained by forward propagation of error bounds.

In this Chapter, we focus on the FP mode and briefly touch on arbitrary precision mode and validated mode. The symbolic mode will be treated in depth in a later Chapter.

§2. Floating Point Arithmetic

It is important to understand some basic properties of machine arithmetic, as this is ultimately the basis of most numerical computation, including arbitrary precision arithmetic. As machine floating-point arithmetic is highly optimized, we can exploit them in our solutions to nonrobustness; this remark will become clear when we treat filters in a later chapter. An excellent introduction to numerical floating point computation and the IEEE Standard is Overton [30].

We use the term **fixed precision arithmetic** to mean any arithmetic system that operates on numbers that have a fixed budget of bits to represent their numbers. Typically, the number is represented by a fixed number of "components". For instance, a floating point number has two such components: the mantissa and the exponent. Each component is described by a natural number. The standard representation of natural numbers uses the well-known positional number system in some **radix** $\beta \geq 2$. An alternative name for radix is **base**. Here β is a natural number and the **digits** in radix β are elements of the set $\{0, 1, \ldots, \beta - 1\}$. A positional number in radix $\beta \geq 2$ is represented by a finite sequence of such digits. For instance, humans communicate with each other using $\beta = 10$, but most computers use $\beta = 2$.

We will call such numbers **positional number systems**. There are two main ways to represent positional numbers on a computer, either fixed point or floating point.

¶1. Fixed Point Systems. A **fixed point system** of numbers with parameters $m, n, \beta \in \mathbb{N}$ comprises all numbers of the form

$$\pm d_1 d_2 \ldots d_n . f_1 f_2 \ldots f_m \tag{1}$$

and $d_i, f_j \in \{0, 1, \ldots, \beta - 1\}$ are digits in base β. Typically, $\beta = 2, 10$ or 16. Fixed point systems are not much used today, except for the special case of $m = 0$ (i.e., integer arithmetic).

It is instructive to briefly look at another class of fixed-precision arithmetic, based on rational numbers. Matula and Kornerup [23] gave a study of such systems. In analogy to fixed point and floating point numbers, we also **fixed-slash** and **floating-slash** rational numbers. In the fixed-slash system, we consider the set of rational numbers of the form $\pm p/q$ where $0 \leq p, q \leq n$ for some n. The representable numbers in this system is the well-known **Farey Series** F_n of number theory. In the floating-slash system, we allow the number of bits allocated to the numerator and denominator to vary. If L bits are used to represent a number, then we need about $\lg L$ bits to indicate this allocation of bits (*i.e.*, the position of the floating slash). Matula and Kornerup address questions of naturalness, complexity and accuracy of fixed-precision rational arithmetic in their paper. Other proposed number system include Hensel's p-adic numbers (e.g., [13, 20]) and continued fractions (e.g., [6]). Note that in p-adic numbers and continued fractions, the number of components is unbounded. For general information about about number systems, see Knuth [19].

All these alternative number systems ultimately need a representation of the natural numbers $\mathbb{N}$. Besides the standard β-ary representation of $\mathbb{N}$, we mention an alternative[1] called **β-adic numbers**. A digit in β-adic number is an element of $\{1, 2, \ldots, \beta\}$, and a sequence of such digits $d_n d_{n-1}, \ldots, d_1 d_0$ represents the number $\sum_{i=0}^{n} d_i \beta^i$. This equation is identical to the one for β-ary numbers; but since the d_i's are non-zero, every natural number has a unique β-adic representation. In particular, 0 is represented by the empty sequence. In contrast to β-adic numbers, the usual β-ary numbers are non-unique: $.1 = 0.100 = 00.1$.

¶2. Floating Point Systems. Given natural numbers $\beta \geq 2$ and $t \geq 1$, the **floating point system** $F(\beta, t)$ comprises all numbers of the form

$$r = m \times \beta^{e-t+1} = \frac{m}{\beta^{t-1}} \beta^e \tag{2}$$

where $m, e \in \mathbb{Z}$ and $|m| < \beta^t$. We call β the **base** and t the **significance** of the system. The pair (m, e) is a **representation** of the number r, where m is the **mantissa** and e the **exponent** of the representation. When exponents are restricted to lie in the range

$$e_{\min} \leq e \leq e_{\max},$$

we denote the corresponding subsystem of $F(\beta, t)$ by

$$F(\beta, t, e_{\min}, e_{\max}). \tag{3}$$

Note that $F(\beta, t)$ is just the system $F(\beta, t, -\infty, +\infty)$. When r is represented by (m, e), we may write

$$\text{float}(m, e) = r.$$

[1] There is a conflict in terminology here when numbers of the form $m2^n$ $(m, n \in \mathbb{Z})$ are called **dyadic numbers**. Such numbers are also known as binary floating point numbers.

For instance, $\text{float}(2^{t-1}, 0) = 1$ and $\text{float}(2^{t-1}, 2) = 4$ in $F(2, t)$.

Sometimes, e is called the **biased exponent** in (2) because we might justifiably[2] call $e - t + 1$ the "exponent". Using the biased exponent will facilitate the description of the IEEE Standard, to be discussed shortly. Another advantage of using a biased exponent is that the role of the t parameter (which controls precision) and the role of $e_{\min}, e_{\max}$ (which controls range) are clearly separated. Thus $e_{\max}$ limits the largest absolute value, and $e_{\min}$ limits the smallest non-zero absolute value. The expression m/β^{t-1} in (2) can be written as $\pm d_1.d_2d_3 \cdots d_t$ where m is a t-digit number $m = \pm d_1 d_2 \cdots d_t$ in β-ary notation. As usual, $d_i \in \{0, 1, \ldots, \beta - 1\}$ are β-ary digits. The equation (2) then becomes

$$r = \pm\beta^e \times d_1.d_2d_3 \cdots d_t = \text{float}(m, e). \tag{4}$$

In this context, d_1 and d_t are (respectively) called the **leading** and **trailing digits** of m. The number $\pm d_1.d_2d_3 \cdots d_t = m/\beta^{t-1}$ is also called the **significand** of r.

In modern computers, β is invariably 2. We might make a case for $\beta = 10$ and some older computers do use this base. In any case, all our examples will assume $\beta = 2$.

We classify the representations (m, e) into three mutually disjoint types:
(i) If $|m| \geq \beta^{t-1}$ or if $(m, e) = (0, 0)$, then (m, e) is a **normal representation**. When $m = 0$, we just have a representation of zero. Thus, the normal representation of non-zero numbers amounts to requiring $d_1 \neq 0$ in (4).
(ii) If $e = e_{\min}$ and $0 < |m| < \beta^{t-1}$ then (m, e) is a **subnormal representation**. Note that when $e_{\min} = -\infty$, there are no subnormal representations since e and m are finite values (by assumption).
(iii) All other (m, e) are **denormalized representations**.

In the following discussion, we assume some $F = F(\beta, t, e_{\min}, e_{\max})$. Numbers in F are said to be **representable** or **floats**. **Normal** and **subnormal numbers** refers to numbers with (respectively) normal and subnormal representations.

We claim that *every representable number is either normal or subnormal, but not both.* In proof, first note that the normal numbers and subnormal numbers are different: assuming $e_{\min} > -\infty$, the smallest non-zero normal number is $\text{float}(\beta^{t-1}, e_{\min}) = \beta^{e_{\min}}$, which is larger than the largest subnormal number $\text{float}(\beta^{t-1} - 1, e_{\min})$. Next, consider any denormalized representation (m, e). There are two possibilities: (a) If $m = 0$ then $float(m, e) = 0$ which is normal. (b) If $m \neq 0$ then $|m| < \beta^{t-1}$. So the leading digit of m is 0. Let $d_i = 0$ for $i = 1, \ldots, k$ and $d_{k+1} \neq 0$ for some $k \geq 1$. Consider the representation $(m\beta^\ell, e - \ell)$ where $\ell = \min\{k, e - e_{\min}\}$. It is easy to see that this is either normal ($\ell = k$) or subnormal ($\ell < k$). This shows that $\text{float}(m, e)$ is either normal or subnormal, as claimed. The transformation

$$(m, e) \longrightarrow (m\beta^\ell, e - \ell), \tag{5}$$

which is used in the above proof, is called **normalization** (even though the result might actually be subnormal).

We claim that *normal and subnormal representations are unique and they can easily be compared.* Subnormal representations are clearly unique. They are also smaller than normal numbers. To compare two subnormal representations, we just compare their mantissas. Next consider two normal representations, $(m, e) \neq (m', e')$. We may assume that $mm' > 0$ since otherwise the comparison can be based on the signs of m and m' alone. If $e = e'$ then clearly the comparison is reduced to comparing m with m'. Otherwise, say $e > e'$. If $m > 0$ then we conclude that $\text{float}(m, e) > \text{float}(m', e')$ as shown in the following:

$$\text{float}(m, e) = \frac{m}{\beta^{t-1}}\beta^e \geq \beta^e \geq \beta^{e'+1} > \frac{m'}{\beta^{t-1}}\beta^{e'} = \text{float}(m', e').$$

If $m < 0$, we similarly conclude that $\text{float}(m, e) < \text{float}(m', e')$.

¶3. Resolution and Range. In the system $F = F(\beta, t, e_{\min}, e_{\max})$, there are two related measures of the "finest resolution possible". One is the **machine epsilon**, $\varepsilon_M := \beta^{1-t}$, which may be defined to be the

[2] It is less clear why we use "$e - t + 1$" instead of "$e - t$". Mathematically, "$e - t$" seems preferable but "$e - t + 1$" is a better fit for the IEEE standard.

distance from 1.0 and the next larger representable number, i.e., float$(\beta^{t-1}+1, 0)$. More important for error analysis is the **unit roundoff**, defined as

$$\mathbf{u} := \varepsilon_M/2 = \beta^{-t}. \tag{6}$$

We wish to define the "range of F". If y is normal then

$$\beta^{e_{\min}} \leq |y| \leq \beta^{e_{\max}} \left(\beta - \beta^{1-t}\right). \tag{7}$$

Thus we define the **normal range** of F to be

$$(-\beta^{1+e_{\max}}, -\beta^{e_{\min}}] \cup [\beta^{e_{\min}}, \beta^{1+e_{\max}}). \tag{8}$$

The **subnormal range** is the open interval $(-\beta^{e_{\min}}, \beta^{e_{\min}})$. Note that 0 is normal but lies in the subnormal range. The **range** of F is the union of the normal and subnormal ranges of S.

A striking feature of F is the non-uniform distribution of its numbers. Informally, the numbers in F becomes more and more sparse as we move away from the origin. This non-uniformity is both a strength and weakness of F. It is a strength because the range of F is exponentially larger than could be expected from a uniformly distributed number system with the same budget of bits. It is a weakness to the extent that algorithms and error analysis based on F are harder to understand.

To understand this non-uniform distribution, we need only consider the non-negative portion of the range of F, $[0, \beta^{1+e_{\max}})$. Subdivide this into half-open intervals of the form $I_e := [\beta^e, \beta^{e+1})$ for $e \in [e_{\min}, e_{\max}]$,

$$[0, \beta^{e_{\max}}) = I_{-\infty} \uplus I_{e_{\min}} \uplus I_{e_{\min}+1} \uplus \cdots \uplus I_{e_{\max}}$$

where $\uplus$ denotes disjoint union and $I_{-\infty}$ is defined to be $[0, \beta^{e_{\min}})$. Note that except for 0, the representable numbers in $I_{-\infty}$ are precisely the subnormal numbers.

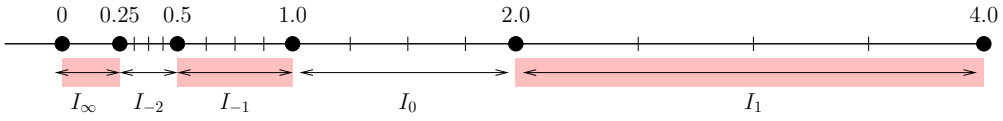


Figure 1: Non-uniform intervals in $F(2, 3, -2, 1)$: normal numbers.

For the example in Figure 1, each interval I_e (for $e \geq e_{\min}$) has exactly 4 normal numbers. In general, each interval I_e contains all the normal representations of the form (m, e), with $\beta^{t-1} \leq m < \beta^t$. There are exactly $\beta^{t-1}(\beta - 1)$ numbers in I_e (this is because in (4), there are $\beta - 1$ choices for d_1 and β choices for $d_2, \ldots, d_t$). The numbers in interval I_e are uniformly spaced β^{e-t+1} apart; multiplying this by the number of normal numbers in I_e, we obtain $\beta^e(\beta - 1)$, which is the width of the interval I_e.

¶4. Rounding. We are familiar with the concept of rounding to the nearest integer: for any real x, we can **round down** to $\lfloor x \rfloor$ (floor function) or **round up** to $\lceil x \rceil$ (ceiling function).

To discuss rounding in general, we must first generalize the ceiling and floor functions so that the role played by integers can be played by any closed set $G \in \mathbb{R}$. Then $\lfloor x \rfloor_G$ and $\lceil x \rceil_G$ denotes the closest number in $G \cup \{\pm\infty\}$ such that

$$\lfloor x \rfloor_G \leq x \leq \lceil x \rceil_G. \tag{9}$$

Thus, $x \in G$ if and only if $\lfloor x \rfloor_G = \lceil x \rceil_G = x$. Viewing G as the "rounding grid", we are rounding to the nearest grid value. We shall be using the floating point system $F = F(\beta, t, e_{\min}, e_{\max})$ as our rounding grid.

There are 7 major "rounding modes". Each mode is identified by a particular **rounding function**, $\mathrm{fl} : \mathbb{R} \to F$. We have the general requirement that all rounding functions satisfy

$$\mathrm{fl}(x) \in \{\lceil x \rceil, \lfloor x \rfloor\} \subseteq F \cup \{\pm\infty\}.$$

We have already seen the two rounding functions:

$$\mathrm{fl}_{1(x)} = \lfloor x \rfloor \tag{10}$$

$$\mathrm{fl}_{2(x)} = \lceil x \rceil \tag{11}$$

Let us describe the remaining 5 rounding functions. The next two rounding modes are **round towards zero** and **round away from zero**. These corresponds to the rounding functions

$$\mathrm{fl}_{3(x)} = \begin{cases} \lfloor x \rfloor & \text{if } x \geq 0 \\ \lceil x \rceil & \text{if } x < 0 \end{cases}. \tag{12}$$

$$\mathrm{fl}_{4(x)} = \begin{cases} \lceil x \rceil & \text{if } x \geq 0 \\ \lfloor x \rfloor & \text{if } x < 0 \end{cases}. \tag{13}$$

$$\tag{14}$$

Another two rounding modes are **round to even** and **round to odd**, respectively. They depend on an additional structure of F: each number $y \in F$ is classified as even or odd, called the **parity** of y. Moreover, two consecutive numbers in F have opposite parity. By convention, the parity of 0 is even, and this uniquely fixes the parity of each number in F. This notion of parity generalizes the usual notion of even or odd integers. Now we may define the corresponding rounding functions

$$\mathrm{fl}_{5(x)} = \begin{cases} x & \text{if } x \in F \\ \text{the value in } \{\lfloor x \rfloor, \lceil x \rceil\} \text{ with odd parity} \end{cases} \tag{15}$$

$$\mathrm{fl}_{6(x)} = \begin{cases} x & \text{if } x \in F \\ \text{the value in } \{\lfloor x \rfloor, \lceil x \rceil\} \text{ with even parity} \end{cases} \tag{16}$$

The last rounding mode is **rounding to nearest**, with rounding function denoted $\mathrm{fl}^*(x)$ or $\lfloor x \rceil$. This is intuitively clear: we choose $\mathrm{fl}^*(x)$ to satisfy the equation

$$|\mathrm{fl}^*(x) - x| = \min\{x - \lfloor x \rfloor, \lceil x \rceil - x\}.$$

Unfortunately, this rule in incomplete because of the possibility of ties. So we invoke one of the other six rounding modes to break ties! Write $\mathrm{fl}_i^*(x)$ (for $i = 1, \ldots, 6$) for the rounding function where tie-breaking is determined by $\mathrm{fl}_i(x)$. Empirically, the variant fl_6^* has superior computational properties, and is the default in the IEEE standard. Hence it will be our default rule, and the notation "$\lfloor x \rceil$" will refer to this variant. Thus, $\lfloor 1.5 \rceil = 2$ and $\lfloor 2.5 \rceil = 2$. Call this the **round to nearest/even** function.

> **Parity Again.** We give an alternative and more useful computational characterization of parity. Recall that each number in F has a unique normal or subnormal representation (m, e); we say $\mathrm{float}(m, e)$ is even iff m is even. Let us prove this notion of parity has our originally stated properties. Clearly, 0 is even by this definition. The most basic is: *there is a unique even number in the set* $\{\lfloor x \rfloor, \lceil x \rceil\}$ *when* $x \notin F$. Without loss of generality, assume $0 \leq \lfloor x \rfloor < \lceil x \rceil$ where $\lfloor x \rfloor = \mathrm{float}(m, e)$ and $\lceil x \rceil = \mathrm{float}(m', e')$. If $e = e'$ then clearly m is even iff m' is odd. If $e = e' - 1$ then $m = \beta^t - 1$ and $m' = \beta^{t-1}$. Again, m is even iff m' is odd. There are two other possibilities: $\lfloor x \rfloor = -\infty$ or $\lceil x \rceil = +\infty$. To handle them, we declare $\pm\infty$ to be even iff β is even.

Let $\mathrm{fl}(x)$ be any rounding function relative to some floating point system F. Let us prove a basic result about fl:

Theorem 1. *Assume $x \in \mathbb{R}$ lies in the range of F and $fl(x)$ is finite.*
(i) Then

$$fl(x) = x(1 + \delta), \qquad |\delta| < 2\mathbf{u} \tag{17}$$

and

$$fl(x) = \frac{x}{1 + \delta'}, \qquad |\delta'| < 2\mathbf{u}. \tag{18}$$

(ii) If $fl(x) = ft^*(x)$ is rounding to nearest, then we have

$$ft^*(x) = x(1+\delta), \qquad |\delta| < \mathbf{u} \tag{19}$$

and

$$ft^*(x) = \frac{x}{1+\delta'}, \qquad |\delta'| \leq \mathbf{u}. \tag{20}$$

Proof. (i) Suppose $x \in I_e$ for some e. If $x = \beta^e$ then $\mathrm{fl}(x) = x$ and the theorem is clearly true. So assume $|x| > \beta^e$. The space between consecutive representable numbers in I_e is β^{e-t+1}. Writing $\Delta = \mathrm{fl}(x) - x$, we obtain $\mathrm{fl}(x) = x + \Delta = x(1 + \Delta/x) = x(1 + \delta)$ where

$$|\delta| = \left|\frac{\Delta}{x}\right| < \frac{\beta^{e-t+1}}{|x|} < \beta^{-t+1}. \tag{21}$$

This proves (17) since $\mathbf{u} = \beta^{1-t}/2$. Note this bound holds even if $\mathrm{fl}(x)$ is the right endpoint of the interval I_e. Similarly, if $\Delta' = x - \mathrm{fl}(x)$ then $x = \mathrm{fl}(x) + \Delta' = \mathrm{fl}(x)(1 + \Delta'/\mathrm{fl}(x)) = x(1 + \delta')$ where

$$|\delta'| = \left|\frac{\Delta'}{\mathrm{fl}(x)}\right| < \frac{\beta^{e-t+1}}{|\mathrm{fl}(x)|} \leq \beta^{-t+1}. \tag{22}$$

(ii) This is similar to (i) applies except that $|\Delta| \leq \mathbf{u}$ (not strict inequality). The analogue of (21) is $|\delta| < \mathbf{u}$, but the analogue of (22) is $|\delta'| \leq \mathbf{u}$ (not strict inequality). **Q.E.D.**

Example 1 (The IEEE Floating Point Systems).
The IEEE single precision numbers is essentially[3] the system

$$F(\beta, t, e_{\min}, e_{\max}) = F(2, 24, -127, 127)$$

with range roughly $10^{\pm 38}$. The IEEE double precision numbers is essentially the system

$$F(\beta, t, e_{\min}, e_{\max}) = F(2, 53, -1023, 1023)$$

with range roughly $10^{\pm 308}$. Note that $e_{\min} = -e_{\max}$ in both systems. The unit roundoffs for these two systems are

$$\mathbf{u} = 2^{-24} \approx 5.96 \times 10^{-8} (\text{single}); \tag{23}$$
$$\mathbf{u} = 2^{-53} \approx 1.11 \times 10^{-16} (\text{double}). \tag{24}$$

The **formats** (i.e., the bit representation) of numbers in these two systems are quite interesting because it is carefully optimized. The bit budget, i.e., total number of bits used to represent the single and double precision numbers, is 32 and 64 bits, respectively. In the double precision format, 53 bits are dedicated to the mantissa ("significand") and 11 bits for the exponent. In single precision, these are 24 and 8 bits, respectively. We will return later to discuss how this bit budget is used to achieve the representation of $F(2, t, e_{\min}, e_{\max})$ and other features of the IEEE standard.

¶5. Standard Model of Floating Point Arithmetic. Let $\circ$ be any basic floating point operation (usually, this refers to the 4 arithmetic operations, although square-root is sometimes included). Let $\circ'$ be the corresponding operation for the numbers in F. The fundamental property of fixed precision floating point arithmetic is this: if $x, y \in F$ then

$$x \circ' y = \mathrm{fl}(x \circ y). \tag{25}$$

This assumes $\circ$ is binary, but the same principle applies for unary operations. Let us call any model of machine arithmetic that satisfies (25) the **strict model**. Thus, the strict model together with Theorem 1 implies the following property

$$x \circ' y = \begin{cases} (x \circ y)(1 + \delta), & |\delta| < \mathbf{u}, \\ \frac{(x \circ y)}{(1+\delta')}, & |\delta'| \leq \mathbf{u}, \end{cases} \tag{26}$$

where $\mathbf{u}$ is the unit roundoff (see (6) and (23)). Any model of machine arithmetic that satisfies (26) is called a **standard model** (with unit roundoff $\mathbf{u}$). Note that if $x \circ y \in F$, then the strict model requires that $\delta = 0$; but this is not required by the standard model. All our error analysis will be conducted under the standard

[3] The next section will clarify why we say this correspondence is only in "essence", not in full detail.

model. NOTATION: As an alternative[4] to the notation $x \circ' y$, we often prefer to use the **bracket notation** "$[x \circ y]$".

We refer to [19, Chapter 4] for the algorithms for arithmetic in floating point systems, and about general number systems and their history. Here is a brief overview of floating point arithmetic. It involves a generic 3-step process: suppose $F = F(\beta, t)$ for simplicity, and we want to perform the binary operation $x \circ' y$ in F,

1. (Scaling) First "scale" the operands so that they share a common exponent. It makes sense to scale up the operand with the smaller magnitude to match the exponent of the operand with larger magnitude: if $x = m2^e$ and $y = n2^f$ where $e \geq f$, then scaling up means y is transformed to $(n/2^{e-f})2^e$. The scaled number may no longer be representable. In general, some truncation of bits may occur.

2. (Operation) Carry out the desired operation $\circ$ on the two mantissas. This is essentially integer arithmetic.

3. (Normalization) Truncate the result of the operation back to t digits of precision. Normalize if necessary.

The Scaling Step is the key to understanding errors in floating point arithmetic: after we scale up the smaller operand, its mantissa may require much more than t digits. All hardware implementation will simultaneously truncate the scaled operand. But truncated to what precision? We might guess that truncating to t digits is sufficient (after all the final result will only have t digits). This is almost right, with one exception: in the case of addition or subtraction, we should truncate to $t+1$ digits. This extra digit is called the **guard digit**. Without this, the hardware will fail to deliver a standard model (26). This was a standard "bug" in hardware before the IEEE standard (Exercise).

Lemma 2 (Sterbenz). *Let a, b be positive normal numbers, and $\frac{1}{2} \leq \frac{a}{b} \leq 2$.*
(i) $a - b$ is representable.
(ii) If we perform subtraction of a and b using a guard digit, we get the exact result $a - b$.

Proof. Note that (ii) implies (i). To show (ii), let the normal representations of a and b be $a = a_1.a_2 \cdots a_t \times 2^{e(a)}$ and $b = b_1.b_2 \cdots b_t \times 2^{e(b)}$, where $a_1 = b_1 = 1$ and $e(a)$ denotes the exponent of the floating point representation of a. Assume $a \geq b$ (if $a < b$, then the operation concerns $-(b-a)$ where the same analysis applies with a, b interchanged). Our assumption on a/b implies that $e(a) - e(b) \in \{0, 1\}$. Consider the case $e(a) - e(b) = 1$. We execute the 3 steps of scaling, operation and normalization to compute $a - b$. To scale, we rewrite b as $0.b_1b_2 \cdots b_t \times 2^{e(a)}$. This new representation needs $t+1$ bits, but with the guard bit, we do no truncation. The subtraction operation has the form

$$\begin{array}{cccccccc} & a_1 & . & a_2 & \cdots & a_t & 0 \\ - & 0 & . & b_1 & \cdots & b_{t-1} & b_t \\ \hline & c_0 & . & c_1 & \cdots & c_{t-1} & c_t. \end{array} \tag{27}$$

Thus $a - b = c_0.c_1 \cdots c_t \times 2^{e(a)}$. It suffices to show that $c_0 = 0$, so that after normalization, the non-zero bits in $c_1 \cdots c_t$ are preserved. Note that $a \leq 2b$; this is equivalent to $a_1.a_2 \cdots a_t \leq b_1.b_2 \cdots b_t$. Therefore $c_0.c_1 \cdots c_t = a_1.a_2 \cdots a_t - 0.b_1b_2 \cdots b_t \leq 0.b_1b_2 \cdots b_t$. This proves $c_0 = 0$. Note that $a-b$ might be a subnormal number. The other possibility is $e(a) = e(b)$. But this case is slightly simpler to analyze. **Q.E.D.**

Note that we cannot guarantee exact results when forming the sum $a + b$, under the assumptions of Sterbenz's Lemma. Complementing Sterbenz's lemma is another "exact" result from Dekker (1971): let $\widetilde{s}$ be the floating point result of adding a and b where we make no assumptions about a/b. Dekker shows, in base $\beta = 2$, the sum $a + b$ can be expressed exactly as $\widetilde{s} + \widetilde{e}$ where $\widetilde{e}$ is another floating point number computed from a, b. See Chapter 4.

We note that in general, the problem of exact rounding is a difficult problem. It is called the Table Maker's Dilemma [27, 22]. The foundations of exact rounding and its connection to Transcendental Number Theory is given in [43].

Exercises

[4] In our notation, "$\mathrm{fl}(x \circ y)$" is not the same as "$[x \circ y]$". The former is simply applying the rounding operator $\mathrm{fl}(\cdot)$ to the exact value $x \circ y$ while the $[\cdots \circ \cdots]$ refers to applying the floating point operation $\circ'$. The $[\cdots \circ \cdots]$ is attributed to Kahan.

Exercise 2.1: Unless otherwise noted, assume the $F(2,t)$ system.
(i) Give the normal representations of $-1, 0, 1$.
(ii) Give the representations of the next representable number after 1, and the one just before 1.
(iii) Give the IEEE double formats of the numbers in (i) and (ii).
(iv) Give the binary representation of machine epsilon in the IEEE double format.
(v) True or False: for any x, $\lceil x \rceil$ and $\lfloor x \rfloor$ are the two closest representable to x. ♢

Exercise 2.2: Determine all the numbers in $F = F(2,3,-1,2)$. What are the subnormal numbers? What is $\mathbf{u}$ and the range of F? ♢

Exercise 2.3: Consider arithmetic in the system $F(\beta,t)$.
(i) Show that the standard model (26) holds for multiplication or division when the scaled operand is truncated to t digits (just before performing the actual operation).
(ii) Show that the standard model (26) fails for addition or subtraction when we truncate to t digits.
(iii) Give the worst case error bound in (ii).
(iv) Show that the standard model holds for addition and subtraction if we have a guard digit. ♢

Exercise 2.4: (i) Give an actual numerical example to show why the guard digit in the scaled operands is essential for addition or subtraction.
(ii) State the error bound guaranteed by addition or subtraction when there is no guard bit. This should be weaker than the standard model. ♢

Exercise 2.5: (Ferguson) Generalize the lemma of Sterbenz so that the hypothesis of the lemma is that $e(a-b) \leq \min\{e(a), e(b)\}$ where $e(a)$ denotes the exponent of a in normal representation. ♢

Exercise 2.6: The area of a triangle with side-lengths of a, b, c is given by a formula

$$\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \qquad s = (a+b+c)/2.$$

This formula was derived in Book I of *Metrica*, by Heron who lived in Alexandria, Egypt, from approximately 10 to 75 A.D.. If the triangle is needle-like (say, c is very small compared to a, b) the straightforward evaluation of this formula using machine arithmetic can be very inaccurate.
(i) Give a straight forward `C++` implementation of this formula. Use the input data found in the first three columns of the following table:

No.	a	b	c	Naive	Kahan's
1	10	10	10	43.30127019	43.30127020
2	−3	5	2	**2.905**	Error
3	100000	99999.99979	0.00029	**17.6**	9.999999990
4	100000	100000	1.00005	500**10.0**	50002.50003
5	99999.99996	99999.99994	0.00003	**Error**	1.118033988
6	99999.99996	0.00003	99999.99994	**Error**	1.118033988
7	10000	50000.000001	15000	**0**	612.3724358
8	99999.99999	99999.99999	200000	**0**	Error
9	5278.64055	94721.35941	99999.99996	**Error**	0
10	100002	100002	200004	0	0
11	31622.77662	0.000023	31622.77661	0.**447**	0.327490458
12	31622.77662	0.0155555	31622.77661	246.**18**	245.9540000

Values in bold font indicate substantial error. Your results should be comparable to the results in the first 4th column.
(ii) Now implement Kahan's prescription: first sort the lengths so that $a \geq b \geq c$. If $c - (a-b) < 0$ then the data is invalid. Otherwise use the following formula

$$\Delta = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}.$$

In these formulas, the parenthesis are significant. Compare your results with the 5th column of the above table.
(iii) Convert your programs in both parts (i) and (ii) into `CORE` programs and run at level III, with guaranteed precision of 64 bits. What conclusions do you draw?

NOTE: This problem is derived from[5] Kahan's paper "Miscalculating Area and Angles of a Needle-like Triangle", July 19, 1999. ♢

End Exercises

§3. The IEEE Standard

The official name for this standard is the "IEEE Standard 754 for Binary Floating-Point Arithmetic" [16, 10]. There is a generalization called the IEEE Standard 854 (1987) which applies to any base and to any word length. It is important to understand some basic properties of this standard because all modern computer arithmetic subscribes to it. This standard was precipitated by a growing problem in numerical computation in the 1980s. As FP computation grew in importance, hardware implementations of floating point arithmetic began to proliferate. The divergence among these architectures caused confusion in the computing community, and numerical software became largely non-portable across platforms. In 1985, the IEEE established the said standard for hardware designers. See the book of Patterson and Hennessy [31] for some of this history. We should properly understand the true significance of this standard vis-á-vis our robustness goal.

- It ensures consistent performance across platforms. This is no mean achievement, considering the confusion preceding its introduction and acceptance.
- Because of its rational design, it can reduce the frequency of nonrobust behavior. But we emphasize that *it does not completely eliminate nonrobustness.*
- There are issues beyond the current standard that remain to be addressed. A key problem is high-level language support for this standard [9]. The IEEE Standard is usually viewed as a hardware standard. Most programming languages still do not support the IEEE standard in a systematic way, or consistently across platforms.

In the previous section, we had presented the system $F(\beta, t, e_{\min}, e_{\max})$ that forms the mathematical foundation of floating point arithmetic. But in an actual computer implementation, we need other features that goes beyond mathematical properties. The IEEE standard provides for the following:

- Number formats. We already mentioned the single and double formats in the previous section. Another format that is expected to have growing importance is the **quadruple precision floating point format** that uses 128 bits (four computer words).
- Conversions between number formats.
- Subnormal numbers. When a computed value is smaller than the smallest normal number, we say an **underflow** has occurred. Using subnormal numbers, a computation is said to achieves a gradual underflow (see below). The older IEEE term for "subnormal numbers" is "denormal numbers".
- Special values (NaN, $\pm\infty$, ± 0). The values $\pm\infty$ are produced when an operation produces a result outside the range of our system. For instance, when we divide a finite value by 0, or when we add two very large positive values. The condition that produces infinite values is called **overflow**. But operations such as $0/0$, $\infty - \infty$, $0 \times \infty$ or $\sqrt{-1}$ result in another kind of special value called **NaN** ("not-a-number"). There are two varieties of NaN (quiet and signaling). Roughly speaking, the quiet NaN represents indeterminate operation which can propagate through the computation (without stopping). But signaling NaN represents an invalid operation (presumably this condition must be caught by the program). Similarly, the zero value has two varieties ± 0.

[5] Available from `http://cs.berkeley.edu/~wkahan/Triangle.pdf`. The `CORE` version of this program may be found with the `CORE` distribution. The above table contains a small correction of the original table in Kahan. The bug was discovered by the `CORE` program, and confirmed by Kahan.

- Unambiguous results of basic operations $(+, -, \times, \div)$ as well as more advanced ones (remainder and $\sqrt{}$). Transcendental functions are not in the standard per se. The basic rule is simple enough to understand: if $\circ'$ is the machine implementation of a binary operation $\circ$, then let $x \circ' y$ should return $\mathrm{fl}(x \circ y)$, i.e., the *correctly rounded* result of $x \circ y$. This applies to unary operations as well.

- Unambiguous results for comparisons. Comparisons of finite values is not an issue. But we need to carefully define comparisons that involve special values. In particular, $+0 = -0$ and NaN is non-comparable. For instance, $NaN < x$ and $NaN \geq x$ and $NaN = x$ are all false. This means that in general, $not(x < y)$ and $(x \geq y)$ are unequal. Comparisons involving $\pm\infty$ is straightforward.

- 4 Rounding modes: to nearest/even (the default), up, down, to zero. These have been discussed in the previous section.

- 5 Exceptions and their handling: invalid result, overflow, divide by 0, underflow, inexact result. The philosophy behind exceptions is that many computation should be allowed to proceed even when they produce infinities, NaN's and cause under- or overflows. The special values ($\pm\infty$ and NaN) can serve to indicate such conditions in the eventual output. On the other hand, if there is a need to handle detect and handle such conditions, IEEE provide the means for this. A single arithmetic operation might cause one or more of the Exceptions to be signals. The program can trap (catch) the signals if desired. Interestingly, one of the exceptions is "inexact result", i.e., when the result calls for rounding.

¶6. Format and Encoding. Let us focus on the IEEE double precision format: how can we represent the system $F(2, 53, -1023, 1023)$ using 64 bits? The first decision is to allocate 11 bits for the exponent and 53 bits for the mantissa. To make this allocation concrete, we need to recall that modern computer memory is divided into 32-bit chunks called (computer) **words**; this is illustrated in Figure 2.

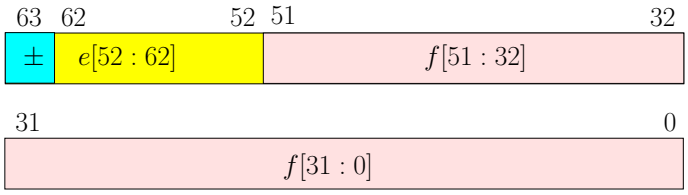


Figure 2: Format of a Double Precision IEEE Float

This physical layout is important to understand if one were to program and manipulate such representations. Two consecutive words are used to represent a double precision number. The bits of a floating point number f are indexed by $0, 1, \ldots, 63$. The 11 exponent bits are in $f[52 : 62]$. The remaining bits, $f[0 : 51]$ and $f[63]$ are allocated to the mantissa. Bit $f[63]$ represents the sign of the mantissa. Although this sign bit logically belongs to the mantissa, it is physically separated from the rest of the mantissa.

We do not allocate any bits for sign in the exponent. So the 11 exponent bits represent an unsigned integer f between 0 and 2047, but we view f as encoding the signed exponent $e = f - 1023$ (here 1023 is called[6] the **exponent bias**). By avoiding a sign for the exponent, we have saved a "half-bit" using this exponent bias trick (Exercise). So e ranges from -1023 to $+1024$. We shall reserve the value $e = 1024$ for special indications (e.g., representing infinity – see below). Hence $e_{\min} = -1023$ and $e_{\max} = 1023$.

Since one of the 53 bits in the mantissa is used for sign, we have only 52 bits to represent the absolute value of the mantissa. Here, we use another trick to gain one extra bit: for normal numbers, the leading bit of a 53-bit mantissa is always 1. So this leading bit does not need to be explicitly represented! But what about subnormal numbers? In this case we declare the implicit bit to be 0. How shall we know whether a number is normal or subnormal? IEEE standard declares that this is determined by the exponent e. A number with exponent e is normal if $e > -1023$, and it is subnormal if $e = -1023$.

The preceding discussion can be directly transferred to the representation of $F(2, 24, -127, 127)$ by the IEEE single precision format: with a 32-bit budget, we allocate 8 bits to the exponent and 24 bits to the

[6] This is distinct from the "biased exponent" idea discussed in the representation (m, e) where e is called the biased exponent.

mantissa. The unsigned exponent bits represents a number between 0 and 255. Using an exponent bias of 127, it encodes an exponent value e between -127 and 128. Again, the exponent $e = -127$ indicates subnormal numbers, and the value $e = 128$ is used for special indication, so $e_{\min} = -127$ and $e_{\max} = 127$.

Let us discuss how the special values are represented in the single or double precision formats. Note that ± 0 is easy enough: the mantissa shows ± 0 (this is possible since there is a dedicated bit for sign), and exponent is 0. The infinite values are represented as $(\pm 0, e_{\max})$ Finally, the NaN values are represented by $(m, e_{\max})$ where $|m| > 0$. If $|m| \geq 2^{t-1}$ then this is interpreted as a quiet NaN, otherwise it is interpreted as a signaling NaN.

■

Let us briefly mention how bits are allocated in quad-precision floating point numbers: 15 bits for the exponent and 113 bits for the mantissa. This is essentially the system $F(2, 113, -16382, 16383)$. The IEEE Standard also provide for extensions of the above types, but we shall not discuss such extensions.

¶7. Is the IEEE doubles really $F(2, 53, -1023, 1023)$? We had hinted that the IEEE double precision may not correspond exactly to $F(2, 53, -1023, 1023)$. Indeed, it is only a proper subset of $F(2, 53, -1023, 1023)$. To see this, we note that there are 2^{53} numbers in $F(2, 53, -1023, 1023)$ with exponent -1023. However, there are only 2^{52} numbers with exponent -1023 in the IEEE standard. What are the missing numbers? These are numbers of the form $\text{float}(m, -1023)$ where $|m| \geq 2^{-52}$. Why are these missing? That is because of the implicit leading bit of the 53-bit mantissa is 0 when $e = -1023$. This means m must have the form $m = 0.b_1 b_2 \cdots b_{52}$. For instance, $\text{float}(2^{-52}, -1023) = 2^{-1023}$ is not representable.

In general, all the normal numbers with exponent $e_{\min}$ are necessarily missing using the IEEE's convention. This creates an undesirable non-uniformity among the representable numbers; specifically, there is a conspicuous gap between the normal and subnormal numbers. But consider the alternative, where we get rid of the special rule concerning the implicit bit in case $e = e_{\min}$. That is, we use the rule that the implicit bit is 1 even when $e = e_{\min}$. The creates a different gap – namely, the gap between 0 and the next representable number is $2^{e_{\min}}$. Thus, the special rule for implicit leading bit gives us 2^{-t+1} values to fill this gap. So, this is where the missing numbers go! This tradeoff is apparently worthwhile, and is known as the **graceful degradation towards** 0 feature of IEEE arithmetic. Why don't we split the difference between the two gaps? See Exercise.

Exercises

Exercise 3.1:

(i) What are the numbers in in $F(2, 24, -127, 127)$ that are missing from the IEEE single precision floats?

(ii) In general, what are the missing numbers in the IEEE version of $F(2, t, e_{\min}, e_{\max})$?

(iii) Recall the two kinds of gaps discussed in the text: the gap G_0 between 0 and the smallest representable number, and the gap G_1 between normal and subnormal numbers. The IEEE design prefers to fill G_0. This is called graceful degradation towards 0 policy. Why is this better than filling in the gap G_1?

(iv) But suppose we split the difference between G_0 and G_1. The numbers with exponent $e_{\min}$ will be split evenly between the two gaps. One way to do this is to look at the mantissa: if the mantissa is odd, we let the implicit bit be 1, and if the mantissa is even, the implicit bit is 0. What are the pros and cons of this proposal? ♢

Exercise 3.2: Refer to the example in Figure 1. The "missing numbers" under the IEEE scheme are those in the interval I_{-2}. The gradual underflow feature is achieved at the expense of moving these numbers into the interval $I_{-\infty}$. The result does not look pretty. Work out a scheme for more uniform floating-point grid near 0: the idea is to distribute 2^{-t+1} values uniformly in $I_{-\infty} \cup I_{\min}$ (say, filling in only the even values) What are the gap sizes in this range? ♢

Exercise 3.3: By using the biased exponent trick instead of allocating a sign bit to the exponent, what have we gained? ♢

End Exercises

§4. Error Analysis in FP Computation

The ostensible goal of error analysis is to obtain an error bound. But [14, p. 71] ("The purpose of rounding error analysis") notes that the actual bound is often less important than the insights from the analysis. The quantity denoted **u** from the previous section will be an important parameter in such analysis. Without the right tool, bounding the error on the simplest computation could be formidable. The art in error analysis includes maintaining the error bounds in a form that is reasonably tight, yet easy to understand and manipulate.

¶8. Summation Problem. To illustrate the sort of insights from error analysis, we will analyze an extremely simple code fragment which sums the numbers in an array $x[1..n]$:

```
s ← x[1];
for i ← 2 to n do
    s ← s + x[i]
```

Let x_i be the floating point number in $x[i]$ and $s_i = \sum_{j=1}^{i} x_i$. Also let $\widetilde{s_i}$ be the value of the variable s after the ith iteration. Thus $\widetilde{s_1} = x_1$ and for $i \geq 2$,

$$\widetilde{s_i} = [\widetilde{s}_{i-1} + x_i]$$

where we use the bracket notation "$[a \circ b]$" to indicate the floating operation corresponding to $a \circ b$. Under the standard model, we have

$$\widetilde{s_i} = (x_i + \widetilde{s}_{i-1})(1 + \delta_i) \tag{28}$$

for some δ_i such that $|\delta_i| \leq \mathbf{u}$. For instance, $\widetilde{s_2} = (x_1 + x_2)(1 + \delta_2)$ and

$$\widetilde{s_3} = ((x_1 + x_2)(1 + \delta_2) + x_3)(1 + \delta_3).$$

One possible goal of error analysis might be to express $\widetilde{s_n}$ as a function of s_n, n and **u**. We shall see to what extent this is possible.

The following quantity will be useful for providing bounds in our errors. Define for $n \geq 1$,

$$\gamma_n := \frac{n\mathbf{u}}{1 - n\mathbf{u}}.$$

Whenever we use γ_n, it is assumed that $n\mathbf{u} < 1$ holds. This assumption $n\mathbf{u} < 1$ is not restrictive in typical applications. For instance, with the IEEE single precision numbers, it means $n < 2^{24}$ which is about 16 billion. Note that $\gamma_n < \gamma_{n+1}$. Another useful observation is this:

$$n\mathbf{u} < 0.5 \Longrightarrow \gamma_n < 2n\mathbf{u}.$$

The following lemma is from Higham [14, p. 69]:

Lemma 3. *Let $|\delta_i| \leq \mathbf{u}$ and $\rho_i = \pm 1$ for $i = 1, \ldots, n$. If $n\mathbf{u} < 1$ then*

$$\prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \theta_n, \tag{29}$$

where $|\theta_n| \leq \gamma_n$.

Before proving this lemma, it is worthwhile noting why the expression on left-hand side of (29) is of interest: you can view this as the accumulated relative error in a quantity (e.g., in $\widetilde{s_i}$) that is the result of a sequence of n floating point operations. *Proof.* We use induction on n. When $n = 1$, the lemma follows from

$$(1 + \delta_1) \leq 1 + \mathbf{u} < \frac{1}{1 + \mathbf{u}} = 1 + \frac{\mathbf{u}}{1 - \mathbf{u}} = 1 + \gamma_1$$

and

$$(1+\delta_1)^{-1} \leq (1-\mathbf{u})^{-1} = 1 + \frac{\mathbf{u}}{1-\mathbf{u}} = 1+\gamma_1.$$

Assuming the lemma for $n \geq 1$, we will prove it for $n+1$. We consider two cases. (1) If $\rho_{n+1} = 1$, then we have

$$(1+\theta_n)(1+\delta_{n+1}) = 1+\theta_{n+1}$$

where $\theta_{n+1} = \theta_n + \delta_{n+1}\theta_n + \delta_{n+1}$. Thus

$$\begin{aligned} |\theta_{n+1}| &\leq \gamma_n + \mathbf{u}\gamma_n + \mathbf{u} \\ &= \frac{n\mathbf{u} + n\mathbf{u}^2 + \mathbf{u}(1-n\mathbf{u})}{1-n\mathbf{u}} \\ &\leq \gamma_{n+1}. \end{aligned}$$

(2) If $\rho_{n+1} = -1$, we have

$$\frac{1+\theta_n}{1+\delta_{n+1}} = 1+\theta_{n+1}$$

where $\theta_{n+1} = \frac{\theta_n - \delta_{n+1}}{1+\delta_{n+1}}$. Thus

$$\begin{aligned} |\theta_{n+1}| &\leq \frac{\theta_n + \mathbf{u}}{1-\mathbf{u}} \\ &< \frac{(n+1)\mathbf{u}}{(1-n\mathbf{u})(1-\mathbf{u})} \\ &< \gamma_{n+1}. \end{aligned}$$

Q.E.D.

Using this lemma, it is now easy to show:

Lemma 4.
(i) $\widetilde{s}_n = \sum_{i=1}^n x_i(1+\theta_{n-i+1})$ *where* $|\theta_i| \leq \gamma_i$.
(ii) If all x_i*'s have the same sign then* $\widetilde{s}_n = s_n(1+\theta)$ *where* $|\theta| \leq \gamma_n$.

Proof. (i) easily follows by induction from (28). To see (ii), we note that when $xy \geq 0$ and $\alpha \leq \beta$, then

$$\alpha x + \beta y = \gamma(x+y) \tag{30}$$

for some $\alpha \leq \gamma \leq \beta$. Thus $x_1(1+\theta_1)+x_2(1+\theta_2) = (x_1+x_2)(1+\theta)$ for some $\min\{\theta_1,\theta_2\} \leq \theta \leq \max\{\theta_1,\theta_2\}$. The result follows by induction. **Q.E.D.**

What insight does this analysis give us? The proof of Lemma 4(i) shows that the backward error associated with each x_i is proportional to its depth in the expression for s_n. We can reduce the error considerably by reorganizing our computation: if the summation of the numbers $x_1, \ldots, x_n$ is organized in the form of a balanced binary tree, then we can improve Lemma 4(i) to

$$\widetilde{s}_n = \sum_{i=1}^n x_i(1+\theta_i) \tag{31}$$

where each $|\theta_i| \leq \gamma_{1+\lceil \lg n \rceil}$. We leave this as an exercise.

¶9. Forward and Backward Error Analysis. Let y be a numerical variable. As a general notation, we like to write $\widetilde{y}$ for an approximation to y, and also δy for their absolute difference $\widetilde{y} - y$. Typically, $\widetilde{y}$ is some computed floating point number. There are two main kinds of error measures: absolute or relative. **Absolute error** in $\widetilde{y}$ as an approximation to y is simply $|\delta y|$. **Relative error** is defined by

$$\varepsilon(\widetilde{y}, y) = \frac{|\delta y|}{|y|}.$$

This is defined to be 0 if $y = \widetilde{y} = 0$ and ∞ if $y = 0 \neq \widetilde{y}$. Generally speaking, numerical analysis prefers to use relative error measures. One reason is that relative error for floating point numbers is built-in; this is clear from Theorem 1.

In error analysis, we also recognized two kinds of algorithmic errors: forward and backward errors. Let $f : X \to Y$ be a function with $X, Y \subseteq \mathbb{C}$. Suppose $\widetilde{y}$ is the computed value of f at $x \in Y$ and $y = f(x)$. How shall we measure the error in this computation? Conceptually, forward error is simple to understand – it measures how far off our computed value is from the true value. Again, this can be absolute or relative: so the **absolute forward error** is $|\delta y|$ and the **relative forward error** is $\varepsilon(\widetilde{y}, y)$. The backward error is how far is x from the input $\widetilde{x}$ for which $\widetilde{y}$ is the exact solution. More precisely, the **absolute backward error** of $\widetilde{y}$ is defined to be the infimum of $|\delta x| = |\widetilde{x} - x|$, over all $\widetilde{x}$ such that $\widetilde{y} = f(\widetilde{x})$. If there is no such $\widetilde{x}$, then the absolute backward error is defined to be ∞. The **relative backward error** of $\widetilde{y}$ is similarly defined, except we use $\varepsilon(\widetilde{x}, x)$ instead of $|\delta x|$.

In general, X, Y are normed spaces (see Appendix). The forward error is based on the norm in range space Y, while backward error is based on the norm in domain space X. Consider our analysis of the summation problem, Lemma 4. The computed function is $f : \mathbb{C}^n \to \mathbb{C}$. Also, let take the ∞-norm in $X = \mathbb{C}^n$. Part (i) gives us a relative backward error result: it says that the computed sum $\widetilde{s}_n$ is the correct value of inputs that are perturbed by at most γ_n. Part (ii) gives us a forward error result: it says that the relative error $\varepsilon(\widetilde{s}_n, s_n)$ is at most γ_n.

It turns out that backward error is more generally applicable than forward error for standard problems of numerical analysis. Let us see why. Note that Lemma 4(ii) has an extra hypothesis that the x_i's have the same sign. How essential is this? In fact, the hypothesis cannot be removed even for $n = 2$. Suppose we want to compute the sum $x + y$ where $x > 0$ and $y < 0$. Then we do not have the analogue of (30) in case $x + y = 0$. Basically, the possibility of cancellation implies that no finite relative error bound is possible.

Less the reader is lulled into thinking that backward error analysis is universally applicable, we consider an example [14, p. 71] of computing the outer product of two vectors: $A = xy^T$ where x, y are n-vectors and $A = (a_{ij})_{i,j} = (x_i y_j)_{i,j}$ is a $n \times n$-matrix. Let $\widetilde{A} = (\widetilde{a}_{ij})$ be the product computed by the obvious trivial algorithm. The forward analysis is easy because $\widetilde{a}_{ij} = a_{ij}(1 + \theta)$ where $|\theta| \leq \mathbf{u}$. But there is no backwards error result because $\widetilde{A}$ cannot be written as $\widetilde{x}\widetilde{y}^T$ for any choice of $\widetilde{x}, \widetilde{y}$, since we cannot guarantee that $\widetilde{A}$ is a rank-one matrix.

In general, we see that for problems $f : X \to Y$ in which the range $f(X)$ is a lower dimensional subset of Y, no backward error analysis is possible. Standard problems of numerical analysis are usually not of this type.

We can combined forward and backward error analysis. In Lecture 3, we return to this issue.

EXERCISES

Exercise 4.1: The inequality $|\theta_n| \leq \gamma_n$ in Lemma 3 is actually a proper inequality, with one possible exception. What is this? ♢

Exercise 4.2: Extend the error analysis for summation to the scalar product problem: $s = \sum_{i=1}^n x_i y_i$. ♢

Exercise 4.3: Write a recursive program for summing the entries in the array $x[1..n]$ so that the error bound (31) is attained. Also, prove the bound (31) for your scheme. ♢

Exercise 4.4: We have noted that the *relative* forward error analysis for $s = \sum_{i=1}^n x_i$ is not possible in general. Carry out the *absolute* forward error analysis. ♢

Exercise 4.5: Suppose we introduce a "composite" error combining relative with absolute. More precisely, let $\varepsilon(\widetilde{y}, y) = \min\{|y - \widetilde{y}|, |y - \widetilde{y}|/|y|\}$. Thus, in the neighborhood around $y = 0$, it is the absolute error that takes effect, and in the neighborhood of $|y| = \infty$, the relative error takes effect. Show that we can how provide a forward error analysis for the general summation problem. ♢

END EXERCISES

§5. Condition Number and Stability

Despite its acknowledged importance in numerical analysis, the concept of "stability" has largely remained an informal notion. The book of Higham [14], for instance, says [14, p. 8] that an algorithm for $y = f(x)$ is "backward stable" if the backward error $\varepsilon(\widetilde{x}, x)$ is "small" in some context-specific sense. The book of Trefethen and Bau [37] devoted several chapters to stability, and offers the most explicit definition of stability. This is reproduced below. Before delving into stability, let us consider a more primitive concept: the condition number.

Suppose

$$f : X \to Y \tag{32}$$

where X, Y are normed spaces. If we want to distinguish between the norms in these two spaces, we may write $\|\cdot\|_X$ and $\|\cdot\|_Y$. But generally, we will omit the subscripts in the norms. For $x \in X$, we again use our convention of writing $\widetilde{x}$ for some approximation to x and

$$\delta x := \widetilde{x} - x.$$

The condition number of f at $x \in X$ is a non-negative number (possibly infinite) that measures the sensitivity of $f(x)$ to x; a larger condition number means greater sensitivity. Again we have the absolute and relative versions of condition number.

The **absolute condition number** is defined as

$$\kappa^A f(x) := \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|f(\widetilde{x}) - f(x)\|}{\|\delta x\|}.$$

The **relative condition number** is defined as

$$\begin{aligned} \kappa^R f(x) &:= \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \left(\frac{\|f(\widetilde{x}) - f(x)\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right) \\ &:= \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \left(\frac{\|f(\widetilde{x}) - f(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|f(x)\|} \right). \end{aligned} \tag{33}$$

When f is understood or immaterial, we may drop the subscripts from the κ-notations. For similar reasons, we may also drop the superscript A or R and simply write 'κ'. Viewing $\kappa : X \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ as a function, we call κ the **condition number function** of f. REMARK: we could also consider how an absolute error in X affects the relative error in Y, or how a relative error in X affects the absolute error in Y. This gives rise to 2 additional concepts of relative error.

Suppose $X \subseteq \mathbb{C}^m$, $Y \subseteq \mathbb{C}^n$ and $f = (f_1, \ldots, f_n)$ is differentiable. We write $\partial_i f$ for the partial derivative function $\frac{\partial f}{\partial x_i}$, and write $\partial_i f[x]$ for the value of this derivative at a given $x \in \mathbb{C}^m$. Let $J_f(x)$ be the **Jacobian** of f at x: this is a $n \times m$ matrix whose (i, j)-th entry given by $\partial_j f_i[x]$. The norm $\|J_f(x)\|$ on the Jacobian will be that induced by the norms in X and Y, namely,

$$\|J_f(x)\| = \sup_x \|J_f(x) \cdot x\|_Y$$

where x range over those elements of X satisfying $\|x\|_X = 1$. Then the absolute and relative condition numbers are given by

$$\kappa^A f(x) = \|J_f(x)\| \tag{34}$$

and

$$\kappa^R f(x) = \frac{\|J_f(x)\| \cdot \|x\|}{|f(x)|}. \tag{35}$$

EXAMPLE 1. Let us give a proof of (35) for the case $f : \mathbb{C}^m \to \mathbb{C}$. By Taylor's theorem with remainder,

$$f(x + \delta x) - f(x) = J_f(x) \circ \delta x + R(x, \delta x)$$

where $\circ$ denotes the dot product of $J_f(x) = [\partial_1 f[x], \ldots, \partial_n f[x]]$ with $\delta x = (\delta x_1, \ldots, \delta x_n) \in \mathbb{C}^n$. Also, we have $\frac{|R(x,\delta x)|}{\|\delta x\|} \to 0$ as $\|\delta x\| \to 0$. Taking absolute values and multiplying by $\frac{\|x\|}{|f(x)| \cdot \|\delta x\|}$, we get

$$\frac{|f(x + \delta x) - f(x)|}{\|\delta x\|} \cdot \frac{\|x\|}{|f(x)|} = \frac{\|x\|}{|f(x)|} \left(\left| J_f(x) \circ \frac{\delta x}{\|\delta x\|} \right| + \frac{|R(x, \delta x)|}{\|\delta x\|} \right).$$

Taking the limsup of the lefthand side as $\|\delta x\| \to 0$, we obtain $\kappa^R f(x)$. But the righthand side equals $\frac{\|x\|}{|f(x)|}\|J_f(x)\|$, by definition of $\|J_f(x)\|$. This completes our demonstration.

EXAMPLE 2. Let us compute these condition numbers for the problem $f : \mathbb{C}^2 \to \mathbb{C}$ where $f(x) = x_1 + x_2$ and $x = (x_1, x_2)^T$. The Jacobian is

$$J_f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right] = [1, 1]$$

Assume the norm on $X = \mathbb{C}^2$ is the ∞-norm, so $\|x\| = \max\{|x_1|, |x_2|\}$. Thus $\|J_f(x)\| = 2$. From (34), we obtain $\kappa^A f(x) = 2$. From (35), we obtain

$$\kappa^R = \frac{2\max\{|x_1|, |x_2|\}}{|x_1 + x_2|}.$$

Thus $\kappa^R = \infty$ when $x_1 + x_2 = 0$.

¶10. Stability of Algorithms. Condition numbers are inherent to a given problem. But the concept of stability depends on the particular algorithm for the problem. We may view an algorithm for the problem (32) as another function $\widetilde{f} : X' \to Y'$ where $X' \subseteq X$ and $Y' \subseteq Y$. In fact, we might as well take $X' = \mathrm{fl}(X)$ and $Y' = \mathrm{fl}(Y)$, i.e., the representable elements of X, Y. The stability of $\widetilde{f}$ is the measure of how much $\widetilde{f}$ deviates from f. Intuitively, we want to define $\widetilde{\kappa}_f(x)$, the (relative or absolute) condition number of $\widetilde{f}$ at x. But we need to be careful since $X' = \mathrm{fl}(X), Y' = \mathrm{fl}(Y)$ are discrete sets. The stability of an algorithm is clearly limited by the condition numbers for the problem. Ideally, a stable algorithm should have the property that $\widetilde{\kappa}_f(x) = O(\kappa_f(x))$.

We now come to our key definition: an algorithm $\widetilde{f}$ for f is **stable** if there exists nonnegative constants $C_0, C_1, \mathbf{u}_0$ such that for all $\mathbf{u}$ $(0 < \mathbf{u} < \mathbf{u}_0)$ and all $x \in X$, and all[7] $\widetilde{x} \in X'$,

$$\frac{\|\widetilde{x} - x\|}{\|x\|} \leq C_0\mathbf{u} \quad \Rightarrow \quad \frac{\|\widetilde{f}(x) - f(\widetilde{x})\|}{\|f(\widetilde{x})\|} \leq C_1\mathbf{u}. \tag{36}$$

According to Trefethen and Bau, the constants C_0, C_1 may not depend on x but may depend on f. In practice, the requirement that $\mathbf{u}$ must be smaller than some $\mathbf{u}_0$ is not an issue. To understand this definition of stability, let us explore some related concepts:

- The requirement (36) is a little subtle: for instance, it might be more obvious to require that for all $\widetilde{x} \in X'$,
$$\frac{\|\widetilde{f}(\widetilde{x}) - f(\widetilde{x})\|}{\|f(\widetilde{x})\|} \leq C_1\mathbf{u}. \tag{37}$$
This would not involve $x \in X \setminus X'$. Of course, (37) is the relative forward error in $\widetilde{f}(x)$. Indeed, this constitutes our definition of **forward stability**.

- If, in the definition of stability above, the constant C_1 is chosen to be 0, we say the algorithm $\widetilde{f}$ is **backward stable** for f. In other words, we can find $\widetilde{x}$ subject to (36) *and* $\widetilde{f}(x) = f(x)$.

- In contrast to forward or backward stability, the view of (36) is to compare $\widetilde{f}(x)$ to the "correct solution $f(\widetilde{x})$ of a nearly correct question $\widetilde{x}$". So this concept of stability contains a mixture of forward and backward error concepts.

- We may rewrite (36) using the standard big-Oh notations:
$$\frac{\|\widetilde{x} - x\|}{\|x\|} = O(\mathbf{u}) \quad \Rightarrow \quad \frac{\|\widetilde{f}(x) - f(\widetilde{x})\|}{\|f(\widetilde{x})\|} = O(\mathbf{u}).$$

[7] Our definition is at variance from [37, p. 104] because they require only the existence of $\widetilde{x} \in X'$, such that $\frac{\|\widetilde{x}-x\|}{\|x\|} \leq C_0\mathbf{u}$ and $\frac{\|\widetilde{f}(x)-f(\widetilde{x})\|}{\|f(\widetilde{x})\|} \leq C_1\mathbf{u}$.

In the big-Oh notations of equations (36) we view $\mathbf{u}$ as varying and approaching 0. As noted above, the implicit constants C_0, C_1 in these big-Oh notations do not depend on x but may depend on f. Typically, $X \subseteq \mathbb{C}^m$ for some fixed m. Thus the implicit constants C_0, C_1 are allowed to depend on m. When $X = \cup_{m\geq 1}\mathbb{C}^m$, then it seems that we should allow the constant C_0, C_1 to have a weak dependence of x: in particular, we would like C_0, C_1 to depend on size(x) where size(x)=m when $x \in \mathbb{C}^m$.

EXAMPLE 3: From Theorem 1, we conclude that the standard algorithms for performing arithmetic operations $(+, -, \times, \div)$ are all stable.

EXAMPLE 4: Consider the problem of computing eigenvalues of a matrix A. One algorithm is to compute the characteristic polynomial $P(\lambda) = \det(A - \lambda I)$, and then find roots of $P(\lambda)$. Trefethan and Bau noted that this algorithm is not stable.

There is a rule of thumb in numerical analysis that says

$$\text{Forward Error} \leq \text{Condition Number} \times \text{Backward Error}.$$

Here is one precise formulation of this insight.

Theorem 5 ([37, p. 151]). *If $\widetilde{f}$ is a backward stable algorithm for f then the relative forward error satisfy*

$$\frac{\|\widetilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\kappa_f(x)\mathbf{u}).$$

Proof. By definition of $\kappa = \kappa^R f(x)$, for all $\varepsilon > 0$ there exists $\delta > 0$ such that

$$\sup_{\|\delta x\|\leq\delta} \left(\frac{\|f(x+\delta) - f(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|f(x)\|} \right) \leq \kappa + \varepsilon. \tag{38}$$

By definition of backward stability, for all $\mathbf{u} < \mathbf{u}_0$ and all $x \in X$, there is δx such that

$$\frac{\|\delta x\|}{\|x\|} = O(\mathbf{u}) \tag{39}$$

and

$$\widetilde{f}(x) = f(x + \delta x). \tag{40}$$

Plugging (40) into (38), we conclude

$$\begin{array}{lll} \frac{\|\widetilde{f}(x)-f(x)\|}{\|\delta x\|}\frac{\|x\|}{\|f(x)\|} & \leq & \kappa + \varepsilon \\ \frac{\|\widetilde{f}(x)-f(x)\|}{\|f(x)\|} & \leq & O(\kappa)\frac{\|\delta x\|}{\|x\|} \quad (\text{choose } \varepsilon = O(\kappa)) \\ & = & O(\kappa\,\mathbf{u}) \quad (\text{by (39)}). \end{array}$$

Q.E.D.

Corollary 6. *If the condition number function of $f : X \to Y$ is bounded, and $\widetilde{f}$ is a backward stable algorithm for f, then $\widetilde{f}$ is a forward stable algorithm for f.*

Exercises

Exercise 5.1: Give the proof (35) in the general case of $f : \mathbb{C}^m \to \mathbb{C}^n$. ♢

Exercise 5.2: (Trefethan and Bau) Compute $\kappa^R f(x)$ for the following functions.
(i) $f : \mathbb{C} \to \mathbb{C}$ where $f(x) = x/2$.
(ii) $f : \mathbb{R} \to \mathbb{R}$ where $f(x) = \sqrt{x}$ and $x > 0$.
(iii) $f : \mathbb{R} \to \mathbb{R}$ where $f(x) = \tan(x)$ and $x = 10^{100}$. ♢

Exercise 5.3: Compute the condition numbers of the following problems:
(i) Linear function $x \in \mathbb{R}^n \mapsto Ax \in \mathbb{R}^m$ where A is an $m \times n$ matrix (cf. Example 1). This is also called the condition number of A.
(ii) Polynomial root finding: the function $F : \mathbb{C}^n \to \mathbb{C}^n$ where $F(a_0, \ldots, a_{n-1}) = (\alpha_1, \ldots, \alpha_n)$ where $x^n + \sum_{i=0}^{n-1} a_i x^i = \prod_{i=1}^{n}(x - \alpha_i)$ ♢

Exercise 5.4: Show that the above algorithm for eigenvalues in EXAMPLE 4 is not stable. HINT: it is enough to show this for the case where A is a 2×2 matrix: show that the error is $\Omega(\sqrt{\mathbf{u}})$. ♢

End Exercises

§6. Arbitrary Precision Computation

In contrast to fixed precision arithmetic, we now consider number types with no *á priori* bound on its precision, save for physical limits on computer memory, etc. We call[8] this **arbitrary precision computation** (or "AP computation") Arbitrary precision computation is becoming more mainstream, but it is still a small fraction of scientific computation. Unlike the ubiquitous hardware support for FP computation, AP computation is only available through software. It is the computational basis for the fields of computer algebra, computational number theory and cryptographic computations. In our quest for robustness, AP computation is one of the first steps.

Computer numbers with arbitrary precision are called **Big Numbers** and the software for manipulating and performing arithmetic with such numbers are usually called **Big Number Packages**. There are many kinds of Big Numbers. The simplest (and the basis for all the other Big Numbers) is the **Big Integer**. An obvious way to represent a Big Integer is an array or a linked list of computer words. If each word is L bits, we can view the list as a number is base 2^L (typically, $L = 32$). The four arithmetic operations on Big Integers is easily reduced to machine arithmetic on these words. The next kind of Big Number is the **Big Rational**. A Big Rational number is represented by a pair of Big Integers (m, n) representing the rational number m/n. Let us write $m : n$ to suggest that it is the ratio that is represented. The rules for reducing rational number computation to integer computation is standard. Big Integers are usually represented internally in some kind of positional representation (so a Big Integer is represented by a sequence of digits $a_1 a_2 \cdots a_n$ where $0 \le a_i < \beta$ for some natural number $\beta > 1$). The third type of Big Number is the **Big Float**: a Big Float number can be viewed as a pair of Big Integers (m, e) representing the floating point number

$$m\beta^e \tag{41}$$

(where m is called the **mantissa** and e the **exponent** and β is the implicit base). In practice, it may be unnecessary to represent e by a Big Integer; a machine long integer may be sufficient. Note that viewing (m, e) as the number (41) is at variance with the system $F(\beta, t)$ introduced in Lecture 2. If we want an analogous system, we can can proceed as follows: assuming that m is also represented in base β, let us define

$$\mu(m) \leftarrow \left\lceil \log_\beta(|m|) \right\rceil . \tag{42}$$

We can think of $\mu(m)$ as the position of the Most Significant Digit (MSD) in m. Then we can alternatively view the pair (m, e) as the number

$$\text{float}(m, e) := \beta^{e-\mu(m)}. \tag{43}$$

Call this the **normalized Big Float** value. Since $\beta^{-1} < m\beta^{-\mu(m)} \le 1$, this means the normalized value lies in the interval $(\beta^{e-1}, \beta^e]$. The advantage of normalization is that we can now compare two Big Floats (of the same sign) by first comparing their exponents; if these are equal, we then compare their mantissas.

Arbitrary precision arithmetic can also be based on Hensel's p-adic numbers [12, 13, 8], or on continued fractions. Unfortunately, these non-standard number representations are hard to use in applications where we need to compare numbers.

[8] Alternatively, "multi-precision computation" or "any precision". However, we avoid the term "infinite precision" since this might be taken to mean "exact" computation.

¶11. On Rationals versus Integers versus Big Floats. Conceptually, Big Rationals do not appear to be different from Big Integers. In reality, Big Rational computations are very expensive relative to Big Integer arithmetic. We view Big Integer arithmetic as our base line for judging the complexity of arbitrary precision computation. One problem with Big Rational numbers, viewed as a pair (p, q) of integers, is their non-uniqueness. To get a unique representation, we can compute the greatest common denominator $m = GCD(p, q)$ and reduce (p, q) to (p', q') where $p' = p/m$ and $q' = q/m$. Without such reductions, the numbers can quickly grow very large. This is seen in the Euclidean algorithm for computing the GCD of two integer polynomials, where the phenomenon is called intermediate expression swell. Karasick et al [18] reported that a straightforward rational implementation of determinants can cause a slow down of 5 orders of magnitude. In their paper, they describe filtering techniques which finally bring the slowdown down to a small constant. This is one of the first evidence of the importance of filters in geometric problems.

On the other hand, Big Float arithmetic is considerably faster than Big Rational arithmetic. *Big Floats recover most of the speed advantages of Big Integer arithmetic while retaining the ability of Big Rationals to provide a dense approximation of the real numbers.* We cannot always replace Big Rationals by Big Floats since the latter represent a proper subset of the rational numbers. However, if approximate arithmetic can be used, the advantages of Big Float should be exploited. This is discussed in Lecture 4.

Let us illustrate the relative inefficiency of rational computations as compared to floating point computation. This example is taken from Trefethen [36]. Suppose we wish to find the roots of the polynomial

$$p(x) = x^5 - 2x^4 - 3x^3 + 3x^2 - 2x - 1.$$

Let us use Newton's method to approximate a root of $p(x)$, starting from $x_0 = 0$. With rational arithmetic, we have the sequence

$$x_0 = 0, x_1 = -\frac{1}{2}, x_2 = -\frac{22}{95}, x_3 = -\frac{11414146527}{36151783550},$$
$$x_4 = -\frac{437115663193076384403256764909499867587929989600855 36}{13863433279008761611840812755838900332126896609091862 5}.$$

The next value is

$$x_5 = -\frac{724391479176820176129001381878925973035003883604754393117804119434357926010580274469629922882064184585670017703551996316651611596343632297460237315758733339908166643200351477598472080210886600668747832494887509884519822479758228984471808467983259225717929917685478944492456273529992130866466313940576741205287553820124064248430069821235453610519870689471522317606875456902898519837650550434545296779211536221568972260935865495519518216876316931568370465908144002495419674804116675018139752278347161906687414800535564210785107754125 0.$$

The growth is the size of the x_i's is quite dramatic (that is exactly the point). Let the "size" of a rational number p/q in lowest terms be the maximum number of digits in p and q. So we see that the sizes of x_2, x_3, x_4, x_5 are (respectively)

$$2, 11, 54, 266.$$

Thus the growth order is 5. This might be expected, since the degree of $p(x)$ is 5. The reader can easily estimate the size of x_6.

In contrast to rational numbers, suppose we use floating point numbers (in fact, IEEE double). We obtain the following Newton sequence:

$$\begin{aligned}
x_0 &= \ \ 0.00000000000000,\\
x_1 &= -0.50000000000000,\\
x_2 &= -0.33684210526316,\\
x_3 &= -0.31572844839629,\\
x_4 &= -0.31530116270328,\\
x_5 &= -0.31530098645936,\\
x_6 &= -0.31530098645933,\\
x_7 &= -0.31530098645933,\\
x_8 &= -0.31530098645933.
\end{aligned}$$

This output is clearly more useful. It also re-inforce our previous remark that it is important to have number representation for which it is easy to compare two numbers (at least for input/output). Positional number representations have this property (we can easily see that the sequence of x_i's is converging).

¶12. The Computational Ring Approach. In general, we are interested in computing in a continuum such as $\mathbb{R}^n$ or $\mathbb{C}$. Most problems of scientific computation and engineering are of this nature. The preceding example might convince us that floating point numbers should be used for continuum computation. But there are still different models for how to do this. We have already seen the standard model of numerical analysis (¶5). One of the problems with the standard model is that it is never meaningful to compute numerical predicates, in which we call for a sharp decision (yes or no). Recall the dictum of numerical analysis – *never compare to zero*. This is a serious problem for geometric computation. For this reason, we describe another approach from [39]. Two other motivations for this approach is that we want to incorporate some algebraic structure into numerical computation, and we want to exploit arbitrary precision (AP) computation.

We introduce numerical computation based on the notion of a "computational ring". We say a set $\mathbb{F} \subseteq \mathbb{R}$ is a **computational ring** if it satisfies the following axioms:

- $\mathbb{F}$ is a ring extension of $\mathbb{Z}$, and is closed under division by 2.
- $\mathbb{F}$ is dense in $\mathbb{R}$
- The ring operations, $x \mapsto x/2$ and exact comparisons in $\mathbb{F}$ are computable.

It is clear the set of dyadic numbers $\mathbb{Z}[1/2]$ is a computational ring, in fact, it is the unique the minimal computational ring. On the other hand, $\mathbb{Q}$, $\mathbb{Z}[1/2, 1/5]$, $\mathbb{Q}[\sqrt{2}]$ or the set of real algebraic numbers can also serve as computational rings. The central questions of this approach is to show that various computational problems defined over the reals are approximable, either in an absolute or relative sense. We will return to this theory later.

The computational ring approach ought to be contrasted to the standard model of numerical analysis ¶5. In particular, deciding zero is a meaningful question, unlike the standard model. This property is critical for many basic problems.

Exercises

Exercise 6.1: Show that the behavior of Trefethan's polynomial under Newton iteration is typical rather than an anomaly. ♢

Exercise 6.2: Discuss the issues that might arise because of the choice of specific computational rings. ♢

End Exercises

§7. Interval Arithmetic

The above example leads us to the concept of **significance arithmetic**. Intuitively, when computing the Newton sequence $x_0, \ldots, x_5$ using rational arithmetic, we are propagating a lots of "insignificant digits". In contrast, computing the same sequence using floating point arithmetic allows us to keep only the insignificant digits. Properly speaking, this notion of "significant digits" is only meaningful for positional number representations, and not to rational number representations. We may regard the last digit in positional numbers as a rounded figure. Thus "1.2" ought to be regarded as a number between 1.15 and 1.25. Informally, significance arithmetic is a set of rules for doing arithmetic with uncertain numbers such as 1.2. For instance, the rules tell us $(1.2)^2$ should not be 1.44 but 1.4, because there is no significance in the last digit in 1.44, and we should not propagate insignicance in our answers. But we would be wrong to treat *all* numbers as uncertain. Often, integers and defined conversion rates or units are exact: there are exactly 60 seconds in a minute and $C = 5(F - 32)/9$ is an exact temperature formula.

We can introduce a precise notion of significance arithmetic for floating point numbers. Assuming the normalized Big Float system (43), we say that the big float number (m, e) has $\mu(m)$ significant digits (see (42)). Instead of the above implied uncertainty in the last digit, we we now introduce an uncertainty $u \geq 0$ into this representation: the triple (m, e, u) represents the interval

$$[\text{float}(m - u, e), \text{float}(m + u, e)]. \tag{44}$$

In this case, the significance of (m, e, u) is defined to be $\mu(m) - \mu(u)$. This means that we can only trust the first $\mu(m) - \mu(u)$ digits of float(m, e). Significance arithmetic is usually traced to the work of Metropolis [2, 24]. In the original Core Library (known as Real/Expr), our BigFloat numbers are based on this representation.

Significance arithmetic can be regarded as a special case of interval arithmetic, where we attempt to only retain the uncertainty warranted by the input uncertainty. In interval arithmetic, each number x is represented by an interval $I = [a, b]$ that contains x. The usual arithmetic operations can be extended to such intervals in the natural way (see below). The development and analysis of techniques for computation with intervals constitute the subject of **interval arithmetic**. Taking a larger view, the subject is also[9] known as **validated computation**. R.E. Moore [26] is the pioneer of interval arithmetic. The book of Rokne and Ratschek [32] gives an excellent account of interval functions. Stahl [35] gives an comprehensive treatment of interval arithmetic generally and interval functions in particular, including extensive experimental comparisons of interval techniques. See [40, 29] for algorithms with a priori guaranteed error bounds for elementary operations, using relative and absolute (as well as a composite) error.

A motivation for interval analysis is the desire to compute error bounds efficiently and automatically. In [1], this is called the "naïve outlook"; instead, it is suggested that the proper focus of the field ought to be about computing with inexact data (e.g., solving a linear system with interval coefficients) and, for problems with exact data, the issues of algorithmic convergence when we approximate the computing using intervals.

¶13. For $D \subseteq \mathbb{R}$, let $\mathbb{I}D$ or $\mathbb{I}(D)$ denote the set of closed intervals $[a, b]$ with endpoints $a, b \in D$ and $a \leq b$. Note that our definition does not insist that $[a, b] \subseteq D$. For instance, a very important example is when $D = \mathbb{F} := \{m2^n : m, n \in \mathbb{Z}\}$, the set of dyadic numbers. Then $\mathbb{I}\mathbb{F}$ denotes the set of intervals whose endpoints are dyadic numbers.

The n-fold Cartesian product of $\mathbb{I}D$ is denoted $(\mathbb{I}D)^n$ or $\mathbb{I}^n D$. Elements of $\mathbb{I}^n D$ are called **boxes** (or **n-boxes**). A typical box $B \in \mathbb{I}^n D$ has the form $B = I_1 \times \cdots \times I_n$ where the I_j are intervals, called the **jth projection** of B. We also write $I_j(B)$ for the jth projection. So intervals are just boxes with $n = 1$.

Let $\ell(I)$ and $u(I)$ denote the lower and upper endpoints of interval I. So $I = [\ell(I), u(I)]$. Sometimes, we also write $\overline{I}$ and $\underline{I}$ for $\ell(I)$ and $u(I)$. Let $m(I) := (\ell(I) + u(I))/2$ and $w(I) := u(I) - \ell(I)$ denote, resp., the **midpoint** and **width** of I. Two other notations are useful: the **mignitude** and **magnitude** of an interval I is

$$mig(I) := \min\{|x| : x \in I\}, \qquad mag(I) := \max\{|x| : x \in I\}.$$

For an n-box B, let its ith component I_i be denoted $I_i(B)$, and so $B = \prod_{i=1}^n I_i(B)$. define its midpoint to be $m(B) := (m(I_1(B)), \ldots, m(I_n(B)))$. We need several width concepts. First, let the **i-th width** be defined as $w_i(B) := w(I_i(B))$ (for $i = 1, \ldots, n$). We may define the **width vector** by $W(B) := (w_1(B), \ldots, w_n(B))$. More useful for us is the **width** and **diameter** of B is given by

$$w(B) := \min\{w_1(B), \ldots, w_n(B)\}, \qquad d(B) := \max\{w_1(B), \ldots, w_n(B)\}. \tag{45}$$

Thus, $w(B) = d(B)$ if $n = 1$. When $w(B) > 0$, we call B a **proper box**; and when $d(B) = 0$, we call B a **point** (or **point box**). So an improper box that is not a point must satisfy $d(B) > 0 = w(B)$. We regard $\mathbb{R}^n$ as embedded in $\mathbb{I}^n\mathbb{R}$ by identifying elements of $\mathbb{R}^n$ with the point boxes.

The endpoints of an interval I is generalized to the **corners** of a box B: these are points of the form $(c_1, \ldots, c_n)$ where c_j is an endpoint of $I_j(B)$. Thus, an n-box B has 2^d corners for some $d = 0, 1, \ldots, n$: if B is proper, $d = n$ and if B is a point, $d = 0$. Two of these corners are given a special notation:

$$\ell(B) := \underline{B} := (\ell(I_1(B)), \ldots, \ell(I_n(B))), \qquad u(B) := \overline{B} := (u(I_1(B)), \ldots, u(I_n(B)))$$

and we continue to call $\ell(B), u(B)$ the "endpoints" of B.

[9] Instead of "validated", such computations are also described as "certified" or "verified", and sometimes "guaranteed". The error bounds in such computations are á posteriori ones. We prefer to reserve the term "guaranteed" for the stronger notion of á priori error bounds.

¶14. Metric and Convergence. We introduce the **Hausdorff metric** on $\mathbb{IR}$ by defining $d_H(A,B) = \max\{|a-b|, |a'-b'|\}$ where $A = [a,a'], B = [b,b']$. This is a metric because $d_H(A,B) \geq 0$ with equality iff $A = B$, $d_H(A,B) = d_H(B,A)$ and finally, the triangular inequality holds: $d_H(A,C) \leq d_H(A,B) + d_H(B,C)$.

We extend this metric to $\mathbb{I}^n\mathbb{R}$ where

$$d_H(A,B) := \max\{d_H(A_i,B_i) : i = 1,\ldots,n\}. \tag{46}$$

Let $\widetilde{B} = (B_0, B_1, B_2, \ldots)$ be an infinite sequence of n-boxes. We say that $\widetilde{B}$ **converges** to a box B, written $\lim_{i\to\infty} B_i = B$ if $\lim_{i\to\infty} d_H(B_i, B) = 0$. If, in addition, the sequence also has the property that $w(B_i) > 0$ and $B_{i-1} \supseteq B_i$, for all i, and $\lim_{i\to\infty} d(B_i) = 0$, then we say that $\widetilde{B}$ is **properly convergent** to B. In this case, B must be an point, i.e., $d(B) = 0$.

So $\widetilde{B}$ can be improperly convergent in two basic ways: either $\lim_{i\to\infty} d(I_i) = d_0 > 0$, or $w(B_i) = 0$ for some i. The importance of proper convergence is discussed below in the context of box functions.

¶15. Representation of Boxes. The most obvious representation of boxes is the **end-point representation**, *i.e.*, a box B is represented by the pair $(\ell(B), u(B)) = (\underline{B}, \overline{B})$.

The **mid-point representation** of B is the pair $(m(B), W(B)/2)$ where $W(B) = (w_1(B), \ldots, w_n(B))$ is the width vector of B. If we think of a box as representation an unknown point $p \in \mathbb{R}^n$, then $m(B)$ is the **representative value** of p and $W(B)/2$ is the **uncertainty** in our knowledge of p. This is reminescent of the representation of bigfloats with error in (44). In general, the mid-point representation of an n-box B is the pair of points $(m(B), W(B)/2)$ where $W(B) = (w_1(B), \ldots, w_n(B))$.

The mid-point representation is more compact than the end-point representation, since we generally use fewer bits. The Big Float package implemented in our `Real/Expr` and `Core Library` (version 1) [42] uses this representation. The down side of this is that arithmetic operations in this representation may be slightly more complicated and its width increases faster. In practice, we may further restrict the uncertainty to a fixed bit budget (see Exercise), adding a further need for "normalization" in the representation. Rump [33] has pointed out that the midpoint representation is uniformly bounded by a factor of 1.5 in optimum radius for the 4 basic arithmetic operations as well as for vector and matrix operations over reals and complex numbers. Moreover, this can take advantage of vector and parallel architectures. We leave it as an exercise to work out the implementation of arithmetic based on midpoint representations.

¶16. Interval Arithmetic. We now extend the basic operations and predicates on real numbers to intervals. But it is useful to see them as special cases of definitions which apply to arbitrary sets of real numbers. In the following, let $A, B \subseteq \mathbb{R}$.

1. We can compare A and B in the natural way: we write $A \geq B$ to mean that the inequality $a \geq b$ holds for all $a \in A, b \in B$. The relations $A \leq B$, $A > B$ and $A < B$ are similarly defined. If $A \leq B$ or $A \geq B$, we say A and B are **comparable**; otherwise they are **incomparable**. If $A \leq B$ and $B \leq A$, then $A = B$ is a point. Note that $A < B$ implies $A \leq B$ and $A \neq B$, but the converse fails. In case A, B are intervals, these relations can be reduced to relations on their endpoints. For instance, $A \leq B$ iff $u(A) \leq \ell(B)$.

2. If $\circ$ is any of the four arithmetic operations then $A \circ B$ is defined to be $\{a \circ b : a \in A, b \in B\}$. It is assume that in case of division, $0 \notin B$. In case A, B are intervals, it is easy to see that $A \circ B$ would be intervals. Moreover, $A \circ B$ can be expressed in terms of operations on the endpoints of A and B:

   - $A + B = [\ell(A) + \ell(B), u(A) + u(B)]$
   - $A - B = [\ell(A) - u(B), u(A) - u(B)]$
   - $A \cdot B = [\min S, \max S]$ where $S = \{\ell(A)\ell(B), \ell(A)u(B), u(A)\ell(B), u(A)u(B)\}$
   - $1/B = [1/u(B), 1/\ell(B)]$
   - $A/B = A \cdot (1/B)$

¶17. Algebraic Properties. The set $\mathbb{IR}$ under these operations has some of the usual algebraic properties of real numbers: $+$ and $\times$ are both commutative and associative, and have $[0,0]$ and $[1,1]$ (respectively) as their unique identity elements. An interval is a zero-divisor if it contains 0. Note that no proper interval I has any inverse under $+$ or $\times$. In general, we have **subdistributivity**, as expressed by the following lemma:

Lemma 7 (Subdistributivity Property). *If A, B, C are intervals, then $A(B+C) \subseteq AB + AC$.*

The proof is easy: if $a(b+c) \in A(B+C)$ where $a \in A$, etc, then clearly $a(b+c) = ab + ac \in AB + AC$.

When is this inclusion an equality? To claim equality, if $ab + a'c \in AB + AC$, then suppose there is some $a'' \in A$ such that $a''(b+c) = ab + a'c$. Thus means $a'' = (ab + a'c)/(b+c)$. In case $bc \geq 0$, then this means a'' is a convex combination of a and a', and so $a'' \in A$. This shows:

$$BC \geq 0 \implies A(B+C) = AB + AC.$$

If $bc < 0$, the equality $A(B+C) = AB + AC$ may be false. For instance, choose $A = [a', a]$, $b = 2, c = -1$. Then $(ab + a'c)/(b+c) = 2a - a'$. Thus $2a - a' > a$ andso $(ab + a'c)/(b+c) \notin A$. Thus we obtain a counterexample with $A = [a', a], B = [2,2], C = [-1,-1]$.

Note that the basic operations $\circ$ are **monotone**: if $A \subseteq A'$ and $B \subseteq B'$ then $A \circ B \subseteq A' \circ B'$. This is also called **isotonicity**.

¶18. Complex Intervals. Most of our discussion concern real intervals. There is the obvious and simple of interval arithmetic for complex numbers: a pair $[a, b]$ of complex numbers represents the complex "interval" $\{z \in \mathbb{C} : a \leq z \leq b\}$ where $a \leq b$ means that $\texttt{Re}(a) \leq \texttt{Re}(b)$ and $\texttt{Im}(a) \leq Im(b)$. Many of our discussions generalize directly to this setting. One generalization of the midpoint representation to complex numbers introduces the geometry of balls: given $z \in \mathbb{C}, r \geq 0$, the pair (z, r) can be viewed as representing the ball $\{w \in \mathbb{Z} : |w - z| \leq r\}$.

¶19. Real versus Dyadic Interval Functions. A function of the form

$$F : \mathbb{I}^n\mathbb{R} \to \mathbb{I}^m\mathbb{R} \tag{47}$$

is called a **interval function**. Interval functions are the central implementation objects for exact numerical algorithms. Unfortunately, the definition (47) is unsuitable for implementation. We cannot hope to implement functions that accept arbitrary input boxes from $\mathbb{I}^n\mathbb{R}$. What we could implement are functions that take dyadic n-boxes and return dyadic m-boxes, i.e., interval functions of the form

$$F : \mathbb{I}^n\mathbb{F} \to \mathbb{I}^m\mathbb{F}. \tag{48}$$

To emphasize that the difference between (47) and (48), we call the former **real interval functions**, and the latter **dyadic interval functions**. Of course, $\mathbb{F}$ could be replaced by any computational ring. Note that it is meaningful to speak of dyadic interval functions as **recursive** in the sense of being computable by halting Turing machines; in contrast, it is non-obvious what "computability" means for real interval functions.

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a real function. For any set $S \subseteq \mathbb{R}^n$, let $f(S)$ denote the set $\{f(a) : a \in S\}$. We can think of f as having been "naturally extended" into a function from subsets of $\mathbb{R}^n$ to subsets of $\mathbb{R}^m$.

A dyadic interval function $F : \mathbb{I}^n\mathbb{F} \to \mathbb{IF}$ is called a **dyadic interval extension** of $f : \mathbb{R}^n \to \mathbb{R}$ if $f(B) \subseteq F(B)$ for every dyadic box $B \in \mathbb{I}^n\mathbb{F}$. A real interval extensions of f can be similarly defined. We use the notation "$\square f$" to denote a (real or dyadic) interval extension of f. We derive from f a real interval function $[[f]]$ given by

$$[[f]](B) := [[f(B)]]$$

where $[[X]] \subseteq \mathbb{R}^n$ is the smallest n-box containing a set X.

¶20. Box Functions. We come to a key definition in our development: a dyadic interval function F is called a **box function** if there exists a real function $f : \mathbb{R}^n \to n$ such that (1) F is an interval extension of f, and (2) for every properly convergent sequence $(B_0, B_1, B_2, \ldots)$ of dyadic n-boxes with $\lim_{i\to\infty} B_i = p \in \mathbb{R}^n$, we have

$$\lim_i d(F(B_i)) = f(\lim_i B_i).$$

We also say F is a **box function for** f, and we usually write $F = \square f$.

This use of proper convergence is critical in this definition because, these are only kind we can implement. Suppose we allow *any* convergent sequence $(B_0, B_1, \ldots)$ in the above definition of box functions. Then we see that the function $f(x) = \exp(x)$ cannot have any box functions. Suppose F is a dyadic interval extension of f. If it is a box function, then for any nonzero $a \in \mathbb{F}$, we can choose $(B_0, B_1, \ldots)$ such that $B_i = [a, a]$ for all i. This sequence converges to a, and so by definition of a box function, $F([a, a]) = \exp(a)$. But this is impossible since $\exp(a)$ is transcendental for any non-zero a. On the other hand, we can compute box functions F with the property that for any properly convergent sequence $(B_0, B_1, \ldots) \to p$, $\lim_i F(B_i) = \exp(p)$. This tells us that we must distinguish between input boxes B where $w(B) = 0$ from those where $w(B) > 0$. The behavior of F on points (i.e., B with $w(B) = 0$) is irrelevant in the definition of box functions. The reason we disregard $F(B)$ where $B = p$ is a point is because it may not impossible to compute an exact value of $f(p)$. Instead of $\exp(x)$, we could use $\sqrt{x}$ as an example; but this requires a treatment of partial functions which we were avoiding to keep this treatment simple.

We next discuss practical construction of box functions. As another example, the midpoint function $m : \mathbb{I}^n\mathbb{F} \to \mathbb{F}$ is a box function. In the Exercise, we will prove some closure properties of box functions.

Another commonly satisfied property of box functions is (inclusion) **isotony**: $B \subseteq B'$ implies $\Box f(B) \subseteq \Box f(B)$. It is easy to check that the four arithmetic operations implemented in the standard way is inclusion isotone. It follows that if $\Box f$ is defined by the interval evaluation of any rational expression defining f.

¶21. Examples of Box Functions. We consider the problem of constructing box functions. It is easy to see that the basic arithmetic operations $(\pm, \times, \div)$ defined above are box functions. It follows that if f is any rational function, and we define $\Box f(B)$ by evaluating a fixed rational expression for f, then the result is a box function for f. Suppose f is a univariate polynomial, $f = \sum_{i=0}^{m} a_i X^i$. Here are some possibilities.

- In most applications, we may restrict ourselves to intervals $I \in \mathbb{IR}$ with a definite sign: i.e., either $I = 0$ or $I > 0$ or $I < 0$. Let us assume $I > 0$ in the following; the case $I < 0$ is treated analogously, and the case $I = 0$ is trivial. We write $f = f^+ - f^-$ where f^+ is just the sum of those terms of f with positive coefficients. Then we may define $f^- := f^+ - f$. If $I = [a, b]$, then define the box function

$$\Box_1 f(I) = [f^+(a) - f^-(b), f^+(b) - f^-a].$$

  It is not hard to verify that $\Box_1 f$ is a box function.

- We can also define box functions by specifying a fully parenthesized expression E for f. For instance, Horner's rule for evaluating f gives rise to the expression

$$E = (\cdots((a_m X + a_{m-1})X + a_{m-2}X + \cdots + a_0).$$

  Now, we can define the box function $\Box_E f(I)$ which returns the interval if we evaluate f on I using the expression E. For instance, if $f = 2X^2 - 3X + 4$ then Horner's expression for f is $E = ((2X - 3)X + 4)$. If $I = [1, 2]$ then

$$\begin{aligned} \Box_E f(I) &= ((2I - 3)I + 4) = (([2, 4] - 3)[1, 2] + 4) \\ &= ([-1, 1][1, 2] + 4) = [-2, 2] + 4] = [2, 6]. \end{aligned}$$

  If E is Horner's expression for f, we may write $\Box_0 f$ instead of $\Box_E f$.

- Consider a third way to define box functions, where E is basically the expression given by the standard power basis of f: namely, we evaluated each term of f, and sum the terms. Call the corresponding box function $\Box_2 f$. We can show that $\Box_2 f$ is the same as $\Box_1 f$. Moreover, for all $I \in \mathbb{IR}$,

$$\Box_0 f(I) \subseteq \Box_1 f(I).$$

  This follows from the subdistributivity property of interval arithmetic.

In some applications, box functions suffice. E.g., Plantinga and Vegter (2004) shows that the isotopic approximation of implicit non-singular surfaces can be achieved using box functions. Sometimes, we want additional properties. For instance, the ability to specify a precision parameter $p > 0$ such that

$$\square f(B; p) \subseteq [[f(B) \oplus E_p]]$$

where $\oplus$ is Minkowsky sum and $E_n = 2^{-p}[-1, 1]^n$.

EXAMPLE 1. Suppose we want to solve the equation $AX = B$ where $A = [a, a'] \neq 0$. Define $\chi(A) = a/a'$ if $|a| \leq |a'|$, and $\chi(A) = a'/a$ otherwise. There is a solution interval $X \in \mathbb{IR}$ iff $\chi(A) \geq \chi(B)$. Moreover the solution is unique unless $\chi(A) = \chi(B) \leq 0$.

Under the Hausdorff metric, we can define the concept of continuity and show that the four arithmetic operations are continuous.

Let $f(x)$ be a real function. If $X \in \mathbb{IR}$, we define $f(x) = [\underline{a}, \overline{a}]$ where $\underline{a} = \min_{x \in X} f(x)$ and $\overline{a} = \max_{x \in X} f(x)$. Let $E(x), E'(x)$ be two real expressions which evaluates to $f(x)$ when the input intervals are improper. The fact that certain laws like commutativity fails for intervals means that E and E' will in general obtain different results when we evaluate them at proper intervals.

EXAMPLE 2. Let the function $f(x) = x - x^2$ be computed by the two expressions $E(x) = x - x^2$ and $E'(x) = x(1-x)$. When x is replaced by the interval $X = [0, 1]$ then $f([0, 1]) = \{x - x^2 : 0 \leq x \leq 1\} = [0, 1/4]$. But $E([0, 1]) = [0, 1] - [0, 1]^2 = [0, 1] - [0, 1] = [-1, 1]$ and $E'([0, 1]) = [0, 1](1 - [0, 1]) = [0, 1][0, 1] = [0, 1]$.

In fact, we have the following general inclusion:

$$f(X) \subseteq E(X)$$

for all $X \in \mathbb{IR}$. In proof, note that every $y \in f(X)$ has the form $y = f(x)$ for some $x \in X$. But it is clear that the value $f(x)$ belongs to $E(X)$, since it can be obtained by evaluating $E(X)$ when every occurrence of X in $E(X)$ is replaced by x.

A simpler example is $E(x) = x - x$ and $f(x) = x - x$. If $X = [0, 1]$ then $E(X) = [-1, 1]$ while $f(X) = 0$ (in fact, $f(Y) = 0$ for any interval Y).

¶22. Rounding and interval arithmetic. Machine floating point numbers can be used in interval arithmetic provided we can control the rounding mode. In the following, assume machine numbers are members of $F(\beta, t)$ for some base $\beta > 1$ and precision $t \geq 1$. We need 2 kinds of rounding: for any real number x, let round up $\mathrm{fl}_{up}(x)$ and round down $\mathrm{fl}_{down}(x)$ be the closest numbers in $F(\beta, t)$ such that $\mathrm{fl}_{down}(x) \leq x \leq \mathrm{fl}_{up}(x)$. Then $A = [a, b] \in \mathbb{IR}$ can be **rounded** as $\mathrm{fl}(A) = [\mathrm{fl}_{down}(a), \mathrm{fl}_{up}(b)]$. If we view $x \in \mathbb{R}$ as an interval, we now have $fl(x) = [\mathrm{fl}_{down}(x), \mathrm{fl}_{up}(x)]$. If $x \in F(\beta, t)$ and the exponent of x is e then the width of $fl(x)$ is β^{-t+e}. Rounding can be extended to the arithmetic operations in the obvious way: if $\circ \in \{+, -, \times, \div\}$ and $\circ'$ is the interval analogue, we define $A \circ' B := fl(A \circ B)$. Inclusion monotonicity is preserved: If $A \subseteq A', B \subseteq B'$ then $A \circ' B \subseteq A' \circ' B'$.

In practice, it is inconvenient to use two rounding modes within a computation. One trick is to store the interval $A = [a, b]$ as the pair $(-a, b)$ and use only round up. Then $\mathrm{fl}(A)$ is represented by the pair $(\mathrm{fl}_{up}(-a), \mathrm{fl}_{up}(b))$.

¶23. Machine arithmetic. One class of results in interval analysis addresses the question: suppose we compute with numbers in $F(\beta, t)$. This is a form of idealized machine arithmetic in which we ignore issues of overflow or underflow. How much more accuracy do we gain if we now use numbers in $F(\beta, t')$ where $t' > t$? If $A = [a, b]$, let the **width** of A be $w(A) := b - a$. $w(\mathrm{fl}(x)) \leq \beta^{e-t}$ where e is the exponent. Any real number x can be represented as

$$x = \beta^e \left(\sum_{i=1}^{\infty} d_i \beta^{-i} \right)$$

with $0 \leq d_i \leq \beta - 1$. This representation is unique provided it is not the case that each digit $d_i = \beta - 1$ for all i beyond some point. Then $\mathrm{fl}_{down}\, x = \beta^e \left(\sum_{i=1}^{t} d_i \beta^{-i} \right)$ and $\mathrm{fl}_{up}\, x \leq \mathrm{fl}_{down}(x) + \beta^{e-t}$. Hence $w(\mathrm{fl}(x)) \leq \beta^{e-t}$. The following is a basic result (see [1, theorem 5, p.45]).

Theorem 8. *Let an algorithm be executed using machine arithmetic in $F(\beta, t)$ and also in $F(\beta, t')$ for some $t' > t$. Assuming both computations are well-defined, then the relative and absolute error bounds for the result is reduced by a factor of $\beta^{t-t'}$ in the latter case.*

¶24. Lipschitz Condition and Continuity. Let $\square f : \mathbb{I}^n D \to \mathbb{R}$ be a box function for some $D \subseteq \mathbb{R}$, and let $r \geq 1$. We say $\square f$ has **convergence of order** r if for every $A \in \mathbb{I}^n D$, there exists a constant K_A such that for all $B \in \mathbb{I}^n \mathbb{R}$, $B \subseteq A$ implies

$$d(\square f(B)) \leq K_A d(B)^r. \tag{49}$$

We say $\square f$ has **linear** (resp., **quadratic**) convergence if $r = 1$ (resp., $r = 2$). When $r = 1$, the constants K_A are called **Lipschitz constants** for A, and $\square f$ is also said to be **Lipschitz**. It is possible to choose global constant K such that $K_A = K$ for all $A \in \mathbb{I}^n D$.

E.g., the addition function $\square f_+ : \mathbb{I}^2\mathbb{R} \to \mathbb{IR}$ given by

$$\square f_+(A, B) = A + B$$

has a global Lipschitz constant $K = 2$:

$$d(\square f_+(A, B)) = d(A + B) = (\overline{A} + \overline{B}) - (\underline{A} + \underline{B}) = w(A) + w(B) \leq 2 \max\{w(A), w(B)\} = 2d(A, B).$$

Similarly the subtraction function has a global Lipschitz constant 2. However, the multiplication function $\square f_\times : \mathbb{I}^2\mathbb{R} \to \mathbb{IR}$ where

$$\square f_\times(A, B) = A \times B$$

is Lipschitz with no global Lipschitz constant:

$$d(\square f_\times(A, B)) = d(AB) \geq \overline{AB} - \overline{AB} \geq \overline{A}(\overline{B} - \underline{B}) = \overline{A} w(B)$$

(assuming $A, B > 0$). We can choose $\overline{A}$ can be arbitrarily large such that $d(A, B) = w(B)$.

¶25. Continuous Function. A box function $\square f$ is **continuous** for any sequence $(B_0, B_1, B_2, \ldots)$ of boxes that converges to a box B, we have $\square f(B_i) \to \square f(B)$. For real functions, being Lipschitz is equivalent to being continuous. This breaks down for box functions:

Lipschitz may not be continuous: suppose $\delta(x) = 1$ if $x > 0$ and $\delta(x) = 0$ otherwise. Let $\square f : \mathbb{IR} \to \mathbb{IR}$ be given by $\square f(B) = B + \delta(w(B))$. Then $w(\square f(B)) = w(B)$, and we see that $\square f$ has global Lipschitz constant 1. However, $\square f$ is not continuous at 0 since the sequence $B_n = [0, 1/n]$ converges to 0, but the sequence $\square f(B_n)$ converges to 1, which is not equal to $0 = \square f(0)$.

Continuous functions may not be Lipschitz: With $B = [0, 1]$, let $\square f : \mathbb{I}B \to \mathbb{I}B$ be the function $\square f(I) = [0, \sqrt{w(I)}]$. Then $\square f$ is continuous but not Lipschitz in $\mathbb{I}B$ because the sequence of quotients

$$w(\square f(I))/w(I) = \sqrt{w(I)}/w(I) = 1/\sqrt{w(I)}, \qquad w(I) \neq 0$$

is unbounded.

¶26. Centered Forms. We next seek box functions that has quadratic convergence, i.e., $r = 2$ in (49). This can be attained by general class of box functions called **centered forms**. The basic idea of centered forms is to use a Taylor expansion about the midpoint $m(I)$ of the interval I argument. Moore originally conjectured that the centered forms of f has quadratic convergence. This was first shown by Hansen and generalized by Nickel and Krawcyk. Below, we shall reproduce a simple proof by Stahl [35]. The topic of centered forms has been a key area of research in validated computing.

Let us begin with the simplest case, $n = 1$ and where f is a polynomial of degree d. Letting $c \in \mathbb{R}$ be given. Then we can do the Taylor expansion of f about the point c:

$$\begin{aligned} f(x) &= f(c) + f'(c)(x - c) + \frac{1}{2} f''(c)(x - c)^2 + \cdots + \frac{1}{d!} f^{(d)}(c)(x - c)^d \\ &= f(c) + \sum_{i=1}^{d} f^{[i]}(c)(x - c)^i \end{aligned} \tag{50}$$

where, for simplicity, we write $f^{[i]}(x)$ for $\frac{1}{i!}f^{(i)}(x)$, called the normalized ith derivative of f.

We define $\square f(I)$ to be the interval evaluation of the expression in (50), i.e.,

$$\square f(I) = f(c) + f'(c)(I-c) + \frac{1}{2}f''(c)(I-c)^2 + \cdots + \frac{1}{d!}f^{(d)}(c)(I-c)^d. \tag{51}$$

If I is given and we can freely choose c, then perhaps the most useful choice for c is $c = m(I)$. This ensures that the interval $I - c$ is a **centered interval**, i.e., has the form $[-a, a]$ for some $a \geq 0$. For simplicity, we shall write[10] $[\pm a]$ for $[-a, a]$ if $a \geq 0$, or $[a, -a]$ if $a < 0$.

The interval evaluation of (51) can advantage of such centered intervals in implementations. We have the following elementary properties of interval operations: let $a, b, c \in \mathbb{R}$.

(a) $c[\pm a] = -c[\pm a] = [\pm ca] = ca[\pm 1]$

(b) $[\pm a] + [\pm b] = [\pm(|a| + |b|)]$

(c) $[\pm a] - [\pm b] = [\pm(|a| + |b|)]$

(d) $[\pm a] \times [\pm b] = [\pm ab]$

From (a), we conclude that all centered intervals can be expressed as a scalar multiple of the canonical centered interval, $[\pm 1]$. Basically, such centered intervals are parametrized by one real number, and we need only one arithmetic operation to carry out any interval arithmetic operation. As consequence of these elementary rules, we see that the sub-distributive law is really an identity:

$$[\pm c]([\pm a] + [\pm b]) = [\pm c][\pm a] - [\pm b][\pm b][\pm c][\pm a] + [\pm b][\pm b]$$

since both sides are equal to

$$c(|a| + |b|)[\pm 1].$$

Consider shifts of centered forms, i.e., intervals represented as $I = c \pm [\pm b]$. Of course, this can represent all intervals, but we are interested in basic properties of I in terms of c, b. In particular, we have $m(I) = c$. We will shortly consider division by I, and we need the following property. Clearly,

$$0 \in I \Leftrightarrow |c| \leq |b|.$$

Next, assume $|c| > |b|$. Hence $0 \notin I$ and the expression $[\pm a]/(c + [\pm b])$ is well-defined. Then we have

$$\frac{[\pm a]}{c + [\pm b]} = \frac{[\pm a]}{[c - |b|, c + |b|]} = \frac{[\pm a]}{|c| - |b|}. \tag{52}$$

It follows that

$$w\left(\frac{[\pm a]}{c + [\pm b]}\right) = \frac{2|a|}{|c| - |b|}. \tag{53}$$

We are now ready to evaluate a polynomial expression $f(x) = \sum_{i=0}^{d} c_i x^i$ at a centered interval $[\pm a]$ where $a > 0$. We have

$$\sum_{i=0}^{d} c_i[\pm a]^i = c_0 + \left(\sum_{i=1}^{d} |c_i|a^i\right)[\pm 1] \tag{54}$$

$$= c_0 + [\pm b] \tag{55}$$

where $b = \sum_{i=1}^{d} |c_i|a^i$. Hence the basic algorithm for $\square f([\pm a])$ goes as follows:
1. Compute all the normalized Taylor coefficients at $c = m(I)$, i.e., $f^{[i]}(c)$ for $i = 0, \ldots, d$.
2. Compute $b = \sum_{i=1}^{d} |f^{[i]}(c)|a^i$ (note the i in the summation begins with $i = 1$).
3. Return $|f(c)| \pm [\pm b]$.

[10] This notation is consistent with our convention for errors where we write "$x \pm \varepsilon$" to denote a number of the form $x + \theta\varepsilon$ for some $\theta \in [-1, 1]$. Then $[x \pm \varepsilon]$ can denote the interval $\{y : x - \varepsilon \leq y \leq x + \varepsilon\}$. Centered intervals $[\pm\varepsilon]$ corresponds to the case $x = 0$.

Moreover, it is clear that

$$m(\square f(I)) = f(c)$$

and

$$w(\square f(x)) = b = \sum_{i=1}^{d} |f^{[i]}(c)|a^i.$$

Note that we can easily check if $0 \in \square f(I)$ since this amounts to $|f(c)| \leq b$.

¶27. Quadratic Convergence of Centered Forms. The expression (50) for the function $f : \mathbb{R} \to \mathbb{R}$ can be rewritten in the form

$$f(x) = f(c) + D(x, c)(x - c)$$

for some $D : \mathbb{R}^2 \to \mathbb{R}$. In the polynomial case (easily extended to analytic functions) we have

$$D(x, c) = \sum_{i=1}^{d} f^{[i]}(c)(x - c)^{i-1}.$$

This can be generalized to the multivariate setting: if $f : \mathbb{R}^n \to \mathbb{R}$ and $D : \mathbb{R}^{2n} \to \mathbb{R}$ satisfy

$$f(\mathbf{x}) = f(\mathbf{c}) + D(\mathbf{x}, \mathbf{c}) \circ (\mathbf{x} - \mathbf{c}) \tag{56}$$

for all $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ and $\circ$ denotes scalar product, then we call $D = D_f$ a **slope function** for f.

A function

$$z : \mathbb{I}^n\mathbb{R} \to \mathbb{R}^n \tag{57}$$

such that $z(B) \subseteq B$ is called a **centering function**. For instance, we can define $z(B)$ to be the midpoint $m(B)$. An Lipschitz function of the form

$$\square D : \mathbb{I}^n\mathbb{R} \to \mathbb{I}^n\mathbb{R} \tag{58}$$

is a **slope extension** of D relative to z if for all n-box B,

$$D(B, z(B)) \subseteq \mathbb{I}D(B).$$

Then we call the interval function defined by

$$\square f(B) := f(z(B)) + \square D(B) \circ (B - z(B)) \tag{59}$$

a **centered form** function for f. Note that

$$f(B) \subseteq f(z(B)) + D(B, \mathbf{c})(B - z(B)) \subseteq f(z(B)) + \square D(B) \circ (D - z(B)),$$

i.e., $\square f$ is an inclusion function for f. Moreover, since each of the functions in this expressions are box function, $\square f$ is a box function.

We now present a simple proof of Stahl [35, Theorem 1.3.37] that centered form functions has quadratic convergence. It is useful to introduce the **signed mignitude function**: for interval I, let

$$smig(I) := \begin{cases} \overline{I} & \text{if } \overline{I} < 0 \\ 0 & \text{if } 0 \in I \\ \underline{I} & \text{if } \underline{I} > 0. \end{cases}$$

For instance, $smig([2, 3]) = 2$, $smig([-2, 2]) = 0$, $smig([-3, -2]) = -2$. We extend this to boxes $B = I_1 \times \cdots \times I_n$,

$$smig(B) = (smig(I_1), \ldots, smig(I_n)).$$

We have the following inclusion

$$B \subseteq z(B) + [-1, 1]w(B). \tag{60}$$

It is enough to prove this for an interval: if $z \in I$ then $I \subseteq z + [-1, 1]w(I)$.

In (60), we can replace $z(B)$ by $smig(B)$ and thus obtain a "big" interval that contains B. In contrast, the next lemma shows that we can obtain an "small" interval using $smig$ that is contained in $f(B)$. Both these tricks will be used in the quadratic convergence proof.

Lemma 9. *For all n-boxes B,*

$$f(z(B)) + smig(\square D(B)) \circ (B - z(B)) \subseteq f(B).$$

We leave the verification of this lemma to an exercise.

Theorem 10 (Quadratic Convergence). *The centered form $\square f$ of f convergences quadratically: for all $A \subseteq \mathbb{I}^n\mathbb{R}$, there is a constant K_A such that if $B \in \mathbb{I}^n\mathbb{R}$, $B \subseteq A$, then*

$$d(\square f(B)) \leq K_A d(B)^2.$$

Proof. Let $B = I_1 \times \cdots \times I_n$ be an n-box contained in some $A \in \mathbb{I}^n\mathbb{R}$, and let $\mathbf{c} = (c_1, \ldots, c_n) = z(B)$. Also, let $\square D(B) = (\square D_1(B), \ldots, \square D_n(B))$. Recall that $\square D$ is Lipschitz (see (58)) so there exists a constant $k_A > 0$ such that $d(\square D_i(B)) \leq k_A d(B)$. We make the argument by a sequence of elementary arguments:

$$\begin{array}{rcll}
\square f(B) & = & f(\mathbf{c}) + \sum_{i=1}^n \square D_i(B)(I_i - c_i) & (\text{by Definition of } \square f) \\
& \subseteq & f(\mathbf{c}) + \sum_{i=1}^n \{smig(\square D_i(B)) + [-1,1]w(\square D_i(B))\}\,(I_i - c_i) & (\text{by (60)}) \\
& \subseteq & f(\mathbf{c}) + \sum_{i=1}^n \{smig(\square D_i(B))(I_i - c_i)\} + \{[-1,1]w(\square D_i(B))(I_i - c_i)\} & (\text{by subdistributivity}) \\
& \subseteq & f(B) + \sum_{i=1}^n \{[-1,1]w(\square D_i(B))\}\,(I_i - c_i) & (\text{by Lemma 9}) \\
& = & f(B) + \sum_{i=1}^n [-1,1]w(\square D_i(B))w(I_i) & \\
& \subseteq & f(B) + [-1,1]d(B)\sum_{i=1}^n w(\square D_i(B)) & (\text{since } w(I_i) \leq d(B)) \\
& \subseteq & f(B) + [-1,1]d(B)\sum_{i=1}^n k_A d(B) & (\text{since } \square D \text{ is Lipschitz}) \\
& = & f(B) + [-1,1]d(B)^2(nk_A) & .
\end{array}$$

Thus $w(\square f(B)) \leq K_A d(B)^2$ where $K_A = nk_A$. **Q.E.D.**

¶28. Box Rational Functions. We extend the previous development to rational functions. We begin with the identity when $f(x) = p(x)/q(x)$:

$$f(x) - f(c) = \frac{p(x) - p(c) - f(c)(q(x) - q(c))}{q(x)}. \tag{61}$$

Then by Taylor's expansion,

$$\begin{array}{rcl}
f(x) - f(c) & = & \dfrac{\sum_{i\geq 1} p^{[i]}(c)(x-c)^i - f(c)\sum_{i\geq 1} q^{[i]}(c)(x-c)^i}{\sum_{i\geq 0} q^{[i]}(c)(x-c)^i} \\
& = & \dfrac{\sum_{i\geq 1}(p^{[i]}(c) - q^{[i]}(c))(x-c)^i}{\sum_{i\geq 0} q^{[i]}(c)(x-c)^i} \\
& = & \dfrac{\sum_{i\geq 1} t_i(x-c)^i}{\sum_{i\geq 0} q^{[i]}(x-c)^i}
\end{array}$$

where

$$t_i := p^{[i]}(c) - q^{[i]}(c) \tag{62}$$

This last expression yields our **standard center form** for rational functions:

$$\square f(I) := f(c) + \frac{\sum_{i\geq 1} t_i(I-c)^i}{\sum_{i\geq 0} q^{[i]}(I-c)^i}. \tag{63}$$

Moreover, if $c = m(I)$, then $I - c$ is a centered interval and thus $\square f(I)$ can be easily evaluated similar to the polynomial case.

We can further generalize the preceding development in two ways. One direction of generalization is to consider multivariate rational functions. A somewhat less obvious direction to consider the "higher order centered forms": for any $k = 1, \ldots, n$, we may generalize (61) and the subsequent Taylor expansion to obtain:

$$f(x) - \sum_{i=0}^{k-1} f^{[i]}(c)(x-c)^i = \frac{\sum_{i\geq k} t_{k,i}(x-c)^i}{\sum_{i\geq 0} q^{[i]}(x-c)^i} \tag{64}$$

where

$$t_{k,i} = p^{[i]}(c) - \sum_{j=0}^{k-1} \binom{i}{j} f^{[j]}(c) q^{[i-j]}(c). \tag{65}$$

Note that (62) is just the case $k = 1$.

If we replace x by I in the expression (64), we obtain the **kth order centered form** $\square_k f(I)$. Thus (63) corresponds to $k = 1$. As shown in [32, Section 2.4], the higher order centered forms are at least as good than lower order ones in the sense that

$$\square_{k+1} f(I) \subseteq \square_k f(I).$$

For a polynomial f, this inclusion is always an identity: $\square_k f(I) = \square_1 f(I)$. But for non-polynomial f, the inclusion is strict for general I.

¶29. Krawczyk's Centered Form. The number of arithmetic operations to compute the above centered forms for $f = p/q$ is $\Theta(n^2)$, where $n = \deg(p) + \deg(q)$. Krwaczyk (1983) described another centered form which uses only $\Theta(n)$ arithmetic operations.

Let $f = f(X_1, \ldots, X_m)$ be a rational function and $B \in \mathbb{I}^m\mathbb{R}$ is contained in the domain of f. We call $G \in \mathbb{IR}$ an **interval slope** of f in B if

$$f(x) - f(c) \subseteq G \cdot (x - c), \qquad \text{for all } x \in B$$

where $c = m(B)$.

We provide a method to compute G from any straightline program S for f. Such a straightline program is a finite sequence of steps. The ith step ($i = 1, 2, \ldots m$) introduces a brand new variable u_i. Each step is an assignment statement, of one of the following type:

1. $u_i \leftarrow X_j$ $(j = 1, \ldots, m)$
2. $u_i \leftarrow c$ $(c \in \mathbb{R})$
3. $u_i \leftarrow u_j \circ u_k$ $(j < i, k < i$ and $\circ \in \{\pm, \times, \div\})$

So each u_i represents a rational function in $X_1, \ldots, X_n$. We say S computes the rational function represented by u_m, the last variable. We convert S to S' as follows:

- $G_i \leftarrow 1$ if the ith step is $u_i \leftarrow X_j$
- $G_i \leftarrow 0$ if the ith step is $u_i \leftarrow c$
- $G_i \leftarrow G_j \pm G_k$ if the ith step is $u_i \leftarrow u_j \pm u_k$
- $G_i \leftarrow$ ... if the ith step is $u_i \leftarrow u_j \times u_k$
- $G_i \leftarrow$ ... if the ith step is $u_i \leftarrow u_j / u_k$

We leave it as an exercise to show that G_m is an interval slope for f.

Exercises

Exercise 7.1: Show that the following operations are box functions:
(a) Arithmetic functions
(b) The composition of two box functions.
(c) Any centering function $z : \mathbb{I}^n\mathbb{R} \to \mathbb{R}^n$ where $z(B) \subseteq B$. ♢

Exercise 7.2: Consider interval arithmetic in the midpoint representation. Describe algorithms to perform the 4 arithmetic operations, for each of the following forms of interval arithmetic.
(a) Assume that an interval is represented by a pair (x, u) of real numbers, with $u \geq 0$, representing the interval $[x - u, x + u]$.
(b) Assume the interval is represented by triple (m, e, u) of integers, with $u \geq 0$, representing the

interval $[(m-u)2^e, (m+u)2^e]$.
(c) Assume the same interval representation as in (c), but suppose u is required to be less than U (e.g., $U = 2^3 2$). You must now describe a normalization algorithm – how to convert (m, e, u) to a normalized form where $u < U$. Discuss the tradeoffs for this restricted representations. ♢

Exercise 7.3: Prove the assertions in Example 1. ♢

End Exercises

§8. Additional Notes

Until the 1980's, floating point arithmetic are often implemented in software. Hardware implementation of floating point arithmetic requires an additional piece of hardware ("co-processor"), considered an add-on option. Today, floating point processing is so well-established that this co-procesor is accepted as standard computer hardware. The fact that the floating point numbers is now the dominant machine number representation is, in retrospect, somewhat surprising. First note some negative properties of FP computation, as compared to fixed point computation:

(1) Algorithms for FP arithmetic are much more complicated. This fact is obvious at the hardware level: an examination of the physical sizes of computer chips for FP arithmetic units and for integer arithmetic units will show the vast gap in their relative complexity.

(2) Error analysis for FP computation is much harder to understand. One reason is that spacing between consecutive representable numbers is non-uniform, in contrast to fixed-point numbers. In addition, around 0, there is further non-uniformity because 0 is a singularity for relative error representation.

Item (1) is an issue for hardware designers. This led the early computer designers (including von Neumann) to reject it as too complicated. In contrast, fixed point arithmetic is relatively straightforward. Item (2) contributes to the many (well-known and otherwise) pitfalls in FP computation. There are many anecdotes, examples and lists of numerical pitfalls (sometimes called "abuses") collected from the early days of numerical computing. Most of these issues are still relevant today. (e.g., [38, 34]).

Despite all this, FP computation has become the *de facto* standard for computing in scientific and engineering applications. Wilkinson, especially through his extensive error analysis of floating point computations, is credited with making floating point arithmetic better understood and accepted in main stream numerical computing. First, let us note that criterion (1) is no longer an critical issue because the algorithms and circuit design methodology for FP arithmetic are well-understood and relatively stable. Also, minimizing hardware size is usually not the bottleneck in today's world of very large scale integrated (VLSI) circuits. But what are some advantages of FP computation? The first advantage is range: for a given budget of bits, the range of numbers which can be approximated by floating point numbers is much larger than, say using fixed point representation. This was critical in allowing scientific computations in many domains to proceed (in the days of slower and clumsier hardware, this spell the difference between being able to complete one computation or not at all). In some domains, this advantage is now less importance with advancing computer speed and increasing hardware complexity. The second advantage is speed: the comparison here is between floating point arithmetic with rational arithmetic. Both floating point numbers and rational numbers are dense in the reals, and are thus useful for approximating real computation. The speed of floating point arithmetic is reduced to integer arithmetic plus some small overhead (and integer arithmetic is considered to be fast). In contrast to rational arithmetic is considerably slower than integer arithmetic, and easily suffer from rapid growth in bit lengths. This phenomenon was illustrated at the end of ¶11.

§9. APPENDIX: Concepts from Numerical Analysis

¶30. Norms. We assume vectors in $x \in \mathbb{C}^n$ (or $\mathbb{R}^n$). In approximations and error analysis we need to have some notion of size or magnitude of vectors. This is captured by the concept of a norm. A **norm** on $\mathbb{C}^n$ is a function $N : \mathbb{C}^n \to \mathbb{R}$ such that for all $x, y \in \mathbb{C}^n$,

- $N(x) \geq 0$, with equality iff $x = 0$.
- $N(\alpha x) = |\alpha| N(x)$ for all $\alpha \in \mathbb{C}$.
- $N(x + y) \leq N(x) + N(y)$

The main example of norms are the **p-norms** for any positive integer p. This is denoted $N(x) = \|x\|_p$ and defined as

$$\|x\|_p := \sqrt[p]{|x_1|^p + \cdots |x_n|^p}$$

where $x = (x_1, \ldots, x_n)^T$. The special cases of $p = 1$ and $p = 2$ are noteworthy: $\|x\|_1 = \sum_{i=1}^n |x_i|$, $\|x\|_2 = \sqrt{\sum_{i=1}^n |x_i|^2}$. The 2-norm is differentiable and invariant under an orthogonal transformation of the space. We can also generalize this to $p = \infty$ and define

$$\|x\|_\infty := \max\{|x_1|, \ldots, |x_n|\}.$$

A fundamental fact is that any two norms N and N' are equivalent in the following sense: there exists positive constants $c < C$ such that for all $x \in \mathbb{C}^n$,

$$c \cdot N(x) \leq N'(x) \leq C \cdot N(x).$$

To show this, it suffices to show that any norm N is equivalent to the ∞-norm. Write $x = \sum_{i=1}^n x_i e^i$ where e^i is the ith elementary vector. Then

$$N(x) \leq \sum_{i=1}^n |x_i| N(e^i) \leq \|x\|_\infty \sum_{i=1}^n N(e^i)$$

so it is sufficient to choose $C = \sum_{i=1}^n N(e^i)$. Now, consider the unit sphere under the ∞-norm, $S = \{x \in \mathbb{C}^n : \|x\|_\infty = 1\}$. Since S is compact, the norm function $N : S \to \mathbb{R}$ achieves its mininum value at some $x^0 \in S$. Let $N(x^0) = c$. If $\|x\|_\infty = b$ then we have

$$N(x) = N(b \cdot \frac{x}{b}) = bN(\frac{x}{b}) \geq b \cdot c = c\|x\|_\infty.$$

This completes the proof.

A vector space X with a norm $\|\cdot\| : X \to \mathbb{R}$ is called a **normed space**. An infinite sequence $(x_1, x_2, \ldots)$ in X is **Cauchy** if for all $\varepsilon > 0$ there is an $n = n(\varepsilon)$ such that for all $i > n$, $\|x_i - x_{i+1}\| < \varepsilon$. A normed space X is complete if every Cauchy sequence $x_1, x_2, \ldots$ has a limit $x^* \in X$, i.e., for all $\varepsilon > 0$, there is an n such that for all $i > n$, $\|x_i - x^*\| < \varepsilon$. A complete normed space is also called a **Banach space**.

¶31. Componentwise Error Analysis. A major use of norms is in error analysis. For instance, we can quantify the difference between a computed matrix $\widetilde{A}$ and the corresponding exact matrix A by using norms: the size of their difference can be given by $\|\widetilde{A} - A\|$. We call this **normwise error**.

However, we can also measure this error in a componentwise manner. For this, it is useful to introduce a notation: if $A = [a_{ij}]_{i,j}$ then $|A| = [|a_{ij}|]_{i,j}$ is obtained by replacing each entry of A by its absolute value. Moreover, if A, E are two real matrices with the same dimensions, we can write $A \leq E$ to mean each component of A is at most the corresponding component of E. We call E is a **componentwise bound** on the error of $\widetilde{A}$ if $|\widetilde{A} - A| \leq E$. For instance, E can be the matrix whose entries are all equal to some value $u \geq 0$. In general, componentwise bounds is a refined tool for bounding errors of individual entries of $\widetilde{A}$.

¶32. Distance to the closest singularity. The numerical stabilty of a numerical problem is directly influenced by its distance to the nearest singularity. We show the following result from Turing (and Banach in the 1920s). It was first shown by Gastinel for arbitrary norms in 1966 [17]. For a non-singular square matrix A, let

$$\delta_T(A) := \inf_S \left\{ \frac{\|S - A\|}{\|A\|} \right\}$$

where S ranges all singular matrices. Thus $\delta_T(A)$ is the relative distance from A to the nearest singular matrix S. The subscript T refers to Turing.

Theorem 11 (Turing).

$$\delta_T(A) = \frac{1}{\|A^{-1}\|\|A\|}.$$

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983. Translated from German by Jon Rokne.

[2] R. L. Ashenhurst and N. Metropolis. Error estimates in computer calculation. *Amer. Math. Monthly*, 72(2):47–58, 1965.

[3] D. H. Bailey. Multiprecision translation and execution of Fortran programs. *ACM Trans. on Math. Software*, 19(3):288–319, 1993.

[4] M. Benouamer, D. Michelucci, and B. Péroche. Boundary evaluation using a lazy rational arithmetic. In *Proceedings of the 2nd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 115–126, Montréal, Canada, 1993. ACM Press.

[5] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. on Math. Software*, 4:57–70, 1978.

[6] D. G. Cantor, P. H. Galyean, and H. G. Zimmer. A continued fraction algorithm for real algebraic numbers. *Math. of Computation*, 26(119):785–791, 1972.

[7] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[8] K. Dittenberger. *Hensel Codes: An Efficient Method for Exact Numerical Computation*. PhD thesis, Johannes Kepler Universitaet, Linz, 1985. Diplomarbeit, Institute for Mathematics.

[9] S. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. Ph.D. thesis, New York University, 1999.

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

[11] P. Gowland and D. Lester. A survey of exact arithmetic implementations. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 30–47. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[12] R. T. Gregory and E. V. Krishnamurthy. *Methods and Applications of Error-Free Computation*. Springer-Verlag, New York, 1984.

[13] E. C. R. Hehner and R. N. S. Horspool. A new representation of the rational numbers for fast easy arithmetic. *SIAM J. Computing*, 8(2):124–134, 1979.

[14] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[15] T. Hull, M. Cohen, J. Sawchuk, and D. Wortman. Exception handling in scientific computing. *ACM Trans. on Math. Software*, 14(3):201–217, 1988.

[16] IEEE. ANSI/IEEE Standard 754-1985 for binary floating-point arithmetic, 1985. The Institute of Electrical and Electronic Engineers, Inc., New York.

[17] W. Kahan. Numerical linear algebra. *Canadian Math. Bull.*, 9:757–801, 1966.

[18] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Trans. on Graphics*, 10:71–91, 1991.

[19] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Boston, 2nd edition edition, 1981.

[20] N. Koblitz. *P-adic numbers, p-adic analysis, and zeta-functions*. Springer-Verlag, New York, 1977.

[21] S. Krishnan, M. Foskey, T. Culver, J. Keyser, and D. Manocha. PRECISE: Efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation. In *17th ACM Symp. on Comp. Geometry*, pages 274–283, 2001.

[22] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Trans. Computers*, 47(11):1235–1243, 1998.

[23] D. W. Matula and P. Kornerup. Foundations of finite precision rational arithmetic. *Computing*, Suppl.2:85–111, 1980.

[24] N. Metropolis. Methods of significance arithmetic. In D. A. H. Jacobs, editor, *The State of the Art in Numerical Analysis*, pages 179–192. Academic Press, London, 1977.

[25] D. Michelucci and J.-M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9):961–975, 1997.

[26] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[27] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[28] N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[29] K. Ouchi. Real/Expr: Implementation of an Exact Computation Package. Master's thesis, New York University, Department of Computer Science, Courant Institute, Jan. 1997. From `http://cs.nyu.edu/exact/doc/`.

[30] M. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, 2000. To appear.

[31] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1994.

[32] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions*. Horwood Publishing Limited, Chichester, West Sussex, UK, 1984.

[33] S. M. Rump. Fast and parallel interval arithmetic, 200X.

[34] K. Schittkowski. *Numerical Data Fitting in Dynamical Systems – A Practical Introduction with Applications and Software*. Kluwer Academic Publishers, 2002.

[35] V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. Ph.D. thesis, Johannes Kepler University, Linz, 1995.

[36] L. N. Trefethen. Computing with functions instead of numbers. *Mathematics in Computer Science*, 1(1):9–19, 2007. Inaugural issue on Complexity of Continous Computation. Based on talk presented at the Brent's 60th Birthday Symposium, Weierstrass Institute, Berlin 2006.

[37] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[38] von Siegfried M. Rump. How reliable are results of computers? *Jahrbuch Überblicke Mathematik*, pages 163–168, 1983. Trans. from German *Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?*

[39] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[40] C. K. Yap. On guaranteed accuracy computation. In Chen and Wang [39], chapter 12, pages 322–373.

[41] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.

[42] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.

[43] C. K. Yap and J. Yu. Foundations of exact rounding. In *Proc. WALCOM 2009*, volume 5431 of *Lecture Notes in Computer Science*, 2009. To appear. Invited talk, 3rd Workshop on Algorithms and Computation, Kolkata, India.

Lecture 3
GEOMETRIC COMPUTATION, I

Among numerical software, those that might be described as "geometric" are notoriously non-robust. Understanding geometric computation is key to solving the non-robustness problem. In this chapter, we introduce the reader to some simple geometric problems related to the convex hull. Various forms of non-robustness can be illustrated using such examples.

§1. Convex Hull of Planar Points

The prototype geometric objects are points and lines in the plane. We begin with one of the simplest problems in geometric computing: computing the convex hull of a set of points in the plane. The one-dimensional analogue of convex hull is the problem of computing the maxima and minima of a set of numbers. Like sorting (which also has a geometric analogue in higher dimensions), our convex hull problem has been extensively studied and there are many known algorithms known for the problem. This chapter will introduce several such algorithms, to investigate their non-robustness properties. We first develop a convex hull algorithm in a leisurely fashion.

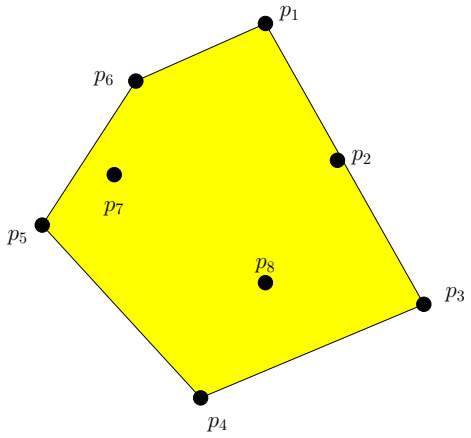


Figure 1: Convex hull of a set of points

The computational problem is this: given a set S of points in the plane, compute its convex hull. Figure 1 displays the convex hull of the set $S = \{p_1, p_2, \ldots, p_8\}$ of points. The output is essentially a convex polygon determined by the five points p_1, p_6, p_5, p_4, p_3. These points are said to be **extreme** in S (see the Appendix of this Chapter for some the basic definitions).

The output needs to be specified with some care: we not only want to determine the extreme points in S, but we also want to list them in some specific circular order, either clockwise or counter clockwise order. For specificity, choose the clockwise order. Thus, for the input set in Figure 1, we want to list the output set $\{p_1, p_6, p_5, p_4, p_3\}$ in a specific order $(p_1, p_3, p_4, p_5, p_6)$ or any cyclic equivalent, such as $(p_4, p_5, p_6, p_1, p_3)$.

A possible ambiguity arises. Are points on the relative interior of a convex hull edge considered "extreme"? In Figure 1, the point p_2 just inside the convex hull. But suppose p_2 lies on the edge $[p_1, p_3]$: should it be included in our convex hull output? According to our definition in the Appendix, points in the relative interior of the edge $[p_1, p_3]$ are not considered extreme. It is also possible to define our convex hull problem so that such points are required to appear in our circular list.

¶1. Checking Extremity of a Point. One approach to computing a convex hull of S is to reduce it to checking if a point $p \in S$ is extremal in S. We eventually show that if we retain some extra information in extremity checking, we can link them into a circular list corresponding to a convex hull.

So how do we check if a point p is extreme in S? First we make a simple observation: it is easy to find at least one extremal points: if $p_{\min} \in S$ is the lexicographically smallest point in S, then p is extremal. If S has at least two points, then the lexicographically largest point $p_{\max} \in S$ is another extremal point.

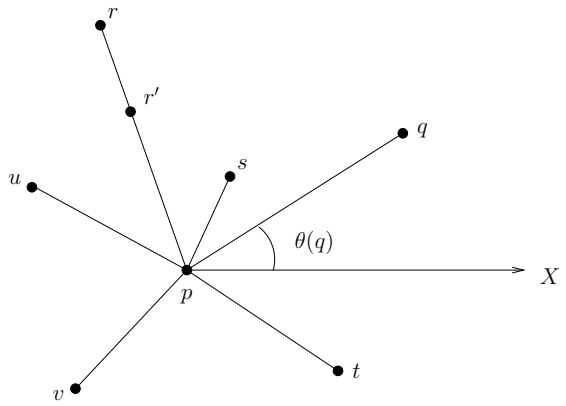


Figure 2: Circular sorting S about p.

In general, we can easily reduce the question of extremity of p in S to another problem: circular sorting of $S \setminus \{p\}$ about p. This is the problem where, for each point $q \in S \setminus \{p\}$, define $\theta(q) = \angle(Xpq)$ to be the angle swept as the ray $\overrightarrow{pX}$ turn counter-clockwise to reach the ray $\overrightarrow{pq}$. Here, X is any point to the right of p (see Figure 2). The problem of **circular sorting** of S about the point p amounts to sorting the set

$$\{\theta(q) : q \in S \setminus \{p\}\}.$$

For instance, sorting the set S in Figure 2 gives the output (q, s, r, r', u, v, t). In case of ties for two points (e.g., r, r'), their relative order is not important for our purposes (e.g., we could also output (q, s, r', r, u, v, t)).

Let $(p_1, \ldots, p_{n-1})$ be output of circular sorting $S \setminus \{p\}$ about p. Then p is extreme iff there is some $1 \leq i \leq n$ such that

$$\theta(p_i) - \theta(p_{i-1}) > \pi. \tag{1}$$

When $i = 1$, we must interpret $\theta(p_0)$ in (1) as $\theta(p_n) - 2\pi < 0$.

Circular sorting has interest in its own right (see Graham scan below). But for the purpose of deciding whether p is extreme, circular sorting is an overkill. Let us develop a simpler method.

Here is one approach, illustrated in Figure 3. Let $p = (p_x, p_y)$ and $S' = S \setminus \{p\}$. We split S' into two sets $S' = S_a \uplus S_b$ where S_a comprise those points $q \in S'$ that lies *above* the X-axis: either $q_y > p_y$ or ($q_y = p_y$ and $q_x > p_x$). Similarly, S_b comprise those points $q \in S'$ lying *below* the X-axis. Thus, a point v on the positive X-axis would belong to S_a while a point u on the negative X-axis would belong to S_b (see Figure 3). Define

$$\theta_a^- := \min\{\theta(q) : q \in S_a\}, \quad \theta_a^+ := \max\{\theta(q) : q \in S_a\}$$

When S_a is empty, $\theta_a^- = +\infty$ and $\theta_b^+ = -\infty$. Similarly let $\theta_b^- := \min\{\theta(q) : q \in S_b\}$ and $\theta_b^+ := \max\{\theta(q) : q \in S_b\}$. Note that $0 \leq \theta_a^- \leq \theta_b^+ < \pi$ if S_a is non-empty, and $\pi \leq \theta_b^- \leq \theta_b^+ < 2\pi$ if S_b is non-empty. The following is easy to see:

Lemma 1. *Point p is extreme in S iff one of the following three conditions hold:*
1. S_a or S_b is empty.
2. $\theta_b^+ - \theta_a^- < \pi$.
3. $\theta_b^- - \theta_a^+ > \pi$.

Based on this lemma, it is now easy to give an $O(n)$ time algorithm to check for extremity:

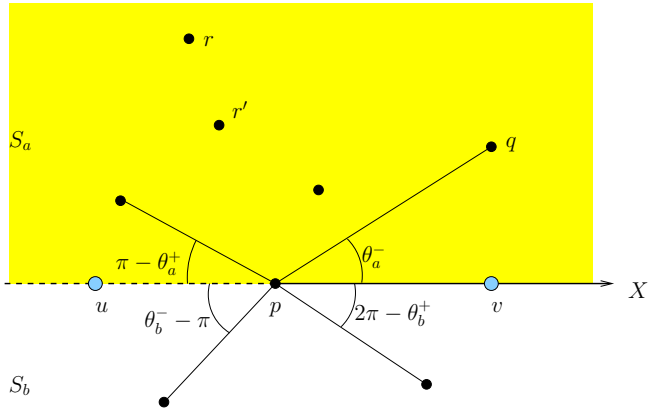


Figure 3: Checking extremity of point p.

Extremity Test (Conceptual)
Input: A set of points S and $p \in S$.
Output: True if p is extreme in S and false otherwise.
1. $\theta_a^- \leftarrow \theta_b^- \leftarrow +\infty$, $\theta_a^+ \leftarrow \theta_b^+ \leftarrow -\infty$. ◁ *Initialization*
2. for each $q \in S \setminus \{p\}$ ◁ *Main Loop*
3. if $(q.Y > p.Y)$ or $(q.Y = p.Y$ and $q.X > p.X)$
4. then $\theta_a^- \leftarrow \min\{\theta_a^-, \theta(q)\}$ and $\theta_a^+ \leftarrow \max\{\theta_a^+, \theta(q)\}$.
5. else $\theta_b^- \leftarrow \min\{\theta_b^-, \theta(q)\}$ and $\theta_b^+ \leftarrow \max\{\theta_b^+, \theta(q)\}$.
6. if $[(\theta_a^-$ or θ_b^- is infinite) or $(\theta_b^+ - \theta_a^- < \pi)$ or $(\theta_b^- - \theta_a^+ > \pi)]$
return True.
7. else
return False.

We leave the correctness proof to an Exercise. Note that this algorithm assumes the ability to compute the angles $\theta(q)$ for $q \in S$. As discussed next, this is not a practical assumption.

Exercises

Exercise 1.1:
(i) Prove the correctness of Lemma 1.
(ii) Prove the correctness of the (Conceptual) Extremity Test. ♢

Exercise 1.2: Develop another conceptual Extremity Test in which we assume that the two extreme points $p_{\min}$ and $p_{\max}$ have already been found, and we want to test another point $p \in S \setminus \{p_{\min}, p_{\max}\}$ for extremity. HINT: The idea is that having the extra information about $p_{\min}$ and $p_{\max}$ should simplify our task. To further simplify the problem, you may assume that p lies below the line through $p_{\min}$ and $p_{\max}$. ♢

Exercise 1.3: Generalize Lemma 1 to three dimensions. Give a corresponding algorithm to check extremity of a point in $S \subseteq \mathbb{R}^3$, in analogy to the previous exercise. ♢

End Exercises

§2. Some Geometric Predicates

We consider the above Extremity Test based on Lemma 1 to be a "conceptual" algorithm, not a practical one. This is because the algorithm requires the computation of angles $\theta(q)$, which are transcendental[1] quantities. Thus, the computed angles are necessarily approximate, and it is non-trivial to turn this into an exact algorithm. More generally, we must be wary of the Real RAM model often assumed in theoretical algorithms. Real computational difficulties, including undecidable questions, might lurk hidden its abstraction. To avoid the computation of angles, we now introduce an important low-level tool in Computational Geometry: the signed area of an oriented triangle.

¶2. Signed Area. Let A, B, C be three points in the plane. The **signed area** of ABC is half of the following determinant

$$\Delta(A,B,C) := \det \begin{bmatrix} a & a' & 1 \\ b & b' & 1 \\ c & c' & 1 \end{bmatrix} \tag{2}$$

$$= \det \begin{bmatrix} a & a' & 1 \\ b-a & b'-a & 0 \\ c-a & c'-a & 0 \end{bmatrix} = \det \begin{bmatrix} b-a & b'-a' \\ c-a & c'-a' \end{bmatrix} \tag{3}$$

where $A = \mathsf{Point}(a, a'), B = \mathsf{Point}(b, b')$, etc.

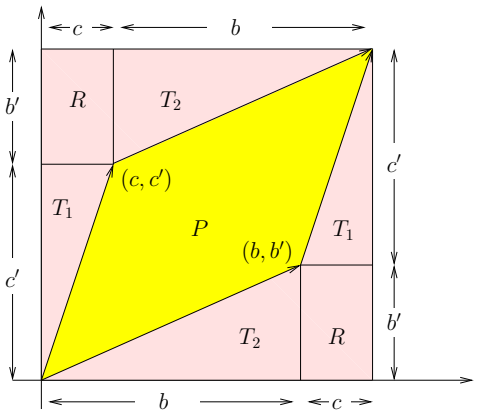


Figure 4: Area of parallelogram determined by vectors $(b, b'), (c, c')$.

To justify of this formula, we assume without loss of generality that $a = a' = 0$. First assume that both vectors (b, b') and (c, c') are in the first quadrant, with vector (b, b') making a larger angle with the positive x-axis than vector (c, c'). Referring to Figure 4, the area of the parallelogram P is equal to the area of the enclosing rectangle $(b + b')(c + c')$ minus the area of the two smaller rectangles $2R = 2bc'$, minus the area of the 4 triangles $2(T_1 + T_2) = bb' + cc'$. This yields

$$\Delta(A,B,C) = (b+b')(c+c') - (2bc' + bb' + cc') = b'c - bc'.$$

The above figure assumes that the angle $\theta(B) \leq \theta(C)$. In case $\theta(B) > \theta(C)$, we obtain the same area with an negative sign. We leave it as an exercise to verify this formula even when the 2 vectors are not both in the first quadrant.

This formula generalizes to volume in 3 and higher dimensions. For instance, the signed volume of the tetrahedron defined by four points $A, B, C, D \in \mathbb{R}^3$, given in this order, is

$$\frac{1}{6} \det \begin{bmatrix} A & 1 \\ B & 1 \\ C & 1 \\ D & 1 \end{bmatrix} \tag{4}$$

[1] A transcendental computation is one that involves non-algebraic numbers such as π. Assuming the input numbers are algebraic, then $\theta(q)$ is generally non-algebraic. Algebraic and transcendental numbers will be treated in detail later in the book.

where the row indicated by $(A, 1)$ is filled with the coordinates of A followed by an extra "1", etc. The factor of $1/6$ will be $1/d!$ in d-dimensions.

¶3. The Orientation Predicate. Using the signed area, we define several useful predicates. The orientation predicate is defined as

$$\mathsf{Orientation}(A, B, C) = \mathtt{sign}(\Delta(A, B, C)).$$

Thus the orientation predicate is a three-valued function, returning $-1, 0$ or $+1$. In general, a **predicate** is any function with a (small) finite number of possible outcomes. In case there are only two (resp., three) possible outcomes, we call it a **truth predicate** (resp., **geometric predicate**). Truth predicates are well-known and widely used in mathematical logic and computer science. Geometric predicates are perhaps less well-known, but they arise naturally as in the following examples:

- Relative to a triangle, a point can be **inside**, **outside**, or **on the boundary**.
- Relative to an oriented hyperplane, a point can be **left**, **right**, or **on the hyperplane**.
- Two lines in the plane can be **intersect** in a single point, be **coincident**, or **non-intersecting**.
- Two triangular areas can be **disjoint**, have a **containment** relationship, or their boundaries may **intersect**.

Geometric relations with more than 3 values are clearly also possible. E.g., the relation between two lines in 3-D space is 4-valued because they are either co-planar (in which case we can classify them into the 3 cases as above), or they can be skew. We also note that 3-valued predicates have a well-known non-geometric application in which the 3 values are interpreted as **true**, **false** and **unknown**. It is then viewed as a generalization of 2-valued logic.

What is the significance of the orientation predicate? One answer is immediate: when the output is 0, it means that A, B, C are collinear since the area of the triangle is zero exactly in this case. For instance if two of the points are coincident, then the output is 0. What about the non-zero outputs? Let us do a simple experiment: let $A = (0, 0), B = (1, 0)$ and $C = (0, 1)$. Then we see that

$$\Delta(A, B, C) = \det \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = 1.$$

Thus, if you look from the point $(0, 0)$ towards the point $(1, 0)$, then $(0, 1)$ is to the left. In general:

> CLAIM: *If A, B, C are not collinear, and you look from point A to point B, then point C lies to the left if and only if* $\mathsf{Orientation}(A, B, C) = +1$.

We argue as follows: fix $A = (0, 0)$ and $B = (1, 0)$ and let C vary continuously, starting from the position $(0, 1)$. As long as C does not enter the X-axis, the area $\Delta(A, B, C)$ remains non-zero. Since the area is a continuous function, we conclude that area will remain positive, i.e., $\mathsf{Orientation}(A, B, C) = +1$. By the same token, as long as C lies strictly below the X-axis, the area remains negative and so $\mathsf{Orientation}(A, B, C) = -1$. Moreover, the area vanishes iff C lies on the X-axis. This proves our CLAIM when A, B are the special points $(0, 0)$ and $(1, 0)$.

In general, if A, B, C is an arbitrary non-collinear triple, there is a unique a rigid transformation T that transforms (A, B) to $(T(A), T(B)) = ((0, 0), (1, 0))$: first translate A to $(0, 0)$, then rotate B to the positive X axis, and finally scale B to $(1, 0)$. A translation, rotation and scaling transformation is represented by a 3×3 matrix with positive determinant (Exercise). Thus T has positive determinant. Moreover, we see that

$$\Delta(T(A), T(B), T(C)) = \det(T)\Delta(A, B, C). \tag{5}$$

Since $\det(T) > 0$, the transformation is orientation preserving. Finally, we note that (A, B, C) is a left-turn (in the sense of the CLAIM) iff $T(C)$ is in the upper half plane. Hence our CLAIM holds generally.

¶4. Some Truth Predicates. The orientation predicate is a geometric (3-valued) predicate. We now define several useful truth (2-valued) predicates from this single geometric predicate.

In the CLAIM above, we have already implicitly defined the left-turn predicate:

$$\mathsf{LeftTurn}(A,B,C) \quad \equiv \quad \text{“}\mathsf{Orientation}(A,B,C) > 0\text{”}.$$

Similarly, the $\mathsf{RightTurn}(A,B,C)$ predicate gives the truth value of “$\mathsf{Orientation}(A,B,C) < 0$”.

Our goal is to reduce the Extremity Test to truth predicates such as $\mathsf{LeftTurn}$. Unfortunately, $\mathsf{LeftTurn}$ by itself is not enough in the presence of “degenerate” inputs. For our extremity problem, a set of points is degenerate if three input points are collinear. We need new predicates to take into account the relative disposition of three collinear points.

For instance, in Figure 2, when we discover that p, r, r' are collinear, we want to discard r' as non-extreme. So we define a helper predicate, called $\mathit{Between}(A,B,C)$, which evaluates to true iff A, B, C are collinear with B strictly between A and C. We define

$$\mathit{Between}(A,B,C) \quad \equiv \quad \text{“}\mathsf{Orientation}(A,B,C) = 0 \text{ and } \langle A-B, C-B\rangle < 0\text{”}$$

where $\langle u, v\rangle$ is just scalar product of two vectors. To see this, from elementary geometry, we see that $\langle A-B, C-B\rangle = \|A-B\| \cdot \|C-B\| \cos\theta$ provided $A \neq B$ and $B \neq C$ and θ is angle between the vectors $A-B, C-B$. Since $\langle A-B, C-B\rangle < 0$, we know that $A \neq B$, $B \neq C$ and $\cos\theta < 0$. Since A, B, C are collinear, $\theta = 0$ or $\theta = \pi$. We conclude that θ must in fact be π; this proves that B lies strictly between A and C.

Define the **modified Left Turn Predicate** as:

$$\mathit{LeftTurn}^*(A,B,C) \quad \equiv \quad \text{“}\mathsf{Orientation}(A,B,C) > 0 \text{ or } \mathit{Between}(C,A,B)\text{”}.$$

The modified Right Turn Predicate is similar,

$$\mathit{RightTurn}^*(A,B,C) \quad \equiv \quad \text{“}\mathsf{Orientation}(A,B,C) < 0 \text{ or } \mathit{Between}(C,A,B)\text{”}.$$

Note that in both calls to the *Between* predicate, the arguments are (C, A, B) and not (A, B, C).

¶5. Enhanced Extremity Test. We now return to our conceptual Extremity Test. Our plan is to modify it into a practical algorithm by using the various predicates which we just developed.

Before installing the modified turn predicates, we make another adjustment: instead of computing the angle θ_a^-, we only remember the point in S_a that determines this angle. This point is denoted p_a^-. Similarly, we replace the other 3 angles by the points p_a^+, p_b^- and p_b^+. Initially, these points are initialized to the Null value. Moreover, the $\mathit{LeftTurn}^*(p,q,r)$ predicate always return true if any of the arguments is Null.

We also enhance our output as follows: if the point p is extreme in S, we output a triple (r,p,q) such that p, q are also extreme in S and $\mathsf{LeftTurn}(r,p,q) =$ True. if the point p is non-extreme in S, we output 3 points $u, v, w \in S \setminus \{p\}$ such that p in contained in the triangle (u,v,w) and $\mathsf{LeftTurn}(u,v,w)$. This is captured by another truth predicate defined as follows:

$$\mathit{Inside}(p,u,v,w) \quad \equiv \quad \mathit{LeftTurn}^*(p,u,v) \wedge \mathit{LeftTurn}^*(p,v,w) \wedge \mathit{LeftTurn}^*(p,w,u). \tag{6}$$

The above considerations finally lead to the following algorithm for the extremity test. It is best to understand this algorithm as a refinement of the original Conceptual Extremity Test.

Extremity Test
Input: A set of points S and $p \in S$. Assume $|S| \geq 2$.
Output: If p is extreme in S, output (r, p, q) such that
r, q are extreme and $\mathsf{LeftTurn}(r, p, q) = \textsc{True}$.
Else, output the points $u, v, w \in S \setminus \{p\}$ such that
p lies inside triangle (u, v, w) and $\mathsf{LeftTurn}(r, p, q) = \textsc{True}$.

1. ▷ *INITIALIZATION:*
   $p_a^- \leftarrow p_b^- \leftarrow p_a^+ \leftarrow p_b^+ \leftarrow \textsc{Null}$.
2. ▷ *MAIN LOOP:*
   for each $q \in S \setminus \{p\}$
3. if $(q.Y > p.Y)$ or $(q.Y = p.Y$ and $q.X > p.X)$
4. then
   if *LeftTurn*$^*(p_a^-, p, q)$ then $p_a^- \leftarrow q$
   if *RightTurn*$^*(p_a^+, p, q)$ then $p_a^+ \leftarrow q$
5. else
   if *LeftTurn*$^*(p_b^-, p, q)$ then $p_b^- \leftarrow q$
   if *RightTurn*$^*(p_b^+, p, q)$ then $p_b^+ \leftarrow q$
6. ▷ *OUTPUT FOR EXTREME CASES:*
   if $(p_a^- = \textsc{Null})$ then return (p_b^+, p, p_b^-).
   if $(p_b^- = \textsc{Null})$ then return (p_a^+, p, p_a^-).
   if *LeftTurn*$^*(p_a^+, p, p_b^-)$ then return (p_a^+, p, p_b^-)
   if *LeftTurn*$^*(p_b^+, p, p_a^-)$ then return (p_b^+, p, p_a^-)
7. ▷ *OUTPUT FOR NON-EXTREME CASES:*
   if *Inside*(p, p_a^-, p_a^+, p_b^-) then return (p_a^-, p_a^+, p_b^-)
   else return (p_a^-, p_b^-, p_b^+)

Note that above algorithm uses only two kinds of predicates: the modified left- and right-turns.

Exercises

Exercise 2.1:
(i) Give the general formula for the 3×3 matrix T that transforms any pair (A, B) of distinct points to $((0, 0), (1, 0))$.
(ii) What is T when $(A, B) = ((-2, 3), (3, 2))$? ♢

Exercise 2.2: A fundamental property of the left turn predicate is that

$$\mathsf{LeftTurn}(A, B, C) \equiv \mathsf{LeftTurn}(B, C, A) \equiv \mathsf{LeftTurn}(C, A, B).$$

Kettner and Welzl [3] reported that for a randomly generated set of 1000 points in the unit box $[0, 1) \times [0, 1)$, about one third of the $\binom{1000}{3}$ triples (A, B, C) of points fails this identity when evaluated in IEEE single precision coordinates. No failures were observed in IEEE double precision coordinates. Please repeat their experiments. ♢

Exercise 2.3:
(i) Implement the Extremity Test algorithm.
(ii) Run your algorithm on the following input points taken from [2].

$$\begin{aligned}
p_1 &= (\ 7.30000\,00000\,00019\,4,\ 7.30000\,00000\,00016\,7),\\
p_2 &= (24.00000\,00000\,00068,\quad 24.00000\,00000\,00061\ ),\\
p_3 &= (24.00000\,00000\,0005,\quad 24.00000\,00000\,00053\ ),\\
p_4 &= (\ 0.50000\,00000\,00016\,21,\ 0.50000\,00000\,00012\,43),\\
p_5 &= (8, 4),\\
p_6 &= (4, 9),\\
p_7 &= (15, 27),
\end{aligned}$$

$$\begin{aligned} p_8 &= (26, 25), \\ p_9 &= (19, 11). \end{aligned}$$

♢

Exercise 2.4: Develop an Extremity Test for 3 dimensional points. Fix a set $S \subseteq \mathbb{R}^3$ and a point $p \in S$. To see if p is extremal, you want to find two other points q, r such that relative to the plane $H(p, q, r)$ spanned by $\{p, q, r\}$, all the remaining points of S lies strictly to one side of this plane. ♢

End Exercises

§3. Simplified Graham Scan

There are probably a dozen known convex hull algorithms. One of the earliest algorithms is called the "Graham Scan", from Ron Graham (1975). The idea of Graham scan was to first perform an angular sorting of the input set S about an arbitrary point $p^* \notin S$ (e.g., p^* may be origin). If $(p_1, \ldots, p_n)$ is this sorted list, we then incrementally compute the convex hull of the prefix $(p_1, \ldots, p_i)$ for $i = 2, \ldots, n$. There is a very simple stack-based algorithm to achieve this incremental sorting.

We now describe variant that is described by Kettner and Welzl [3] (it is a variant of what is known as "Andrew Sort"). The basic idea is choose let $(p_1, \ldots, p_n)$ be the lexicographical sorting of S. This can be interpreted as choosing p^* sufficiently far to the west and south of S for the Graham scan; see Exercise. Besides its simplicity, we will also use it to illustrate an algorithmic idea called "conservative predicates".

> *Reducibility Among Some Geometry Problems:* Convex hull in 1-D is computing the max-min of a set of points. Convex hull in 2-D is rather like sorting since a 2-D convex hull is almost like a linearly sorted list. Indeed, this section shows how convex hull in 2-D can be reduced to sorting in 1-D. Similarly, convex hull in 3-D can be reduced to sorting in 2-D (there is a notion of "sorting points" in k-D). Another such reduction is that Voronoi diagrams (which will be introduced later) in k-dimensions and be reduced to convex hull in $(k+1)$-dimensions.

It is a generally useful device to split the 2-D convex hull problem into two subproblems: computing an upper hull and a lower hull. Recall that

$$p_{\min}, p_{\max} \in S$$

are two extremal points of the convex hull of S (assuming S has at least two points). Hence, we may relabel the points on the convex hull so that

$$(p_{\min}, p_1, \ldots, p_{k-1}, p_k, p_{\max}, q_\ell, q_{\ell-1}, \ldots, q_1)$$

is the counter-clockwise listing of the convex hull of S. Here, $k \geq 0$ and $\ell \geq 0$. Then we may call the two subsequences

$$(p_{\min}, p_1, \ldots, p_{k-1}, p_k, p_{\max}), \qquad (p_{\min}, q_1, \ldots, q_{\ell-1}, q_\ell p_{\max})$$

the **lower** and **upper hulls** of S. See Figure 5 for an illustration.

By symmetry, it is sufficient to show how to compute the lower hull of S. Suppose we begin by computing the lexicographic ordering of all the points in S, discarding any duplicates we find. Let this lexicographic ordering be

$$L = (p_{\min}, p_1, p_2, \ldots, p_m, p_{\max}) \tag{7}$$

where $p_{i-1} <_{\texttt{LEX}} p_i$ for all $i = 1, \ldots, m+1$, with $p_0 = p_{\min}, p_{m+1} = p_{\max}$.

Clearly, the lower hull of S is a subsequence of L. In general, any lexicographically sorted sequence of the form (7) that contains the lower hull as a subsequence, with the first and last elements $p_{\min}$ and $p_{\max}$, is called a **liberal hull**. Temporarily, we call a point "good" if it belongs to the lower hull, and "bad" otherwise.

The idea of Kettner and Welzl is to begin with a liberal hull, and successively eliminate bad points. A simple observation is this:

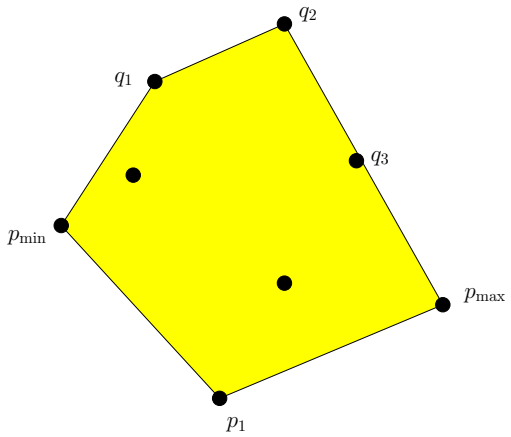


Figure 5: Lower $(p_{\min}, p_1, p_{\max})$ and upper $(p_{\min}, q_1, q_2, q_3, p_{\max})$ hulls

Observation 1. *Let (p, q, r) be 3 points in a liberal hull with $p <_{\text{LEX}} q <_{\text{LEX}} r$.*
(a) If the predicate

$$\mathsf{LeftTurn}(p, r, q)$$

holds, then q is bad.
(b) If q is bad, then there exists some p, r such that (a) holds.

So we can use this predicate to test the badness of any point q as long as $q \notin \{p_{\min}, p_{\max}\}$. How can we systematically apply this observation to eliminate bad points? It turns out that to test $q = p_i$, it is enough to choose $(p, q, r) = (p_{i-1}, p_i, p_{i+1})$, provided we do this systematically.

Assume that the points are distinct and have been lexicographically sorted in an array $P[0..n-1]$, and further $p_0 = p_{\min}$ and $p_{n-1} = p_{\max}$. Then one way to test for bad points is to go from left to right in the array. Suppose that the subarray $P[0..i-1]$ has been tested, and all the good points have been moved (left-shifted) into $P[0..k]$ for some $k \leq i - 1 < m$. Therefore $P[0..k]$ is actually a lower hull for $P[0..i-1]$. Since

$$P[0..k]; P[i..n-1] = P[0..k; i..n-1]$$

is a liberal hull, we can use the above observation to test if $P[k]$ is bad: if

$$\mathsf{LeftTurn}(P[k], P[k-1], P[i])$$

holds, then we can discard $P[k]$ (this amounts to setting $k \leftarrow k - 1$). If $P[k]$ is not bad, we extend our induction hypothesis by incrementing k, and then moving $P[i]$ to the kth array location:

$$k := k + 1; \quad P[k] \leftarrow P[i].$$

This can summarized be captured in the following algorithm:

```
LowerHullScan(S)
Input: S is a set n > 1 points.
Output: k and array P[0..k] representing the lower hull of S.
    ▷ Initialization
    Compute the lexicographic sort of S.
        Let the sorted sequence be P[0..m] with P[i-1] <_LEX P[i] (1 ≤ i ≤ m < n)
        If m ≤ 1, return(m, P).
        k ← 1
    ▷ Main Loop
    for (i = 2; i ≤ m; i ← i + 1)
        ▷ Invariant: P[0..k] is a lower hull of P[0..i-1].
        while (k > 1 and LeftTurn(P[k], P[k-1], P[i]))
            k ← k - 1
        k ← k + 1; P[k] ← P[i]
    P[k+1] ← P[m]; return(k + 1, P)
```

¶6. How to Use Conservative Predicates. Let us now add a new twist from Kettner and Welzl: suppose we implement an imperfect predicate, $\mathsf{LeftTurn}'(A, B, C)$ with the property that for all points A, B, C, we have

$$\mathsf{LeftTurn}'(A, B, C) \Rightarrow \mathsf{LeftTurn}(A, B, C).$$

Then $\mathsf{LeftTurn}'$ is called a **conservative version** of LeftTurn. In this setting, we may describe the original predicate as **exact**.

In terms of discarding bad points, if we use $\mathsf{LeftTurn}'$, we may sometimes fail to detect bad points. But that is OK, because we are interested in liberal lower hulls. Of course, if we define $\mathsf{LeftTurn}'(A, B, C)$ to be always false, this is not very useful. The point is that we can implement conservative predicates $\mathsf{LeftTurn}'(A, B, C)$ which are (1) very efficient, and (2) agrees with the exact predicate most of the time. If property (2) holds, we say the predicate is **efficacious**.

If we use a conservative version of LeftTurn in our LowerHullScan Algorithm above, we would have computed a liberal lower hull. To compute the true lower hull, we proceed in three phases as follows:

- Sorting Phase: sort the points lexicographically.
- Filter Phase: run the while loop of the LowerHullScan Algorithm using a conservative $\mathsf{LeftTurn}'$ predicate.
- Cleanup Phase: run the while loop again using the exact LeftTurn predicate.

As borne out by experiments [3], this can speedup the overall algorithm. This speedup may sound somewhat surprising: logically speaking, the filter phase is optional. Nevertheless, filter phase is relatively fast when we use an efficient conservative predicate. If the conservative predicate is not efficacious, then the filter phase is wasted, and the algorithm is faster without it. But for a large class of inputs (especially randomly generated ones) many bad points will be eliminated by the filter phase. This will greatly speed up the Cleanup phase, and hence the overall algorithm.

This filter idea is a very general paradigm in exact geometric computation. There is another further idea: sometimes, it is possible to "verify" if something is good more efficiently than calling the exact LeftTurn predicate. In this case, the Clean Up phase call a "verify left turn predicate" first. Only when verification fails, we will call exact predicate. We will encounter these ideas of filtering and verification again.

¶7. How to Construct Conservative Predicates Let us address the issue of constructing conservative predicates which are efficient and efficacious. The general idea is to exploit fast floating point hardware operations. There is a generally applicable method: implement the operations using interval arithmetic based on machine doubles. This slows down the computation by a small constant factor.

Suppose we implement the left-turn predicate on (A, B, C) by first evaluating the determinant $\Delta(A, B, C) = (b-a)(c'-a') - (b'-a')(c-a)$ where $A = (a, a')$, etc. This is done is machine double, and then we do some

simple checks to be sure that the positive sign of the computed value D is trustworthy. If so, we can output TRUE and otherwise FALSE. For instance, we can use Sterbenz' Theorem from Lecture 2.

¶8. Conservative or Liberal? Many computations, like our convex hull computation, can be viewed as an iterative process of accepting good points and rejecting bad points. There are two kinds of errors in such a computation: rejecting a good point or accepting a bad point. We then classify algorithms that may make mistakes into two camps: a **liberal algorithm** never erroneously reject a good point, while a **conservative algorithm** never erroneously accept a bad point. But a liberal algorithm may err in accepting a bad point while a conservative algorithm may err in rejecting a good point. This terminology was used in [5] for discussing testing policies in computational metrology.

Kettner and Welzl focused on predicates. If P, Q are two predicates over a common domain, we say P is a **conservative version** of Q if $P(x)$ is true implies $Q(x)$ is true. Similarly, P is a **liberal version** of Q if $P(x)$ is false implies $Q(x)$ is false.

When we use a conservative version of a correct predicate to reject points, the resulting algorithm is actually a liberal one! The converse is also true: using a liberal version to accept points will result in a conservative algorithm. By this terminology, the Kettner-Welzl algorithm is liberal, not conservative.

§4. The Gift Wrap Algorithm

Using the modified extremity test, we could construct simple convex hull algorithm in two steps. (1) First detect all extreme points, by going through each points of S. (2) Linking these extreme points into a circular list, using the adjacency information provided by the extended Extremity Test. But we can combine steps (1) and (2) into a more efficient algorithm. Suppose we have found a single extreme point using our extended Extremity Test:

$$(p_1, p_2, p_3) \leftarrow \textsc{Extremity Test}(S, p).$$

Assuming $p_1 \neq p_3$, we already have three extreme points to start our convex hull: $CH = (p_1, p_2, p_3)$. In general, given the partial convex hull

$$CH = (p_1, \ldots, p_i), \quad i \geq 3$$

we try to extend it by calling

$$(q_1, q_2, q_3) \leftarrow \textsc{Extremity Test}(S, p_i). \tag{8}$$

Note that q_2 will be equal to p_i. There are two possibilities: in case q_3 equals p_1, then the current convex hull is complete. Otherwise, we extend the partial convex hull to $CH = (p_1, \ldots, p_i, q_3)$.

A further improvement is to replace (8) by a more specialized routine:

$$q_3 \leftarrow \textsc{WrapStep}(S, p_{i-1}, p_i). \tag{9}$$

Here is the implementation of this specialized test. The name "Wrap Step" will be clear later.

```
WrapStep(S, p, q)
Input: S is a set n > 2 of points,
       and for all s ∈ S, Orientation(q, p, s) ≥ 0.
Output: extreme point r ∈ S such that Orientation(q, p, r) ≥ 0.
1.  r ← q.
2.  for s ∈ S,
                if LeftTurn*(r, p, s) then r ← s.
```

To use the WrapStep primitive, we first find initial extreme point p_1. We could obtain this using the Extremity Test, but this can be avoided: among the subset of points in S whose y-coordinates are minimum, we can pick p_1 so that $p_1.X$ is maximum. Then we call $WrapStep(S, p_0, p_1)$ where p_0 is defined to be $p_1 - \mathsf{Point}(1, 0)$. Note that triple (S, p_0, p_1) satisfies the preconditions for inputs to $WrapStep$.

The resulting algorithm is presented here:

```
Gift Wrap Algorithm
Input: A set S of n > 2 of points.
Output: The convex hull of S, CH(S) ← (p_1, ..., p_h), h ≥ 2.
1.  FIND INITIAL EXTREME POINT:
        Let p_1 be any point in S.
        for s ∈ S,
            if (s.Y < p_1.Y) or (s.Y = p_1.Y and s.X > p_1.X)
                then p_1 ← s.
2.  ITERATION:
        CH ← () (empty list).
        q ← p_1 − (1, 0) and r ← p_1.
        do
            Append r to CH.
            p ← q and q ← r.
            r ← WrapStep(p, q).
        while
            r ≠ p_1.
```

What is the complexity of this algorithm? If h is the size of the convex hull, then this algorithm executes the do-loop h times. Since each iteration takes $O(n)$ time, the overall complexity is $O(nh)$.

This algorithm is called the "Gift Wrap Algorithm" since it is analogous to how one wraps a gift item (albeit in 3-dimensions).

REMARK: The gift wrap algorithm can even replace the initialization loop for finding the first extreme point p_1. If we know in advance any two points

$$p_{-1}, p_0 \tag{10}$$

which are not in S, but guaranteed to form an edge in the convex hull of $S \cup \{p_{-1}, p_0\}$, we could begin our iteration with these two points. See Exercises.

¶9. Robustness Issues. Numerical nonrobustness problems in the Gift Wrap algorithm can arise from the *WrapStep* primitive. Let $(p_1, p_2, \ldots, p_h)$ is the true convex hull, with $h \geq 4$. Let us examine the possible ways that this can arise: assume the partial hull (p_1, p_2) has been computed correctly, but the next point is wrongly computed to be q instead of p_3. Evidently, p_2, q, p_3 must be nearly collinear. Assume that q is further away from p_2 than p_3 is. This is illustrated in Figure 6.

[TAKE FROM ANATOMY PAPER...]

Next, suppose that p_4 (the true 4th point on the convex hull) satisfies $\mathsf{LeftTurn}(p_4, q, p_2) = \text{True}$. So it is possible that our giftwrap algorithm produces the partial hull (p_1, p_2, q, p_3). It is also possible that our giftwrap algorithm produces (p_1, p_2, q, p_3, p_4).

We leave as an Exercise to construct an specific choice of the points $S = \{p_1, p_2, p_3, p_4, q\}$ with exactly this outcome, assuming all numbers are machine doubles and the arithmetic is computed using the IEEE standard.

Exercises

Exercise 4.1: Verify the area formula for all possible disposition of the vectors (b, b') and (c, c'). ♢

Exercise 4.2: Our extremity algorithm uses two kinds of predicates. Modify the algorithm so that only the modified LeftTurn predicate is used. ♢

Exercise 4.3: Device another $O(n)$ time extremity test, based on incremental insertion. Suppose you want to test if p is extremal in S. Let $S_i = \{p_1, \ldots, p_i\}$ for $i = 1, \ldots, n$ and $S_n = S \setminus \{p\}$. Suppose p is extremal in S_i, how do you check if p is extremal in S_{i+1}? You will need to maintain some witness of the extremity of p in S_i. ♢

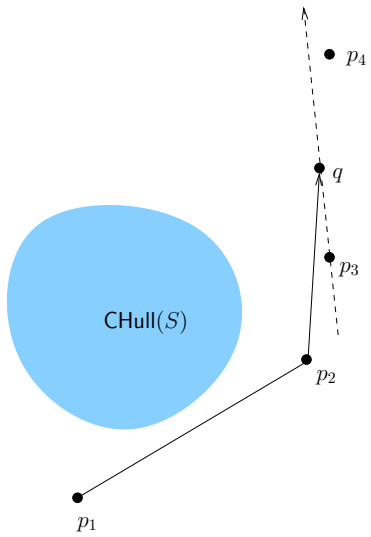


Figure 6: Error in Gift Wrap Algorithm

Exercise 4.4: Give a formula for the signed area of a simple polygon whose vertices are $A_1, A_2, \ldots, A_n$ (for $n \geq 3$). Let $A_i = \mathsf{Point}(x_i, y_i)$ for each i. ♢

Exercise 4.5: Prove the signed area of a tetrahedron in 3 dimensions. Extend this to the volume formula for a d-dimensional simplex for $d > 3$. ♢

Exercise 4.6: Suppose we can easily find a pair of artificial points p_{-1} and p_0 as in (10). Modify the Gift Wrap algorithm to produce such an algorithm. Note that such a pair of points can be manufactured easily if we admit symbolic (non-standard) numbers. ♢

Exercise 4.7: Is it possible to construct an example (assuming imprecise arithmetic) so that the Gift Wrap Algorithm goes into an infinite loop? ♢

End Exercises

§5. Study of an Incremental Algorithm

We now describe an incremental algorithm for computing the convex hull of a point set. The robustness issues here are more intricate because the algorithm is slightly more complicated than the previous one.

Nevertheless, its basic algorithmic idea is simple: assuming that we have computed the convex hull H_i of the first i points, we now update H_i to H_{i+1} by inserting the next point. The algorithm stops when we reach H_n.

In general, the **incremental paradigm** for geometric algorithms is quite powerful. When combined with randomization, this is often the method of choice in implementation. Indeed, randomization is often trivial – it amounts to introducing the points (or more generally, geometric object) into the construction in a random order. However, we will not dwell on this, but refer to excellent texts [1, 4]. The incremental algorithm for convex hull in every dimension is called the Beneath-Beyond algorithm (see [Edelsbrunner, page 147]).

THIS Lecture is taken from [2].

Exercises

Exercise 5.1: Assume the existence of a data type POINT and a predicate $\mathsf{RightTurn}(p, q, r)$. Use a doubly linked list to represent L (e.g. the STL-data type XXXX). ♢

End Exercises

§6. Other Convex Hull Methods

REMARK: The material from "Classroom Examples" is to be integrated into this chapter!!
It is instructive to see several other algorithms for the convex hull of points.

¶10. Graham Scan. Another popular convex hull algorithm, in fact the earliest $O(n \log n)$ algorithm is from R. L. Graham. The attractive feature here is the natural application of a standard data structure of computer science – the pushdown stack. We first describe the original version of this algorithm.

1. The first step is to find the $P = \mathsf{Point}(x, y)$ which is lexicographically minimum (minimize x and then minimize y). Let $P_0 = \mathsf{Point}(x_0, y_0)$ be this point. Clearly, P_0 is extremal be any definition of this word.

2. Circular Sort the other points about P_0, as explained earlier. If two points have the same angle, drop the one which is closer to to P_0. Let
$$P_0, P_1, \ldots, P_{n-1}$$
be the list of sorted points. Note that P_{n-1} is on the convex hull.

3. Create a stack S of points. Initials S with (P_{n-1}, P_0). In general, if $S = (Q_0, Q_1, \ldots, Q_k)$ were $k \geq 2$, then let *top* refer to Q_k and *next* refer to Q_{k-1}. To push an element Q onto S would make $S = (Q_1, \ldots, Q_k, Q)$. Similarly, for popping an element.

4. We process the points P_i in the i-th stage. If P_i is to the left of the line from *next* to *top* then push(P_i). Else, pop the stack.

A much more robust version of Graham scan is known as "Andrew's Version". In this version, we replace circular sorting by lexicographical sorting of the points: $p \leq_{lex} q$ iff $p_x < q_x$ or ($p_x = q_x$ and $p_y < q_y$). Since this sorting does not require any arithmetic, it is fully robust. The resulting sequence is a x-monotone in a saw-teeth fashion. We then compute the upper and lower hull of this sequence using the Graham scan.
[See paper of Kettner-Welzl]

¶11. Lower Bounds It can be shown that within a reasonable model of computation, every convex hull algorithm requires $\Omega(n \log n)$ steps in the worst case. In this sense, we usually call an algorithm with running time $O(n \log n)$ an "optimal algorithm". But witness the next result.

¶12. Ultimate Convex Hull Algorithm A remarkable algorithm was discovered by Kirkpatrick and Seidel: their algorithm takes time $O(n \log h)$. Thus, it is simultaneously better than gift-wrapping and $O(n \log n)$. Notice that we manage to beat other "optimal algorithms". Of course, this is no contradiction because we have changed the rules of the analysis: we took advantage of another complexity parameter, h. This parameter h is a function of some **output size parameter**. Algorithms that can take advantage of the output size parameters are said to be **output-sensitive**. This is an important algorithmic idea. In terms of this h parameter, a lower bound of $\Omega(n \log h)$ can be shown.

¶13. Randomized Incremental Algorithms We will return to this later, to introduce two other important themes in computational geometry: randomization and incremental algorithms. Another direction is to generalize the problem to the "dynamic setting". Here the point set S is considered a dynamic point set which can change over time. We can insert or delete from S. Incremental algorithms can be regarded as the solution for the "semi-dynamic" case in which we can only insert, but not delete points. Yet another direction is to consider the convex hull computation in higher (or arbitrary) dimensions.

Exercises

Exercise 6.1: Prove the correctness of Graham Scan. ♢

Exercise 6.2: Implement the Andrew Scan. ♢

End Exercises

§7. APPENDIX: Convexity Concepts

We introduce some basic concepts of convexity in an Euclidean space $\mathbb{R}^d$ of dimension $d \geq 1$. We will mostly discuss $d = 2$ or $d = 3$.

If $p, q \in \mathbb{R}^d$ are two points, then the closed **line segment** joining them is the set of points of the form

$$\lambda p + (1 - \lambda)q$$

for $0 \leq \lambda \leq 1$. This segment is denoted $[p, q]$. When λ is unrestricted, we obtain the **line** through p and q, denoted $\overline{p, q}$. If $a_0, \ldots, a_d \in \mathbb{R}$, then the set of points $p = (p_1, \ldots, p_d) \in \mathbb{R}^d$ satisfying the linear equation $a_0 + \sum_{i=1}^d a_i p_i = 0$ is called a **hyperplane** (for $d = 2$ and $d = 3$, we may simply call it a line or a plane, respectively). A hyperplane H defines two **open half-spaces**, corresponding to those points p such that $a_0 + \sum_{i=1}^d a_i p_i > 0$ or $a_0 + \sum_{i=1}^d a_i p_i < 0$, respectively. Let these be denoted H^+ and H^-, respectively. The corresponding closed half-spaces are denoted H_0^+ and H_0^-. A set $R \subseteq \mathbb{R}^d$ of points is convex if for all $p, q \in R$, we $[p, q] \in R$. A point p is **extreme** in R if there is a hyperplane H containing p such that H^+ or H^- contains $R \setminus \{p\}$. The point is **weak extreme** if there is a hyperplane H containing p such that H_0^+ or H_0^- containes R. A **supporting hyperplane** of R is any hyperplane H through a point of R such that R lies in one of the two closed half-spaces defined by H.

We now define the **convex hull** of a set $R \subseteq \mathbb{R}^d$ to be the smallest closed convex set containing R. We denote it by $\mathsf{CHull}(R)$. A **convex polyhedron** is a convex set R with a finite number of extreme points. A convex polyhedron might be unbounded. If R is also bounded, it is called a **convex polytope**. A **polyhedron** (resp. **polytope**) is a finite union of convex polyhedras (resp. polytopes). In 2-dimensions, a polytope is call a **polygon**.

A convex polytope is completely determined by its set of extreme vertices. Thus we can regard the output of our convex hull algorithm as a representation of a convex polygon. In our problem specification, we demanded a little more: namely we want an explicit representation of the "adjacency relation" among the extreme points. We say two extreme points p, q of a convex set R are **adjacent** if there is a hyperplane H containing p, q such either H^+ or H^- contains the set $R \setminus H$. Thus, in the circular list $(p_1, p_2, \ldots, p_k)$ that is output by our planar convex hull algorithm, the point p_i is adjacent to p_{i-1} and p_{i+1} where subscript arithmetic is modulo k.

References

[1] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1997. Translated by Hervé Brönnimann.

[2] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computation. *Comput. Geometry: Theory and Appl.*, 40(1):61–78, 2007.

[3] L. Kettner and E. Welzl. One sided error predicates in geometric computing. In K. Mehlhorn, editor, *Proc. 15th IFIP World Computer Congress, Fundamentals - Foundations of Computer Science*, pages 13–26, 1998.

[4] K. Mulmuley. *Computational Geometry: an Introduction through Randomized Algorithms*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1994.

[5] C. K. Yap and E.-C. Chang. Issues in the metrology of geometric tolerancing. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robot Motion Planning and Manipulation*, pages 393–400, Wellesley, Massachusetts, 1997. A.K. Peters. 2nd Workshop on Algorithmic Foundations of Robotics (WAFR).

"Accordingly, the defendant is found Not Guilty.
However, this Court is of the opinion that variable-precision floating point is perilous, and its use should be restricted to qualified professionals."

— Dirk Laurie
in "Variable-precision Arithmetic Considered Perilous — A Detective Story"
Electronic Trans. on Numerical Analysis, Vol. 28, pp. 168–173, 2008.

Floating point arithmetic has numerous engineering advantages: it is well-supported... the Challenge is to demonstrate that a reliable implementation can result from the use of floating point arithmetic.

– Steve Fortune

Lecture 4
ARITHMETIC TECHNIQUES

Since numerical errors are the cause of nonrobustness, it stands to reason that if we make numerical computations more accurate, we improve the robustness of our programs. In this section, we look at some techniques for improving numerical accuracy. In particular, we look at floating point techniques to determine the correct sign of a sum, and also at robust rotations using rational angles.

§1. More Accurate Machine Arithmetic

Since numerical error is the source of non-robustness, it is natural to try to improve upon the standard floating point arithmetic. Can we improve on the conventional floating point arithmetic, which has already reached a highly optimal design under the IEEE Standard? More generally, what techniques are available in the FP mode of computation?

In conventional floating point systems, the number of bits allocated to the mantissa and the exponent are fixed. Matsui and Iri [21] proposed a system where these can vary. But let us look at a remarkable proposal of Clenshaw, Olver and Turner [5] called **level-index arithmetic** (LI).

First, any non-negative real number x is assigned a unique **level** $\ell(x) \in \mathbb{N}$ defined by the property that $\ln^{(\ell(x))}(x) \in [0,1)$, where $\ln^{(n)}(x)$ denotes n applications of the natural logarithm ln. The **index** of x defined by $i(x) := \ln^{(\ell(x))}(x)$. The function $\psi(x) = \ell(x) + i(x)$ is called the **generalized logarithm**; its inverse $\phi(x)$ is the **generalized exponential** defined by

$$\phi(x) = \begin{cases} x & \text{if } 0 \le x < 1 \\ e^{\phi(x-1)} & \text{else.} \end{cases}$$

Thus any real number x can be represented by a triple $LI(x) := (s(x), \ell(|x|), i(|x|))$ where $s(x) \in \{\pm 1\}$ is defined so that $s(x) = +1$ iff $x \ge 0$.

The symmetric version of LI is called **symmetric LI** or SLI [6]. In SLI, a non-zero number x with magnitude < 1 is represented by the level index representation of $1/x$, otherwise it basically has the same representation as before. More precisely, $x \in \mathbb{R}$ is represented by a quadruple

$$SLI(x) := (r(x), s(x), \ell(|x|^{r(x)}), i(|x|^{r(x)})) \tag{1}$$

where $r(x) = \pm 1$. We define $r(x) = -1$ iff ($|x| < 1$ and $x \neq 0$). Suppose $SLI(x) = (r', s', \ell', i')$. Then $x = 0$ iff $i' = 0$. Assuming $i' \neq 0$, the triple (s', ℓ', i') is either $LI(x)$ or $LI(1/x)$ (depending on whether $r' = +1$ or -1).

Now consider a fixed-precision version of the SLI representation: suppose we have a fixed budge t of bits to represent a number x. We use two bits for $r(x), s(x)$. It is suggested that, for all practical purposes, allocating 3 bits to $\ell(x)$ suffices. Indeed, with $\ell(x) = 5$, x can reach numbers as high as $2^{2^{5,500,000}}$. Thus $t - 5$ bits are available for representing the index, $i(x)$. There are many remarkable properties of this representation: its range is so hugh that perhaps only astronomers would fully appreciate it. The representable numbers have gaps that are much more smoothly distributed than standard floating point representation. Overflow and underflow are practically eliminated: assuming the 3-bit scheme and $t < 5,500,000$, there is no overflow or underflow in the four arithmetic operations [20]. Nevertheless, the complexity of the algorithms for basic arithmetic in LI/SLI, and their less familiar error analysis, conspire to limit its wider use. On the other hand, in the early days of modern computing, that was exactly the same prognosis for floating-point arithmetic vis-à-vis fixed-point arithmetic. Hence there is perhaps hope for SLI.

Another way to improve the accuracy of fixed precision arithmetic is to introduce some accurate primitives for critical operations. One such critical operation is the scalar product $\sum_{i=1}^{n} a_i b_i$ of two vectors. In other words, for n within a reasonable range, we would like this scalar product to yield the closest representable number. The usefulness of such extensions for problems such as line segment intersection was pointed out in [24]. In recent years, there has appeared in hardware an operation called FMA (**fused multiply and add**). This is the ternary operation of $FMA(a, b, c) = ab + c$. E.g., we can implement an accurate scalar product operation using only n FMA's.

Yet another direction is to go to **arbitrary precision arithmetic** (AP mode). In the AP mode, one can either use arbitrary precision floating point numbers, or use arbitrary precision rational numbers. The former is inherently an approximation-based computation while the latter is error-free when a computation uses only the rational operations $(\pm, \times, \div)$. The last section of this chapter explores techniques for performing rotations using arbitrary precision rational numbers.

The use of **interval arithmetic** or more generally, **validated computation**, [23, 1, 19] is also major direction for achieving robustness. Interval arithmetic is independent of the FP mode: it is just as useful in arbitrary precision computation. The interval bounds can rapidly increase, unless one counteracts this by actively reducing these intervals, using increasing precision. In this sense, arbitrary precision computation is a better setting for applications of interval arithmetic.

Software and Language Support. Ultimately any approach to robustness ought to be supported by appropriate software tools, so that it is easy to implement robust algorithms using the particular approach. For instance, to support interval methods, an extension of `Pascal` called `Pascal-SC` has been developed [2]. This language supports validated arithmetic with facilities for rounding, overloading, dynamic arrays and modules. Further extensions have been made in the languages `PASCAL-XSC`, `C-XSC` and `ACRITH-XSC`. The language **Numerical Turing** [14, 15, 16] has a concept of "precision block" whereby the computation within such a block can be repeated with increasing precision until some objective is satisfied. Despite the success of the IEEE Arithmetic in hardware, it has yet to be fully at the level of of programming languages. In particular, the proper treatment of arithmetic exceptions [8] is still lacking.

§2. Summation Problem

The previous section discuss some general proposals to use more general representations for a suitable subsets of the real numbers. These ideas suggests that, for any given bit-budget, there are extensions of the standard floating point representation that can increase the range and accuracy of the standard representations.

In this section, we consider another line of attack: we focus on certain critical computations and see if we can improve the accuracy of performing these operations in the standard model (Lecture 2, §5) of numerical computation.

We focus on the **summation problem**, which is to compute the sum

$$S = \sum_{i=1}^{n} a_i \tag{2}$$

of a sequence $a_1, a_2, \ldots, a_n$ of input numbers. Mathematically, summation is a triviality, but within FP computation, this is[1] no longer true. Such summations are central to more complex operations such as matrix-vector multiplications or the scalar product of two vectors. At any rate, it is the simplest computation with a cascaded sequence of primitive arithmetic steps which we can study. In Chapter 2, we already analyzed the error for the obvious algorithm for this problem. Higham [13, chapter 4] treats this problem in detail.

In this section, we want to introduce a remarkable algorithm called "compensated summation". By way of motivation, consider the sum $\widetilde{s} = [a + b]$ where a, b are floating point numbers with $|a| \geq |b|$. Here $[a + b]$ uses the bracket notation to denote floating point arithmetic in the standard model discussed in Chapter 2.

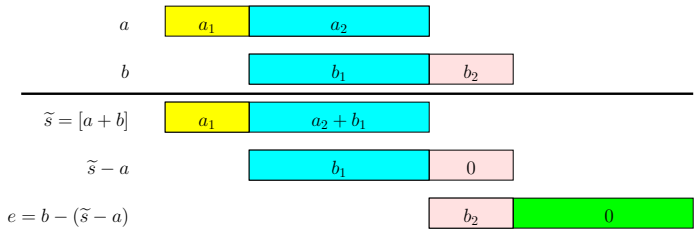


Figure 1: Estimated error, $e = (b - ((a + b) - a))$.

In Figure 1, we have illustrated a sequence of three computational steps starting from a pair of floating point numbers a, b:

$$\begin{aligned} \widetilde{s} &:= a + b \\ \widetilde{t} &:= \widetilde{s} - a \\ e &:= b - \widetilde{t} \end{aligned}$$

The results of these operations are aligned to give intuition to an error analysis of these steps. From studying this figure, a reader might find it quite plausible that the computed quantity

$$e = (b - ((a + b) - a)),$$

has the property that $|e|$ is the error in $\widetilde{s}$ as an approximation to $a + b$. Indeed, this is a result of Dekker:

Lemma 1 (Dekker (1971)). *Let $|a| \geq |b|$ and compute the expression*

$$\widetilde{e} = [b - [\widetilde{s} - a]] = [[a - [a + b]] + b].$$

If the floating point arithmetic is in base 2, then

$$a + b = \widetilde{s} + \widetilde{e}. \tag{3}$$

In other words, $|\widetilde{e}|$ is precisely the error in $\widetilde{s}$.

This gives rise to the following summation algorithm.

Compensated Sum Algorithm
Input: Floating point numbers $a_1, \ldots, a_n$
Output: Approximate sum $s = \sum_{i=1}^{n} a_i$

```
s ← 0; e ← 0;
for i = 1 to n
    temp ← s;
    a ← [a_i + e];    ◁ Compensated summand
    s ← [temp + a];   ◁ Cumulative sum
    e ← [[temp − s] + a];   ◁ Error estimate
```

[1] This is true of many issues in FP computation: mathematical trivialities that aren't.

The idea is to estimate the error of the addition at each step, and to add this error back to the partial sum. Note that this error estimate is not exact because Dekker's analysis depends on the inequality $|a| \geq |b|$ which we do not guarantee. Nevertheless, it can be shown (see [18, pp. 572–573] and [11]) that the computed value is equal to

$$\widetilde{s} = \sum_{i=1}^{n}(1+\delta_i)a_i, \qquad |\delta_i| \leq 2u + O(nu^2). \tag{4}$$

So the relative backwards error in each argument a_i is about $2u$. This is much better than the corresponding error in our analysis in Lecture 2: we showed that the relative backwards error is $\gamma_{n-i+1} \approx (n-i+1)u$ for each a_i $(i = 1, \ldots, n)$.

Exercises

Exercise 2.1:

(i) Implement the Compensated Sum Algorithm in same standard programming language.
(ii) Randomly generate sequences of floating point numbers in the range $[-1, 1]$ and compute their sum using three methods:
(A) the straightforward method in machine double precision,
(B) your compensated sum algorithm from (a), and
(C) an exact algorithm using BigFloats.

Use sequences of length $5, 10, 50, 100, 1000$, and compute some statistics about their relative accuracy and speeds. ♢

Exercise 2.2: Prove Dekker's lemma. ♢

Exercise 2.3: Prove the error bound (4). ♢

Exercise 2.4: Let $x_1, \ldots, x_n$ be a sequence of numbers. Consider the following five strategies for forming their sum:

- (O) Random reordering (if your input is randomly generated, you do not need to do anything).
- (A) Sort the numbers in non-decreasing order and add them in a simple loop.
- (B) As in (A), but first sort the numbers in non-increasing order.
- (C) Put the numbers in a min-priority queue, and at each iteration, extract two numbers, and insert their sum back into the queue. We halt when there is only one number in the queue. [This method is called "Psum" in [13, p. 90].
- (D) As in (C), except we use a max-priority queue.

(i) Experimentally, show that when the numbers are positive, (A) and (C) are superior to (B) and (D), respectively.
(ii) Give examples in which (B) and (D) are better than (A) and (C).
(iii) Compare the performance of (A) and (C) for positive numbers. ♢

End Exercises

§3. Sign of Sum Problem

A simplification of the Summation Problem is the following **Sign of Sum Problem** (SoS Problem): *Given a sequence $a_1, a_2, \ldots, a_n$ of numbers, determine the sign of the sum $S = \sum_{i=1}^{n} a_i$.* An exact solution for SoS can be used to solve the non-robustness of problems such as convex hulls. To see this, observe that all convex hull algorithms ultimately depend on the predicate $\mathsf{Orientation}(p, q, r)$. According to the principles of exact geometric computation (EGC), the convex hull would be computed exactly provided all such predicates are evaluated without error. But the Orientation predicate can be reduced to the sign of a

2×2 determinant, which is an expression of the form $ad - bc$; this is the SoS Problem with $n = 2$. For a more accurate analysis, we should view it as a special 3×3 determinant, which expands to

$$\Delta = (b - a)(c' - a') - (c - a)(b' - a') = bc' - ac' - ac' - cb' + ab' + ca'. \tag{5}$$

This is SoS with $n = 6$.

Ratschek and Rokne [25] introduced an exact algorithm for SoS, called[2] ESSA. But before going into the details of ESSA, let us develop some intuitions. Suppose you are given a sequence of numbers, and you want to determine the sign of its sum. How can you organize the arithmetic to improve its reliability over the naive summation? The simplest idea is to break the sum into two parts, comprising the positive and negative numbers. Assume the positive part is $A = \sum_{i=1}^{k} a_i$ (with $a_i > 0$) and the negative part is $B = -\sum_{j=1}^{n-1} b_j$ (where $b_j > 0$). Then we just have to compare A and B.

We now focus on improving the summation process. We only need to focus on the first sum A. Suppose the a_i's are sorted in increasing order and we form the cumulative sum from smallest to largest. The intuition reason for this is clear – we are trying to minimize the truncation error by adding comparable size numbers (recall that in floating point addition, we first align the two numbers at their binary point, truncate the least significant bits of the smaller number, and then add). But this sorting is not optimal, and does not always add two numbers that are as comparable as possible. To fix this, we next put all the a_i's into a min-priority queue and at each step, pull two elements from the queue, add them, and insert the sum back into the queue. Call this the **priority queue SoS Algorithm**.

The Priorty Queue SoS Algorithm seems pretty good. Of course, it is not exact. We now address the ESSA Algorithm which is based on a similar idea, but it achieves exactness by tracking any truncation error. We assume the arithmetic is in the floating point system $F(2, t)$ (for some t) and

$$n \leq 2^{t-1}. \tag{6}$$

Recall from Chapter 2 that $F(2, t)$ comprise all numbers of the form $b_1.b_2 \ldots b_t \times 2^e$ where $b_i \in \{0, 1\}$ and $e \in \mathbb{Z}$. The exponent does not overflow or underflow in $F(2, t)$. By sorting the inputs in a preprocessing step, we may further assume

$$a_1 \geq a_2 \geq \cdots a_k > 0 > a_{k+1} \geq \cdots \geq a_n.$$

By renaming the negative inputs a_{k+i} as $-b_i$ $(i = 1, \ldots, \ell = n-k)$ the problem is now reduced to determining the sum

$$S = A - B := \sum_{i=1}^{k} a_i - \sum_{j=1}^{\ell} b_j.$$

Let E_i be the exponent part of a_i and F_j the exponent part of b_j. Thus, $2^{E_i} \leq |a_i| < 2^{E_i+1}$. The algorithm goes as follows:

[2] ESSA stands for Exact Sign of Sum Algorithm. Beware that the other acronym "SoS" also refers to "Simulation of Simplicity", a technique for treating data degeneracies from Edelsbrunner and Muecke.

ESSA
Input: a_i, b_j $(i = 1, \ldots, k, j = 1, \ldots, \ell)$ as above.
Output: The sign of $S = \sum_i a_i - \sum_j b_j$

1. (BASIS) Terminate with the correct output in the following cases:
   - 1.1 If $k = \ell = 0$ then $S = 0$.
   - 1.2 If $k > \ell = 0$ then $S > 0$.
   - 1.3 If $\ell > k = 0$ then $S < 0$.
   - 1.4 If $a_1 \geq \ell 2^{F_1+1}$ then $S > 0$.
   - 1.5 If $b_1 \geq k 2^{E_1+1}$ then $S < 0$.
2. (AUXILIARY VARIABLES)
   $a' = a'' = b' = b'' = 0$;
3. (PROCESSING THE LEADING SUMMANDS)
   CASE $E_1 = F_1$:
   If $a_1 \geq b_1$ then $a' \leftarrow a_1 - b_1$
   Else $b' \leftarrow b_1 - a_1$;
   CASE $E_1 > F_1$:
   $u \leftarrow 2^{F_1+1}$;
   $a' \leftarrow a_1 - u$, $a'' \leftarrow u - b_1$;
   CASE $E_1 < F_1$:
   $v \leftarrow 2^{E_1+1}$;
   $b' \leftarrow b_1 - u$, $b'' \leftarrow u - a_1$;
4. (UPDATE BOTH LISTS)
   Discard a_1, b_1 from list.
   Insert a', a'' into the a-list (only non-zero values need to be inserted).
   Insert b', b'' into the b-list (only non-zero values need to be inserted).
   Update the values of k, ℓ and return to Step 1.

To see the correctness of this algorithm, we note the following properties:

- *Basis.* The correctness of Step 1 is easy. For instance, Step 1.4 is true because $\sum_{j=1}^{\ell} b_j < \ell 2^{F_1+1}$.
- *Each arithmetic operation is error free.* This concerns only Step 3: In case $E_1 = F_1$, the value $a' \leftarrow a_1 - b_1$ is computed exactly. In case $E_1 > F_1$, then $a'' \leftarrow u - b_1$ is exact because the exponent of u is exactly one more than the exponent of b_1 and so (because of the guard bit) the scaling of b_1 before the operation incurs no loss in precision. To see that $a' \leftarrow a_1 - u$ is also exact, we use the fact that the non-execution of Step (1.4) implies $a_1 < \ell 2^{F_1+1}$ and hence by assumption (6), we have $a_1 < 2^{F_1+t}$. This implies $E_1 \leq F_1 + t - 1$. In the operation $a_1 - 2^{F_1+1}$, we need to scale the operand 2^{F_1+1} by right shifting $E_1 - F_1 - 1$ positions. Since $E_1 - F_1 - 1 \leq t$, and again because of the guard bit, no truncation error occurs.
- *Invariance of sum S.* The main computation is represented by Step 3. At the end of Step 3, we verify that the following invariant holds

$$a_1 - b_1 = (a' + a'') - (b' + b'').$$

  This implies that updating the a- and b-lists in Step 4 preserves the value $S = A - B$ where A, B are the sum of the two lists.
- *Partial Correctness.* From the invariance of the sign, and the correctness of Step 1, this proves that the algorithm is correct if it halts.
- *Termination.* This follows by observing that the largest element in either the a-list or the b-list is reduced in each iteration.

Although termination is guaranteed but it may be slow in the worst case. In practice, this worst case behaviour may be an issue. If we apply ESSA to our 2×2 determinant $ad - bc$, there are two places where the

numbers may overflow or underflow in a finite precision number system (not $F(2,t)$ but $F(2,t,e_{\min},e_{\max})$). First the values a,b,c,d really arise as the difference of two floating point numbers So this may cause an overflow or underflow in the values a,b,c,d. The possibility of overflow or underflow is reduced if we view the problem as an ESSA problem with $n=6$ as in (5). Each of the 6 terms is a produce of two floating point numbers, and a potential overflow still exists; but there are well-known ways to handle this [25].

Exercises

Exercise 3.1:
(i) Complete the proof that the ESSA algorithm is correct.
(ii) Describe modifications to the algorithm when exponents are limited in the range $[e_{\min},e_{\max}]$. ♢

Exercise 3.2: Consider the Priority Queue SoS Algorithm.
(i) Give examples where this algorithm is inexact.
(ii) Improve this algorithm by working on the priority queues for A and B at the same time. ♢

Exercise 3.3: Implement ESSA. Experimentally compare its performance against the following alternative methods:
(E) Using ESSA
(N) Naive Method (no reordering), just add and compare.
(S) Separate the input numbers into positive and negative parts, and add them separately. Then determine their sign.
(C) As in (S), but use the Compensated Sum method to add the separate positive and negative parts.

We want to use two classes of inputs:
(i) One class of inputs should be a random input sequences of numbers taken from the range $[-1,1]$. Use sequences of length $5,10,50,100,1000$.
(ii) The second class of inputs are those that sum to zero. For this, general sequences of numbers from the range $[0,1]$, and then append the negation of the input. Then randomly permute the numbers before feeding to your algorithms.

As usual, compute some statistics about their relative accuracy and speeds. ♢

Exercise 3.4: (i) Analyze the worst case behavior of ESSA.
(ii) Give numerical examples with this behavior. ♢

Exercise 3.5: Show that for all $p\geq 1$, if $2^{-p}b<a<2^pb$ then the error in computing $a-b$ is at most $p-1$ bits. ♢

End Exercises

§4. Evaluation Strategies for Orientation Predicate

In the previous two sections, we studied two numerical problems (Summation and Sign of Sum). In this section, we focus on a more specifically geometric problem.

The orientation predicate is a critical one in many elementary problems of computational geometry. In this section, we explore some heuristics for the accuracy of floating point evaluation of this predicate. Note that even if we are ultimately interested in the *exact* sign of sum, it is useful to have heuristic techniques which may not be always correct, but are usually correct. Such techniques, if they are efficient, can be used as filters in exact computation.

There are essentially 3 evaluation strategies for the orientation predicate $\mathsf{Orientation}(p,q,r)$ (see Lecture 3,§4). We consider the strategies for choosing one of p, q or r as the **pivot** for forming the 2×2 determinant $ad-bc$. For instance, if p is the pivot then the entries a,b,c,d are the coordinates of $q-p$ and $r-p$:

$$a=q.x-p.x,\quad b=q.y-p.y,\quad c=r.x-p.x,\quad d=r.y-p.y.$$

Which is the best choice of a pivot?

Fortune [10] suggests to choose the pivot which minimizes

$$|ad| + |bc|. \tag{7}$$

If we interpret $|ad|$ and $|bc|$ as the area of suitable boxes involving the coordinates of p, q, r, than this amounts to choosing the pivot whose corresponding boxes have least total area.

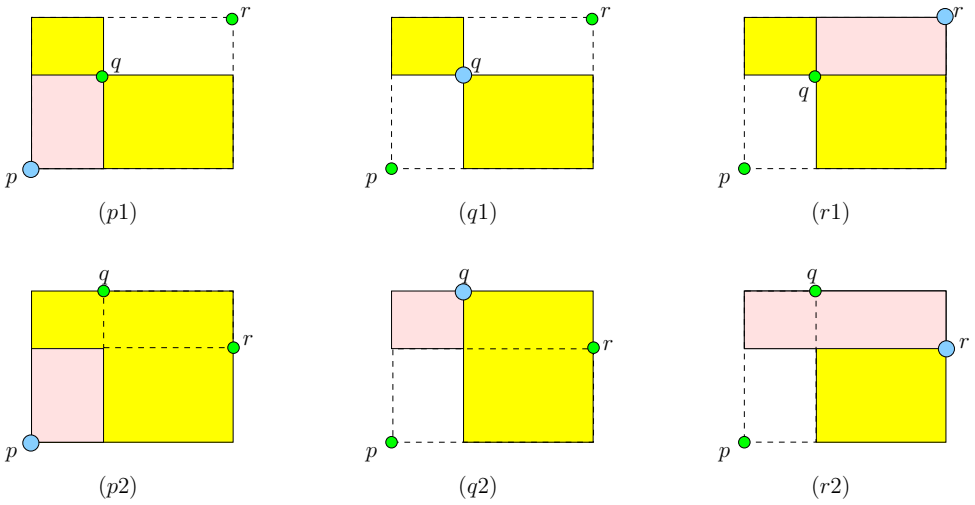


Figure 2: Evaluation Strategies: pivots are indicated by the larger (blue) circles.

Let B be the smallest axes-parallel box containing p, q, r. In terms of pivot choices that depends on $|ad|$ and $|bc|$, there are basically two distinct configurations of p, q, r. These correspond to the two rows in Figure 2: the top row corresponds to the case where two points (say p, r) are corners of B. The bottom row corresponds to the case where only one point (say p) is a corner of B. In each row, we show the three possible choice of pivots: p, q or r. The two boxes corresponding to $|ad|$ and $|bc|$ are drawn, and any overlap region is shown in darker shade. It is clear from these pictures that in the top row, pivoting at q will minimize the area (7); in the bottom row, the minimum is achieved by pivoting at q or r. Summarizing all these cases: the minimum is achieved if we pivot at the points whose x-coordinate is the median among the x-coordinates of p, q, r. Naturally, we could also use the y-coordinates.

How can we justify this heuristic? Recall (Chapter 2) that in the standard floating point model, each machine operation $\circ \in \{+, -, \times, \div, \sqrt{\cdot}\}$ satisfy the following property:

$$[x \circ y] = (x \circ y)(1 + \varepsilon)$$

where $|\varepsilon| \leq \mathbf{u}$ (the unit roundoff), and the bracket notation $[\cdots \circ \cdots]$ indicates the machine operation corresponding to $\circ$. The numbers a, b, c, d in the orientation predicate are not exact, but obtained as the difference of two input numbers. Let the floating point results of these differences be a', b', c', d'. Thus

$$\begin{aligned}
a' &= a(1+\varepsilon_a), \qquad b' = b(1+\varepsilon_b), \qquad c' = c(1+\varepsilon_c), \qquad d' = d(1+\varepsilon_d) \\
[a'd'] &= ad(1+\varepsilon_a)(1+\varepsilon_d)(1+\varepsilon_{ad}) \\
[b'c'] &= bc(1+\varepsilon_b)(1+\varepsilon_c)(1+\varepsilon_{bc)} \\
[[a'd'] - [b'c']] &= \big(ad(1+\varepsilon_a)(1+\varepsilon_d)(1+\varepsilon_{ad}) - bc(1+\varepsilon_b)(1+\varepsilon_c)(1+\varepsilon_{bc)}\big)\,(1+\varepsilon_{abcd}), \\
&= (ad)(1+\theta_0) - (bc)(1+\theta_1)
\end{aligned}$$

where each $|\varepsilon_i| \leq \mathbf{u}$ and hence $|\theta_j| \leq \gamma_4$ (where γ_4 is defined in Chapter 2.4). *Assuming that the ε_i's are independent*, the *worst case* error absolute error is $(|ad| + |bc|)\gamma_4$. This error is minimized when we minimize $|ad| + |bc|$.

Let $\mathsf{Orientation}(p, q, r, k)$ be the evaluation of $\mathsf{Orientation}(p, q, r)$ using the k-th point as pivot. That is, $k = 1$ means pivot at p; $k = 2$ means pivot at q, and $k = 3$ means pivot at r. The code goes like this:

Fortune's Pivoting Strategy
Input: point p, q, r
Output: approximate Orientation(p, q, r)
If $(p.x < q.x)$ then
If $(q.x < r.x)$ then $k = 2$,
Else if $(p.x < r.x)$ then $k = 3$,
Else $k = 1$.
Else If $(p.x < r.x)$ then $k = 1$,
Else $(q.x < r.x)$ then $k = 3$,
Else $k = 2$.
Return(Orientation(p, q, r, k).

This is only a heuristic is because the errors (the various ε_i's) are not really independent. So it is possible that a deeper analysis will give a better worst case bound. The Exercises ask you to explore other heuristics for choosing the pivot.

The above implementation was discussed and used in an stable algorithm of Fortune for maintaining Delaunay Triangulations.

Exercises

Exercise 4.1: Conduct some experimental studies to investigate the effectiveness of Fortune's pivoting strategy. ♢

Exercise 4.2: Consider implementations of Fortune's pivoting strategy, trying to minimize the number of arithmetic operations:
(i) What is the maximum and minimum number of comparisons on any input p, q, r?
(ii) The goal is to determine the median value in $p.x, q.x, r.x$. An alternative strategy here is to begin by computing the differences $q.x - p.x$ and $r.x - p.x$. What does one do next?
(iii) Compare the approach in (ii) with the one in the text. ♢

Exercise 4.3: Extend the pivot-choice heuristic of Fortune to the 3-dimensional orientation predicate. Conduct some experiments to see the efficacy of your strategy. ♢

Exercise 4.4: Explore other pivot choice heuristics for Orientation(p, q, r). Experimentally compare Fortune's heuristic with the pivot corresponding to the smallest (or largest) angle in the triangle $\Delta(p, q, r)$. ♢

Exercise 4.5: Analyze more carefully the optimal choice of a pivot for the 2-dimensional orientation predicate. In particular, do not assume that the round-off errors of the operations are independent. ♢

Exercise 4.6: Apply the same heuristic to the evaluation of the InCircle predicate. ♢

End Exercises

§5. Exact Rotation

So far, we considered arithmetic techniques based on fixed precision floating point arithmetic. This section considers arbitrary precision arithmetic. One approach to nonrobustness is to require that all arithmetic operations are performed exactly. The simplest class of problems where such **exact arithmetic approach** is applicable is when our computation require only rational operations $(\pm, \times, \div)$, and the input numbers are rational. For instance, computing the convex hull of a set of points can be treated by this approach.

But what if irrational operations are called for? One class of operations that is frequently desired in geometric modeling is rigid transformations, defined to be transformations that preserve distance and

orientation. In the plane, rigid transformations can be decomposed into a translation $(x, y) \mapsto (x + a, y + b)$ where $a, b \in \mathbb{R}$ composed with a rotation, $(x, y) \mapsto (x, y) \cdot R(\theta)$ where

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

is a 2×2 matrix representing rotation by angle θ. The entries in $R(\theta)$ involve the transcendental functions $\sin(\theta), \cos(\theta)$. Such functions are a problem for exact computation. In this section, we look at some special classes of angles for which $\sin(\theta), \cos(\theta)$ can be computed exactly.

A number $z \in \mathbb{C}$ is algebraic if it is the root of a polynomial $P(X) \in \mathbb{Z}[X]$, $P(z) = 0$. For instance, $\mathbf{i} = \sqrt{-1}$ is algebraic as it is the root of $P(X) = X^2 + 1$. A basic fact is that the algebraic numbers are closed under the four arithmetic operations. If $z \in \mathbb{C}$ is not algebraic, it is **transcendental**. For instance, $\pi = 3.1415926\cdots$ and $e = 2.718281828\cdots$ are transcendental. We recall a basic result from transcendental number theory. Lindemann's theorem says that if $\theta \in \mathbb{C}$ is non-zero and algebraic, then e^θ is transcendental.

Lemma 2. *If $\theta \in \mathbb{R}$ is non-zero then either θ or $\cos\theta$ must be transcendental.*

Proof. Clearly, for any θ, either both $\cos\theta, \sin\theta$ are algebraic or both are transcendental. If θ is transcendental, we are done. Otherwise, Lindemann's theorem implies that $e^{\mathbf{i}\theta}$ is transcendental. Then the identity $e^{\mathbf{i}\theta} = \cos\theta + \mathbf{i}\sin\theta$ implies that $\cos\theta$ and $\sin\theta$ must be transcendental. **Q.E.D.**

We also recall the Gelfond-Schneider theorem: if $a, b \in \mathbb{C}$ such that b is algebraic, $b \notin \{0, 1\}$, and a is algebraic and irrational, then b^a is transcendental. Most of the "well-known angles" such at 45° or 60° have the property that their sine or cosines are rational. Of course, these well-known angles are commensurable with π. Two quantities a, b are commensurable means that a/b are rational. How general is this observation? The following is from [4]:

Lemma 3. *If $\cos\theta$ and θ/π are algebraic, the θ and π are commensurable.*

Proof. Write $\theta = \alpha\pi$ for some α. By assumption, α is algebraic. But we must prove that it is, in fact, rational. Since we assume $\cos\theta$ is algebraic, so is $e^{\mathbf{i}\theta} = \cos\theta + \mathbf{i}\sin\theta$. Hence

$$e^{\mathbf{i}\theta} = e^{\mathbf{i}\pi\alpha} = \left(e^{\mathbf{i}\pi}\right)^\alpha = (-1)^\alpha.$$

By Gelfond-Schneider, if α is irrational, then $(-1)^\alpha$ is transcendental. But since $(-1)^\alpha$ is algebraic, we conclude that α is rational. **Q.E.D.**

We could carry out exact comparisons and the basic arithmetic operations exactly (without error) on algebraic numbers; the same cannot be said for transcendental numbers. These ideas will be pursued in later chapters. To ensure robustness in rigid transformations, we must therefore be more selective in the set of angles for which we allow for rotation. Informally, call these the **admissible angles**. We note some candidates for admissible angles:

(A) Restrict θ to be **Pythagorean**, i.e., both $\sin\theta$ and $\cos\theta$ are rational numbers.

(B) Restrict θ to be **geodetic**, i.e., the square of $\sin\theta$ (equivalently $\cos\theta$) is rational.

(C) Restrict θ to be **commensurable with** π i.e., θ/π is a rational number. For short, we just say "commensurable".

(D) Restrict θ so that $\cos\theta$ is algebraic.

Canny, Donald and Ressler [3] introduced the computational problem of approximating arbitrary angles by Pythagorean angles. They did not explicitly define Pythagorean angles, but defined a **rational sine** to be[3] a rational solution s to the equation $s^2 + c^2 = 1$. In any case, the definition of Pythagorean angles

[3] They did not explicitly say whether c ought to be rational or not; but it is clear from their development that c is intended to be rational. But if c is not required to be rational, this implicitly defines a class of angles that is between Pythagorean and geodetic.

deserves comment. For instance, a less restrictive notion of admissibility is to require only one of $\sin\theta, \cos\theta$ to be rational. Thus non-Pythagorean angles such as $30°$ and $60°$ would become admissible. But $45°$ would still not be admissible. So we may further liberalize our notion of admissible to any angle θ where at least one of $\sin\theta, \cos\theta, \tan\theta$ is rational. This last class of admissible angles is clearly a subset of the geodetic angles. It is even a proper subset because $\theta = \arcsin(\sqrt{2}/\sqrt{5})$ is geodetic but $\sin\theta, \cos\theta, \tan\theta$ are not rational.

The definition of geodetic angles is from Conway et al [7]. They also call an angle **mixed geodetic** if it is a rational linear combination of geodetic angles. For instance, $45°$ is geodetic but $75° = (30+45)°$ and $1° = (1/30)30°$ are mixed geodetic. The set of mixed geodetic angles forms a vector space over $\mathbb{Q}$ They [7] constructed a basis for this vector space. From this, we achieve a classification of all rational linear relations among mixed geodetic angles. An elegant notation comes out of this theory: for any real number $x \in \mathbb{R}$, define the **angle** $\angle x$ as follows:

$$\angle x = \begin{cases} n\pi/2 = n90° & \text{if } x = n \in \mathbb{Z} \\ \angle n + \arcsin(\sqrt{x}) & \text{if } x = n + r, 0 < r < 1, n \in \mathbb{Z}. \end{cases} \tag{8}$$

Observe that if x is rational, then $\angle x$ is a geodetic angle. For instance, $\angle 0 = 0°, \angle(1/4) = 30°, \angle(1/2) = 45°, \angle(3/4) = 60°, \angle 1 = 90°$.

Commensurable Angles. Let us first show that *commensurable angles have algebraic cosines.* I.e., angles that are admissible under (B) is admissible under (D). To see this, we use **Chebyshev polynomials** of the first kind. These are integer polynomials $T_n(X)$ of degree n ($n \geq 0$) which are defined recursively by

$$T_{n+1}(X) = 2XT_n(X) - T_{n-1}(X),$$

starting from $T_0(X) = 1$ and $T_1(X) = X$. The next few polynomials are

$$T_2(X) = 2X^2 - 1, \quad T_3(X) = X(4X^2 - 3), \quad T_4(X) = 8X^4 - 8X^2 + 1,$$
$$T_5(X) = X(16X^4 - 20X^2 + 5).$$

For any θ, we have

$$T_n(\cos\theta) = \cos n\theta. \tag{9}$$

If θ is commensurable, then $n\theta$ is a multiple of π for some n. From (9), we conclude that $T_n(\cos\theta) = \cos n\theta = \pm 1$. This relation proves that $\cos\theta$ is algebraic, as we claimed. But in fact, Jahnel [17] show the stronger statement that $2\cos\theta$ is an algebraic integer (Exercise).

Commensurable angles are clearly closed under addition, subtraction and multiplication by rationals. We claim that angles with algebraic cosines have the same closure properties. Closure under addition and subtraction from the usual formulas for $\cos(\theta + \theta')$ and $\cos(\theta - \theta')$. If $p, q \in \mathbb{Z}$, $q \neq 0$, and $\cos\theta$ is algebraic, then the equation $T_q(\cos(\theta/q)) = \cos\theta$ shows that $\cos(\theta/q)$ is algebraic; also $T_p(\cos(\theta/q)) = \cos((p/q)\theta)$ shows that $\cos((p/q)\theta)$ is algebraic.

Lemma 4 (Jahnel). *The following are equivalent:*
(i) The angle θ is commensurable and $\cos\theta$ is rational.
(ii) $2\cos\theta \in \{0, \pm 1, \pm 2\}$.

Proof. Clearly, (ii) implies (i). To show that (i) implies (ii), note that

$$2\cos 2\theta = (2\cos\theta)^2 - 2 \tag{10}$$

is just the identity $\cos 2\theta = 2\cos^2\theta - 1$ in disguise. Assume $2\cos\theta = m/n$ where $m, n \in \mathbb{Z}$, $\texttt{GCD}(m, n) = 1$ and $n \neq 0$. Substituting into (10), we get $2\cos 2\theta = (m^2 - 2n^2)/n^2$. Note that $\texttt{GCD}(m^2 - 2n^2, n^2) = 1$ for, if $p > 1$ and $p|(m^2 - 2n^2)$ and $p|n^2$, then $p|n$ and hence $p|m$, contradiction. Repeated application of the transformation $2\cos\theta \mapsto 2\cos 2\theta$ gives the sequence

$$2\cos 2\theta, 2\cos 4\theta, 2\cos 8\theta, \ldots, 2\cos 2k\theta, \ldots, \qquad (k \geq 1).$$

Moreover, each $2\cos 2k\theta = f_k(m, n)/n^{2^k}$, for some integer function $f_k(m, n)$ that is relatively prime to n. But, by the commensurablity of θ, there is some ℓ, j when $2\ell\theta = 2j\pi$ and hence $\cos 2\ell\theta = \cos 2j\pi = 1$. If

$n \neq \pm 1$, then the denominator of $\cos 2k\theta$ is growing arbitrarily large as $k \to \infty$. This contradicts the fact that $\cos 2\ell\theta = 1$ for some ℓ. Hence $n = \pm 1$. Therefore $2\cos\theta = \pm m$. This implies $m = 0, \pm 1, \pm 2$. Hence $\cos\theta \in \{0, \pm\frac{1}{2}, \pm 1\}$. **Q.E.D.**

Corollary 5. *Let θ be Pythogorean. Then θ is commensurable iff $\theta \notin \{90°, 0°, 180°, 60°, 120°\}$.*

Proof. Just note that the condition Lemma 4(ii) is equivalent to $\theta \notin \{90°, 0°, 180°, 60°, 120°\}$. **Q.E.D.**

For instance, this shows that $\arcsin(3/5)$ and $\arcsin(5/13)$ are incommensurable.

Theorem 6. *Let θ be incommensurable. Then*

$$\mathbb{Z}\theta := \{(n\theta) \bmod 2\pi : n \in \mathbb{Z}\}$$

is a dense subset of $[0, 2\pi)$.

Proof. Given $\phi \in [0, 2\pi)$ and $N \in \mathbb{N}$, we must find $n \in \mathbb{Z}$ such that

$$|\phi - (n\theta) \bmod 2\pi| < 2\pi/N. \tag{11}$$

By the pigeonhole principle, one of the following N intervals

$$[0, 2\pi/N), [2\pi/N, 4\pi/N), [4\pi/N, 6\pi/N), \ldots, [2(N-1)\pi/N, 2\pi)$$

must contain two members $a < b$ from the set $\{(i\theta) \bmod 2\pi : i = 0, \ldots, N\}$ which has $N+1$ members. Then $0 < b - a < 2\pi/N$. We claim that $b - a \in \mathbb{Z}\theta$. To see this, let $a = i_a\theta - 2j_a\pi$ and $b = i_b\theta - 2j_b\pi$ where $i_a, i_b = 0, \ldots, N$ and $j_a, j_b \in \mathbb{Z}$. Then $b - a = (i_b - i_a)\theta - 2(j_b - j_a)\pi \in \mathbb{Z}\theta$. Let $M = \lceil 2\pi/(b-a) \rceil$. Clearly, $M > N$. So there is a unique $k \in \{0, 1, \ldots, M-1\}$ such that $k(b-a) \leq \phi < (k+1)(b-a)$. Therefore, we may choose $n = k(i_b - i_a)$ to achieve the conclusion in (11). **Q.E.D.**

Thus, we can use the set $\mathbb{Z}\theta$ to arbitrarily approximate any desired angle ϕ to within any desired error bound.

Pythagorean Equation. In the following, we will study Pythagorean angles. Note that rotation by Pythagorean angles preserves the rationality of input points. This problem was studied by Canny, Donald and Ressler [3].

The Pythagorean equation is the equation

$$x^2 + y^2 = z^2 \tag{12}$$

to be solved in integers. Our goal is to characterize all solutions (x, y, z) to this equation. Since the signs of x, y, z do not matter, let us assume they are non-negative: $x \geq 0, y \geq 0, z \geq 0$. As the solutions (x, y, z) where $xyz = 0$ are easily characterized, we further assume positivity: $x \geq 1, y \geq 1, z \geq 1$. A solution (x, y, z) such that $\texttt{GCD}(x, y, z) = 1$ is said to be **primitive**. We may restrict attention to primitive solutions.

It is now easy to see that we have a bijection between positive primitive solutions (x, y, z) to (12) and Pythagorean angles θ $(0 < \theta < \pi/2)$, as given by

$$(\sin\theta, \cos\theta) = (x/z, y/z). \tag{13}$$

Note that if d divides any two values in x, y, z then it divides the third. Hence primitivity is the same as saying that *any* two components in (x, y, z) are co-prime. In particular, there is at most one even number among x, y, z.

Claim: *Exactly one of x or y is even.* Note that the square of an odd number is congruent to 1 modulo 8, and the square of an even number is congruent to 0 modulo 4. If both x and y are odd, then $x^2 + y^2$ is congruent to 2 modulo 8. But this cannot represent the square of any number. This proves our claim.

By symmetry, we assume solutions (x, y, z) in which y is even but x, z are odd. Since $z + x$ and $z - x$ are even, let $m' := (z + x)/2, n' := (z - x)/2$. Note that $\texttt{GCD}(m', n') = 1$ because if k divides m' and n' then k divides $m' + n' = z$ and k divides $m' - n' = x$, we get a contradiction. If $y = 2y'$ then $4y'^2 = (z + x)(z - x)$

and hence $y'^2 = m'n'$. This implies $m' = m^2$ and $n' = n^2$ for some m, n and $\texttt{GCD}(m, n) = 1$. We conclude the solution (x, y, z) we seek must have the form $(x, y, z) = (m' - n', 2y', m' + n')$, or

$$(x, y, z) = (m^2 - n^2, 2mn, m^2 + n^2) \tag{14}$$

for some co-prime $m, n \in \mathbb{N}$ with $m \geq n$. Conversely, it is easy to see that every co-prime $m, n \in \mathbb{N}$ gives rise to a primitive solution of the desired form. Thus all positive, primitive solutions to the Pythagorean equation is completely characterized by (14).

Pythagorean Angle Problem. Define the **length** of a rational number p/q in reduced form to be $\lg |q|$ (roughly the bit length of q in binary notation). Canny, Donald and Ressler [3] posed and solved the following problem: *on input θ, ε, a rational number $0 < s \leq 1$ such that $|\arcsin(s) - \theta| < \varepsilon$ and the length of s is as short as possible.* In practice, we might be happy if the length of s is within $O(1)$ bits of the shortest.

Since the angle θ may be hard to specify exactly in the input, we prefer the alternative specification: *given an input interval $I \subseteq [0, 1]$, to find a rational number $0 < s \leq 1$ such that s lies in the interval I, and the length of s is shortest.* We expect that $I = [\underline{s}, \overline{s}]$ will have rational endpoints. Note that the set of angles with rational sine does not include all angles with rational cosines. So an alternative formulation is: given $I \subseteq [0, 1]$, to to compute a rational number $0 < r < 1$ such that $r \in I$ or $\sqrt{1 - r^2} \in I$, and the length of r is shortest.

We now present one of their solutions: ...

In 3-dimensions, the generalization of the rotation matrix $R(\theta)$ is an orthonormal 3×3 matrix. Milenkovic and Milenkovic [22] considered the 3-dimensional analogue of Pythagorean angles, i.e., orthonormal matrices whose entries are rational numbers. They also show that such matrices are dense among orthonormal ones.

Exercises

Exercise 5.1: Given θ, suppose we computed some approximations $s \approx \sin\theta$ and $c \approx \cos\theta$. In particular, assume $s^2 + c^2 \neq 1$. The rotation of a point p by angle θ is given by $R \cdot p$ where $R = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$, and treating p as a vector and $R \cdot p$ as matrix-vector multiplication. Clearly, $\det R$ may not be 1 in general. Discuss real applications where this is a problem, and where this is not a problem. HINT: problems arise when you need to compute intersections of transformed and untransformed objects. ♢

Exercise 5.2: Let $\theta = \arcsin(3/5)$. We want to use the set $\mathbb{Z}\theta$ for approximation of angles. Given any real interval $I = [a, b] \subseteq [0, 2\pi)$, give an algorithm to find the smallest $|n|$ such $(n\theta) \,\mathbf{mod}\, 2\pi$ lies in I. ♢

Exercise 5.3: Improve Theorem 6.
(i) Can the $|n|$ in (11) be upper bounded as a function of N?
(ii) Can you show that $\mathbb{N}\theta := \{(n\theta) \,\mathbf{mod}\, 2\pi : n \in \mathbb{N}\}$ is dense? ♢

Exercise 5.4: Show that (B) and (C) are not comparable: there is a geodetic angle that is incommensurable, and there is a commensurable angle that is not geodetic. ♢

Exercise 5.5: [12, p.237] Show that $\arccos(1/3)$ is incommensurable. NOTE: This angle is about $70°31'44''$, the dihedral angle of a regular tetrahedron. ♢

Exercise 5.6: (Jörg Jahnel) Let θ be commensurable.
(i) $2\cos\theta$ is an algebraic integer, and all its conjugates have absolute value ≤ 2. HINT: This statement is a generalization of Lemma 4. Indeed, the proof imitates the proof of Lemma 4, except that we must decompose $2\cos\alpha$ into a product of distinct ideals, and argue that none of the ideals has negative exponent.
(ii) If x is a quadratic integer and $|x| < 2$ and $|\overline{x}| < 2$ then $x \in \{\pm\sqrt{2}, \pm\sqrt{3}, \pm\frac{1}{2} \pm \sqrt{5}\}$.
(iii) If $\cos\theta$ is a quadratic irrationality, then $2\cos\theta \in \{\pm\sqrt{2}, \pm\sqrt{3}, \pm\frac{1}{2} \pm \sqrt{5}\}$. Morover, all these values are achievable:

$$\begin{aligned}(\theta, 2\cos\theta) = &(45°, \sqrt{2}), (135°, -\sqrt{2}), (30°, \sqrt{3}),\\ &(150°, \sqrt{3}), (36°, \frac{1}{2} + \frac{1}{2}\sqrt{5}), (144°, -\frac{1}{2} - \frac{1}{2}\sqrt{5}),\\ &(72°, -\frac{1}{2} + \frac{1}{2}\sqrt{5}), (108°, \frac{1}{2} - \frac{1}{2}\sqrt{5}).\end{aligned}$$

(iv) If $2\cos\theta$ is a cubic irrationality then its minimal polynomial $x^3 + ax^2 + bx + c$ satisfies $|a| < 6, |b| < 12, |c| < 8$. Among these, there are exactly 4 irreducible polynomials: $p_1 : x^3 - x^2 - 2x + 1$, $p_2 : x^3 + x^2 - 2x - 1$, $p_3 : x^3 - 3x + 1$, $p_4 : x^3 - 3x - 1$.
(v) For each $i = 1, \ldots, 4$, determine the angles $\alpha_i, \beta_i, \gamma_i$ such that $2\cos\alpha_i, 2\cos\beta_i, 2\cos\gamma_i$ are the zeros of p_i. ♢

End Exercises

§6. Additional Notes

A clear expression of the goal of achieving robustness using fixed-precision arithmetic is stated in Fortune's paper [10].

"The challenge is to develop floating-point analyses of a wide variety of geometric algorithms. Such analyses should lead to efficient, reliable, easily-implemented geometric algorithms.

A general theory, easily applicable to an arbitrary algorithm, appears to be a distant goal. An important component of a general theory would be a way of transforming exact predicates into approximate predicates. Such a transformation must have two properties: first, the truth of approximate predicates has to be testable using floating-point arithmetic; second reasoning usng approximate predicates has to adequately mimic reasoning using exact predicates. An example is the transformation of the 2-d incircle predicate into the D predicate. This transformation satisfies the first property, since incircle can be used to test the D predicate; it does not entirely satisfy the second property.

Many computational geometers attacking robust geometric computation in the early 1990's assumed that the solution must come from fixed-precision floating point computational models. This view is well-summed up in Fortune's introduction: "Floating point arithmetic has numerous engineering advantages: it is well-supported... the Challenge is to demonstrate that a reliable implementation can result from the use of f.p. arithmetic."

Lindemann's theorem is more general than stated in the text. It says that the following expression can never hold:

$$c_1 e^{a_1} + c_2 e^{a_2} + \cdots + c_n e^{a_n} = 0 \tag{15}$$

where $n \geq 1$, the c_i's and a_i's are algebraic, the a_i's distinct and the c_i's are non-zero. This implies Hermite's theorem that e is transcendental: if e is algebraic, then there is an integer polynomial $A(X)$ such that $A(e) = 0$. But $A(e)$ has the form (15), contradiction. This also implies that π is transcendental because of Euler's identity $e^{\mathbf{i}\pi} + 1 = 0$ has the form (15). Finally, this result about π is proof that the ancient quest for squaring the circle is impossible. The quest ask for a ruler-and-compass construction of a square whose area is that of a circle. Without loss of generality, let the circle have unit radius and hence area π. If we can square the circle, we would have constructed a square whose side length is $\sqrt{\pi}$. But it is well-know that all such lengths arising from ruler-and-compass constructions are "constructible reals" (and hence algebraic). Thus, as a length of a square, $\sqrt{\pi}$ must be algebraic. Thus π is algebraic, contradiction.

For a collection of papers on implementation issues, we refer to a special 2000 issue of Algorithmica [9].

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983. Translated from German by Jon Rokne.

[2] G. Bohlender, C. Ullrich, J. W. von Gudenberg, and L. B. Rall. *Pascal-SC*, volume 17 of *Perspectives in Computing*. Academic Press, Boston-San Diego-New York, 1990.

[3] J. F. Canny, B. Donald, and E. Ressler. A rational rotation method for robust geometric algorithms. *Proc. 8th ACM Symp. on Computational Geometry*, pages 251–160, 1992. Berlin.

[4] E.-C. Chang, S. W. Choi, D. Kwon, H. Park, and C. Yap. Shortest paths for disc obstacles is computable. *Int'l. J. Comput. Geometry and Appl.*, 16(5-6):567–590, 2006. Special Issue of IJCGA on Geometric Constraints. (Eds. X.S. Gao and D. Michelucci).

[5] C. Clenshaw, F. Olver, and P. Turner. Level-index arithmetic: an introductory survey. In P. Turner, editor, *Numerical Analysis and Parallel Processing*, pages 95–168. Springer-Verlag, 1987. Lecture Notes in Mathematics, No.1397.

[6] C. Clenshaw and P. Turner. The symmetric level-index system. *IMA J. Num. Anal.*, 8:517–526, 1988.

[7] J. H. Conway, C. Radin, and L. Sadun. On angles whose squared trigonometric functions are rational. *Discrete and Computational Geometry*, 22:321–332, 1999.

[8] S. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. Ph.D. thesis, New York University, 1999.

[9] S. Fortune. Editorial: Special issue on implementation of geometric algorithms, 2000.

[10] S. J. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1):193–213, 1995.

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

[12] R. Hartshorne. *Geometry: Euclid and Beyond*. Springer, 2000.

[13] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[14] Holt, Matthews, Rosselet, and Cordy. *The Turing Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[15] T. Hull, A. Abraham, M. Cohen, A. Curley, C. Hall, D. Penny, and J. Sawchuk. Numerical Turing. *ACM SIGNUM newsletter*, 20(3):26–34, July, 1985.

[16] T. Hull, M. Cohen, J. Sawchuk, and D. Wortman. Exception handling in scientific computing. *ACM Trans. on Math. Software*, 14(3):201–217, 1988.

[17] J. Jahnel. When does the (co)sine of a rational angle give a rational number?, 2004. From `http://http://www.uni-math.gwdg.de/jahnel/linkstopaperse.html`.

[18] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Boston, 2nd edition edition, 1981.

[19] U. Kulisch, R. Lohner, and A. Facius, editors. *Perspectives on Enclosure Methods*. Springer-Verlag, Vienna, 2001.

[20] D. Lozier and F. Olver. Closure and precision in level-index arithmetic. *SIAM J. Numer. Anal.*, 29:1295–1302, 1990.

[21] S. Matsui and M. Iri. An overflow/underflow-free floating-point representation of numbers. *J. Inform. Process*, 4(3):123–133, 1981.

[22] V. Milenkovic and V. Milenkovic. Rational orthogonal approximations to orthogonal matrices. *Proc. 5th Canadian Conference on Computational Geometry*, pages 485–490, 1993. Waterloo.

[23] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[24] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *Proc. 3rd ACM Sympos. Comput. Geom.*, pages 119–125, 1987.

[25] H. Ratschek and J. Rokne. Exact and optimal convex hulls in 2d. *Int'l. J. Comput. Geometry and Appl.*, 10(2):109–130, 2000.

Lecture 5
GEOMETRIC APPROACHES

We look at some geometric approaches to nonrobustness. An intriguing idea here is the notion of "fixed precision geometry". After all, if nonrobustness is a geometric phenomenon, it makes sense to modify the geometry to reflect the fixed precision we find in numbers. There are many ways to create such ersatz geometries*, which in various ways try to recover the basic properties of standard Euclidean geometry. We shall illustrate several such geometries through a discussion of "fixed precision lines".*

§1. What is a Finite Precision Line?

Since qualitative errors is geometric in nature, we can attempt to modify the underlying geometry to be achieve robust behavior. E.g., with finite precision arithmetic, it seems reasonable to replace standard Euclidean geometry by some finite precision geometry. We have already met such a geometry in Chapter 1: we interpreted the standard epsilon-tweaking trick as manipulating new kinds of "points" and "lines", namely, fat points and fat lines.

But there are many potential candidates for finite-precision geometries, even for the simple concept of lines. We illustrate with some common answers [24]. to the question "what is a finite precision line?" Each answer can be viewed as representative of a general approach to finite precision geometry.

- Interval Geometry: We use the term "interval geometry" to refer any approach which replaces standard geometric objects by "fat geometric objects". A "fat point" can be a disc, but it can be generally be any simply connected bounded region. A "fat line" can be any region bounded by two infinite polygonal paths (probably with other properties as well). This is illustrated in Figure 1(a). This is the geometric analogue of interval arithmetic, and has been proposed in many different settings, e.g., [18, 12]. Guibas and Stolfi [6] investigated a form of such geometry ("ε-geometry") by focusing on geometric ε-predicates that can have yes/no/uncertain values. Interval geometry is also attractive from another viewpoint: in the field of mechanical design and manufacture, the tolerancing of geometric objects [23, 13] is a critical problem. The computational aspects of this field is being pursued under the name of computational metrology [9, 25, 14].

- Topologically Consistent Geometry: A line can be distorted into a curve, which in practice would be a polyline. This is illustrated in Figure 1(b). The goal of distortion is to preserve some desired topological property of the original lines. For instance, polylines arising in when computing the arrangements of lines aim to avoid introducing extraneous intersections. Consider the following grid model studied by Greene and Yao [4]. Let G

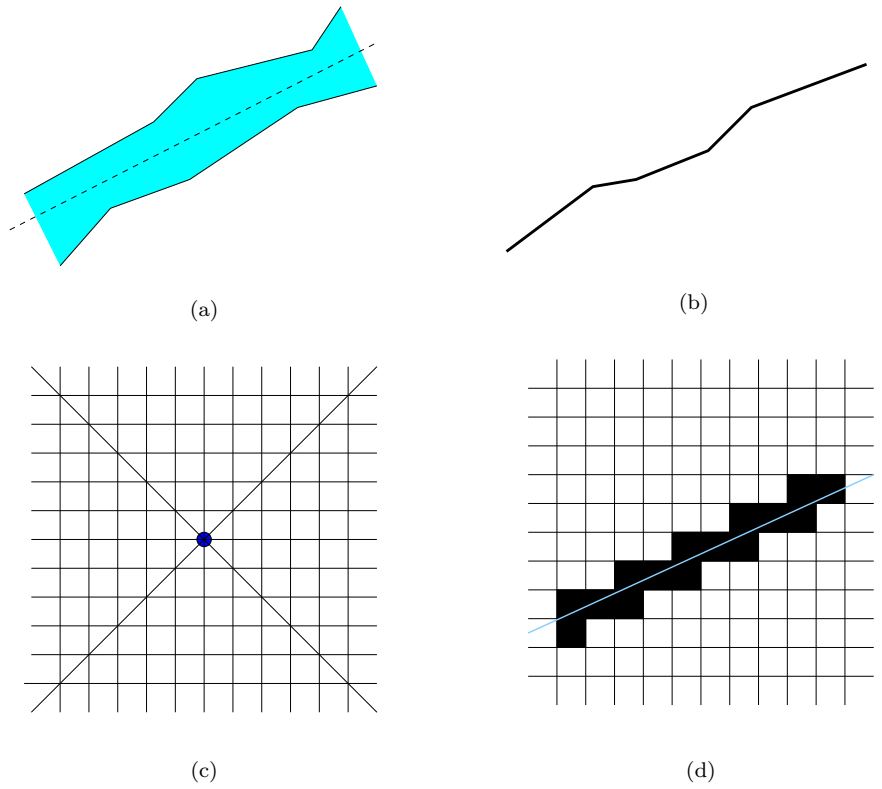


Figure 1: Finite Precision Line Geometry

be a grid, typically $G = \mathbb{Z} \times \mathbb{Z}$. Assume the input are segments whose end points lie on this grid. For each input pair (s, s') of intersecting line segments, we move the intersecting point to a close grid point $p \in G$. The segment s, s' are now converted into polysegments with the same end points as before, but now passing through p. Greene and Yao require the transition from s into a polysegment not to cross any gridpoint, leading to polysegments with continued fraction type properties.

Sugihara and Iri [21, 22] propose topological consistency as a fundamental principle for achieving robustness.

- Rounded Parameter Geometry: Assume our geometric objects live in some parametric space. For instance, lines can be represented in the form $L = \mathsf{Line}(a, b, c)$ as in Chapter 1. This corresponds to the Euclidean line with equation $aX + bY + c = 0$. We want to round the line parameters (a, b, c) to some (a', b', c') where a', b', c' comes from some finite set of representable values (say L-bit integers). We call $\mathsf{Line}(a', b', c')$ a **rounded line**. Sugihara [19] [CHECK!] discusses several approaches to rounding lines. For instance, we might like to restrict $|a'|, |b'|, |c'|$ to lie in the range $[0, 2^L)$. But in fact, it turns out that we get a "nicer class" of such rounded lines is obtained if we let $|c'|$ lie in the slightly larger range $[0, 4^L)$. Such a set of lines is illustrated in Figure 1(c).

- Discretized Geometry: This is well-known in computer graphics: a line is a suitable set of pixels on a finite screen. In contrast to the previous rounding in parameter space, in discretized geometry we discretize the underlying Euclidean space where the geometric object lives. Figure 1(d)

illustrates a discretized line represented as a set of darkened pixels. This is called a Bresenham line in computer graphics. In the field of image processing, an area called "digital geometry" investigates properties of such discretized representation of objects.

Software and Language Support. Regardless of the approach to robustness, it is an important to provide software support for the particular approach. Thus, the development of `Pascal SC` as an extension of `Pascal` for scientific computation supports robust arithmetic with facilities for rounding, overloading, dynamic arrays and modules [1]. These facilities have been extended and made available in other languages: `PASCAL-XSC`, `C-XSC` and `ACRITH-XSC`. The language **Numerical Turing** [8, 10, 11] has a concept of "precision block" whereby the computation within such a block can be repeated with increasing precision until some objective is satisfied. While there are well-known numerical libraries such as `LAPACK`, newer efforts such as `LEDA` and `CGAL` aim to provide robust primitives and basic algorithms and data structures as building blocks for geometric applications. Important primitives that ought to be supported by languages or libraries include scalar product [16] and accurate determinant evaluation. More generally, we could extend programming languages to support geometric objects and their manipulation [17]. In the area of programming languages, the proper treatment of arithmetic exceptions [2] ought to become a standard part of high-level language support.

[NOTE: We should considerably expand the details in the four bullets.]

§2. Robust Line Segment Intersection

We consider another fundamental problem in computational geometry: given a set S of line segments in the plane, we want to compute their arrangement. This is a very basic problem that arises in VLSI circuit design, hidden line elimination, clipping and windowing, physical simulations, etc. The arrangement of S is a **1-skeleton** (or network) determined by the line segments and their intersections: a 1-skeleton is basically a graph that is embedded in the plane: its vertices are the endpoints of segments and the intersection points. Its edges are the open portions of segments bounded by two vertices. This is illustrated in Figure 2.

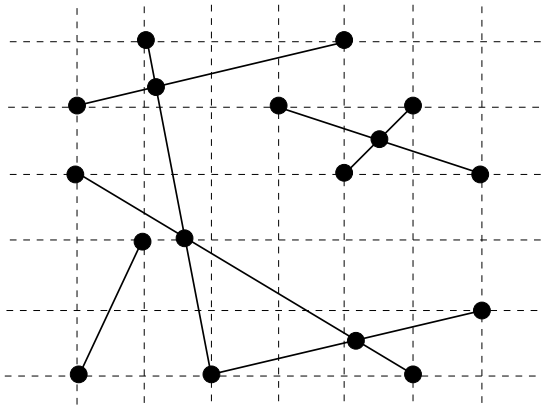


Figure 2: 1-Skeleton of a set of line segments

There are several algorithms for this problem. The asyptotically best algorithm is from Chazelle and Edelsbrunner, with running time $O(k+n\log n)$ where

k is the number of intersection points and n is the number of segments. Here k is the output parameter and ranges from 0 to n^2. More practical algorithms based on randomization have been given Clarkson and also Mulmuley. Algorithms that takes into account robustness issues have been studied by Greene and Yao [4], Milenkovic [15], and Sugihara [20]. The approach of Hobby [7], Guibas and Marimont [5] and Goodrich, et al [3] are based on the concept of "snap rounding", which we will treat in detail.

Recently, Preparata et al considered another issue, namely modifying the primitives using in these algorithms so that the algebraic degree is as small as possible. Although this clearly can help FP computations, it is useful even in AP computation.

Bentley-Ottman Algorithm. The classic algorithm for this problem is from Bentley and Ottmann. This algorithm takes time $O((n+k)\log n)$, and is quite easy to describe. Later we will implement Hobby's snap rounding using this algorithm as its basis [7].

Let us briefly review the Bentley-Ottmann algorithm. We assume a vertical sweepline L with equation $x = x_0$. The line sweeps from left to right (so x_0 is increasing). We have priority queue Q, the **event queue**, initialized to store the end points of each input segment. Each point in Q represents an **event**. An point (x, y) in Q has priority x. Initially, all the points are either **start events** or **stop events**, corresponding to the leftmost or rightmost endpoints of a segment. As L sweeps from left to right, we pull events from Q. At any moment, we keep track of the set of segments that intersect L. These segments are sorted by their the y-coordinate of their intersection with L. We store them in a balanced binary tree T. Moreover, for each pair of adjacent segments in T, if they intersect at a point q to the right of L, then we insert q into Q. Such a q represents the third kind of event, an **intersection event**.

Here is how we update T with each event point. If stop event, we delete the corresponding segment from T If a start event, we insert the corresponding segment into T If intersection event, we exchange the relative order of the two intersecting segments in T.

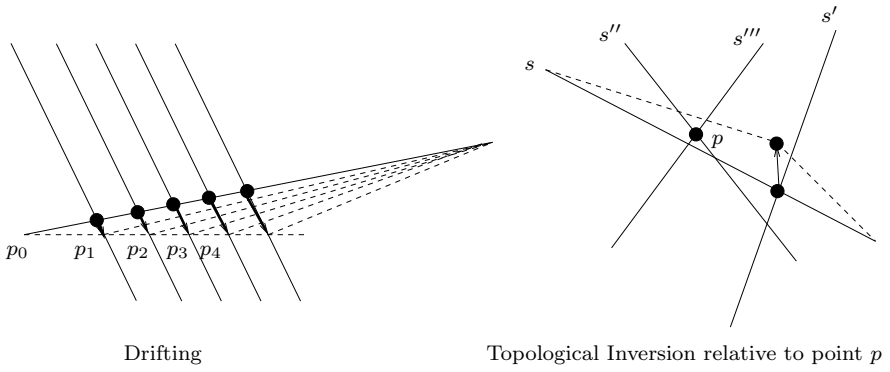
Yao-Greene Grid Model. We consider the following finite-precision model of plane geometry: let $G_2 = \mathbb{Z} \times \mathbb{Z}$ be the **(unit) grid**. All representable points must come from G_2. Suppose s, s' are two representable line segments (i.e., their endpoints are in G_2). If they intersect at a point p, this point is generally not representable. So we must round it to some nearby grid point p'. If $s = [a, b]$, then p' no longer lie on $[a, b]$ – failing a fundamental identity we discussed in Lecture 1:

$$OnLine(Intersect(s, s'), s)$$

is identically true whenever $Intersect(s, s')$ is defined. To restore this identity, we now replace line segments with **polysegments** which is just a finite polygonal path $[a_1, a_2, \ldots, a_n]$. Thus s is now the polysegment $[a, p', b]$ and there is a similar modification for s'. In general, for a polysegment s, we can "snap" a point p in s to a grid point p' and change s accordingly.

This defines our basic model, first studied by Greene and Yao for this problem. What are the issues?

- Unbounded Change: when an intersection point is snapped, other intersection points may move as a result. It is not hard to see that this movement may be unbounded even if the snap distance is bounded.



Drifting Topological Inversion relative to point p

Figure 3: Issues in Snap Rounding

- Cascaded and New Intersections: when we snap a point, we may generate new intersections. If we now snap these new intersections points, we may obtain other new ones. Thus we have a cascading effect. How often do we need to do this? Does this terminate?
- Drifting: Each time we modify a polysegment, the result may drift further and further away from its original position. Is there a bound on this drift? See Figure 3(a) for an illustration of drifting.
- Topological changes: See Figure 3(b) illustrates the possibility of a polysegment moving past an intersection point. We call this an "inversion". A milder version of topological change is to have two points collapse into one (this is called "degeneration").
- Braiding: Two segments intersect in a connected component. If our polysegments intersect is several connected components, we call this "braiding". Even if there were no inversion, braiding can happen.

This list of problems seems formidable [4]. So it is somewhat surprising to see the simple solution described next.

What is an acceptable topological change? Topological change is inevitable in finite precision computation. But we distinguish between **degeneration** (which is acceptable) versus **inversion** (which is not). For instance, if an edge in a planar arrangement of segments is collapsed to a point and it does not affect other relations (except for the obvious ones forced by this collapse), this is an acceptable "degeneration". On the other hand, if a vertex inside a cell is modify to lie outside the cell, this is an unacceptable "inversion". In the presence of degeneration, there is a trivial solution for any segment arrangement: collapse everything to a point. So, our requirements ought to be supplemented by some metric properties such as a bound of the amount of degeneration. Greene and Yao imposed a very strong one, that the deformation of a segment must not cross any grid point. The resulting polysegments has nice properties related to the continued fraction process. We will study another approach from Hobby.

Snap Rounding. It is convenient to introduce the half-open interval $R_1 = (-\frac{1}{2}, \frac{1}{2}]$. Also let $R_2 = R_1 \times R_1$. Call R_1 and R_2 the **rounding interval** or **square**, respectively. It defines the rounding model for us: a number x is rounded to the unique integer $(x + R_1) \cap \mathbb{Z}$, sometimes denoted $\lfloor x \rceil$. In other

words, we round up in case of ties: 2.5 rounds to 3. Similarly, a point p is rounded to the unique point $(p + R_2) \cap G_2$, also denoted $\lfloor p \rceil$. Note that "+" here is the Minkowski sum.

We can view rounding from the view point of the grid: let $\overline{R}_1$ denote the interval $[-\frac{1}{2}, \frac{1}{2})$ (thus, $\overline{R}_1 = -R_1$). If $i \in \mathbb{Z}$, then $\lfloor x \rceil = i$ iff $x \in i + \overline{R}_1$. Similarly, we may define $\overline{R}_2 = \overline{R}_1 \times \overline{R}_1$ and define the square of influence of a grid point $g \in G_2$ to be $g + \overline{R}_2$.

Let S be a set of segments, where each pair of segments intersect transversally (we do not allow overlap). Define H to be the set comprising all the end points of segments in S and also all intersection points. So H is finite. Let the rounded point set $\lfloor H \rceil$: these are called the "hot points".

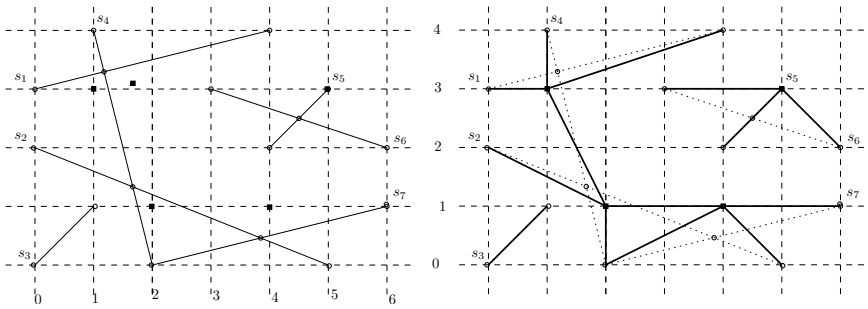


Figure 4: Snap Rounding Based on Hot Points

A polysegment s is said to be **near** a point p if $p + \overline{R}_2 \cap s$ is non-empty. If a polysegment s that passes near a hot point p, we snap s towards p. More precisely, If $q \in p + \overline{R}_2 \cap s$, then we snap q to p (recall the definition of this snap operation above). It is not hard to see that this does not depend on our choice of q. But in the sequel, we will assume that q is the point in s closest to p. The number of vertices in s increases by 1.

Hobby's Theorem on Snap Rounding. Given a segment $s \subseteq \mathbb{R}^2$, define the map $D_T(s)$ to comprise all the segments $[a, b]$ such that there exists $[a', b'] \subseteq s$ such that

$$\lfloor H \rceil \cap ([[a', b'] + R_2) = \{a, b\}.$$

If $[a', b']$ is a maximal segment with this property, we call $[a', b']$ the "preimage" of $[a, b]$. Hence, each segment in $D_T(s)$ has at least one preimage in s. Below we will prove that these preimages are very well-behaved (in particular, there is a unique preimage for each segment in $D_T(s)$).

Lemma 1 *For any segment s there is a direction $(1, \delta)$ (where $\delta = \pm 1$) such that no two points in $G_2 \cap (s + R_2)$ have the same $x + \delta y$ value.*

Proof. If s has positive slope, let $\delta = +1$ else $\delta = -1$. Let $(x_i, y_i) \in s$ for $i = 1, 2$, and $(\lfloor x_i \rceil, \lfloor y_i \rceil) \in (s + R_2) \cap G_2$. Then

$$\lfloor x_1 \rceil + \delta \lfloor y_1 \rceil \neq \lfloor x_2 \rceil + \delta \lfloor y_2 \rceil$$

unless the two rounded points are identical. To see this, suppose the slope is positive and $\lfloor x_1 \rceil + \lfloor y_1 \rceil = c$ for some c. If $\lfloor x_2 \rceil + \lfloor y_2 \rceil = c$, then it is easy to see that they must be diagonal points of a grid square. Indeed, they must be the NW and SE corners of some grid square. But this is impossible. **Q.E.D.**

A collection S of segments are **essentially disjoint** if any pair are disjoint or intersect at their end points only. We also write $D_T(S)$ for the set union of $D_T(s)$, ranging over all $s \in S$.

Lemma 2 *If S have endpoints in T and are essentially disjoint, then the set $D_T(S)$ are also essentially disjoint.*

Proof. Let $s \in S$. The previous lemma ensures that there is a coordinate system

$$(\xi, \eta) = \left(\frac{x + \delta y}{\sqrt{2}}, \frac{-\delta x + y}{\sqrt{2}} \right)$$

such that the endpoints of segments in $D_T(s)$ all have all different ξ coordinates. The transformation $(x, y) \mapsto (\xi, \eta)$ is given by

$$\begin{pmatrix} \xi \\ \eta \end{pmatrix} = M \begin{pmatrix} x \\ y \end{pmatrix}, \qquad \text{where} \quad M = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & \delta \\ -\delta & 1 \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Q.E.D.

The transformation is a rotation, *i.e.*, $\det M = 1$ and the inverse M^{-1} is given by its transpose M^T. E.g., if $\delta = 1$, the points $(\pm 1, 1)$ map to $(1/\sqrt{2}, 0)$ and $(0, 1/\sqrt{2})$.

Lemma 3 *The set of these preimages forms a cover of s. Each segment $D_T(s)$ has a unique preimage.*

Proof. The fact that they form a cover of s follows from the fact that if any maximal open subsegment $(a, b) \subseteq s$ that is dijoint from any preimages, then there must be a preimage of the form $[b, c] \subseteq s$ (for some c). This implies that $[a, c]$ is contained in a preimage, contradicting the assumption that $[b, c]$ is maximal. Uniqueness is trivial by the maximality requirement. **Q.E.D.**

Lemma 4 *If the endpoints in $D_T(S)$ maps to $q_1 = (\xi_1, \eta_1), \ldots, q_m = (\xi_m, \eta_m)$, and $\xi_1 < \xi_2 < \cdots < \xi_m$, then they describe a monotonic piecewise linear function F on the interval $[\xi_1, \xi_m]$ given by*

$$F(\xi) = \frac{\eta_i(\xi_{i+1} - \xi) + \eta_{i+1}(\xi - \xi_i)}{\xi_{i+1} - \xi_i}$$

when $\xi_i \leq \xi \leq \xi_{i+1}$.

Proof. This follows from the fact that the preimages of $[q_i, q_{i+1}]$ are ordered along the segment s in the expected way. [Incomplete] **Q.E.D.**

If $s' \in S$ is a distinct segment, then a similar function F' based on $D_T(s')$ can be defined.

Note that the slope of s' may be different from that of s. This will lead to different coordinate systems for F and F'. To fix this, we may first rotate the coordinate system so that s and s' have both positive slopes or both negative slopes. It is easy to see this is always possible.

These functions approximate the lines ℓ and ℓ' through s and s' (respectively).

Since s, s' are essentially disjoint, we may assume THAT IN THE (ξ, η)-coordinate system, wlog that s is below s' whenever they intersect a common vertical line $\xi = \xi_0$. So it suffices to show that

$$F(\xi) \leq F'(\xi) \tag{1}$$

for all $\xi \in \{\xi_1, \ldots, \xi_m\} \cup \{\xi'_1, \ldots, \xi'_{m'}\}$.

Parametrize the lines ℓ so that $(\xi, \ell(\xi)) \in \ell$ for all ξ. Similarly, assume $(\xi, \ell'(\xi)) \in \ell'$. Since the segment in $D_T(s) \subseteq s_j + R_2$, and similar for s', the difference $F(\xi) - \ell(\xi)$ is limited to the range of the η coordinates in R_2. This ranges over the intervals $(-\frac{1}{2}, \frac{1}{2})$ or $[-\frac{1}{2}, \frac{1}{2})$. [CHECK]

Then our assumptions that $\ell(\xi) \leq \ell'(\xi)$ for all ξ is the range of interest implies that

$$F'(\xi_i) \geq \eta_i, \qquad \text{if } \ell'(\xi_i) \geq \eta_i + \frac{1}{2}.$$

Otherwise, $\ell'(\xi_i) \in \eta_i + R_1$ and the definition of D_T forces $F'(\xi_i) \geq \eta_i$. incomplete

Similarly, we argue that $F(\xi'_j) \leq \eta'_j$. This proves equation (1).

Snap Rounding Based on Bentley-Ottmann's Algorithm. For simplicity, we may assume that the input segments are already representable.

STEP 1. We modify the Bentley-Ottman algorithm as follows. We first run the original algorithm as a "first pass". The intersection points are symbolically represented (by the pair of defining segments). Let H be the set of all endpoints and the intersection points found in this way.

STEP 2. Compute the set of hot points: $\lfloor H \rceil$.

STEP 3. Now we want to snap each segment $s \in S$ to all the hot points p that it is near to. More precisely, for each s we compute a set of pairs $\{(q_i, p_i) : i = 1, \ldots, k\}$ (for some k) such that p_i is a hot point that s is near to, and q_i is the point in s closest to p_i. Then we snap q_i to p_i, in an natural order determined by the q_i's along s. But how do we detect these p_i's? We can do this by modifying the same sweepline algorithm of Bentley-Ottman: for each hot point p, we add the four unit segments corresponding to the boundary of $p + \overline{R}_2$ to the input set. Thus the segment s is near to p iff it intersects one of these unit segments of p.

We can take advantage of the fact that these unit segments are horizontal or vertical and in special positions relative to the grid G_2. An efficient implementation (using the idea of batching, and keeping the unit segments separate from the original segments) is given in the original paper of Hobby.

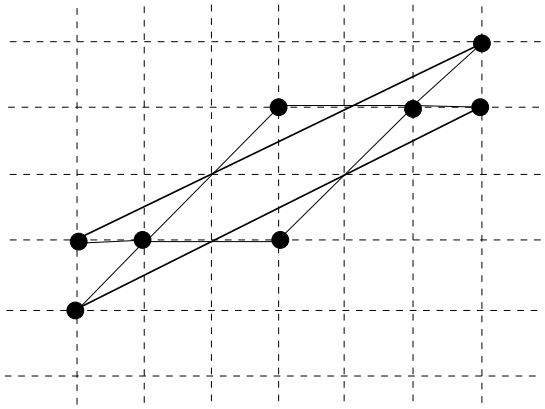


Figure 5: Braiding Behavior in Snap Rounding

Other Approaches. Yao and Greene's. Milenkovic's Sugihara's.

Exercises

Exercise 2.1:
(i) Complete the proof that the ESSA algorithm is correct.
(ii) Describe modifications to the algorithm when exponents are limited in the range $[e_{\min}, e_{\max}]$. ♢

Exercise 2.2: Fix a precision $t \geq 1$. Consider snap rounding in the following "floating point grid" $G_2^t = F(2,t) \times F(2,t)$. Show an example that shows that the snap rounding theorem of Hobby fails in GF_2. ♢

End Exercises

References

[1] G. Bohlender, C. Ullrich, J. W. von Gudenberg, and L. B. Rall. *Pascal-SC*, volume 17 of *Perspectives in Computing*. Academic Press, Boston-San Diego-New York, 1990.

[2] S. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. Ph.d. thesis, New York University, 1999.

[3] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[4] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. *IEEE Foundations of Computer Sci.*, 27:143–152, 1986.

[5] L. Guibas and D. Marimont. Rounding arrangements dynamically. In *Proc. 11th ACM Symp. Computational Geom.*, pages 190–199, 1995.

[6] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. *ACM Symp. on Comp. Geometry*, 5:208–217, 1989.

[7] J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geometry: Theory and Appl.*, 13:199–214, 1999.

[8] Holt, Matthews, Rosselet, and Cordy. *The Turing Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[9] T. H. Hopp. Computational metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 207–217, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[10] T. Hull, A. Abraham, M. Cohen, A. Curley, C. Hall, D. Penny, and J. Sawchuk. Numerical Turing. *ACM SIGNUM newsletter*, 20(3):26–34, July, 1985.

[11] T. Hull, M. Cohen, J. Sawchuk, and D. Wortman. Exception handling in scientific computing. *ACM Trans. on Math. Software*, 14(3):201–217, 1988.

[12] D. Jackson. Boundary representation modeling with local tolerances. In *Proc. Symp. on Solid Modeling Foundations and CAD/CAM applications*, pages 247–253, 1995.

[13] N. Juster. Modeling and representation of dimensions and tolerances: a survey. *CAD*, 24(1):3–17, 1992.

[14] K. Mehlhorn, T. Shermer, and C. Yap. A complete roundness classification procedure. In *13th ACM Symp. on Comp. Geometry*, pages 129–138, 1997.

[15] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annual Symp. on Foundations of Comp.Sci.*, pages 500–506. IEEE Computer Society, 1989.

[16] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *Proc. 3rd ACM Sympos. Comput. Geom.*, pages 119–125, 1987.

[17] M. G. Segal. Programming language support for geometric computations. Report csd-90-557, Dept. Elect. Engrg. Comput. Sci., Univ. California Berkeley, Berkeley, CA, 1990.

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

[19] K. Sugihara. On finite-precision representations of geometric objects. *J. of Computer and System Sciences*, 39:236–247, 1989.

[20] K. Sugihara. An intersection algorithm based on Delaunay triangulation. *IEEE Computer Graphics Appl.*, 12(2):59–67, 1992.

[21] K. Sugihara and M. Iri. Two design principles of geometric algorithms in finite precision arithmetic. *Applied Mathematics Letters*, 2:203–206, 1989.

[22] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation – an approach to robust geometric algorithms. *Algorithmica*, 27:5–20, 2000.

[23] H. B. Voelcker. A current perspective on tolerancing and metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 49–60, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[24] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004. Expanded from 1997 version.

[25] C. K. Yap and E.-C. Chang. Issues in the metrology of geometric toleranc-ing. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robot Motion Planning and Manipulation*, pages 393–400, Wellesley, Massachusetts, 1997. A.K. Peters. 2nd Workshop on Algorithmic Foundations of Robotics (WAFR).

You might object that it would be reasonable enough for me to try to expound the differential calculus, or the theory of numbers, to you, because the view that I might find something of interest to say to you about such subjects is not prima facie *absurd; but that geometry is, after all, the business of geometers, and that I know, and you know, and I know that you know, that I am not one; and that it is useless for me to try to tell you what geometry is, because I simply do not know.*

— G.H.Hardy, in "What is Geometry?"

1925 Presidential Address to the Mathematical Association

Lecture 6
Exact Geometric Computation

We characterize the notion of "geometric computation" by pointing out some common features. Along the way, we give one answer to the age-old question: what is geometry? This analysis will lead to a general prescription for attacking nonrobustness in geometric computations.

Throughout the history of mathematics, different answers to this age-old question has been given. From intuitive popular accounts of geometry [4] to quests for the scope and place of geometry [1, 6, 1] The geometric vein is evident is almost every branch of mathematics. Although we begin with geometry, we are soon led to algebra, and throughout the development of geometry, we see a strong interplay between geometry and algebra. Geometry is highly intuitive, certainly much more than algebra. On the other hand, a purely geometric approach cannot progress without a proper algebraic foundation. This is the impulse behind René Descartes' program to algebraize geometry. Once we introduce the Cartesian plane, the intuitive relationship between points and lines can now be reduced to algebraic relations that are amenable to automatic proofs and calculations. Further abstractions are possible: in Felix Klein's Erlangen Program, the essence of geometry is reduced to invariant groups. Modern algebraic geometry has taken this abstraction to yet another level; the progression in these abstractions are described under different "epochs" by Jean Dieudonné's [2].

There is a way to formalize the intuitive geometry without algebra, going back to Euclid. This culminated in Hilbert's axiomatic or formal mathematics. Hilbert also used geometry as the launching point for this work. The formal (logical) approach and the algebraic viewpoints of geometry is given a new synthesis in Alfred Tarski's view of "elementary geometry and algebra". But Tarski's profound insight is the view that real elementary geometry is first order theory of semialgebraic sets. Moreover, there is a decision procedure for this theory based on a generalization of Sturm theory. In this lecture, we will review this aspect of Tarski's work and subsequent development by Collins.

We should mention another aspect of geometry: the combinatorial vein which is associated with the name of Paul Erdös. This connection is important from the computational viewpoint: the complexity of geometric algorithms are often intimately related to combinatorial bounds. The topological connection is another mode of abstraction (again, algebraization will play a central role).

To all these insights about the nature of geometry, we now add the computational perspective.

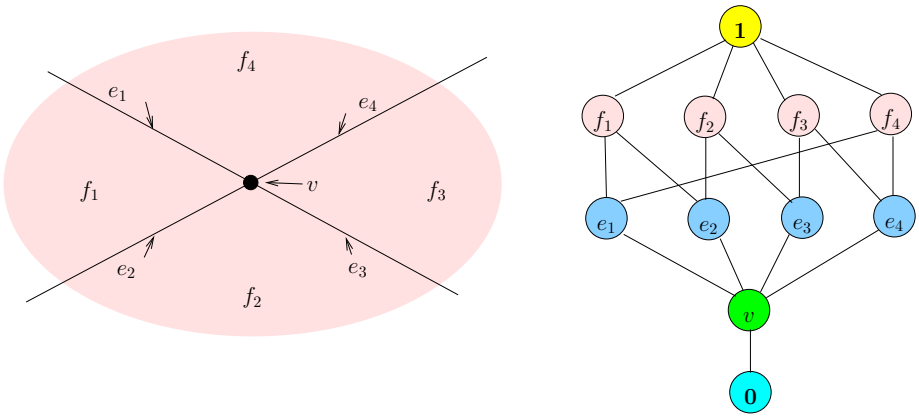
§1. What is Geometry?

When is a computation said to be "geometric"? One answer is "when it involves geometric objects". But what are geometric objects? We begin with the uncontroversial remark that notions such as points, lines, hyperplanes, polytopes, triangulated surfaces are geometric objects *par excellence*. We further observe that the set of all points, the set of all lines, and other basic geometric objects show a common feature: each set constitute a continuum of parametrized geometric objects of the same type. Our notation in the introduction, $\mathsf{Point}(x, y)$ and $\mathsf{Line}(a, b, c)$, hints are this property. In general, we can view geometric objects as $OBJ(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are numerical parameters. We might say this is Descartes' general insight, that geometric objects of a fixed type constitute a parametric space that can be investigated algebraically.

Next, there is a sense of space in geometry. Geometric objects are located in space, and hence they can be in in "spatial relationship" with each other. Thus, $\mathsf{Point}(x, y)$ and $\mathsf{Line}(a, b, c)$ each corresponds to a suitable subset of $\mathbb{R}^2$. It is this embedding in a common substrate that allows spatial relationships to be defined. Examples of spatial relationships include inside/outside, disjoint/intersecting, touching (or incidence), neighborhood (or adjacency) sidedness, co-incidence or other syzyzetic relations. For instance:

- A point can be inside a polytope.
- A line can lie on a hyperplane or be parallel to it, or intersect it transversally.
- Three points may be collinear.
- In dimensions 3 or more, four points may co-coplanar; and when co-planar, they may also be co-circular.
- Two faces of a polytope may be adjacent to each other.

Such relationships are numerically defined, but more importantly, they are[1] **discrete**. A collection of geometric objects will define an implicit collection of such relationships; typically, these relationships can be represented by combinatorial structures such as graphs with parametric labels on its vertices and edges (see [9]). For instance, a collection of points define a convex polytope (i.e., its convex hull). This polytope has facets of various dimensions which are in incidence (or adjacency) relationships. The **incidence graph** [3] is then used to represent this relationship among the facets.



(a) Arrangement of 2 lines (b) Incidence Graph

Figure 1: Incidence Graph

Figure 1(a) shows a pair of lines in the plane, and the partition of the plane by these two lines into subsets of dimension 0 (the point v), dimension 1 (the line segments e_i's) and dimension 2 (the faces f_i's). These subsets are related in Figure 1(b) by an incidence graph.

More generally, let us define this concept for a collection H of n hyperplanes in d-dimensions: this collection partitions space into pairwise disjoint regions called **faces**, where each face f has a dimension $\dim(f)$ between 0 and d. Two faces f, g are **incident** (on each other) provided (1) $|\dim(f) - \dim(g)| = 1$ and (2) either $f \subseteq \overline{g}$ or $g \subseteq \overline{f}$. Here, $\overline{g}$ denotes closure of a set g under the usual topology. The incidence graph $I(H)$ of H comprises a node $n(f)$ for each face f, and an edge from $n(f)$ to $n(g)$ provided $\dim(f) = \dim(g)+1$ and f, g are incident. It is convenient to introduce two **improper faces** denoted $\mathbf{0}$ and $\mathbf{1}$ where $\dim(\mathbf{0}) = -1$ and $\dim(\mathbf{1}) = d + 1$. Moreover, every 0-dimensional face is incident on $\mathbf{0}$ and every d-dimensioan face is incident on $\mathbf{1}$.

Thus, in general, we are dealing with collection of geometric objects in some special relations, which we might term a "geometric complex" for lack of a better term. We continue to write $OBJ(x_1, \ldots, x_n)$ for such

[1] Hoffmann [5] calls them "symbolic".

an object. What is important to realize is that OBJ contains combinatorial data (like graphs) as well as numerical parameters $x_1, \ldots, x_n$. Moreover, the number of numerical parameters for a given class of objects need not be fixed or bounded. This suggests the mnemonic,

$$GEOMETRIC = NUMERIC + COMBINATORIAL \tag{1}$$

But the mere presense of numbers and combinatorial structures in a computation alone does not qualify a computation to be called gemetric. This can be explained by way of two non-examples.
(a) Computing the determinant of a matrix M is non-geometric, even though the matrix structure is a combinatorial structure (albeit very regular one) and there are numerical entries.
(b) Computing the shortest path in a graph with edge weights (say, using Djikstra's algorithm) is likewise non-geometric, even though it has a combinatorial structure (the graph) and numerical values (the weights).

What is implicit in the formula (1) is that there are certain **consistency constraints** that must hold between the numeric and the combinatorial parts. In the extreme case, the numeric part completely determines the combinatorial part. In these two non-examples of geometric computation, there is no relationship between the combinatorial and its numerical parts.

¶1. Remark. Euclidean geometry is the default "geometry" in our discussions. Robustness considerations for other geometries studied in mathematics such as hyperbolic geometry, projective geometry, etc, should have similar considerations as those in the Euclidean case.

§2. What Exact Geometric Computation?

The general outline is this:

- First we prescribe the Exact Geometric Computation (EGC) solution.
- Then we show how EGC is possible if we have computable zero bounds.
- Finally we outline an implementation technique called precision-driven evaluation.

¶2. The EGC Prescription. Having analyzed the concept of "geometry", let us now see how it shows up during a computation. For our purposes, we may view a computation as a possibly infinite rooted tree T, with bounded branching at each step. Each non-terminal node of T will be classified as a **construction step** if it has exactly one child, otherwise it is a **branching step**. An example of a construction step might be

$$\boxed{\begin{array}{l} x \leftarrow x + 1; \\ x \leftarrow f(y, z); \end{array}}$$

As example of a branching step would be

$$\boxed{\begin{array}{l} \mathsf{if}\ z \geq 0\ \mathsf{then\ goto}\ L; \\ \vdots \\ L : \cdots \end{array}}$$

where L is a label. A **test value** is any quantity z that appears in a branching step. In modern computer languages, such "go-to" statements are packaged into case statements, while loops, and so on. The program itself is a finite set of instructions. But when the all the loops and recursions a program is "unrolled" into all possible computation paths, we obtain an infinite branching tree T. Although in principle, binary branching suffices, it is most natural to consider three-way branching for geometric computation because many geometric predicates are naturally 3-valued. E.g. is a point in/out/on a triangle? Is the area of a triangle 0/positive/negative?

Many geometric computations may be classified into one of the following categories: (1) constructing geometric complexes, (2) deriving geometric relationship among geometric complexes, and (3) searching in

geometric complexes. Regardless of the category, our understand of geometry implies that the geometric relationships of a particular input instance are encoded by branching choices during a computation.

Although T is infinite, a correct program must halt for any input instance. Halting computations correspond to a path from the root to a leaf. Suppose our computation is to output a geometric object $OBJ(y_1, \ldots, y_k)$ with parameters $y_1, \ldots, y_k$. If the input has parameters $\mathbf{x} = (x_1, \ldots, x_n)$ then each $y_i = y_i(\mathbf{x})$ is a function of the input numbers $x_1, \ldots, x_n$. Thus, all inputs $\mathbf{x}$ that end up at a specific leaf will yield the same parametrized object $OBJ(y_1(\mathbf{x}), \ldots, y_k(\mathbf{x}))$. Although the parameters may vary with $\mathbf{x}$, the combinatorial structure of the output is invariant at each leaf. We come to this general conclusion: *if ensure that every branch is correct, we guarantee the correct combinatorial structure.*

Here then is the prescription of the **Exact Geometric Computation Approach** (EGC): *it is to ensure that all branches for a computation path are correct.*

¶3. The Zero Problem. This seems almost trivial, but what are the issues? We must not forget that the computed values $y_i = y_i(\mathbf{x})$ are all approximate. The EGC prescription does not explicitly tell us how accurately to compute the y_i's. But it does tell us that the test values $z = z(\mathbf{x})$ that appear in our branching steps must be approximated to enough accuracy to determine its sign! Thus, we say that the central problem of EGC is the **sign problem**, to determine the sign of a numerical constant z. Of course, this would be trivial if z as given in an explicit form, say as machine number. Instead, z is given as an expression $z(\mathbf{x})$ in terms of the input parameters. Typically, we may reduce the sign problem to the simpler **zero problem**: to decide if $z(\mathbf{x}) = 0$. If z turns out to be non-zero, then assuming the ability to approximate z to any precision, we can eventually decide whether $z > 0$ or $z < 0$. The zero problem is a very deep question in mathematics, but in particular, it is intimately connected to questions in transcendental number theory.

For our purposes, we can pose the zero problem as follows. Fix a set Ω of real operators and let $Expr(\Omega)$ denote the set of expressions over Ω. A typical example is

$$\Omega_1 = \{\pm, \times, \div\} \cup \mathbb{Z}.$$

Note that the constants $a \in \mathbb{Z}$ are considered 0-ary operators. Thus each expression $e \in Expr(\Omega_1)$ denotes a rational number $\texttt{val}(e) \in \mathbb{Q}$, if defined. The reason that $\texttt{val}(e)$ may be undefined is because $\div$ is a partial operator ($a \div b$ is undefined if $b = 0$). These expressions can be viewed as rooted trees, but in general, we view them as directed acyclic graphs (DAG's). In general, there is a natural evaluation function

$$\texttt{val} : Expr(\Omega) \to \mathbb{R}$$

which is a partial function if Ω contains any partial function such as $\div$.

So the question before us is this: given an expression $e \in Expr(\Omega)$, to decide if $\texttt{val}(e)$ is defined and if so, return its sign. If we can solve this problem for an algorithm whose numerical operators belong to Ω, then we carry out the EGC prescription for this algorithm.

¶4. Precision-Driven Evaluation So, to achieve EGC, we need to determine the sign of each test value. A simple solution is to determine the sign of *all* numerical values, regardless of whether it is a test value or not. But wastefulness aside, we run into a more basic difficulty: what should be the precision to which each numerical value z be approximated? We cannot tell in advance – it depends on subsequent use of z in the evaluation of another value that depends on z!

To discuss errors/precision, it is useful to introduce this notation: let $z, \widetilde{z}, p \in \mathbb{R}$. An expression of the form "$z\langle p\rangle$" is a shorthand for "$z(1 \pm 2^{-p})$". Therefore

$$\widetilde{z} = z\langle p\rangle \tag{2}$$

means that $|\widetilde{z} - z| \leq |z|2^{-p}$, i.e., $\widetilde{z}$ is a **p-bit relative approximation** of z. Similarly, "$z[p]$" is a shorthand for "$z \pm 2^{-p}$". Therefore

$$\widetilde{z} = z[p] \tag{3}$$

means that $\widetilde{z}$ is a **p-bit absolute approximation** of z.

In (2) and (3), should p be regarded as the precision or error in $\widetilde{z}$ as an approximation to z? We will make this distinction: if p is given *a priori* (before $\widetilde{z}$ as computed), then we call it **precision**. But if p is

computed *a posteriori* (from z and $\widetilde{z}$), then we call it **error**. E.g., we might say all our computations are carried out to "100 bits of relative precision", and or we might say that a particular approximate value $\widetilde{z}$ has "100 bits of absolute error".

A basic observation (Exercise) is this:

LEMMA 1. $\widetilde{z} = z\langle 1\rangle$ *iff* $\texttt{sign}(z) = \texttt{sign}(\widetilde{z})$.

Thus, computing z to relative one-bit amounts to determining the sign of z. We propose a general solution [10] that is akin to what is often called lazy evaluation in the computing milieu. We store the extire expression $z(\mathbf{x})$ for each numerical quantity z as a function of the input parameters $\mathbf{x}$. When the sign of z is actually needed, we will evaluate the expression for z to determine the sign. But how is this evaluation carried out?

We will regard each node u in an expression as representing an (exact) numerical value $\texttt{val}(u)$, and also storing an approximate numerical values $\widetilde{val}(u)$. (It is convenient to abuse notation and use u and $\texttt{val}(u)$ interchangeably.) In case u is an internal node, it also represents an operation $op(u)$ and it will have a number of children equal to the arity of $op(u)$. E.g., if $op(u)$ is a binary operation, it will have two children. We illustrate this in Figure 2 for a node x In Figure 2(a), we show a multiplication node x (whose value is denoted x, by our abuse of terminology). This node depends two other nodes y and z, and so we have the relation $x = yz$.

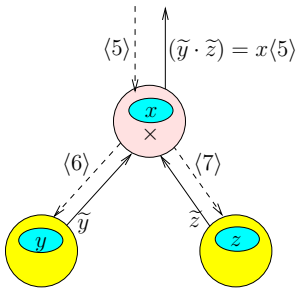


Figure 2: Precision and error propagation at a node

Suppose we want to compute x to p-bits of relative precision. If we require $p+1$-bits of relative precision from y and $p+2$-bits of relative precision from z, then it is easy to see that an exact multiplication of the approximate values will yield the desired result for x. In other words:

$$x = yz, \quad \widetilde{y} = y\langle n+1\rangle, \quad \widetilde{z} = z\langle n+2\rangle. \Rightarrow .\widetilde{y} \cdot \widetilde{z} = x\langle n\rangle. \tag{4}$$

Giving a similar rule for multiplication with absolute error is slightly more involved. On the other hand, we can give a similar rule for addition to absolute precision:

$$x = y + z, \quad \widetilde{y} = y[n+1], \quad \widetilde{z} = z[n+1]. \Rightarrow .\widetilde{y} + \widetilde{z} = x[n]. \tag{5}$$

Giving a similar rule for addition with relative error is slightly more involved. We refer to [8] for this and other rules for the common real operators

$$\pm, \times, \div, \sqrt{\cdot}, \exp, \log .$$

Suppose E_z is the expression for z. For simplicity, we imagine E_z to be a tree rooted at z. The leaves of E_z contain the input parameters $\mathbf{x} = (x_1, \ldots, x_n)$. At each node y of E_z, we store an arbitrary precision (bigfloat) approximation $\widetilde{y}$ for y.

§3. Algebraic Background

We assume some familiarity with the following tower of algebraic structures:

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}.$$

The prime example of a ring is $\mathbb{Z}$ (integers), and the rational numbers $\mathbb{Q}$ is our prime example of a field. So we are mainly interested in computation over suitable subsets of the real numbers $\mathbb{R}$, but ocassionally the complex numbers $\mathbb{C}$. Also, we write $\mathbb{A}$ for the set of **real algebraic numbers**. The set of all algebraic numbers is therefore given by $\mathbb{A}[\mathbf{i}]$.

We will try to state our results as concretely as possible. But student less familiar with abstract algebra may you may substitute these prime examples whenever we say "ring" or "field". Much of this material is summarized from my book [7].

Let K be a field. If X is a variable, let $K[X]$ denote the polynomial ring with coefficients in R. A typical polynomial $A = A(X)$ has the form

$$A(X) := \sum_{i=0}^{m} a_i X^i, \qquad a_m \neq 0$$

where each non-zero a_i is called a **coefficient** of A. Here, a_m is called the **leading coefficient** and m is called the **degree** of $A(X)$. These are denoted $\texttt{lead}(A) = a_m$ and $\deg(A) = m$ respectively. In case $A(X) = 0$, we define[2] $\texttt{lead}(0) = 0$ and $\deg(0) = -\infty$. If $\texttt{lead}A = 1$ then A is said to be **monic**. If $\alpha \in K$ and $A(\alpha) = 0$, we say α is a **root** of $A(X)$. There are several ways to measure the size of polynomials that takes into account its coefficients. For any number $p \geq 1$, define the p**-norm**,

$$\|A(X)\|_p := \sqrt[p]{\sum_{i=0}^{m} |a_i|^p}.$$

We can also define $\|A(X)\|_\infty = \max_{i=0}^{m} |a_i|$, The case $p = 2$ and $p = \infty$ (respectively) are also called the (Euclidean) length and height of A.

A pair of polynomials $p, q \in K[X]$ defines a **rational function** in X which is usually denoted p/q. Moreover, if p'/q' is another rational function, then p/q and p'/q' are declared to be equal if and only if $pq' = p'q$. The set of rational functions is denoted $K(X)$.

In case $K = \mathbb{C}$, we have the Fundamental Theorem of Algebra which says that every $p(X) \in \mathbb{C}[X]$ has exactly $\deg(p)$ complex roots. A number $\alpha \in \mathbb{C}$ is said to be **algebraic** if it is the root of a polynomial $p \in \mathbb{Z}[X]$. If further, p is monic, then α ia called an **algebraic integer**. This generalizes our usual concept of integers.

¶5. UFD. Let R be a ring, $a, b \in R$. A domain is a ring in which $ab = 0$ implies $a = 0$ or $b = 0$. We say a **divides** b if there is a c such that $ac = b$. If $ab = 0$ and $b \neq 0$ then we call a a zero-divisor. Thus a domain is a ring whose only zero-divisor is 0. A **unit** a is an element that divides 1. Alternatively, the equation $ax = 1$ has a solution x. E.g., $\mathbb{Z}$ is a domain in which the only units are ± 1 In a field, all non-zero elements are units. A **unique factorization domain** (UFD) is a domain in which every element a can be written as a product $a = \prod_{i=1}^{k} p_i$ of irreducible elements p_i; this product is unique up to units in the sense that if $a = \prod_{i=j}^{\ell} q_j$ then $k = \ell$ and by suitable reordering, each $p_i = u_i q_i$ were u_i are units. The Fundamental Theorem of Arithmetic tells us that $\mathbb{Z}$ is a UFD. A basic result of Gauss says that if R is a UFD if and only if $R[X]$ is a UFD.

In a UFD, we can define the concept of a GCD of a set of elements. In the ring $K[X]$ over a field K, Euclid's algorithm can be extended to compute the GCD of two polynomials.

¶6. Resultants. A basic tool in algebraic number theory is the concept of resultants. In the following, assume D is a UFD. Given $p, q \in D[X]$, we define their resultant $\texttt{res}(p, q) \in D$ to be the determinant of the

[2]Sometimes, $\deg(0)$ is defined as $+\infty$ instead of $-\infty$. Both definitions have advantages, but neither seems superior: for instance, exactly one of the following statements is true in either definition: (1) "the division algorithm for A, B produces Q, R such that $A = BQ + R$ with $\deg R < \deg B$". (2) "Every complex polynomial of degree m has exactly m roots". Of course, in practice, we just have to add the qualification that A is non-zero. If we define $\deg(0) = +\infty$, then the condition "$\deg A \leq d$" can be used to exclude the case $A = 0$, but then the condition "$\deg A > \deg B$" may produce some surprises. We have opted to avoid surprises.

Sylvester matrix $S(p,q)$ of p,q. If $m = \deg p$ and $n = \deg q$, then $S(p,q)$ is a $m+n$ square matrix whose first n rows are filled with the coefficients of

$$X^{n-1}p, X^{n-2}p, \ldots, Xp, p$$

and the remaining m rows are filled with the coefficients of

$$X^{m-1}q, X^{m-2}q, \ldots, Xq, q.$$

If $p = \sum_{i=0}^{m} p_i X^i$ and $q = \sum_{j=0}^{n} q_i X^j$ then

$$S(p,q) \quad = \quad \left[\begin{array}{ccccccccc|c}
p_m & p_{m-1} & \cdots & & p_0 & & & & & X^{n-1}p \\
 & p_m & p_{m-1} & \cdots & & p_0 & & & & X^{n-2}p \\
 & & \ddots & & & & \ddots & & & \vdots \\
 & & & p_m & p_{m-1} & \cdots & & & p_0 & p \\
\hline
q_n & q_{n-1} & \cdots & q_1 & q_0 & & & & & X^{m-1}q \\
 & q_n & q_{n-1} & \cdots & q_1 & q_0 & & & & X^{m-2}q \\
 & & \ddots & & & & \ddots & & & \vdots \\
 & & & q_n & q_{n-1} & \cdots & & & q_0 & q
\end{array}\right]$$

$$\overbrace{\begin{array}{cccccccc} X^{m+n-1} & X^{m+n-2} & \cdots & X^n & X^{n-1} & \cdots & \cdots & X^0 \end{array}}$$

Note that $X^i p$ is a polynomial of degree $m+i$ and its coefficients fill the ith row of $S(p,q)$. The jth column contains the coefficients of X^{m+n-j}, and thus the leading coefficient of $X^i p$ will be in the $(i, n-i)$th entry of $S(p,q)$. The first n rows will therefore contain the coefficients of p but each is a right-shift of the previous row. The last m row is similarly filles with right-shifted coefficients of q.

We now prove a basic lemma:

Lemma 2. $\texttt{GCD}(p,q)$ *is a constant iff* $\texttt{res}(p,q) \neq 0$.

Proof. Suppose $\texttt{res}(p,q) = 0$. This means $\det(S) = 0$ where $S = S(p,q)$. So a non-trivial linear combination of the rows of S vanishes, i.e.,

$$w \cdot S = 0 \tag{6}$$

where

$$w = (u_{n-1}, u_{n-2}, \ldots, u_0, v_{m-1}, v_{m-2}, \ldots, v_0)$$

is a nonzero row vector of length $m+n$. If

$$x = (X^{m+n-1}, X^{m+n-2}, \ldots, X, 1)^T$$

is a column vector of length $m+n$, then (6) is equivalent to

$$w \cdot S \cdot x = 0.$$

This latter equation can be re-written as the polynomial equation

$$U(X)p(X) + V(X)q(X) = 0 \tag{7}$$

where $U(X) = \sum_{i=0}^{n-1} u_i X^i$ and $V(X) = \sum_{j=0}^{m-1} u_j X^j$. Note that U and V has degree at most $n-1$ and $m-1$, respectively, and since w is nonzero, we have $VU \neq 0$. Since D is a UFD, (7) implies that $p(X)$ divides $V(X)q(X)$. Thus,

$$\texttt{GCD}(p(X), V(X))\texttt{GCD}(p(X), q(X)) = p(X)$$

$$\deg(\texttt{GCD}(p(X), V(X))) + \deg(\texttt{GCD}(p(X), q(X))) = m = \deg(p(X)).$$

But $\deg(\texttt{GCD}(p(X), V(X))) \leq \deg(V(X)) \leq m-1$. Hence $\deg(\texttt{GCD}(p(X), q(X))) \geq 1$. This proves our claim that $\texttt{GCD}(p, q)$ is non-constant.

Conversely, suppose $\texttt{GCD}(p, q) = g(X)$ is non-constant. Then, letting $U(X) := q(X)/g(X)$ and $V(X) := -p(X)/g(X)$, we see that (7) holds. This implies (6) holds for some non-zero w. This shows S is singular, or $\texttt{res}(p, q) = \det(S) = 0$. **Q.E.D.**

The concept of a resultant can be generalized to the the notion of subresultants. We state some basic properties of resultants: let A, B be polynomials with $\deg A = m, \deg B = n$, $\texttt{lead}A = a, \texttt{lead}B = b$. Also let the roots of A and B be $\alpha_1, \ldots, \alpha_m$ and $\beta_1, \ldots, \beta_n$ (resp.).

Lemma 3.
(i) $\texttt{res}((X-\alpha)p, q) = q(\alpha)\texttt{res}(p, q)$.
(ii) $\texttt{res}(A, B) = a^n \prod_{i=1}^m B(\alpha_i)$.
(iii) $\texttt{res}(A, B) = (-1)^{mn} b^m \prod_{j=1}^n A(\beta_j)$.
(iv) $\texttt{res}(A, B) = a^n b^m \prod_{i=1}^m \prod_{j=1}^n (\alpha_i - \beta_j)$.

It is easy to show (ii)-(iv) from (i). The proof of (i) is somewhat involved, but can be achieved by a direct computation. Also, we have

Lemma 4. *Let* $A(\alpha) = 0, B(\beta) = 0$.
(i) If $\alpha \neq 0$ *then* $1/\alpha$ *is a root of* $X^m A(1/X)$.
(ii) $\beta \pm \alpha$ *is a root of* $\texttt{res}_Y(A(Y), B(X \mp Y))$.
(ii) $\alpha\beta$ *is a root of* $\texttt{res}_Y(A(Y), Y^n B(X/Y))$.

It follows from the above that the set of algebraic numbers forms a field. Moreover, the algebraic integers forms a ring.

¶7. Root Bounds. There is a very large classical literature on root bounds. Here, we content ourselves with a simple one.

Suppose $p(X) = \sum_{i=0}^m a_i X^i$, and $\alpha \neq 0$ is a root of $p(X)$. Then we have the following bound of Cauchy:

$$\frac{|a_0|}{|a_0| + H_0} < |\alpha| < 1 + \frac{H_m}{|a_m|}$$

where $H_0 = \max\{|a_1|, |a_2|, \ldots, |a_m|\}$ and $H_m = \max\{|a_0|, |a_1|, \ldots, |a_{m-1}|\}$.

Let us first prove the upper bound. If $|\alpha| \leq 1$, the result is true. Otherwise, we have

$$|a_m| \cdot |\alpha|^m \leq H_m \sum_{i=0}^{m-1} |\alpha|^i$$

which leads to the stated bound. For the lower bound, consider the polynomial $p(1/X)X^m$ instead.

We also have root separation bounds: this is slightly more involved to prove, and requires the concept of discriminants. The **discriminant** of a polynomial $A(X)$ is defined to be $\text{disc}(A) := (-1)^{\binom{m}{2}}(1/a)\texttt{res}(A, A')$ where $a = \texttt{lead}A$ and A' is the derivative of A. For instance if $A(X) = aX^2 + bX + C$ then $\text{disc}(A) = b^2 - 4ac$. Two important properties of the discriminant are

- $\text{disc}(A) \in D$
- $$\text{disc}(A) = a^{2m-2} \prod_{1 \leq i < j \leq m} (\alpha_i - \alpha_j)^2$$

  where $\alpha_1, \ldots, \alpha_m$ are all the roots of $A(X)$ in the algebraic closure $\overline{D}$ of the domain D.

It follows from the second property that $\text{disc}(A)$ vanishes iff A has multiple roots.

The following bound is from Mahler:

Theorem 5. *Let α, α' be distinct roots of $A(X) \in \mathbb{C}[X]$. Then*

$$|\alpha - \alpha'| > \sqrt{|\mathrm{disc}(A)|}\|A\|_2^{-m+1} m^{-(m+2)/2}(\sqrt{3}.$$

Because of the presense of $\mathrm{disc}(A)$ in the right hand side, this bound is trivial if A has multiple roots. To recover a useful bound, we can replace A by its square-free part (Exercise). There is a generalization due to Davenport which gives a lower bound on a product of differences of roots of A.

¶8. Sturm Theory. Suppose $A, B \subseteq \mathbb{R}[X]$ and $\deg A > \deg B \geq 0$. We define a **generalized Sturm sequence** for (A, B) to be a sequence

$$\overline{A} = (A_0, A_1, \ldots, A_h)$$

where $A_0 = A, A_1 = B$ and for $i = 1, \ldots, h-1$,

$$\beta_i A_{i+1} = \alpha_i A_{i-1} + Q_i A_i$$

where $\alpha_i, \beta_i \in \mathbb{R}$, $Q_i \in \mathbb{R}[X]$ and $\alpha_i \beta_i < 0$, and finally

$$A_h | A_{h-1}.$$

If $B = A'$ (the derivative of A) then we simply call $\overline{A}$ a Sturm sequence for A.

The "standard construction" of such a generalized Sturm sequence is where

$$A_{i+1} = -(A_{i-1} \mod A_i)$$

where $A \mod B = R$ means that there exists $Q \in \mathbb{R}[X]$ such that $A = QB + R$ where $\deg R < \deg B$. By high school division of polynomials, it is seen that Q and R is uniquely determined by these conditions. Moreover, in case $A, B \in \mathbb{Z}[X]$, the standard construction yields a sequence of polynomials where each $A_i \in \mathbb{Q}[X]$. As noted in [7], there are better methods than this standard construction in which the A_i's are computed as integer polynomials. In the following, it is convenient to assume some construction method so that, once A, B are given, the rest of the sequence $\overline{A}$ is determined.

If $\overline{a} = (a_0, a_1, \ldots, a_h)$ is a sequence of real numbers, we define the **sign variation** $\mathtt{Var}(\overline{a})$ of $\overline{a}$ to be the number of sign variations (i.e., transitions from $+$ to $-$ or $+$ to $-$) after we omit all 0 values from the sequence. E.g. $\mathtt{Var}(2, 0, 3.5, -1.2, 0, -10, 0, 0, 3) = 2$. If $\overline{A}$ is a sequence of real polynomials $(A_0(X), \ldots, A_h(X))$ and $a \in \mathbb{R}$ then define

$$\mathtt{Var}_{\overline{A}}(a) := \mathtt{Var}((A_0(a), \ldots, A_h(a))).$$

If $\overline{A}$ is a generalized Sturm sequence for A, B then we also write $\mathtt{Var}_{A,B}(a)$ for $\mathtt{Var}_{\overline{A}}(a)$. This notation is justified as it is easily shown (Exercise) that the sign variation does not depend on particular choice of $\overline{A}$. Moreover, when $B = A'$, we simply write $\mathtt{Var}_A(a)$ instead of $\mathtt{Var}_{A,A'}(a)$.

Theorem 6 (Sturm). *Let $A(X) \in \mathbb{R}[X]$ have positive degree and $A'(X)$ denotes its derivative. If $a < b \in \mathbb{R}$ such that $A(a)A(b) \neq 0$ then the number of distinct real roots of $A(X)$ in the interval $[a, b]$ is equal to*

$$\mathtt{Var}_A(a) - \mathtt{Var}_A(b).$$

Proof. Let $(A_0, A_1, \ldots, A_h)$ be the Sturm sequence of A. For $a \leq c \leq b$, define $v_i(c) := \mathtt{Var}(A_{i-1}(c), A_i(c), A_{i+1}(c))$ for $i = 0, \ldots, h$ (assume $A_{-1}(c) = A_{h+1}(c) = 0$). We initially assume that $C = \mathtt{GCD}(A, A')$ is constant (so A, A') has no common zero).

- For $i = 1, \ldots, h$, if $A_{i-1}(c) = A_i(c) = 0$ then $A_{i-2}(c) = A_{i+1}(c) = 0$. Thus, if there are two consecutive zeros in $(A_0(c), \ldots, A_h(c))$ then the entire sequence is 0.
- So $A_h(c) \neq 0$ (otherwise, $A_{h-1}(c) = 0$ by definition of h, and hence $A_i(c) = 0$ for all i; in particular c is common zero of A, A', contradiction)
- Call c a special value if there is some $i \in \{0, 1, \ldots, h\}$ such that $A_i(c) = 0$. There are finitely many special values. Moreover, $\mathtt{val}_A(c)$ is constant as c varies between two consecutive special values. In other words, $\mathtt{val}_A(c)$ can only change when c passes through a specail value. Thus, we next try to understand how $\mathtt{val}_A(c)$ changes at special values.

- For any c, there exist a subset $I \subseteq \{0, 1, \ldots, h\}$ such that

$$\mathtt{Var}_A(c) = \sum_{i \in I} v_i(c). \tag{8}$$

  For instance, if $\mathtt{val}_A(c) = (A_0(c), A_1(c), \ldots, A_6(c)) = (2, -1, 0, 1, -2, 3, -4)$ then some possible choices of I are $\{0, 4, 6\}$ or $\{1, 3, 5\}$. Moreover, we may choose I such that $i \in I$ implies $A_{i-1}(c)A_{i+1}(c) \neq 0$ unless $i = 0$ or $i = h$. In the preceding example, the choice $I = \{0, 4, 6\}$ has this property, but not $I = \{1, 3, 5\}$.

- CLAIM: For all $i \in I$, we have (1) $v_i(c^-) - v_i(c^+) = 1$ if $i = A_0(c) = 0$, and (2) $v_i(c^-) - v_i(c^+) = 0$ otherwise.

- To see (1), we consider two cases: either A_0 is increasing at c or decreasing at c. If A_0 is increasing at c, then $A'(c) > 0$ and $v_0(c)$ increases by 1 as we pass through c. If A_0 is decreasing at c, we again conclude that $v_0(c)$ increases by 1 through c.

- To see (2), we consider two cases: If $A_i(c) = 0$, then $i > 0$ (because of (1)) and $i < h$ (because $A_h(c) \neq 0$). Then $A_{i-1}(c)A_{i+1}(c) \neq 0$. Then, regardless of the values of $A_i(c^-)$ and $A_i(c^+)$, the value $v_i(c^-) - v_i(c^+)$ is 0. If $A_i(c) \neq 0$, then by our assumption that $A_{i-1}(c)A_{i+1}(c) \neq 0$ unless $i = 0$ or $i = h$, we conclude that $v_i(c^-) - v_i(c^+) = 0$ again.

- Our claim therefore implies that $\mathtt{val}_A(c)$ is constant incrementing each time that c passes through a real zero of A. Thus $\mathtt{Var}_A(a) - \mathtt{val}_A(c)$ equals the number of real zeros of A in $[a, b]$.

Finally, suppose $\deg(A_h) > 0$. The sequence $(A_0/A_h, A_1/A_h, \ldots, A_{h-1}/A_h, 1)$ has the same properties as what we proved in (i). Moreover, the sign variation of this modified sequence at any c, that is not a zero of A_h, is equal to $\mathtt{Var}_A(c)$ **Q.E.D.**

See [7, Theorem 7.3, p. 194] for a very general form of this basic theory of counting sign changes.

Exercises

Exercise 3.1: Prove that $\mathbb{A}[\mathbf{i}]$ is the set of all algebraic numbers. ♢

Exercise 3.2: Let $\overline{A}$ be a generalized Sturm sequence for A, B. Show that the value $\mathtt{Var}_{\overline{A}}(a)$ does not depend on the choice of $\overline{A}$. ♢

Exercise 3.3: Let $A(X) = X^4 - 8X^3 + 2X^2 - 14$ and $B(X) = X^3 + X^3 - 7X^2 + X - 1$.
(i) Compute the standard Sturm sequence for A, B.
(ii) Compute the standard Sturm sequence for A.
(iii) Determine the number of real roots of A. ♢

End Exercises

§4. Exact Numerical Algebraic Number Computation

We now demonstrate that the set $\mathbb{A}$ of real algebraic numbers is a suitable domain for computation. More precisely, our goals in this section are:

- To provide a representation for elements of $\mathbb{A}$.

- To show that the operations $\pm, \times, \div, \sqrt{\cdot}$ and comparisons on $\mathbb{A}$ can be carried out on such representations.

¶9. Isolating Interval Approach. There are several known methods in computer algebrac for achieving the above goals. A standard representation real algebraic numbers is called the **isolating interval** representation. If $\alpha \in \mathbb{A}$, then such a representation of α is a pair $(A(X), I)$ where $A(X) \in \mathbb{Z}[X]$ and I is an isolating interval of $A(X)$ containing α. Moreover, if $w(I) > 0$ then α lies in the interior of I. We will write

$$\alpha \simeq (A(X), I) \tag{9}$$

in this case. Here, "isolating interval" means that I contains a unique real root of $A(X)$. Usually, we also require that $A(X)$ be square-free. The **square-free part** of A is defined as follows:

$$sqfr(A) = \frac{A}{\texttt{GCD}(A, A')}$$

where A' is the derivative of A. If $sqfr(A) = A$, we say A is **square-free**.

Note that we can refine the interval I in (9) very easily: evaluate $A(mid(I))$. If this is zero, we can replace I be $[mid(I), mid(I)]$. Otherwise, if $A(mid(I))A(\underline{I}) < 0$, we replace I by $[\underline{I}, mid(I)]$; otherwise, replace I by $[\overline{I}, mid(I)]$. This process can be repeated as often as we like to make $w(I)$ as small as we wish.

Most computer algebra books will assume that the standard representation in the power basis, i.e., we have a list of the coefficients of $A(X)$. Therefore, when we want to compute a representation of

$$\gamma = \alpha + \beta$$

where $\beta \simeq (B(X), J)$. By the resultant results of the previous section, we know that $C(X) = \texttt{res}_Y(A(Y), B(X - Y))$ contains $\alpha + \beta$ as a root. We can replace $C(X)$ by its square-free part if desired. Finally, we would hope that $I + J$ is an isolating interval of $C(X)$. This can be checked using Sturm sequence of $C(X)$. If so, we can output

$$\gamma \simeq (C(X), I + J)$$

If $I + J$ is not a isolating, we half the width of I and J using the refinement step above, and check again if $I + J$ is an isolating interval of $C(X)$. We can repeat this until $I + J$ is an isolating interval. We can similarly perform the other arithmetic operations. We leave it as an exercise to compute $\sqrt{\alpha}$.

Comparisons between α and β can be decided at once if I and J are disjoint. Otherwise, we can repeat unte the root separation bound, and declare $\alpha = \beta$.

¶10. Expression Approach. We now discuss a somewhat unconventional representation: we let $A(X)$ be an expression. In this case, the operations $\pm, \times, \div$ becomes trivial. E.g., to perform the operation

$$x \leftarrow y + z$$

we simply construct a new node for x and make the two children of this node to be the expressions for y and z.

So the main issue is how to form comparisons. This, in turn, can be reduced to checking if a number is zero. We use the method of **constructive zero bounds**: suppose there is a systematic method to attach root bound $B(u) > 0$ to each node u in an expression e with the property that if $\texttt{val}(u) \neq 0$ then

$$|\texttt{val}(u)| \geq B(u).$$

The construction of B is constructive in the following sense: we can maintain with each node u a fixed set of numerical parameters, $q_1(u), q_2(u), \ldots, q_k(u)$ such that (1),

$$B(u) = \beta(q_1(u), \ldots, q_k(u))$$

where β is a computable function, and for each node u, we can compute its parameters $q_i(u)$ from the set of parameters at its children. In a later section, we will give such a constructive zero bound.

Now, we can use our precision-driven evaluation mechanism on the expression e to approximate e to absolute p-bits for $p = 1, 2, 4, 8, \ldots$ until such p where $|\widetilde{e}| > 2^{-p}$ or $p = B(e)$. If $\widetilde{e}| > 2^{-p}$, then the sign of e is the sign of $\widetilde{e}$. Otherwise we conclude that $e = 0$.

¶11. EVAL The **root isolation problem** is this: given a polynomial $f(X)$, to compute an isolating interval for each of the real zeros of $f(X)$. A traditional approach is to use the Sturm sequence of $f(X)$, as we have seen in the previous section. But this turns out that more efficient methods are possible. In recent years, the Descartes method (based on the Descartes Rule of Signs) has been popular. We now describe an even simpler approach, based on interval evaluation. Like the Descartes Method, we require $f(X)$ to be square-free.

In fact, f need not be a polynomial. Let $f : I_0 \to \mathbb{R}$ be any real function on an interval $I_0 \subseteq \mathbb{R}$ that satisfies:

- The derivative f' is defined on I_0.
- f has finitely many zeros in I_0, and $f(x) = 0$ implies $f'(x) \neq 0$.
- We can compute the box functions $\square f(I)$ and $\square f'(I)$ where $I \subseteq I_0$ are dyadic intervals.
- For any dyadic number $x \in I_0$, we can determine the sign of $f(x)$.

Any such f is said to belong to $PV(I_0)$ (or to PV if $I_0 = \mathbb{R}$). Then we have the following extremely easy algorithm:

```
EVAL(f, I_0):
    Input: f ∈ PV(I_0) where I_0 is a finite dyadic interval
           whose endpoints are not roots of f.
    Output: A list L of isolating intervals for each real root of f in I_0
    ---------------------------------------------------------------
    Initialize two lists, Q ← {I_0} and L ← ∅
    while Q is nonempty
        Remove I from Q
        if 0 ∉ □f(I), discard I
        elif 0 ∉ □f'(I)
            if (f(Ī)f(I̲) < 0), L.append(I)
        else
            Insert I_0 = [I̲, mid(I)] and I_1 = [mid(I), Ī] into Q
            if f(mid(I)) = 0, L.append([mid(I), mid(I)])
        return(L)
```

§5. Cylindrical Algebraic Decomposition

Elementary Geometry usually associated with the study of geometric figures in 2 or 3 dimensions and their relationships, as taught in high school. This tradition goes back to the Greeks as well as many ancient civilizations. It is Descartes' major innovation to consider geometric objects as parametric objects. This allows us to reduce geometric questions to numerical computation.

To see the tremendous power of this approach, consider what the alternative might be. Indeed, historically there is another development of geometry, alongside with the algebraization: this is based on the axiomatic approach. We could not do any geometric computation in the usual understanding of geometric computation. Instead, the computational focus in this parallel development is theorem proving. Here the idea is to encode theorems as statements in first order logic, and to deduce them from the axioms. This turned out to be notoriously difficult – only fairly trivial theorems could be proved by first order logic provers. Subsequent clarification by Wu Wen-Tsun indicated why – most elementary theorems are true only "generically", not universally. Moreover, the algebraization approach was hugely successful. Nevertheless, there is an interesting convergence of the logical approach with the algebraization approach in the work of Alfred Tarski. According to Tarski, elementary geometry can be encoded in the first order theory of real closed fields. This turned out to be extremely fruitful: it led to a decision procedure for this language. But in the hands of G.E.Collins, this developed a more geometric viewpoint leading to a procedure to decompose Euclidean space into what is now known as a Cylindrical Algebraic Decomposition (CAD). In some sense, this is the superalgorithm which allows us to solve all questions of elementary geometry. Our goal in this section is to give the basic elements of computing CAD.

¶12. Sign-Invariant Decomposition. Consider the following a planar geometric figure consisting of a line $f = 0$ and a circle $g = 0$ where

$$f = X - Y, \qquad g = (X-1)^2 + (Y-1)^2 - 1.$$

This is illustrated in Figure 3(a).

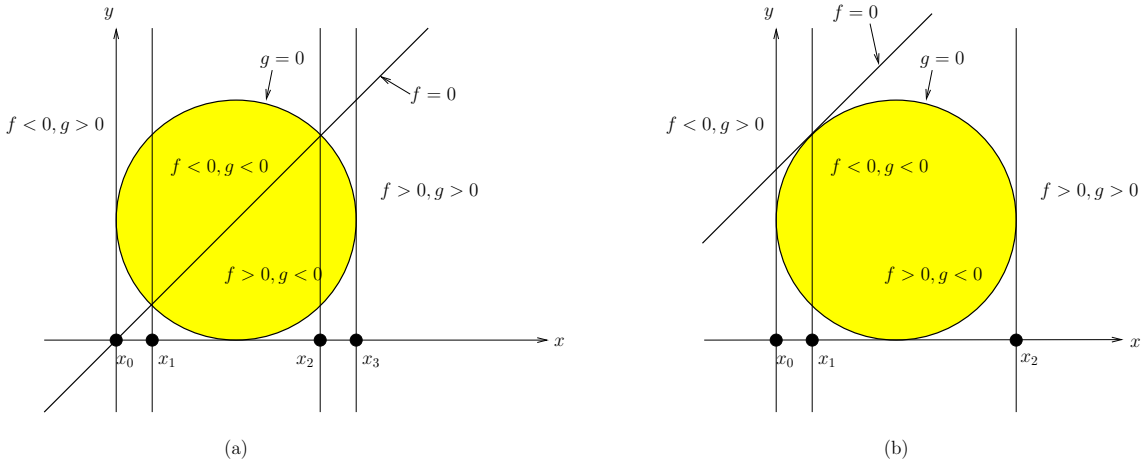


Figure 3: Sign-invariant regions of a line $f = 0$ and a circle $g = 0$.

The plane $\mathbb{R}^2$ is partitioned by these equations into connected subsets of dimensions, with the property that the sign of $\Sigma = \{f, g\}$ is invariant in each subset. We call such regions a **sign-invariant region**. Consider the following four sign conditions:

$$\{f < 0, g < 0\}, \quad \{f < 0, g > 0\}, \quad \{f > 0, g < 0\}, \quad \{f > 0, g > 0\}.$$

Each of them determine a unique 2-dimensional sign-invariant region. Next consider the following sign conditions:

$$\{f = 0, g < 0\}, \quad \{f = 0, g > 0\}, \quad \{f > 0, g = 0\}, \quad \{f > 0, g = 0\}.$$

The first condition again determines a unique sign-invariant region. However, because of the equality condition, any sign regions is necessarily 1-dimensional. But the second condition $\{f = 0, g > 0\}$ determines two sign-invariant regions. The third and fourth conditions again determine a unique sign-invariant region. Finally, there are two 0-dimensional sign-invariant regions corresponds to sign condition

$$\{f = 0, g = 0\}$$

They correspond to the two intersection points of the line and the circle.

In general, if $F \subseteq \mathbb{R}[X_1, \ldots, X_n] = \mathbb{R}[\mathbf{X}]$ is a set of polynomials, we say a set $R \subseteq \mathbb{R}^n$ is **F-invariant** if each polynomial in F has a definite sign throughout R. The partition of $\mathbb{R}^n$ into the collection of maximal connected subsets that are F-invariant is called a **F-invariant decomposition** of $\mathbb{R}^n$.

Thus, our example shows that the $\{f, g\}$-invariant decomposition of $\mathbb{R}^2$ has 11 sign-invariant regions.

¶13. Cylindrical Cell Decomposition. All the sign-invariant regions in the preceding example has an additional property: they are each homeomorphic to $\mathbb{R}^d$ if the region is d-dimensional. Such regions will be called **cells** or **d-cells**. In general, sign-invariant regions need not be cells, as we can easily imagine regions that are non-simply connected for $d \geq 2$. We will next subdivide these sign-invariant into smaller sets which will always be cells.

First, we consider certain special values of x,

$$x_0 < x_1 < x_2 < x_3$$

where each x_i is the x-projection of the intersection of the two curves $f = 0, g = 0$, or the x-extremal points of the curve $g = 0$. Note that $f = 0$ has no x-extremal point. We can define a point p to be x-extremal for a general curve $f = 0$ if $f(p) = f_Y(p) = 0$ where $f_Y = \frac{\partial f}{\partial Y}$.

We can say that these special values determine a **cell decomposition** of the x-axis (i.e., $\mathbb{R}$) where the cells are

$$CAD(\emptyset) : \{x < x_0\}, \{x_0\}, \{x_0 < x < x_1\}, \{x_1\}, \{x_1 < x < x_2\}, \{x_2\}, \{x_2 < x < x_3\}, \{x_3\}, \{x_3 < x\}. \tag{10}$$

Note that these cells are ordered in a natural way, and their dimensions alternate between 1- and 0-dimension. This pattern will be repeated.

Next, for each cell C' of (10), we consider the **cylinder** $C' \times \mathbb{R}$. We now look at the intersection of the F-invariant regions with this cylinder. This will decompose $C' \times \mathbb{R}$ into a sequence of cells. If case C' is a 0-cell, we obtain another sequence analoguous of (10). In case C' is a 1-cell, we obtain a sequence

$$\begin{aligned} CAD(C') : \quad & \{y < g_0(x)\}, \{g_0(x) = 0\}, \{g_0(x) < y < g_1(x)\}, \{g_1(x) = 0\}, \{g_1(x) < y < g_2(x)\}, \\ & \{g_2(x) = 0\}, \{g_2(x) < y < g_3(x)\}, \{g_3(x) = 0\}, \{g_3(x) < y\}. \end{aligned} \tag{11}$$

where we write "$\{g_i(x) < y < g_{i+1}(x)\}$" as shorthand for the set $\{(x, y) : x \in C', g_i(x) < y < g_{i+1}(x)\}$. Here, each $g_i : C' \to \mathbb{R}$ is an implicit function locally determined by either $f = 0$ or $g = 0$ over the cell C'. Note that the dimension of the cells in (11) alternate between 2- and 1-dimensions.

The union of all the cells of in $CAD(C')$ as C' range over $CAD(\emptyset)$ will constitute our desired decomposition of $\mathbb{R}^2$. We will call this set of cells the **F-cylindrical decomposition** or a **cylindrical algebraic decomposition** (CAD) of $\mathbb{R}^2$.

¶14. Tarski's Theorem. To understand the significance of the cylindrical nature of a CAD, we return to the logical roots of this definition. Consider the following first order sentence in the theory of closed real fields:

$$\phi_1 : \quad (\forall X)(\exists Y)[(X > Y) \wedge ((X-1)^2 + (Y-1)^2 > 1)]. \tag{12}$$

How can we check if this sentence is true? It is not that easy without some geometric insights! But suppose we rewrite it in the following form:

$$\phi_1 : \quad (\forall X)(\exists Y)[(f(X, Y) > 0) \wedge (g(X, Y) > 0)]$$

where f, g are the polynomials in our example of Figure 3(a). We will call

$$M(X, Y) : (f(X, Y) > 0) \wedge (g(X, Y) > 0)$$

the **matrix** of this form of ϕ_1. Then we begin to see this geometrically: it sufficies to search among the cells of the CAD for $\{f, g\}$. Indeed, searching for all x amounts to scanning each cell C' of $CAD(\emptyset)$ of (10). For each C', we want to know if there is a cell of $CAD(C')$ for which $f > 0$ and $g > 0$. In each case, we can see that there is such a cell. Hence ϕ_1 is a valid.

To make the search more explicit, we write it as a doubly nested loop:

```
(∀X)(∃Y)[M(X,Y)]:
    for C' ∈ CAD(∅)
        Found ← false
        for C ∈ CAD(C')
            if (M(C))
                Found ← true; Break;
        if (not Found), return(false)
return(true)
```

In the inner for-loop, we write "$M(C)$" for the if-clause. This needs to be explained: recall that $M(X, Y)$ is $(f(X, Y) > 0) \wedge (g(X, Y) > 0)$. Then $M(C)$ is interpreted as the clause "$(f(C) > 0) \wedge (g(C) > 0)$". More

precisely, we can pick *any* $(x_0, y_0) \in C$ and evaluate "$(f(x_0, y_0) > 0) \wedge (g(x_0, y_0) > 0)$". This outcome of this predicate evaluation does not depend of the particular choice of (x_0, y_0) since $M(X, Y)$ is sign-invariant on C.

Next, consider the following sentences:

$$\begin{aligned}
\phi_2 : &\quad (\forall X)(\exists Y)[(f > 0) \wedge (g < 0)],\\
\phi_3 : &\quad (\exists X)(\forall Y)[(f > 0) \wedge (g > 0)],\\
\phi_4 : &\quad (\exists X)(\forall Y)[(f > 0) \wedge (g < 0)].
\end{aligned}$$

For ϕ_2, our search in $CAD(C')$ fails when $C' = \{x < x_0\}$. Therefore ϕ_2 is invalid. The previous doubly nested loop can be used, except that we replace the if clause by $(f(C) > 0) \wedge (g(C) < 0)$. For ϕ_3 and ϕ_4, we use a somewhat different doubly nested loop:

```
(∃X)(∀Y)[M(X,Y)]:
    for C' ∈ CAD(∅)
        Found ← true
        for C ∈ CAD(C')
            if not M(C)
                Found ← false; Break;
        if(Found) return(true);
```

A point $\alpha \in C$ is called a **sample point** of C. We see that in decision problem for sentences, the availability of a sample point $(x_0, y_0) \in C$ for each cell is extremely useful: the abstract predicate $M(C)$ can be replaced by the explicit predicate $M(x_0, y_0)$. We shall see in our algorithms that such sample points can be recursively constructed.

In general, for any sentence ϕ, we can put it into the prenex form:

$$(Q_1 X_1)(Q_2 X_2) \cdots (Q_n X_n)[M(X_1, \ldots, X_n)] \tag{13}$$

where all the quantifiers $Q_1, \ldots, Q_n$ appear as a prefix, followed by the matrix $M(X_1, \ldots, X_n)$. This matrix is a Boolean combination of atomic formulas of the form

$$f_i(X_1, \ldots, X_n) \circ 0, \qquad i = 1, \ldots, m$$

where f_i is a polynomial and $\circ \in \{<, >, =, \neq, \geq, \leq\}$. Let $F = \{f_1, \ldots, f_m\}$ be all these polynomials. Then we can compute a F-cylindrical decomposition of $\mathbb{R}^n$. The truth of ϕ can be reduced to a n-nested loop to search the cells of the decomposition. Thus, being able to compute CAD's will allow us to decide any sentence of this theory. This is Tarski's fundamental decidability result.

¶15. Resultants as Projection. We want to interpret Lemma 2 as saying that the resultant is a kind of projection operator.

In general, if p, q are multivariate polynomials, we write $\mathtt{res}_Y(p, q)$ for the resultant in which we treat p, q as univariate polynomials in one of the variables Y.

In the following two lemmas, let $p, q \in \mathbb{Z}[X, Y]$, and

$$r(X) = \mathtt{res}_Y(p, q) \in \mathbb{Z}[X].$$

Moreover, let $\mathtt{lead}_Y(p), \mathtt{lead}_Y(q) \in \mathbb{Z}[X]$ denote the leading coefficients of p, q, viewed as polynomials in Y.

Before we apply Lemma 2, we prove a helper lemma:

Lemma 7. *Set*

$$p_0(Y) := p(\alpha_0, Y), \qquad q_0(Y) := q(\alpha_0, Y)$$

for some $\alpha_0 \in \mathbb{C}$. *If* $\mathtt{lead}_Y(p)(\alpha_0) \neq 0$ *or* $\mathtt{lead}_Y(q)(\alpha_0) \neq 0$, *then*

$$r(\alpha_0) = \mathtt{lead}_Y(p)(\alpha_0)^m \mathtt{res}(p_0, q_0) \tag{14}$$

for some $m \geq 0$.

Proof. Wlog, assume $\texttt{lead}_Y(p)(\alpha_0) \neq 0$. Thus $\deg(p_0) = \deg_Y(p)$. However, $\deg(q_0) = \deg_Y(q) - m$ for some $m \geq 0$. Clearly, $r(X) = \det S(X)$ where $S(X)$ is the Sylvester matrix of p, q. So $r(\alpha_0) = \det S(\alpha_0)$. But $\texttt{res}(p_0, q_0) = \det S_0$ for another Sylvester matrix. Moreover, S_0 is obtained from $S(\alpha_0)$ by deleting the first m rows and first m columns of $S(\alpha_0)$. From the shape of the Sylvester matrices, our claim follows. **Q.E.D.**

Lemma 8 (Projection Lemma). *et $\alpha_0 \in \mathbb{C}$ and $r(X)$ be non-vanishing. Then the following two statements are equivalent:*
(i) $r(\alpha_0) = 0$.
(ii) Either
(ii-a) there exists $\beta_0 \in \mathbb{C}$ such that $p(\alpha_0, \beta_0) = q(\alpha_0, \beta_0) = 0$, or
(ii-b) $\texttt{lead}_Y(p)(\alpha_0) = \texttt{lead}_Y(q)(\alpha_0) = 0$.

Proof. (i)⇒(ii): Suppose $r(\alpha_0) = 0$. We may assume that (ii-b) does not hold, i.e., $\texttt{lead}_Y(p)(\alpha_0) \neq 0$ or $\texttt{lead}_Y(q)(\alpha_0) \neq 0$. Otherwise, by Lemma 7, we see that $r(\alpha_0) = 0$ iff $\texttt{res}(p_0, q_0) = 0$. Hence $\texttt{res}(p_0, q_0) = 0$. By Lemma 2, we further conclude that $\texttt{GCD}(p_0, q_0) = g(Y)$ for some polynomial $g(Y)$ with positive degree. Thus, there exists $\beta_0 \in \mathbb{C}$ such that $g(\beta_0) = 0$. This implies $p_0(\beta_0) = q_0(\beta_0) = 0$. This implies condition (ii).

(ii)⇒(i): Conversely, suppose (ii) holds. As in the proof of Lemma 7, let $S(X)$ be the Sylvester matrix of p, q such that $r(X) = \det S(X)$. If $\texttt{lead}_Y(p)(\alpha_0) = \texttt{lead}_Y(q)(\alpha_0) = 0$, then it is clear that $r(\alpha_0) = \det S(\alpha_0) = 0$ since the first column of $S(\alpha_0)$ is entirely zero. Hence, we may assume from (ii) that there exists $\beta_0 \in \mathbb{C}$ such that $p_0(\beta_0) = q_0(\beta_0)$. This implies $\texttt{GCD}(p_0, q_0)$ is non-constant. By Lemma 2, this means $\texttt{res}(p_0, q_0) = 0$. Moreover, (14) implies that $r(\alpha_0) = 0$, as desired. **Q.E.D.**

The following corollary shows why this is called a projection lemma:

Corollary 9. *Let $p, q \in \mathbb{Z}[X, Y]$. If $\texttt{res}_Y(p, q)$ does not vanish, then the set $\texttt{Zero}(res_Y(p, q)$ contains the x-projections of $\texttt{Zero}(p, q)$.*

Proof. Suppose $p(\alpha_0, \beta_0) = q(\alpha_0, \beta_0) = 0$. Then $\texttt{GCD}(p_0, q_0)$ has positive degree, and hence $\texttt{res}_Y(p_0, q_0) = 0$ **Q.E.D.**

Next assume $\texttt{res}_Y(p, p_Y)$ is non-vanishing. According to our corollary, $\texttt{Zero}(res_Y(p, p_Y)$ contains the projections of all the x-extremal points of the curve $p = 0$.

REMARK: An important remark is that even if α_0 is real, this lemma does not guarantee that β_0 would be real. For each real zero α_0 of $\texttt{res}_Y(p, q)$, we want to "lift" α_0 to the *real* point (α_0, β_0) that satisfy $p = q = 0$. To do this lifting process, we look at each real zero α_0 of $\texttt{res}_Y(p, q)$. However, many of these α_0 do not lift to any real points! This can fail for two reasons: either it lift to a complex point (α_0, β_0) or this α_0 satisfy condition (ii-b).

¶16. Language of CAD. We now introduce some useful terminology, in order to give a general description of CAD's.

Let $C \subseteq \mathbb{R}^{n-1}$ be any set. Then the **cylinder** over C is the set $C \times \mathbb{R} \subseteq \mathbb{R}^n$. By a **stack** over C we mean a decomposition of $C \times \mathbb{R}$ into an odd number of cells,

$$C_1 < C_2 < \cdots < C_{2m+1}$$

where for each $i = 1, \ldots, m$:

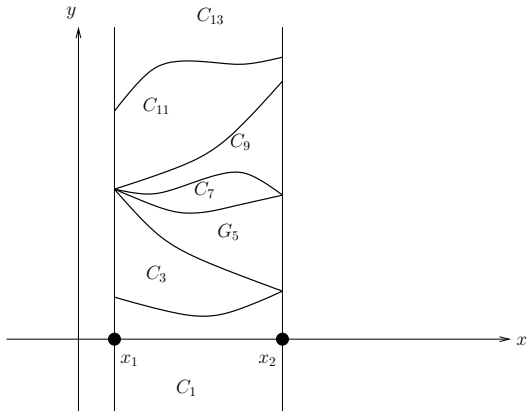
- C_{2i} is called a **section** and is the graph of a real function g_i on C,

$$g_i : C \to \mathbb{R}.$$

- C_{2i-1} is called a **sector** and is given by

$$\{(p, y) \in \mathbb{R}^n : p \in C, g_{i-1}(p) < y < g_{i+1}(p)\}$$

For uniformity in notation, we may assume $g_0 = -\infty$ and $g_{2m+2} = \infty$ in this definition.



(b)

Figure 4: A Stack over C

Such a stack is illustrated in Figure 4.

We now define a **cylindrical decomposition** (CD) to be any finite partition D of $\mathbb{R}^n$ into a collection of cells with the following properties. If $n = 1$, any finite partition of $\mathbb{R}$ into points and open intervals is a CD. Recursively, for $n \geq 2$, there exists a CD D' of $\mathbb{R}^{n-1}$ such that:

- For each $C \in D$, there exists a $C' \in D'$ such that C projects onto C', i.e., $\pi_n(C) = C'$.
- For each $C' \in D'$, the set of $C \in D$ that projects onto C' forms a stack over C'.

If $\pi_n(C) = C'$, we say C is a **child** of C'. We call D' a **projection** of D; conversely D is called an **extension** of D'.

¶17. Projection Operator in the plane. Let $F \subseteq \mathbb{Z}[X_1, \ldots, X_n]$, and D be a CD of $\mathbb{R}^n$.

If each cell C of D is F-invariant, we call D a **cylindrical decomposition** for F. How shall we compute D? If $n = 1$, this is easy (recall that we know how to isolate zeros of polynomials and represent them). Assume $n \geq 2$. The strategy is that we first construct another set $PROJ(F) \subseteq \mathbb{Z}[X_1, \ldots, X_{n-1}]$ such that a cylindrical decomposition D' for $PROJ(F)$ can be extended into a cylindrical decomposition D for F.

The property that makes this possible is the notion of "delineability". We say a function $f : \mathbb{R}^n \to \mathbb{R}$ is **delineable** over a set $S \subseteq \mathbb{R}^{n-1}$ if the set of real zeros of f, restricted to the cylinder $S \times \mathbb{R}$ define the graphs of a finite number of functions of the form

$$g_i : S \to \mathbb{R} \quad (i = 1, \ldots, k)$$

with the following properties:
(i) For $p \in S$,

$$g_1(p) < g_2(p) < \cdots < g_k(p).$$

(ii) For each $i = 1, \ldots, k$, there is a positive integer m_i such that $g_i(p)$ is a root of the polynomial $f(p, X_n)$ of multiplicity m_i.

The graphs of the functions g_i has the form $\{(p, g_i(p)) : p \in S\}$ and are called **f-sections**. These f-sections partition the cylinder $S \times \mathbb{R}$ into sets of the form

$$\{(p, z) : g_i(p) < z < g_{i+1}(p)\}$$

for $i = 0, \ldots, k$. Here, we assume $g_0(p) = -\infty$ and $g_{k+1}(p) = +\infty$. We call these the **f-sectors**. Thus, the cylinder $S \times \mathbb{R}$ is partitioned into k sections and $k + 1$ sectors. If S is a $(n-1)$-cell, then each section is an $(n-1)$-cell, and each sector is an n-cell.

The **order** of a real analytic function $f : \mathbb{R}^n \to \mathbb{R}$ at a point $p \in \mathbb{R}$ is the least k such that some k-th partial derivative of f does not vanish at p. If there is no such k, then the order is ∞. We say f is **order invariant** on a set $S \subseteq \mathbb{R}^n$ if the order of f is constant for all $p \in S$.

Observe that the f-sections and f-sectors are automatically f-sign invariant. We have the following theorem from McCallum:

Theorem 10 (McCallum). *Let $n \geq 2$ and $\overline{X} = (X_1, \ldots, X_{n-1})$. Let $f(\overline{X}, X_n)$ be a polynomial in $\mathbb{R}[\overline{X}, X_n]$ of positive X_n degree. Let $D(\overline{X})$ be the discriminant of $f(\overline{X}, X_n)$ and $D(\overline{X})$ is non-zero. If f is degree-invariant and non-vanishing on $S \subseteq \mathbb{R}^{n-1}$, and D is order-invariant on S, then f is delineable on S and order-invariant on each f-section on S.*

The use of order-invariance in this theorem is somewhat incompatible with the usual emphasis on sign-invariance. Nevertheless, this theorem also shows that order-invariance can be propagated from sign-invariance. Based on this theorem, we may define the following: let F be a **squarefree basis** in $\mathbb{Z}[X_1, \ldots, X_n]$. By squarefree basis, we mean that the polynomials have positive degree in X_n, are primitive, squarefree and pairwise relatively prime. The **projection** of F is the set $PROJ(F) \subseteq \mathbb{Z}[X_1, \ldots, X_{n-1}]$ formed by the union of the following three sets:
(i) $coeff(F)$ is the set of all non-zero coefficients of $f \in F$,
(ii) $disc(F)$ is the set of all non-zero discriminants of $f \in F$,
(iii) $res(F)$ is the set of all non-zero $\mathtt{res}_{X_n}(f, g)$ for $f, g \in F$.

¶18. Additional Issues in CAD. This area has remained an active research area even today (2009). One of the most pressing issue is to improve the complexity of CAD. The number of cells is easily seen to be bounded by a double exponential in n, the number of variables. This double exponential complexity is inherent [Davenport and Heintz, 1987]. Although the worst case complexity of CAD construction has improved dramatically over the years, this is still a bottleneck for widespread applications. There are several ways to improve the complexity. For instance, it is clear that the prenex form is in some sense the worst way to decide sentences – most natural sentences have local structures that can be decoupled and solved separately and combined in an effective way. A simple observation is that CAD size is double exponential, not in n (number of variables), but in the number of alternations of quantifiers in the prenex sentence. This means that, for instance, those with no alternation of quantifiers can be solved much faster (e.g., in polynomial space). Another direction is the development of numerical techniques for CAD [Hong, Collins-McCullum, etc]. The variables in a sentence are not completely independent – for instance, if we discuss the geometry of n points in Euclidean d-space, there are nd variables. This fact can be exploited. In 1983, Schwartz and Sharir applied CAD techniques to the problem of robot motion planning. This introduced an additional issue in CAD construction: the need to determine adjacencies between cells. Note that in this application, the sign-invariant regions are primary, and there is no need to have cylindrical decomposition. This can reduce the number of regions to single exponential.

Exercises

Exercise 5.1: Consider the sentence ϕ in the prenex form of (13). Describe the n-nested loop to evaluate the truth of ϕ given a CAD for the polynomials in the matrix of ϕ. ♢

Exercise 5.2: (Ellipse Problems) Consider the ellipse

$$E : \frac{(X-c)^2}{a^2} + \frac{(Y-d)^2}{b^2} = 1.$$

What are the conditions on a, b, c, d so that E is contained in the unit circle $X^2 + Y^2 = 1$. This can be formulated as the sentence

$$(\forall X)(\forall Y)[\frac{(X-c)^2}{a^2} + \frac{(Y-d)^2}{b^2} \leq 1. \Rightarrow .X^2 + Y^2 \leq 1].$$

Describe how we can use CAD to solve this problem.
NOTE: Lazard (1987) succeeded in solving this with the help of MACSYMA. This problem could not be solved by the available CAD software at that time. ♢

Exercise 5.3: In many applications, not all the real variables in a first order sentence are independent. The most important situation is when the variables are naturally grouped into k-tuples. For instance, let $k = 2$, and our sentence is about points in the plane. How can we take advantage of this in constructing CAD? ♢

End Exercises

§6. APPENDIX: Subresultant Theory

In order to understand Collin's theory of CAD, we will delve deeper into Sturm sequences. Indeed, Tarski's original decision procedure was viewed as a generalization of Sturm sequences. For computational efficiency, we also need to delve into the theory of subresultants. For a more complete treatment, see [7].

A ring R is an **ordered ring** if it contains a subset $P \subseteq R$ with the property that P is closed under addition and multiplication, and for all $x \in R$, exactly one of the following three conditions hold:

$$x = 0, \quad x \in P, \quad -x \in P.$$

Such a set P is called a **positive set** of R. Then R is totally ordered by the relation $x < y$ iff $y - x \in P$.

Throughout the following development, assume D is a UFD that is also ordered. For instance, $D = R[X_1, \ldots, X_n]$ where $R \subseteq \mathbb{R}$ is a UFD. In this case, the positive set $P \subseteq D$ can be taken to be those polynomials whose leading coefficient is a positive number in D.

Why is efficiency a problem for computing the standard Sturm sequences in $D[X]$? The reason is that such sequences assumes that D is a field. If D is not a field, we must replace D by its quotient field, $Q(D)$. In order words, when we compute the remainder of polynomials, $A, B \in D[X]$, the result, $\texttt{rem}(A, B)$ will in general be an element of $Q(D)[X]$, not of $D[X]$. E.g, from $D = \mathbb{Z}$ we must go to $Q(D) = \mathbb{Q}$. This turns out to be very inefficient [7]. We then proceed as follows:

Suppose $A, B \in D[X]$. If $\deg A \geq \deg B$, let us define the **pseudo-remainder** of A, B to be the remainder of $b^{\delta+1}A$ divided by B, where $b = \texttt{lead}(B)$ and $\delta = \deg A - \deg B$, i.e.,

$$\texttt{prem}(A, B) := \texttt{rem}(b^{\delta+1}A, B).$$

It is not hard to see, by looking at the process of long division of polynomials, to see that $\texttt{prem}(A, B) \in D[X]$. If $\deg A < \deg B$, then $\texttt{prem}(A, B) := A$.

Given $A, B \in D[X]$, we define a **polynomial remainder sequence** (PRS) of A, B to be a sequence of polynomials

$$(A_0, A_1, \ldots, A_h)$$

where $A_0 = A, A_1 = B$ and for $i \geq 1$,

$$\beta_i A_{i+1} = \alpha_i A_{i-1} + Q_i A_i \tag{15}$$

for some $\beta_i, \alpha_i \in D$ and $Q_i \in D[X]$ and $\deg A_{i-1} < \deg A_i$. Moreover, the termination of the PRS at A_h is determined by the condition that $A_{h+1} = 0$ where A_{h+1} it is defined by (15) from A_{h-1} and A_h. This h is defined because the degree of the A_i is strictly decreasing.

There are many ways to form the PRS of A, B. One way is to use pseudo remainders is to choose (15) to be:

$$A_{i+1} = \texttt{prem}(A_{i-1}, A_i). \tag{16}$$

This is called the **Pseudo PRS** of A, B. For $D = \mathbb{Z}$, the pseudo PRS will contain generally exponentially large coefficients, and this is not practical. So, instead, we can replace (16) by

$$A_{i+1} = \texttt{prim}(\texttt{prem}(A_{i-1}, A_i)). \tag{17}$$

Here, $\texttt{prim}(A)$ is the **primitive part** of A, defined to be $A/\texttt{cont}(A)$ where $\texttt{cont}(A)$ is the GCD of all the coefficients of A. This produces a PRS that is optimal in terms of coefficient sizes. However, computing $\texttt{prim}(A)$ is quite expensive.

Let $(\beta_1, \ldots, \beta_{h-1})$ where each $\beta_i \in D$. We say that a PRS $(A_0, \ldots, A_h)$ for A, B is **based on** $(\beta_1, \ldots, \beta_{h-1})$ if for $i \geq 1$,

$$A_{i+1} = \frac{\texttt{prem}(A_{i-1}, A_i)}{\beta_i}. \tag{18}$$

We will describe an algorithm for computing a PRS based on a suitable sequence of β_i's in which the β_i's are easy to compute and the coefficients of the PRS remains polynomially bounded in terms of $\deg A_0$. See [7] for proofs.

Here is the algorithm:

```
SubPRS Algorithm
Input A, B ∈ D[X], deg(A) ≥ deg(B) > 0
Output PRS (A_0, ..., A_h) for A, B
▷ INITIALIZATION
        A_0 ← A;   A_1 ← B
        a_0 ← lc(A_0);   a_1 ← lc(A_1)
        d_1 ← deg(A_0) − deg(A_1)
        ψ_0 ← 1;   ψ_1 ← a_1^{d_1}
        β_1 ← (−1)^{d_1+1}
▷ LOOP
    for (i = 1;   true;   i++)
        A_{i+1} ← prem(A_{i−1}, A_i) / β_i      ◁ Exact Division
        a_{i+1} ← lc(A_{i+1})
        if (a_{i+1} = 0)
            h ← i;   break      ◁ Exit For-Loop
        d_{i+1} ← deg(A_i) − deg(A_{i+1})
        ψ_{i+1} ← ψ_i (a_{i+1}/ψ_i)^{d_{i+1}}
        β_{i+1} ← (−1)^{d_{i+1}+1} (ψ_i)^{d_i+1} a_i
```

References

[1] S.-S. Chern. What is Geometry? *Amer. Math. Monthly*, 97(8):679–686, 1990.

[2] J. Dieudonné. *History of Algebraic Geometry*. Wadsworth Advanced Books & Software, Monterey, CA, 1985. Trans. from French by Judith D. Sally.

[3] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[4] D. Hilbert and S. Cohn-Vossen. *Geometry and the Imagination*. Chelsea, 2nd edition edition, 1952.

[5] C. M. Hoffmann. *Geometric and Solid Modeling: an Introduction*. Morgan Kaufmann Publishers, Inc., San Mateo, California 94403, 1989.

[6] A. Tarski. What is Elementary Geometry? In L. Brouwer, E. Beth, and A. Heyting, editors, *Studies in Logic and the Foundations of Mathematics – The Axiomatic Method with Special Reference to Geometry and Physics*, pages 16–29. North-Holland Publishing Company, Amsterdam, 1959.

[7] C. K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.

[8] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[9] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.

[10] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.

Lecture 7
Voronoi Diagrams

We introduce a critical idea of EGC, the technique of root bounds. This will be addressed through the study of Fortune's algorithm for computing the Voronoi diagram of a set of points. The algorithms is based on the plane sweep paradigm. But unlike the algorithms we have seen until now, this one has an interesting twist – it requires the computation of square roots. Thus the direct use of a Big Rational package will not necessarily give us the geometric exactness we need. This leads us into the theory of constructive root bounds.

§1. Introduction

Consider a finite set $S \subseteq \mathbb{R}^2$. We define a certain skeleton (comprising a set of straightline segments) called the **Voronoi diagram** of S. See Figure 1 for an example of such a diagram where $|S| = 11$. The most striking feature of the diagram is the fact that it subdivides the plane into a set of $|S|$ polygonal regions, with each region containing a unique point s of S.

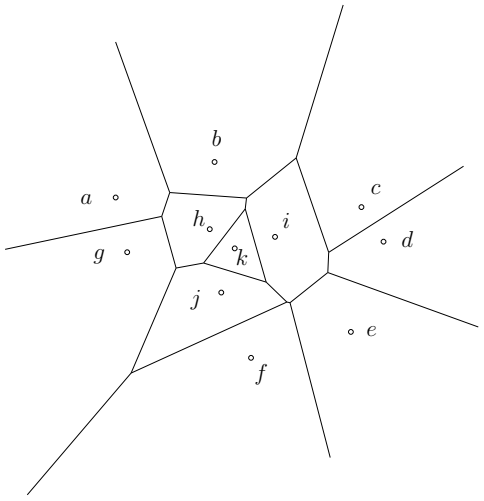


Figure 1: Voronoi diagram of a set $\{a, b, c, \ldots, j, k\}$ of 11 points

We introduce some usual terminology. The points in S is called **sites** because in some applications, S can be viewed as locations on a map. For instance, each site $s \in S$ may be the location of a postoffice, and the polygonal region containing s comprises those points q in the plane that are closest to this postoffice. Then a basic computational problem is to construct a data structure $D(S)$ that can quickly answer, for any query point $q \in \mathbb{R}^2$, the post office $s \in S$ that is closest to q. This is the well-known **Post Office Problem**.

The regions defined by the Voronoi diagram can be precisely discribed as follows: for $p, q \in \mathbb{R}^2$, let $d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$ denote Euclidean distance between them. If $p \neq q$, let $b(p, q)$ denote the **bisector line** between p and q: this is the line that passes through the point $(p + q)/2$ and is normal to the segment $[p, q]$. Also, let $h(p, q)$ denote the open half-space bounded by $b(p, q)$ and containing p. When $p = q$, it is convenient to define $h(p, q)$ to be the entire plane. So $h(p, q) \neq h(q, p)$ unless $p = q$.

Although we have used $\mathbb{R}^2$ for illustration, these concepts are easily extended to any dimension. Fix a non-empty finite set $S \subseteq \mathbb{R}^n$. Define the the **Voronoi region of** s to be

$$V(s) = V_S(s) := \bigcap \{h(s, s') : s' \in S\}.$$

Since $V(s)$ is the intersection of half-spaces, it is is a convex polygonal set. It is non-empty since $s \in V(s)$. The corners of the boundary of $V_S(s)$ are called **Voronoi vertices** and the maximal open line segments bounding $V_S(s)$ are called **Voronoi edges**. Note that a Voronoi edge can be a finite segment, a half-line or a line. Let $K_0(S)$, $K_1(S)$ and $K_2(S)$ denote the set of Voronoi vertices, Voronoi edges and Voronoi regions, respectively.

Let us analyze the Voronoi facets (i.e., vertices, edges, regions) which we just defined. For any point p, let $d(p, S) = \min\{d(p, q) : q \in S\}$ and $C_S(p)$ denote the circle centered at p with radius $d(p, S)$. We call $C_S(p)$ the **largest empty circle** at p because the interior of $C_S(p)$ has empty intersection with S. On the other hand, the set $C_S(p) \cap S$ is non-empty, and the cardinality $|C_S(p) \cap S|$ yields very important information: we classify the point p according to this cardinality.

Lemma 1. *Suppose $p \in \mathbb{R}^2$.*

- *If $C_S(p) \cap S = \{s\}$ then p lies in the Voronoi cell $V_S(s)$.*
- *If $C_S(p) \cap S = \{s, s'\}$ where $s \neq s'$ then p lies on a Voronoi edge. In fact, $p \in b(s, s')$.*
- *If $|C_S(p) \cap S| \geq 3$ then p is a Voronoi vertex. In this case, p is the circumcenter of any three distinct sites $s, s', s'' \in C_S(p) \cap S$.*

We leave the proof as an Exercise.

> **Cell Complex.** Voronoi diagrams are best viewed as a special kind of cell complex. A **d-cell** is any subset of $\mathbb{R}^n$ that is homeomorphic to the open d-ball $B^d = \{p \in \mathbb{R}^d : \|p\| < 1\}$ ($d \geq 1$); a **0-cell** is just a singleton set. Also, the closure of B^d, the closed d-ball, is denoted $\overline{B}^d$. Let K be any non-empty finite collection of pairwise disjoint cells. The **support** of K is $|K| := \cup K \subseteq \mathbb{R}^n$. We call K a **cell complex** if, for each d-cell $C \in K$, $d \geq 1$, there is a continuous function $f_C : \overline{B}^d \to |K|$ such that the restriction of f_C to B^d is a homeomorphism between B^d and C. It can be shown [6, p. 215] that this implies that $f_C(\dot{B}^d)$ is equal to a union of cells in K.

A cell complex K in which every 2-facet is a convex is called a convex subdivision.

Theorem 2. *The union $K(S) = K_0(S) \cup K_1(S) \cup K_2(S)$ is a convex subdivision of the plane.*

Proof. By the previous lemma, p lies in a Voronoi region or a Voronoi edge or a Voronoi vertex depending on whether $|C_S(p) \cap S|$ is 1, 2 or ≥ 3. These conditions are mutually disjoint. Hence K is pairwise disjoint.

By definition, the boundary of each Voronoi region is a union of Voronoi edges and Voronoi vertices. Hence K is a complex.

Since each region is a convex, we conclude that K is a convex subdivision. **Q.E.D.**

In view of this theorem, $K(S)$ is called the **Voronoi complex** of S. Each $f \in K(S)$ is a **Voronoi facet**. The skeleton $K_0(S) \cup K_1(S)$ of this complex is called the **Voronoi diagram** and denoted $Vor(S)$. The Voronoi diagram can also refer to the support of this skeleton; this abuse of language should not cause confusion.

Lemma 3. *The following are equivalent for $|S| \geq 3$:*
(i) The Voronoi diagram of S is connected
(ii) S does not lie on a line.
(iii) Every Voronoi region has a connected boundary.

Proof. (iii) implies (ii): If S lies on a line, then the Voronoi diagram has no vertices and comprises $|S| - 1$ parallel lines.

(ii) implies (iii): A Voronoi region $V(s)$ is star-shaped with respect to s. If $V(s)$ is bounded, then clearly its boundary is connected. So let $V(s)$ be unbounded. Since S does not lie on a line, there exists $s', s'' \in S$ such that (s, s', s'') that forms a non-degenerate triangle. Then $h(s, s') \cap h(s, s'')$ forms a wedge W which contains $V(s)$. Being star-shaped and unbounded, there is a non-empty maximal cone $C \subseteq V(s)$ of rays emanating from s. Thus $C \subseteq W$. [It is easy to argue that the union of set of rays from s must form a

connected set.] It follows that if we shoot any ray from s in a direction outside this cone, the ray will hit a boundary of $V(s)$. Thus the boundary of $V(s)$ is connected.

(iii) implies (i): Let p, q be two points of the Voronoi diagram. Consider the straightline from p to q. It passes through an alternating sequence of Voronoi regions and points on the Voronoi diagram. Let this alternating sequence be $p_0, R_1, p_1, R_2, \ldots, R_k, p_k$ where $p = p_0$ and $q = p_k$. Since each R_i has connected boundary, it means p_{i-1} is connected to p_i. Thus $p = p_0$ is connected to $q = p_k$.

(i) implies (iii): If a Voronoi region $V(s)$ does not have a connected boundary, then it must be bounded by two parallel lines. These two lines determine two connected components of the Voronoi diagram of S. Thus the Voronoi diagram is not connected. **Q.E.D.**

Theorem 4. *Let $|S| = n$. The number of Voronoi vertices is at most $2n - 5$ and the number of Voronoi edges is at most $3n - 6$.*

For each $n \geq 3$, these bounds can be achieved simultaneously by a single configuration.

Proof. If S lies on a line, the result is easy to see. Otherwise, we know that the skeleton is connected. Let v be the number of Voronoi vertices, e the number of Voronoi edges.

Let C be a circle large enough to contain all the Voronoi vertices and sites. We create a new vertex v_∞ outside C, and redirect each infinite Voronoi edge so that, outside of C, they each connect to v_∞ along pairwise disjoint curvilinear paths. See Figure 2.

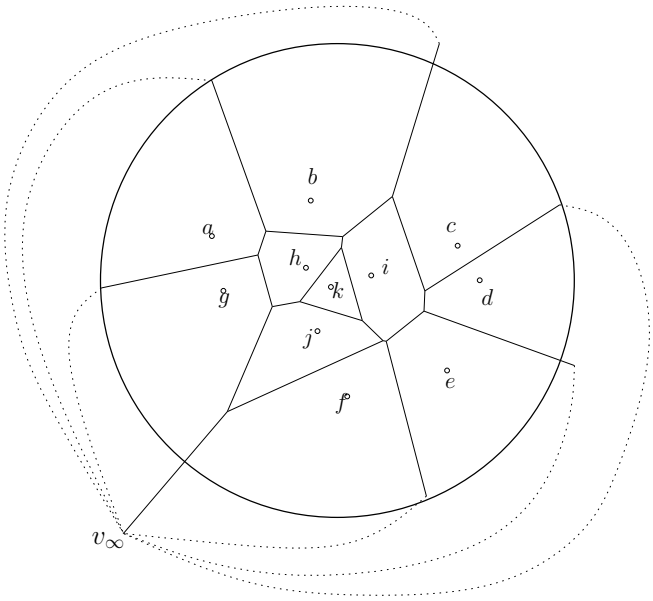


Figure 2: Graph with new vertex v_∞

This creates a connected graph G whose complement defines a set of connected regions. These regions has a one-one correspondence with the Voronoi regions. We now use the Euler formula for a planar graph embedded in the plane,

$$\nu_0 - \nu_1 + \nu_2 = 1 + \beta$$

where ν_i is the number of cells of dimension i and β the number of connected components. Since there are n Voronoi regions, this means $\nu_2(G) = n$. Similarly, we have $\nu_0(G) = v + 1$ (because of the vertex v_∞) and $\nu_1(G) = e$ (the new curves we added are considered part of the infinite edges which we modified). Finally, since G is connected, $\beta(G) = 1$. Thus, the Euler formula becomes

$$(v + 1) - e + n = 2.$$

We need another relation between v and e to get bounds on v and e. Let us count the number I of (vertex, edge) incidences. Counting from the viewpoint of edges, $I = 2e$ since each edge contributes two incidences.

Counting from the viewpoint of vertices, $I \geq 3(v+1)$ since each vertex yields at least three incidences. Thus $2e \geq 3v+3$. Plugging this into the Euler formula, we can either eliminate e or v. The result are the bounds on v and e claimed in this theorem.

We now show that the bounds of the theorem is tight. When $n = 3$, we let S comprise the vertices of an equilateral triangle. Then $v = 1$ and $e = 3$ achieve the bounds of the theorem. Now we can incrementally add successive sites to S so that each addition creates increases the number of Voronoi vertices by 2 and increases the number of Voronoi edges by 3.

For instance, for $n = 4$, we can place the next vertex right at the location of the unique Voronoi vertex (replacing it with 3 new Voronoi vertices), and adding three new Voronoi edges. Alternatively, the next three sites ($n = 4, 5, 6$) can be placed so that these points form the vertices of regular hexagon. In either case, the process can be continued indefinitely. **Q.E.D.**

§2. Fortune's Linesweep

Fix a set S of sites. Again consider a horizontal line $H(t)$ that sweeps from $t = -\infty$ to $t = \infty$. The key idea is to define a curve called the **beach line** $B(t)$ of $H(t)$: the set $B(t)$ comprises those points p such that

$$d(p, H(t)) = d(p, S(t))$$

where $S(t)$ denotes those points that has been swept by time t, i.e., points lying on or below $H(t)$. This is illustrated in Figure 3.

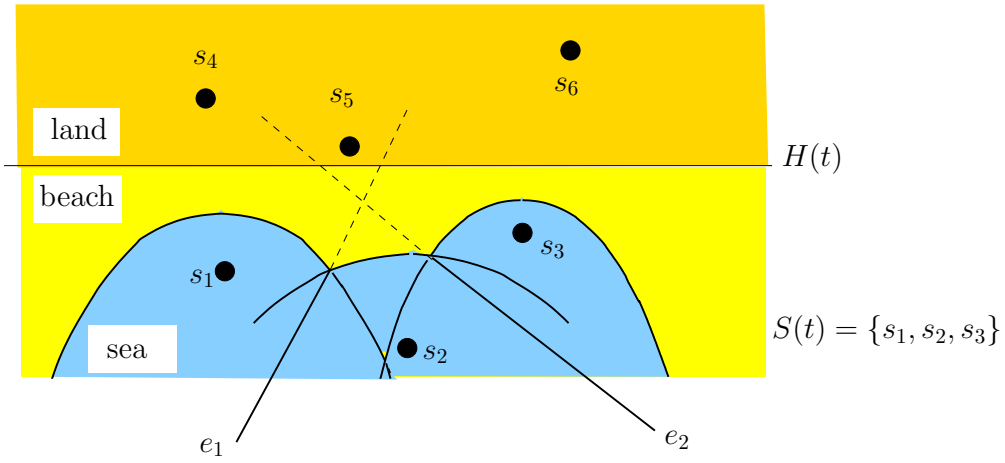


Figure 3: Beach line defined by active sites s_1, s_2, s_3

The next lemma will give a characterization $B(t)$. For each site $s \in S(t)$, define $B_s(t)$ to be the parabola defined by the line $H(t)$ as directrix and s as focus. In Figure 3, the parabolas defined by s_1, s_2, s_3 are indicated.

Lemma 5. *The beach line $B(t)$ is the upper envelop of the set $\{B_s(t) : s \in S\}$ of parabolas.*

Proof. By a fundamental property of parabolas, $B_s(t)$ is comprised of those points that are equidistant from $H(t)$ and s, i.e., $d(p, H(t)) = d(p, s)$. The upper envelop of these parabolas are therefore those points p which achieve $d(p, H(t)) = d(p, S(t))$. **Q.E.D.**

Thus the beach line is a union of a number of parabolic arcs. Each arc is associated with a site $s \in S$; by definition, such sites are said to be **active**. Active sites may become **inactive** when they no longer define arcs on the beach line. Points on the beach line where two consecutive arcs meet are called **breakpoints**. This beach line divides the plane into an upper part called **land** and a lower part called **sea**. The land itself is divided into two parts: the **beach** which lies between $H(t)$ and the beach line, and the **dry land** which lies above $H(t)$. The sites are thus classified as **wet** or **dry** depending on whether they are on dry land or

in the sea; there are no sites on the beach. Thus, active sites lies in the sea. These concepts are illustrated in Figure 3. As we sweep, we maintain the part of the Voronoi diagram *restricted to the sea.* This **partial Voronoi diagram** need not be connected even if the overall Voronoi diagram is connected. E.g., the partial diagram in Figure 3 is comprised of the two disconnected edges e_1 and e_2.

We next examine some basic properties of the beach line. Our main goal is to understand how the beach line transforms as $t \to \infty$. We are interested in the **critical moments** when $B(t)$ change combinatorially.

P1 *The beach line is x-monotone, and its vertical projection covers the entire x-axis.* This follows from the basic properties of parabolas.

As corollary, we can represent $B(t)$ as a concatenation of parabolic arcs,

$$B(t) = (\alpha_1; \alpha_2; \cdots; \alpha_m) \tag{1}$$

listed in order of increasing x. Suppose α_i is part of the parabola defined by $s_i \in S$. Combinatorially, the beach line can be represented by the sequence

$$(s_1, s_2, \ldots, s_m) \tag{2}$$

of active sites. It is important to realize that a site s may appear more than once in this sequence. However, for each pair s, s' of distinct sites, these sites can be adjacent in (2) at most twice (once in the order s, s' and once in the reverse order s', s). That is because the parabolas defined by s and s' can intersect at most twice. We shall next see how this active sequence can change.

P2 *The sea is monotonically advancing with t.* To see this, let L be any vertical line. Then the intersection $B(t) \cap L$ moves monotonically upward with time.

P3 *Every point p in the sea is closer to some wet site s than to any dry site s'.* In proof, let the line segment $[p, s']$ intersect the beachline at some point p'. Note that p' may not be unique. There is an active site s that is closest to p'. Then

$$d(p, s') = d(p, p') + d(p', s') > d(p, p') + d(p', H(t)) = d(p, p') + d(p', s) \geq d(p, s).$$

P4 *Each break point q moves along a Voronoi edge (hence in straight lines).* In proof, let q be the common endpoint of the arcs generated by the active sites s and s'. To show that q lies on a Voronoi edge determined by s and s' we show that $d(q, s) = d(q, s') \leq d(q, s'')$ for any other site s''. If s'' is on the dry land, this follows from P4. If s'' is in the sea, consider the intersection q' between the vertical line L through q and parabola generated by the sweepline $H(t)$ and s''. Note that q' lies below q, since $B(t)$ is the upper envelop. Then $d(q', s'') = d(q', H(t)) > d(q, H(t)) = d(q, s')$. This proves s'' is not closer than s' to q.

P5 *Each point q on an arc of the beach line belongs to the Voronoi region of the cell that generates the arc.* By a similar argument to the one for P4.

P6 Consider three consecutive arcs $\alpha, \alpha', \alpha''$ separated by two consecutive breakpoints p and q. Let s, s', s'' be the sites that generates these arcs. If α, α'' are part of the same parabola (i.e., $s = s''$), then p, q will be diverging with t. Hence α' cannot disappear. This property is clear from the geometry.

P7 *When an arc α' contracts, the first time when it becomes a point v, there are three distinct sites, s, s', s'' such that v is the Voronoi vertex defined by s, s', s''.* This is called the **circle event** parametrized by (s, s', s''). This terminology will become clearer below. In proof, by P6, we know that $s \neq s''$. By P4, we know that the instant α' disappears, it is simultaneously closest to s, s', s''.

P8 *When a new site first becomes wet, it is active and it defines a degenerate arc that is just a line segment (or a ray, in case this is the first site).* This is called the **site event** parametrized by s.

P9 *The beach line can change combinatorially only with a site event or a circle event.* Pf: By P7 and P8, we know that these events can cause combinatorial change in the beach line. We must exclude all other possibilities of creating new arcs in the beach line. Suppose at time t, a new arc appears at position p on some arc define by an active site s. This new arc must come from some wet site s'. Let L be the vertical line through p. Let the parabola $B_s(t)$ intersects L at the point $p(t)$, and similarly, let $p'(t) = B_{s'}(t) \cap L$. At time t^-, $p(t^-)$ is above $p'(t^-)$ but at time t^+, $p(t^-)$ is below $p'(t^-)$. This is impossible (details omitted).

P10 *A site is initially dry, then active (thus wet). Finally it becomes inactive (still wet).* Pf: in other words, inactive sites do not become active again. After it becomes inactive, its Voronoi region is now contained in the sea. It will remain this way since the sea is advancing monotonically.

P11 A parabola can contribute up to n arcs on the beach line. Such examples are easy to generate.

P12 *The beach line has at most* $2n - 1$ *arcs.* Pf: this is because the only way that new arcs appear is through a site event, and there are n such events. Each such event increase the number of arcs by 2 (except the first site event increases the number by 1).

¶1. The Algorithm. As usual, we sweep the horizontal line $H(t)$ from $t = -\infty$ to $t = +\infty$. Call t a **critical moment** if there is a site event or a circle event. If t is non-critical, then the beach line $B(t)$ is a sequence of arcs as in (1). The combinatorial part of this information (i.e., (2)) will be represented by a balanced binary tree T.

We need to detect the successive critical moments using a priority queue Q. Initially, Q stores all the site events. Circle events are added to Q as they arise (see below). Using Q, we can correctly update T.

Finally, we must represent the known portion $Vor_t(S)$ of the Voronoi diagram: this is just the part of $Vor(S)$ that lies below $B(t)$ which we called the partial Voronoi diagram. We will use a half-edge data structure D.

We next consider these three data structures, T, Q, D in detail.

¶2. The Priority Queue Q. We have two kinds of events stored in Q: either a site event s with priority $s.y = s_y$, or a circle event (s, s', s'') with priority equal to the y-coordinate of the highest point of the circumcircle of s, s', s''. Initially, Q contains all the site events. We must discuss how to add circle events into Q. If (2) is the active sequence of sites, then we maintain the invariant that every consecutive triple (s_{i-1}, s_i, s_{i+1}) that can generate a circle event is already in Q. Hence, whenever T is modified we need to check to detect any new circle event that is formed, and to insert it into Q. If the priority of the triple is greater than the current time t, it is inserted into Q. A basic property to show is this: *Every circle event of priority t will placed in Q before time t.*

¶3. The Balanced Binary Tree T. If the arcs and active sites in $B(t)$ are represented as in (1) and (2), then T will have m nodes, where each node stores site. But what is the key value for searching the tree T? Clearly, the x-coordinates of these sites are inapproriate because a site may occur more than once in the sequence. Rather, we use the x-coordinates of the breakpoint defined by any two consecutive sites, s_i, s_{i+1}. This breakpoint must be defined as a function of t: let $b_t(s_i, s_{i+1})$ denote this x-coordinate at time t. For brevity, we may also write $x_i(t)$ for $b_t(s_i, s_{i+1})$. Also, let $x_0(t) = -\infty$ and $x_m(t) = +\infty$. Note that in general, two sites s, s' determine up to two breakpoints since two parabolas with the same directrix intersect in one or two points. If $s.y = s'.y$, then there is a unique break point. Otherwise, we distinguish these two breakpoints by writing $b_t(s, s')$ and $b_t(s', s)$ such that

$$b_t(s, s') < b_t(s', s)$$

provided $s.y < s'.y$. Below, we show how to compute $b_t(s, s')$.

An interesting issue arises in application of binary search trees. In general, there are two ways to store a set of items in a binary search tree. We can either store them exclusively in the leaves, or we can store them

in the internal nodes as well well as the leaves. We call[1] the former **exogenous** and the latter **endogenous** trees. Since keys $x_i(t)$ depends on two consecutive sites, it seems more natural to store the sites only in the leaves. We assume that T is a full-binary tree (i.e., each node has two children) and each node u of the tree stores a pointer to its successor $u.next$ and predecessor and $u.prev$. If u is an internal node, then $u.next$ and $u.prev$ are both leaves. If u is a leaf, then we see that $u.next^2 = u.next.next$ is the next leaf, and $u.prev^2 = u.prev.prev$ is the previous leaf. The leaves therefore stores the sequence $(s_1, \ldots, s_m)$ of active sites.

Note that maintaining all these pointers are straightforward and standard. Moreover, if T is a balanced tree (e.g., an AVL Tree), there will be rotation operations to maintain balance. Such pointers are unaffected by rotations.

We now consider insertion into and deletions from T. These are determined by the site and circle events:

- Site Event s: This amounts to insert a new site s into T using $s.x$ as key and using $t = s.y$ as the current time. Let u be initially the root of T. If u is not a leaf, we compute the breakpoint $b_t(u.prev, u.next)$. If $s.x \geq b_t(u.prev, u.next)$, we set u to the right child of u, else u is set to the left child of u. We continue this process until u is a leaf. Assume that u contains the site s_i. It follows that that

$$x_i(t) \leq s.x < x_{i+1}(t).$$

  Let us first assume the non-degenerate situation where $x_i(t) \neq s.x$. The sequence (2) is thereby transformed to

$$(s_1, \ldots, s_i, s, s_i, \ldots, s_m)$$

  In terms of modifying the tree T, we create a subtree of u with three new leaves. We store s in the middle leaf and store separate copies of s_i in the other leaves. This is illustrated in Figure 4.

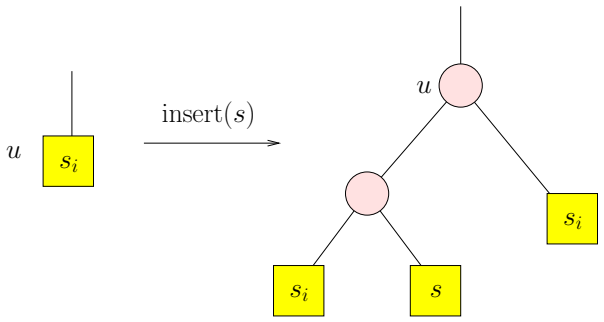


Figure 4: Inserting s into leaf u containing s_i

  We must next check if the new consecutive triples (s_{i-1}, s_i, s) and (s, s_i, s_{i+1}) define circle events in the future (time $> t$). If so, these events are inserted into Q. If $x_i(t) = s.x$, then we only need to create two new leaves as children of u, one storing s_i and the other storing s. We leave the details to the reader.

- Circle Event (s_{i-1}, s_i, s_{i+1}): this amounts to deleting the event s_i from T. Note that $s_{i-1} \neq s_{i+1}$ in this case. Therefore, we obtain a new break point defined by $b_t(s_{i-1}, s_{i+1})$. We must also check if $(s_{i-2}, s_{i-1}, s_{i+1})$ and $(s_{i-1}, s_{i+1}, s_{i+2})$ define circle events in the future (at time $> t$). If so, we insert them into Q.

[1] Our definition of exogenous/endogenous is slightly different than the literature. We think of each item as a pair, item=(data,key). Clearly, each internal node of a binary search tree must store a key. The question is where is the data stored? In the literature, endogenous tree is when each tree node directly stores the data associated with the key; it is exogenous when we only store a pointer to the data.

¶4. The Half-Edge Datastructure D. Note that the partial Voronoi diagram $K_t(S)$ at time t may not be connected even if $K(S)$ is connected. This is seen each time we initiate a site event, at which time an isolated partial edge appears, growing in both directions. To make the partial complex connected, we include the beach line in its representation. Then we can represent the sea as a cell complex whose boundaries are either edges in $K_t(S)$ or arcs from the beach line $B(t)$. Although $K_t(S)$ and $B(t)$ changes continuously with t, the combinatorial structure is unchanged except at an event. One classic representation of such a complex is the half-edge data structure. This structure is illustrated in Figure 5.

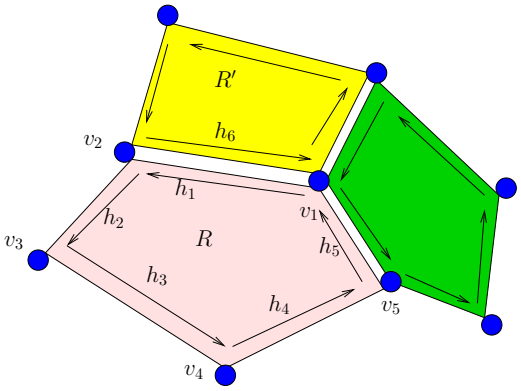


Figure 5: Half-edge data structure

A half-edge data structure D represents an planar graph embedded in the plane. It contains three lists, representing the sets of regions, half-edges and vertices (respectively). The unique feature of this representation is that it stores not "edges" but **half-edges**. Each half edge h is regarded as a directed edge from a start vertex $h.start$ to a stop vertex $h.stop$. E.g., in Figure 5, the half edge h_1 is a directed edge from v_1 to v_2.

Moreover, h has a **twin** h' which is another half-edge, directed from $h.stop$ to $h.start$. Thus,

$$h.start = h.twin.stop, \quad h.stop = h.twin.start.$$

In Figure 5, the twin of half edge h_1 is h_6. We can think of the pair $\{h, h'\} = \{h, h.twin\}$ as an edge e. Why this redundancy? The reason is that each edge e is incident on two regions R and R'. We think of the two two half-edges that represent e as represening the (e, R) and (e, R') incidences. Thus, for each half edge h, we store a field $h.region$ with value R if h represents the incidence between the underlying edge with R. Moreover, we orient the half-edges that are incident on a given region R in a counter-clockwise manner. More precisely, each half-edge h has a field $h.next$ which is another half edge with the property that

$$h.region = h.next.region, \quad h.stop = h.next.start.$$

In this way, by repeated application of the $.next$ field, we can go around the boundary of the $.region$. In Figure 5, the region R has 5 incident half-edges $h_1, \ldots, h_5$, and we have $h_{i-1}.next = h_i$ for $i = 1, \ldots, 5$ (and $h_0 = h_5$).

Each vertex stores its (x, y) coordinate, and also one of the half-edges that points away from the vertex. In some applications, we do not even need the list of regions and may omit the field $h.region$. However, for our Current application, we can store with each region the unique site the determines the corresponding Voronoi region. The boundary edges of regions need not be straightline segments (in this sense, Figure 5 is somewhat misleading). Indeed, for our regions, these edges might be (temporarily) parabolic arcs. With arcs, it makes sense to have regions that have only two bounding half-edges. Indeed, when we insert a new region into D with each site event, the region will initially have only two bounding edges. The reader will find it instructive to carry out the details of creating a new region during a site event.

It is clear that this data structure D stores most of its the information in half-edges. Using D, for instance, we can list the set of regions that are clockwise around a given vertex in linear time. There is

also no need to store the inverse of *.next* because we can also use the available fields to get to the previous half-edge in constant time. We leave these as Exercises.

§3. Algebraic Details

In order to perform the arithmetic operations correctly according to the principles of EGC, we need to make comparisons of numbers accurately. In Fortune's algorithm we encounter irrational numbers for the first time. These numbers involve square roots. Let us investigate this in some detail.

Suppose p, q, r are three sites and c is their circumcenter. If (x, y) is a point on this circle, we have

$$(x - c_x)^2 + (y - c_y)^2 = \rho^2 \tag{3}$$

where ρ is the radius. Plugging $p = (p_x, p_y)$ into this equation, we get $(p_x - c_x)^2 + (p_y - c_y)^2 = \rho^2$, re-written as

$$2p_x c_x + 2p_y c_y + (\rho^2 - c^2) = p^2$$

where $c^2 = c_x^2 + c_y^2$ and $p^2 = p_x^2 + p_y^2$. We can similarly plug q and r into (3) to get two more equations. Writing these 3 equations in the matrix form,

$$\begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix} \begin{bmatrix} 2c_x \\ 2c_y \\ \rho^2 - c^2 \end{bmatrix} = \begin{bmatrix} p^2 \\ q^2 \\ r^2 \end{bmatrix}$$

Rewriting and solving, we obtain

$$\begin{aligned} c_x &= \frac{1}{2\Delta} \begin{vmatrix} p^2 & p_y & 1 \\ q^2 & q_y & 1 \\ r^2 & r_y & 1 \end{vmatrix} \\ c_y &= \frac{1}{2\Delta} \begin{vmatrix} p_x & p^2 & 1 \\ q_x & q^2 & 1 \\ r_x & r^2 & 1 \end{vmatrix} \\ \Delta &= \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}. \end{aligned} \tag{4}$$

The priority of the circle event (p, q, r) is therefore

$$c_y + \rho$$

where

$$\rho = \sqrt{(p_x - c_x)^2 + (p_y - c_y)^2}.$$

In the following, *we assume that the input points p, q, r have L-bit integer coordinates.* It follows from (4) that the center c has coordinates whose numerator and denominators are $3L + O(1)$ and $2L + O(1)$-bit integers (respectively). We can rewrite the priority as a number of the form

$$c_y + \rho = \frac{A + \sqrt{B}}{D} \tag{5}$$

where A, B, D are integers with $3L + O(1)$, $6L + O(1)$ and $2L + O(1)$ bits respectively. In fact, we can choose $D = \Delta$.

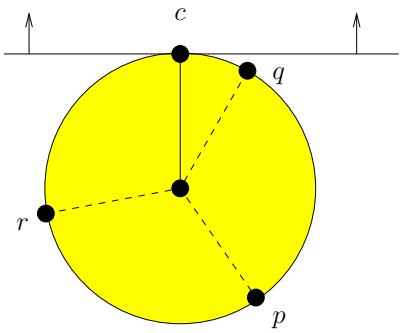


Figure 6: Circle event c generated by sites p, q, r

The λ-notation. If x is a rational number of the form p/q, and p, q are integers with $\lg|p| \leq m$ and $\lg|q| \leq m$, then we call x an $(m:n)$**-bit number**, and write

$$\lambda(x) = (m:n). \tag{6}$$

For instance, if x is an integer, then $\lambda(x) = (m:0)$ for any $m \geq \lg|x|$.
Like the standard big-O notation, the λ-notation is an *upper bound notation*. Thus the equality in (6) is not standard equality, but a form of "$\leq$". In particular, if $\lambda(x) = (m:n)$ holds, then $\lambda(x) = (m':n')$ holds for all $m' \geq n$ and $n' \geq n$. We can thus use big-O expressions in the righthand side of (6). E.g., $\lambda(x) = (O(L^2) : 2L + O(1))$. But it would not make sense to insert lower bound expressions into the righthand side. E.g., "$\lambda(x) = (\Omega(L) : 0)$" is nonsense.

For rough estimates, we can ignore the $O(1)$ terms and simply say that the coordinates of c are $(3L:2L)$-bit rational numbers. Nevertheless, in our implementation, we need to know explicit bounds on these $O(1)$ terms. Here are the explicit constants for the numbers derived in (4) and (5), collected here for future reference:

$$\begin{aligned}
\lambda(p_x) &= (L:0) && (7)\\
\lambda(p^2) &= (1+2L:0) && (8)\\
\lambda(\Delta) &= (4+2L:0) && (9)\\
\lambda(c_x) &= (4+3L:4+2L) && (10)\\
\lambda(D) &= (4+2L:0) && (11)\\
\lambda(A) &= (4+3L:0) && (12)\\
\lambda(B) &= (11+6L:0) && (13)
\end{aligned}$$

In our priority queue operations, we must compare the priorities of events. Priority of site events are L-bit integers while circle events have priority given by (5). Hence the most complicated comparisons are between the priorities of two circle events,

$$\frac{A+\sqrt{B}}{D} : \frac{A'+\sqrt{B'}}{D'} \tag{14}$$

where A, A' are $3L$ bits, etc. How can we make such comparisons exact?

¶5. Method of Repeated Squaring. We can use the well-known method of repeated squaring. This method seems quite reasonable for (14) if we only want to determine *equality or inequality*. But we need to

know more: in case of inequality, we must know which is bigger. We now demonstrate that in this case, the method of repeated squaring is considerably more involved to implement.

We may reduce the desired comparison to the simpler case of checking whether $\alpha + \beta\sqrt{n} \geq 0$ where $\alpha, \beta \in \mathbb{R}, n \in \mathbb{N}$. This is equivalent to the following three disjunctions:

$$(\alpha \geq 0, \beta \geq 0) \quad \vee \tag{15}$$
$$(\alpha \geq 0, \beta \leq 0, \alpha^2 - \beta^2 n \geq 0) \quad \vee \tag{16}$$
$$(\alpha \leq 0, \beta \geq 0, \alpha^2 - \beta^2 n \leq 0). \tag{17}$$

Recursively, if α, β are integer expressions involving other square-roots then we must further expand on each of the predicates $\alpha \geq 0$, $\beta \geq 0$, $\alpha^2 - \beta^2 n \geq 0$.

Let us apply this remark to the original comparison (14), but first rewriting it as $D'(A + \sqrt{B}) \geq D(A' + \sqrt{B'})$ or,

$$(I): \quad D'\sqrt{B} \geq D\sqrt{B'} - E$$

where $E = DA' - D'A$. First, let us only assume $D, D' \neq 0$. Then (I) is equivalent to the disjunction of the following three conjuncts:

$$(II): \quad (D' \leq 0), (D\sqrt{B'} - E \leq 0), (D'^2 B \leq (D\sqrt{B'} - E)^2).$$
$$(III): \quad (D' \geq 0), (D'^2 B \geq (D\sqrt{B'} - E)^2).$$
$$(IV): \quad (D' \geq 0), (D\sqrt{B'} - E \leq 0).$$

These can ultimately be expanded into a Boolean function of the sign of the following 6 expressions:

$$D, D', E, \tag{18}$$
$$D^2 B' - E^2, \tag{19}$$
$$D'^2 B - D^2 B' - E^2, \tag{20}$$
$$4D^2 E^2 B' - (D'^2 B - D^2 B' - E^2)^2. \tag{21}$$

Alternatively, we can expand $(II) \vee (III) \vee (IV)$ into a disjunction of 18 conjuncts involving these expressions.

Let us make a useful observation. The last expression represents a number with $20L + O(1)$ bits. Thus we conclude:

Lemma 6. *The method of repeated squaring, applied to the comparison of the priorities of two circle events can be reduced to the computation of integers that are at most* $20L + O(1)$ *bits long.*

In our application, it is reasonable to assume D, D' to be positive. Then the test (I) is equivalent to the disjunction $(III) \vee (IV)$. But we still have to evaluate a Boolean function of the sign of all the expressions, except D and D', in (18). Thus the $20L + O(1)$ bits bound of Lemma 6 cannot be improved. Below we will derive a true improvement.

¶6. Method of Root Bounds. Another way to do the comparison is to compute $\alpha = (A + \sqrt{B})/D$ and $\alpha' = (A' + \sqrt{B'})/D'$ to sufficient accuracy to distinguish them. Basically, we need a lower bound on $|\alpha - \alpha'|$. Consider the integer polynomials:

$$\begin{aligned} P(X) &= (D \cdot X - (A + \sqrt{B}))(D \cdot X - (A - \sqrt{B})) && (22)\\ &= D^2 X^2 - 2DA \cdot X + (A^2 - B), && (23)\\ P'(X) &= D'^2 X^2 - 2D'A' \cdot X + (A'^2 - B'). && (24) \end{aligned}$$

Then α and α' are roots of

$$\begin{aligned} Q(X) &= P(X)P'(X)\\ &= (DD')^2 X^4 - 2DD'(DA' + D'A)X^3\\ &\quad +(D^2(A'^2 - B') + D'^2(A^2 - B) + 4AA'DD')X^2\\ &\quad -2(AD(A'^2 - B') + A'D'(A^2 - B))X\\ &\quad +(A^2 - B)(A'^2 - B'). \end{aligned}$$

Note that coefficients of X^i has $(12-i)L + O(1)$ bits ($i = 0, \ldots, 4$). Thus $\lg \|Q\|_\infty = 12L + O(1)$. The root separation bound of Mahler (see Lecture 6 and also [12, Corollary 32, Chap. VI]) applied to the square-free polynomial $Q(X)$ of degree m says that

$$-\lg(|\alpha - \alpha'|) < (m-1)\lg(\|Q\|_2) + (m+2)(\lg m)/2.$$

Since $m = 4$,

$$-\lg(|\alpha - \alpha'|) = 36L + O(1).$$

We can improve this using another approach: form the resultant of $P(X)$ and $P'(X+Y)$ with respect to X,

$$R(Y) = Res_X(P(X), P'(X+Y)).$$

Note that $\alpha - \alpha'$ is a root of $R(Y)$. From [12, Lemma 35, Chap. VI], we conclude that

$$-\lg(|\alpha - \alpha'|) = 24L + O(1).$$

¶7. Repeated Squaring again, but for Root Bounds. We further improve our bound by using a trick[2] which superficially look like repeated squaring. Suppose we have an expression $\sum_{i=1}^n a_i\sqrt{b_i}$ whose sign we want to determine. Here $a_i \in \mathbb{Q}$ and $b_i \in \mathbb{N}$. We introduce a new variable ε via the equation $\varepsilon = \sum_{i=1}^n a_i\sqrt{b_i}$. By repeated squaring, we eventually obtain an integer polynomial equation,

$$P(\varepsilon) = 0.$$

Now we can apply the zero bound for ε.

Applying this idea to the critical comparison in Fortune's algorithm, we have

$$\varepsilon = \frac{A + \sqrt{B}}{D} - \frac{A' + \sqrt{B'}}{D'} \tag{25}$$

By repeated squaring, we finally obtain the polynomial equation

$$0 = P(\varepsilon) = \sum_{i=0}^m b_i \varepsilon^i.$$

How large are the coefficients b_i? From the discussion of repeated squaring leading to Lemma 6, we conclude that each b_i has at most $20L + O(1)$ bits. Using Cauchy's bound, we conclude that

$$\begin{aligned} |\varepsilon| &> \left(1 + \frac{\max\{|b_1|, |b_2|, \ldots, |b_m|\}}{|b_0|}\right)^{-1} \\ &\geq (1 + \max\{|b_1|, |b_2|, \ldots, |b_m|\})^{-1} \\ -\lg|\varepsilon| &< 1 + \lg(\max\{|b_1|, |b_2|, \ldots, |b_m|\}) \\ &= 20L + O(1). \end{aligned} \tag{26}$$

¶8. Improved Bound and Explicit Constants. If we are implementing this algorithm, the additive constant "$+O(1)$" in (26) must be known. So let us explicitly compute it:

ε	$=$	$\frac{A+\sqrt{B}}{D} - \frac{A'+\sqrt{B'}}{D'}$	(definition of ε)
$DD'\varepsilon$	$=$	$D'(A+\sqrt{B}) - D(A'+\sqrt{B'})$	(clearing denominators)
$\sqrt{D^2B'}$	$=$	$\sqrt{D'^2B} + (E-\delta)$	(put $E = D'A - DA'$, $\delta = DD'\varepsilon$)
D^2B'	$=$	$D'^2B + (E-\delta)^2 + 2(E-\delta)\sqrt{D'^2B}$	(squaring)
$2(E-\delta)\sqrt{D'^2B}$	$=$	$(D^2B' - D'^2B) - (E-\delta)^2$	(rearranging)
	$=$	$F - (E-\delta)^2$	(put $F = D^2B' - D'^2B$)
$4(E-\delta)^2D'^2B$	$=$	$F^2 + (E-\delta)^4 - 2F(E-\delta)^2$	(squaring)
$(E-\delta)^4$	$=$	$2(E-\delta)^2(F + 2D'^2B) - F^2$	(rearranging)
	$=$	$2(E-\delta)^2G - F^2$	(put $G = D^2B' + D'^2B$)

[2] We are indebted to K. Mehlhorn for pointing out this.

Rewriting the last equation as the polynomial equation $Q(\delta) = 0$, we get

$$\begin{aligned} Q(\delta) &= \delta^4 - \delta^3[4E] + 2\delta^2[3E^2 - G] - 4\delta[E(E^2+G)] + [E^4 - 2E^2G + F^2]. \\ P(\varepsilon) &= Q(DD'\varepsilon) \\ &= \sum_{i=0}^{4} b_i \varepsilon^i. \end{aligned}$$

Since

$$\lg|D| \le 4 + 2L, \quad \lg|A| \le 4 + 3L, \quad \lg|B| \le 11 + 6L.$$

(see (7)), we conclude that

$$\lg|E| \le 9 + 5L, \quad \lg|F| \le 20 + 10L, \quad \lg|G| \le 20 + 10L.$$

We can now read off the coefficients of $P(\varepsilon)$:

$$\begin{aligned} b_4 &= (DD')^4 && (\text{so } \lg|b_4| \le 32 + 16L) \\ b_3 &= -4(DD')^3 E && (\text{so } \lg|b_3| \le 35 + 17L) \\ b_2 &= 2(DD')^2(3E^2 - G) && (\text{so } \lg|b_2| \le 38 + 18L) \\ b_1 &= -4DD'E(E^2 + G) && (\text{so } \lg|b_1| \le 40 + 19L) \\ b_0 &= E^4 - 2E^2G + F^2 && (\text{so } \lg|b_0| \le 41 + 20L) \end{aligned}$$

From (26), we finally obtain the following:

Lemma 7. *If the input numbers are L-bit integers, then the priorities α, α' of any two events in Fortune's algorithm must have a gap of at least*

$$-\lg|\alpha - \alpha'| < 41 + 19L.$$

This is a sharpening of Lemma 6. More importantly, it can be directly used in an implementation of Fortune's algorithm. We compute an approximate priority $\widetilde{\alpha}$ with absolute precision $43 + 19L$ bits, *i.e.*,

$$|\widetilde{\alpha} - \alpha| \le 2^{-43-19L}.$$

Similarly, $\widetilde{\alpha'}$ is a $43 + 19L$ bit approximation of α'. Then we compare $\alpha : \alpha'$ using the following procedure:

COMPARISON PROCEDURE
Input: $43 + 19L$-bit approximations $\widetilde{\alpha}$ and $\widetilde{\alpha'}$.
Output: The outcome of comparison $\alpha : \alpha'$

1. Compute $\varepsilon' = \widetilde{\alpha} - \widetilde{\alpha'}$.
2. If $\varepsilon' > 2^{-42-19L}$, return("$\alpha$ is larger than α'").
3. If $\varepsilon' < -2^{-42-19L}$, return("$\alpha$ is smaller than α'").
4. Return("α is equal to α'").

Justification: assume $\alpha = \widetilde{\alpha} + e$ and $\alpha' = \widetilde{\alpha'} + e'$ where $|e|$ and $|e'|$ is at most $2^{-43-19L}$. If $\varepsilon' > 2^{-42-19L}$ then

$$\begin{aligned} \alpha - \alpha' &= (\widetilde{\alpha} + e) - (\widetilde{\alpha'} + e') \\ &= \varepsilon' + (e - e)' \\ &> 2^{-42-19L} - |e| - |e'| \\ &\ge 0. \end{aligned}$$

The case $\varepsilon' < -2^{-41-19L}$ is similarly justified. The final case is $|\varepsilon'| \le 2^{-42-19L}$.

$$\begin{aligned} |\alpha - \alpha'| &= |(\widetilde{\alpha} + e) - (\widetilde{\alpha'} + e')| \\ &\le \varepsilon' + |e| + |e'| \\ &\le 2^{-42-19L} + 2^{-43-19L} + 2^{-43-19L} \\ &\le 2^{-41-19L}. \end{aligned}$$

This implies $\alpha = \alpha'$.

¶9. Discussion. The advantage of repeated squaring is that only integer operations are used. The disadvantage for using this for determining signs of expressions involving square roots is the need to evaluate a large Boolean function of many smaller predicates.

On the other hand, as first observed in [3], the existence of such root bounds shows that approximate square-roots can be used to make such comparisons. Note that the complexity of approximate square-roots is as fast as multiplication (both in theory and in practice). A bonus of our root bounds is that, we manipulated $19L + O(1)$ bit approximations, not $20L$-bit integers. But the most important advantage of using approximate square-roots seems to be this: suppose our priorities are α_i $(i = 1, 2, \ldots)$. Then for all $i < j$, the repeated squaring method requires the evaluation of $20L$-bit expressions that depend on both i and j. There could be $O(n \log n)$ such expressions to evaluate, assuming an $O(n \log n)$ time implementation of Fortune's algorithm. In contrast, when we use approximate arithmetic we only need to compute $20L$-bit approximate values for $O(n)$ expressions. Thereafter, all comparisons require no arithmetic, just a straight comparison of two $20L$-bit numbers. A further improvement to $15L$-bits was achieved by Sellen and Yap.

§4. A Practical Implementation

Fortune gave an implementation of his algorithm in the `C` langauge. We have converted this program into a Core Library program by introducing the minimal necessary changes. The program can output postscript scripts (to display images) or combinatorial structure of the Voronoi diagram.

Fortune's algorithm is practical and dispenses with the balanced binary tree T. Instead, it uses a bucketing scheme. An empirical study of Delaunay Triangulation algorithms, Drysdale and Su [10] found that for uniformly randomly generated points, the use of an $O(\log n)$ priority queue in Fortune's algorithm did not improve upon the bucketing scheme for less than 2^{17} points. At 2^{17} points, there was a 10% improvement.

The original implementation, assuming machine arithmetic, do not explicitly handle degenerate data: for instance, it assumes that all Voronoi vertices have degree 3. To investigate its behavior with respect to degeneracies, we compute the Voronoi diagram of a set of points on the uniform grid. SHOW RESULTS!

§5. Voronoi Diagram of Line Segments and Circular Arcs

The definition of Voronoi diagrams is capable of many generalizations. We consider the generalization where S is now a set of x-monotone curve segments. The reason for monotonicity property is that we wish to preserve the linesweep paradigm in our algorithm. We assume these segments are either line segments or circular arcs.

To understand this, consider the generalized parabola $B_s(t)$ where s is either a line segment or a circular arc. Note that $H(t)$ can now intersect s over a range of values: in other words, instead of a "site event" we have "start site" and "stop site" events.

Given a site s, we introduce the lines through the endpoints of s which are normal to s. When s is a line segment, this divides the plane into 4 **zones**, corresponding to the influence of the various features of the line segment. See Figure 7(a). Zones z_0 and z_1 comprise those points that are (respectively) closest to one of the two "sides" of s. Zones z_ℓ and z_r comprise those those points that are (respectively) closest to the two endpoints of s. When s is a circular arc, we introduce a similar set of zones. In this case, the side of the arc containing the center of the circle will have a bounded zone z_0, as seen in Figure 7(b). In either case, there is a left and right zone z_ℓ, z_r corresponding to the left and right endpoints of the arc.

Let us consider the **beach line** of a line segment s: this is basically the bisector $b(s, H(t))$ comprising all points that are equidistant from s and $H(t)$. Assume s is non-horizontal with endpoints p and q. In Figure 8(a) we show three arcs $\alpha, \alpha', \alpha''$ of the bisector. These arcs are part of the bisectors of $H(t)$ with (respectively) $q, \ell(s)$ and p. Here $\ell(s)$ is the line defined by s.

It turns out that unless s is a horizontal segment, there are two more arcs α''' and α'''' which may not be easy to see if s is close to horizontal. Altogether $b(s, H(t))$ has 5 arcs, where the arcs alternate between straight and parabolic. To see the existence of α''', we need to show that the parabola α'' intersects the normal through the endpoint p of the segment s (see Figure 9). The parabola will intersect the normal line at p at the point c which is the center of a circle through p and tangent to the current sweepline $H(t)$. It is clear that c exists. Hence α''' exists, and is the angle bisector of the lines $\ell(s)$ and $H(t)$. So α''' is a straight

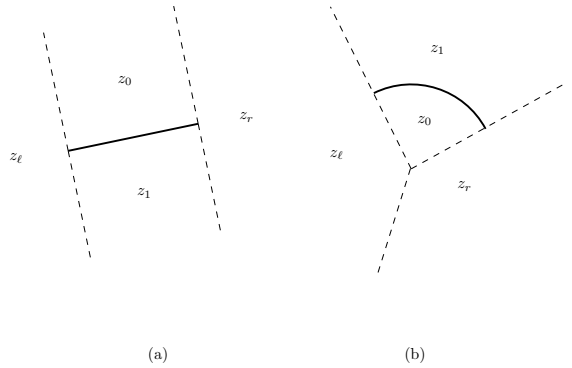


Figure 7: Zones of (a) Line Segment and (b) Circular Arc

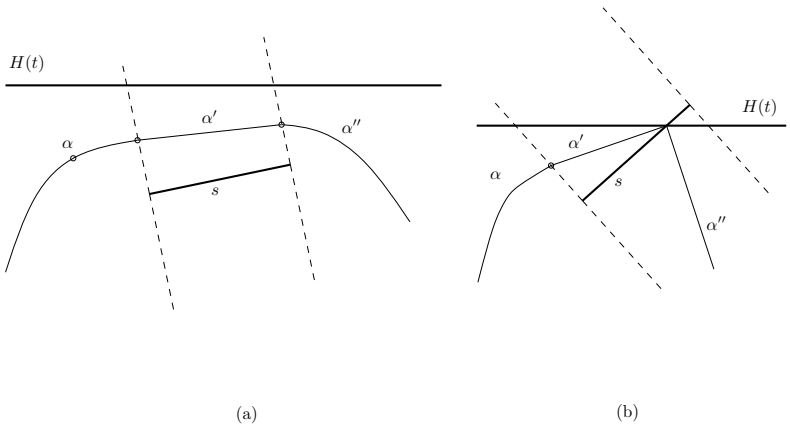


Figure 8: Beachline of a line segment s

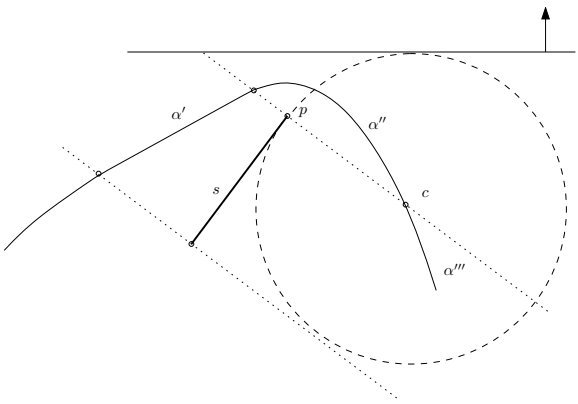


Figure 9: The existence of arc α'''

line segment. But α''' will intersect the normal line through the other endpoint of s, thereby ensuring that α'''' exists. Here α'''' is a parabolic segment with focus q and directrix $H(t)$.

Another possibility is for $H(t)$ to intersect s as in Figure 8(b). In this case, we are only interested in the part of this bisector that lies below $H(t)$. If $H(t)$ intersects s at some point q, then there are two angle bisectors emanating from q. To see that $b(s, H(t))$ will comprise of 4 arcs note that at the moment $H(t)$ touches s, the beach line generated by s is a half-ray emanating southward from s, representing a degenerate parabola, as in the case of point sites. But immediately after this, this parabola would be split into two parts, and joined by two straight line segments. This gives us the 4 arcs illustrated in Figure 8(b). Note that this is the case even when s is vertical.

We can extend this analysis to the case where $H(t)$ has swept past the other endpoint of s. At that moment, the two straightline segments become separated by a new parabolic arc generated by the second endpoint of s.

Next, we consider bisectors $b(s, H(t))$ between a circular arc s and the sweepline $H(t)$. Initially, assume $H(t)$ does not intersect s. The arc is either a cap or a cup (convex upwards or downwards). If the circular arc is a cap, the bisector $B_s(t)$ has a parabolic part separating two linear parts. The focus and directrix of the parabola are (respectively) the center of the arc and the displaced sweepline $H(t + r)$, where r is the radius of the arc. If a cup-arc, then again we have a parabolic arc with directrix $H(t - r)$ and focus at the center of the arc. The parabolic arc is confined to the bounded zone z_0 of the arc Figure 7(b). Outside, the zone, we have parabolic arcs determined by the endpoints of the arc. Note that if $H(t)$ intersects s, the bisector $b(s, H(t))$ ough to be limited to the half-space below $H(t)$. These curves are essentially restrictions of the curves explained as below.

What about circle events? Now we can have Voronoi vertices generated by i linesegments and $3 - i$ endpoints, for any $i = 0, 1, 2, 3$. Call this a **i-circle event**. The original circle event corresponds to a 0-circle event. See Figure 10.

THIS IS A PLACE HOLDER – NO FIGURE YET

Figure 10: Four kinds of Circle Events

We can now modify the original algorithm of Fortune so that instead of site events, we now have **start event** and **stop event** corresponding to the first and last encounter of a site. At a start event, we insert four arcs as in Figure 8(b). At a stop event, we insert one more arc into (whatever remains of) the original four.

The modification for the rest of the algorithm is left as an exercise. The analysis of the geometry and algebraic root bounds, together with implementation, have been carried out by Burnikel [1] in his PhD Thesis (University of Saarbrücken, 1996).

Exercises

Exercise 5.1: Prove Lemma 1. ♢

Exercise 5.2: Describe some concrete ways in which the algorithm of Fortune can fail because of numerical errors in deciding the priority of events. You must provide concrete numerical examples to illustrate how it fails. HINT: think of very degenerate inputs where 4 or more points are co-circular so Voronoi vertices are very close to each other. ♢

Exercise 5.3: Re-derive our root bounds for Fortune's algorithm if the input numbers are binary floats with L-bit mantissas and exponents at most E. ♢

Exercise 5.4: Here are some generally open questions for exploration with Voronoi diagrams:
(i) Given $n \geq 1$, to find a set S with n sites which is the optimal location for n post offices. Various optimality criteria can be studied. For instance, assume S is restricted to the interior of a simple polygon P, and we want to minimize the area of the largest Voronoi region of S within P.
(ii) Suppose S is given, and we want to locate a new site s so that $S \cup \{s\}$ will optimize the criteria of (i). ♢

Exercise 5.5: Show how to do basic operations using the half-edge data structure: (i) getting to the previous half-edge in constant time, (ii) listing all the regions around a vertex in linear time. ♢

Exercise 5.6: Work out the other predicates needed for Fortune's sweep, and compute their corresponding root bounds. ♢

Exercise 5.7: Suppose the input numbers are IEEE Machine doubles. What is the corresponding bound in Lemma 7? ♢

Exercise 5.8: (i) Give the minimal polynomials $P_2(X)$ and $P_3(X)$ for the following numbers: $\alpha_2 = \sqrt{A} + \sqrt{B}$, $\alpha_3 = \sqrt{A} + \sqrt{B} + \sqrt{C}$ for positive but indeterminate integers A, B, C.
(ii) The constant terms of these polynomials are squares. Is this true for $P_n(X)$ for all n? ♢

Exercise 5.9: If $p, q, p', q' \in \mathbb{R}^2$ are points whose coordinates are IEEE machine single precision floats (i.e., numbers of the form $m2^{e-23}$, $|m| < 2^{24}$ and $|e| \leq 127$) we want a separation bound on

$$\delta = \|p - q\| - \|p' - q'\|$$

where $\|p - q\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$. In fact, show that if $\delta \neq 0$ then

$$|\delta| > 2^{431}.$$

HINT: Apply Cauchy's bound to the polynomial $P_2(X)$ obtained in the previous exercise. ♢

Exercise 5.10: The circle through three points p, q, r has equation $C(X, Y) = 0$ where

$$C(X,Y) \quad = \quad \det \begin{bmatrix} X^2+Y^2 & X & Y & 1 \\ p^2 & p_x & p_y & 1 \\ q^2 & q_x & q_y & 1 \\ r^2 & r_x & r_y & 1 \end{bmatrix} \tag{27}$$

♢

Exercise 5.11: Suppose $p, q, p', q' \in \mathbb{R}^2$ are points whose coordinates are represented by machine floats (i.e., IEEE single precision floating point numbers). For this question, they are just numbers of the form $x = m2^{e-23}$ where m, e are integers satisfying $|m| < 2^{24}$ and $|e| \leq 127$. Let

$$\delta = \|p - q\| - \|p' - q'\| \tag{28}$$

where $\|p - q\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$. We want a separation bound on $|\delta|$.
(i) Let $\Delta = M\delta$ where M is the smallest positive integer such that $M\delta$ is equal to $\sqrt{A} - \sqrt{B}$ for some integers A, B. Give upper bounds on M, A and B.
(ii) Give a polynomial $P(X)$ such that $P(\Delta) = 0$, written in terms of A, B.
(iii) Apply Cauchy's bound to conclude that if $\delta \neq 0$ then $|\delta| > 2^{-431}$.
(iv) Extra Credit: Give an actual numerical example that comes as close to your lower bound as possible. ♢

Exercise 5.12: Let $e(x) = f$ where $x = m2^{f-23}$ is a IEEE single precision float, with $|m| < 2^{24}$ and $|f| \leq 127$. In the previous question, suppose $e_0 = \min\{e(p_x), e(p_y), e(q_x), e(q_y), e(p'_x), e(p'_y), e(q'_x), e(q'_y)\}$ and $e_1 = \max\{e(p_x), e(p_y), e(q_x), e(q_y), e(p'_x), e(p'_y), e(q'_x), e(q'_y)\}$. Redo parts (i) and (iii) of the question, where e_0 and e_1 are taken into account. ♢

End Exercises

§6. Additional Notes

The book of Okabe, Boots and Sugihara [9] is devoted to Voronoi diagrams, their generalizations and applications. We follow the site/circle event terminology of Guibas and Stolfi [5]. The beachline terminology is from [4]. The detailed analysis of the root bounds necessary for Fortune's algorithm is from Dubé and Yap [3]. This seems to be the first time[3] when it was realized that numerical approximations could be used to solve geometric computation problems in the "exact geometric sense". The retraction approach to motion planning was first applied to a disc in O'Dunlaing and Yap [8]; this can be extended to other kinds of motion planning problems [7, 11, 2]. Yap had obtained the first correct $O(n \log n)$ algorithm for Voronoi diagrams for line segments and circular arcs using a divide-and-conquer algorithm. Unlike Fortune's linesweep approach, it is based on divide and conquer paradigm, and it can be used in parallel algorithms as well as for constrained Voronoi diagrams.

References

[1] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, Mar. 1996.

[2] J. Cox and C. K. Yap. On-line motion planning: case of a planar rod. *Annals of Mathematics and Artificial Intelligence*, 3:1–20, 1991. Special journal issue. Also: NYU-Courant Institute, Robotics Lab., No.187, 1988.

[3] T. Dubé and C. K. Yap. A basis for implementing exact geometric algorithms (extended abstract), September, 1993. Paper from `http://cs.nyu.edu/exact/doc/`.

[4] M. T. Goodrich, C. Ó'Dúnlaing, and C. Yap. Constructing the Voronoi diagram of a set of line segments in parallel. *Algorithmica*, 9:128–141, 1993. Also: Proc. WADS, Aug. 17-19, 1989, Carleton University, Ottawa, Canada. *Lecture Notes in C.S.* No.382, Springer (1989)12–23.

[5] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[6] J. R. Munkres. *Topology: A First Course*. Prentice-Hall, Inc, 1975.

[7] C. Ó'Dúnlaing, M. Sharir, and C. K. Yap. Retraction: a new approach to motion-planning. *ACM Symp. on Theory of Computing*, 15:207–220, 1983.

[8] C. Ó'Dúnlaing and C. K. Yap. A "retraction" method for planning the motion of a disc. *J. Algorithms*, 6:104–111, 1985. Also, Chapter 6 in *Planning, Geometry, and Complexity*, eds. Schwartz, Sharir and Hopcroft, Ablex Pub. Corp., Norwood, NJ. 1987.

[9] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations — Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons, Chichester, 1992.

[10] P. Su and R. L. S. Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proc. 11th ACM Symp. Comp. Geom.*, pages 61–70, 1995. Vancouver, British Columbia, Canada.

[11] C. K. Yap. Coordinating the motion of several discs. Robotics Report 16, Dept. of Computer Science, New York University, Feb. 1984.

[12] C. K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.

[3] Until then, most papers assumed that to compute exactly, one must reduce all computation to exact integer or rational computations.

A foolish consistency is the hobgoblin of little minds

— Ralph Waldo Emerson

Lecture 8
DEGENERACY AND PERTURBATION

The central concern of theoretical algorithms is to understand the inherent complexity of computational problems. As such, most algorithms on paper are formulated to facilitate their analysis, not meant to be literally implemented. In computational geometry, the reader will often encounter qualifications of the following sort:

> *"Our algorithm accepts an input set S of points. For simplicity, we assume that the points in S are distinct, no 2 points are co-vertical, no 3 points are collinear, no 4 points are co-circular, etc. The reader can easily extend our algorithm to handle inputs that fail these assumptions."*

The inputs that satisfy these assumptions may be said to be **non-degenerate**; otherwise they are **degenerate**. Implementors are left to their own devices to handle degenerate cases. Some papers may not even list the degenerate inputs. Thus, Forrest [2] deplores the "plethora of special cases ... glossed over by theoretical algorithms". Forrest, and also Sedgewick [6], calls this the "bugbear" of geometric algorithm implementation.

In this lecture, we explore some general techniques that can help bridge *this* particular gap between theoretical algorithms and their implementation. Notice we do not address other possible gaps, such as the under-specification of data representation.

§1. A Perturbation Framework

Given an algorithm, let us assume that there is a notion of **valid inputs**. Generally speaking, we expect it to be easy to verify if a given inputs are valid. Among the valid inputs, certain inputs are defined as **degenerate**. Let us call an algorithm **generic** if it works correctly only for non-degenerate inputs; it is **general** if it works correctly for all valid inputs, degenerate or not.

As a simple example, suppose the valid inputs are any sequence of points. If there are n points, $p_1, \ldots, p_n$, we view the input as a sequence

$$\mathbf{a} = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n) \in \mathbb{R}^{2n}$$

where $p_i = (x_i, y_i)$. Thus the set of valid inputs can be identified with $\mathbb{R}^{2n}$ (for each n). But in many algorithms, we assume the input points are distinct. In this case, the valid inputs of size n would be

$$\mathbb{R}^{2n} \setminus \{\mathbf{a} : (\exists i, j)[i \neq j, x_i = x_j, y_i = y_j\}\,.$$

Checking valid inputs in this case is relatively easy, though still non-trivial. Alternatively, the algorithm may detect this error and reject during computation.

The gap between theoretical algorithms and implementable algorithms amounts to converting generic algorithms into general algorithms. Of course, one way to bridge this gap is to explicitly detect all the degenerate cases and handle them explicitly. There is a practical problem with this suggestion:

> (B) The number of degenerate cases can be overwhelming and non-trivial to exhaustively enumerate.

Roughly speaking, the number of special cases for handling degeneracies increases exponentially with the dimension d of the underlying geometry (e.g., $S \in \mathbb{R}^d$). So the explicit treatment of degeneracies can increase

the size of the code by a large factor: we end up with a program where, say 10% of the code uses 90% of the machine cycles, with a large portion of the remaining code rarely executed if at all. This makes the automatic generation of this 90% of the code an attractive goal, if it were possible.

Some feel that the degenerate cases are rare. This is true in some measure-theoretic sense, but ignoring these cases is not an option if we want a robust implementation. Another impulse is to require the algorithm designer to supply details for the degenerate cases as well. Indeed, this has been suggested as a standard for publications in algorithms. This is[1] neither desirable nor enforceable. Such details may not make the most instructive reading for non-implementors. In short, the gap between theoretical algorithms and practical algorithms will not go away: the literature will continue produce some algorithms spelled out in detail, and others less so. What should implementors do?

In many applications situations, the user would not mind if their input is slightly perturbed (this is clearly acceptable if the inputs are inexact to begin with). What does perturbation mean? Recall geometric objects live in a parametrized space, and the input is just the set of parameters (associated with some combinatorial structure). If G is the combinatorial structure (e.g., a graph) and $x = (x_1, \ldots, x_n)$ are the parameters, then the object is $G(x)$. In general, only certain choices of parameters x are **consistent** with G. For any $\varepsilon > 0$, we define an ε-perturbation of $G(x)$ to be another object of the form $G(x')$ where $\|x - x'\| < \varepsilon$. Note that $G(x')$ shares the same combinatorial structure (G) with $G(x)$, and and x' must be consistent with G. In numerical analysis, this ε is called **backwards error** bound.

So we want to exploit this freedom to do perturbation. In particular, we would like to consider automatic tools to choose some $G(x')$ that is non-degenerate. The framework is as follows:

Begin with a generic algorithm A for a problem P. We provide schemes for transforming A into a general algorithm A' for P. But what is A'? It cannot really be a full-fledged algorithm for P since, by definition, it does not really handle degenerate cases. Yet as a general algorithm it does produce an output, even when the input is degenerate. We call this output a "perturbed output". Perturbation calls for another **postprocessing algorithm** B. The algorithm B is supposed to fixed the perturbed output so that it is the correct output for the unperturbed input. This framework is illustrated in figure 1.

The simplistic view of perturbation is to add a small non-zero value to each numerical parameter (within some $\epsilon > 0$ bound). We call this **real perturbation**. But we can also use **infinitesimal perturbation** which can be viewed as perturbation that is smaller than any real perturbation $\epsilon > 0$. This notion can be justified using the theory of infinitesimal numbers. We will consider both these kinds of perturbations.

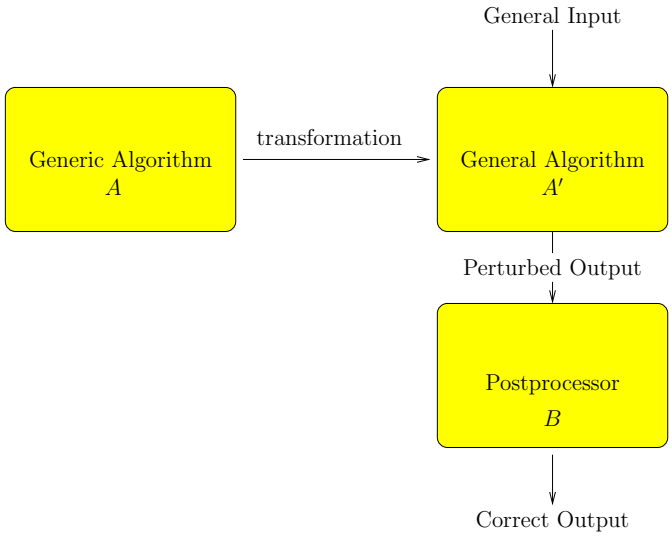


Figure 1: Generic to General Transformation

[1]The algorithmic activity, like all human activity, is multifaceted and serves no single overriding concern. In theoretical papers, implementation details are usually of secondary interest.

¶1. Example. Let us illustrate infinitesimal perturbation with the problem of computing the Voronoi diagram of a point set S. The input S is degenerate if any four points of S are cocircular. If A is a generic algorithm, what does the transformed A' compute? Well, in case a, b, c, d are cocircular (in this order) about an empty circle C of S, then the output of A' will include two Voronoi vertices, say $v(a, b, c)$ and $v(c, d, a)$, which are connected by a Voronoi edge of zero length. Then the postprocessor B has to detect such zero-length edges and remove them.

¶2. Discussion. It has been pointed out that handling degeneracies explicitly may be a good thing (e.g. [1]). This is based on two facts (i) programming the degenerate cases may not be onerous, and (ii) perturbation can increase the run-time of specific instances by an exponential amount. A simple example is the convex hull of a set of n points in d dimensions. If all n points are coincident, the runtime is constant, but the convex hull of n points in general position is $\Theta(n^{\lfloor d/2 \rfloor})$.

Such examples supply some caution when using the generic perturbation framework. On the other hand, examples of complete handling of degeneracies in algorithms are not common and there has been no evidence that they scale easily to harder problems. As for the potential of increasing the worst case cost dramatically, it is clear that such examples (all points coincident) are special. It is possible that general techniques may be developed to automatically reduce such complexity.

The examples do not negate the fact that generic algorithms are much easier to design and code than general algorithms. So when perturbations are acceptable, and there are general tools to cross the chasm from generic to general, users ought to consider this option seriously. A more potentially more serious objection arises: by not constructing a general algorithm in the first place, aren't we doing more work in the end? Instead of one general algorithm, we need to construct a generic A, transform it to A' and finally, provide a postprocessor B. We see in some examples that this is not necessarily harder. First of all, the transformation $A \mapsto A'$ may be quite simple. Basically the idea is to perturbed any input so that it looks nondegenerate. Second, in view of the possible code bloat in a general algorithm, the work involved in constructing A' is negligible. An example of this is the problem of constructing hyperplane arrangements (see below). Finally, the postprocessing algorithm B in these examples turns out to be relatively simple. More importantly, B is not specific to A', but can be used with *any* other algorithm for P. For instance, if P is computing Voronoi diagrams of a point set, then B amounts to detecting Voronoi edges of length 0 and collapsing their end points. Thus B is not specific to A or A'.

¶3. An Example. It is instructive to work out the degenerate cases of some of the algorithms we have encountered so far in these lectures. These are not too difficult, but it can also serve to indicate the kind of tedium that our propose framework can eliminate.

Let us begin with the Bentley-Ottmann algorithm.

Exercises

Exercise 1.1: Take any Voronoi diagram algorithm for a point set (e.g., Fortune's sweep).

(a) Describe the generic version of this algorithm.

(b) Discuss how the possibility of zero-length edges would affect how the arithmetic is implemented. One basically has to beware of dividing by zero. ♢

Exercise 1.2: The exercise will help you appreciate the practical difficulties of enumerating all combinatorial cases, including degenerate cases. In each part (a)-(e) of this question, we consider the combinatorial intersection patterns involving two simple geometric objects, s and t in $\mathbb{R}^d$ ($d = 2$ or 3). A segment is represented by a pair of distinct points in $\mathbb{R}^d$, and a triangle is represented by three non-collinear points in $\mathbb{R}^d$. In each case, you need to do three things:

(1) Classify all non-degenerate (s, t)-intersection patterns.

(2) Classify all degenerate (s, t)-intersection patterns. Do not forget that, for instance, if s, t are both triangles, they might be identical. Summarize your classifications in (1) and (2) by filling in the Table 1.

(3) Give a classification algorithm which, given s and t, will correctly classify the intersection patterns. This classification algorithm must detect non-intersection as well, and must be "parsimonious". A parsimonious (or non-redundant algorithm) is one that does not any comparisons that could have been deduced from earlier comparisons. ♢

	Dim	Type of s	Type of t	No. Nondegen.	No. Degenerate
(a)	2	segment	segment		
(b)	2	segment	triangle		
(c)	2	triangle	triangle		
(d)	3	segment	triangle		
(e)	3	triangle	triangle		

Table 1: Intersection patterns of simple objects

Exercise 1.3: Classify the combinatorial intersection pattern for three intersecting triangles in space. As in the previous exercise, list (1) the non-degenerate cases (2) the degenerate cases, and (3) give a parsimonious classification algorithm. ♢

Exercise 1.4: Work out the degenerate cases for for the problem of constructing a hyperplane arrangement. ♢

Exercise 1.5: Work out the degenerate cases for the beneath-beyond algorithm for 3-D convex hull. ♢

Exercise 1.6: Fix any given combinatorial structure G, and suppose $\mathbb{R}^n$ is the set of valid parameters for G, and there is a polynomial $p(x_1, \ldots, x_n)$ such that for all $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, if x is degenerate for G then $p(x) = 0$. Prove that for any $\varepsilon > 0$ and any $x \in \mathbb{R}^n$, we can always $x' \in \mathbb{R}$ such that $\|x - x'\| < \varepsilon$ which is non-degenerate. ♢

End Exercises

§2. Model of Degeneracy

Our definition of degeneracies in Section 1 ought to called **algorithm-induced degeneracies**. In contrast, an input is **inherently degenerate** if it is degenerate for all algorithms for the problem. An example is an input set S of points for the convex hull problem. If three of the points in S are collinear and they lie on the boundary of the convex hull of S, we would say that S is **inherently degenerate** because any convex hull algorithm must be able to detect and handle this degeneracy. However, if S has three collinear points but all such triples do not on the convex hull of S, then S is not inherently degenerate. S may still be degenerate for some algorithms (e.g., an incremental algorithm might see such a collinear triple). We focus on algorithm-induced degeneracy, as this is more important in practice.

In the introduction, we described some typical non-degeneracy assumptions on input points (coincident points, covertical pairs, collinear triples, cocircular quadruples, etc). Here are more examples of such assumptions, in case the input is a set of lines:

- a vertical line
- 2 parallel lines
- 2 perpendicular lines
- 3 concurrent lines
- 2 intersecting lines (when the lines are embedded in dimension greater that 2)
- the intersection points defined by pairs of lines satisfy the degeneracy conditions specified for point sets.

In case the input set is a set S of points and a set T of lines, we may introduce more such assumptions: (1) an input point lying on an input line, (2) an input line parallel to the line through 2 input points, (3) an input line perpendicular to the line through 2 input points, etc.

¶4. Computational Model. Our algorithm is essentially a Real RAM whose instructions are classified into one of two types: **comparison steps** or **construction steps**. A typical comparison step has the form

$$\mathsf{if}\ (P(y_1,\ldots,y_m))\mathsf{then\ goto}\ L \tag{1}$$

where $P(y_1,\ldots,y_m)$ is a predicate of the variables $y_1,\ldots,y_m$, and L a label of an instruction. If P is true, the the next instruction has label L, otherwise it will be the instruction following the current one in the program. But we also consider a sign comparison step of the form

$$OnSign(g(y_1,\ldots,y_m))\ \mathsf{goto}\ (g_-, L_0, L_+) \tag{2}$$

where g is a real function applied to variables $y_1,\ldots,y_m$. Depending on the sign of $g(y_1,\ldots,y_m)$, the next instruction will be labeled by one of L_-, L_0, or $L+$. A construction step has the form

$$y \leftarrow g(y_1,\ldots,y_m) \tag{3}$$

where y_i are variables and g is some function. Construction steps do not cause any branching.

Note that assume the variables of our RAM are of various types that include reals, integers, Booleans, and possibly other types. We allow algorithm-dependent predicates (e.g., P in (1)) and functions (e.g., g in (3)) to be used directly in our RAM programs.

Assume that the input is a geometric object of the form $G(x)$ where $x \in \mathbb{R}^n$ and G is a directed graph. Our algorithm will have programming variables, and these are of two kinds: real variables that depend on x are **critical**, and all others are non-critical. How do we determine if a real variable y is critical? Each of the input variables $x_1,\ldots,x_n$ are critical. If y ever appears as the right hand side of a construction step as in (3) where any of the variables $y_1,\ldots,y_m$ is critical, then y is critical. A comparison step of the form (2) where any $y_1,\ldots,y_m$ is critical is called a **critical comparison**. The input is deemed **degenerate** if the algorithm takes any zero-branch in any critical comparison. The g used in a critical comparison of the form (2) is called a **critical function**.

¶5. Precise Problems for Imprecise Points. It should be emphasized that inputs are assumed to be exact whenever we discuss perturbation methods. In any case, what do we mean that an input is "imprecise"? What is the algorithm supposed to compute?

Suppose we want to compute the convex hull of a set of "imprecise points". The input is a set S of points together with an $\varepsilon > 0$ parameter. What we want to output is the **precise** convex hull of any set S' of points such that each point in S' is within ε distance from a corresponding point in S. We see that this new problem is as precise as anything we have considered so far. So in some sense, there is really no such thing as an "imprecise input". In any case, we can solve the (precise) problem of computing the convex hull of "imprecise points" just by computing the convex hull S. Of course, we now have the option of not doing so (Exercise).

Note that all such degeneracy conditions can be specified as a polynomial $p(x_1,\ldots,x_m)$ vanishing on some of the input parameters. We may call these **á priori degeneracy conditions**. Usually, these á priori conditions are meant to prevent the occurrence of algorithm induced degeneracies (i.e., **á posteriori degeneracies**). Informally, á priori and á posteriori degeneracies is analogous to compile-time and run-time errors.

Exercises

Exercise 2.1: Provide polynomials whose vanishing corresponds to each to the degeneracy condition listed above. ♢

Exercise 2.2: Discuss how we can solve the (precise) problem of computing the convex hull of imprecise points by taking advantage of the ε parameter. Is it true that a generic convex hull algorithm can always solve such a problem? ♢

End Exercises

§3. Black Box Perturbation Scheme

A critical value is the value of a critical variable. The **depth** of a critical value y is inductively defined: the value of an input parameter $x_1, \ldots, x_n$ has depth 0. The depth of a value y is 1 more than the maximum depth of $y_1, \ldots, y_m$ where $y = g(y_1, \ldots, y_m)$. Note that we use the same symbol 'y' to denote a programming variable as well as its value which is time-dependent – the context should make it clear which is intended. If the values $y_1, \ldots, y_m$ in a critical comparison (3) is never more than d, we say the algorithm has **depth** of d.

We will first describe a scheme described in [8] which works under the following conditions:

- The critical functions are polynomial.
- The test depth is 0.

Later we note possibilities to relax some of these conditions. The idea is to construct a "black box" which, given any non-zero polynomial $p = p(x_1, \ldots, x_m)$ and any sequence of depth 0 parameters $\mathbf{b} = (b_1, \ldots, b_m)$, to output a non-zero sign

$$\mathtt{sign}_{\mathbf{b}}(p) \in \{-1, +1\}.$$

Note that a depth 0 sequence $\mathbf{b}$ is essentially a subsequence of $\mathbf{a}$ in (**??**). Moreover, if $p(\mathbf{b}) = p(b_1, \ldots, b_m)$ is non-zero, then $\mathtt{sign}_{\mathbf{b}}(p)$ is just the sign of $p(\mathbf{b})$. Thus, the interesting case happens when $p(\mathbf{b}) = 0$: in effect, we are computing a "signed zero", $+0$ or -0.

¶6. Using the Black Box. As in figure 1, we begin with a generic algorithm A. We construct the general algorithm A' by replacing each polynomial-test $p(\mathbf{b})$ in A with a call to this black box. The algorithm then continues with the positive or negative branch after the test, depending on $\mathtt{sign}_{\mathbf{b}}(p)$. Even if the input is degenerate, no zero-branch is ever taken.

Before describing our solution, consider the following possible approach. Randomly perturb the inputs $\mathbf{b}$ to some $\mathbf{b}'$ (with $\|\mathbf{b} - \mathbf{b}'\| < \varepsilon$) and evaluate the sign of $p(\mathbf{x}')$. With high probability, $p(\mathbf{b}')$ is non-zero, as desired. But what if the result is zero? Perturb again. Unfortunately, we need to make sure that the new perturbation preserves the signs of all previously evaluated polynomials. This seems hard. Moreover, we must ensure that the perturbation $\mathbf{b}'$ is sufficiently small that it does not change the sign of any non-zero $p(\mathbf{b})$. We abandon this approach for now, although the idea of numerical perturbation can be made to work under special conditions (see Section 4 below).

¶7. Admissible Black Box Schemes. Our scheme is based on the concept of "admissible orderings" which originally arose in Gröbner bases theory.

Let $\mathbf{x} = (x_1, \ldots, x_n)$ be real variables (corresponding to the input parameters (**??**)). Let $\mathrm{PP} = \mathrm{PP}(x_1, \ldots, x_n)$ denote the set of all power products over $x_1, \ldots, x_n$. Thus a typical power product has the form

$$w = \prod_{i=1}^{n} x_i^{e_i}, \qquad e_i \geq 0$$

where the e_i are natural numbers. Note that $1 \in \mathrm{PP}$ corresponding to $e_1 = e_2 = \cdots = e_n = 0$. If $e = (e_1, \ldots, e_n)$ we also write $w = x^e$. The **degree** of x^e is $\sum_{i=1}^n e_i$, also denoted $|e|$. We say a total ordering $\leq_A$ on PP is **admissible** if for all $u, v, w \in \mathrm{PP}$, (1) $1 \leq_A w$, and (2) $u \leq_A v$ implies $uw \leq_A vw$.

There are infinitely many admissible orderings, but the two most familiar ones are **(pure) lexicographical ordering** $\leq_{\mathtt{LEX}}$ and **total degree ordering** $\leq_{\mathtt{TOT}}$. These are defined as follows: let $w = x^e$ and $v = x^d$ where $e = (e_1, \ldots, d_n)$ and $d = (d_1, \ldots, d_n)$.

- $w \leq_{\mathtt{LEX}} v$ iff either $e = d$ or else $e_i < d_i$ where i is the first index where $e_i \neq d_i$.
- $w \leq_{\mathtt{TOT}} v$ iff $|e| < |d|$ or else $w \leq_{\mathtt{LEX}} v$.

Notice that both of these orderings depend on a choice of an ordering of the variables $x_1, \ldots, x_n$. So there are really $n!$ lexicographical (resp. total degree) orderings.

Now let $p(\mathbf{x}) \subseteq \mathbb{R}[\mathbf{x}]$ be a real polynomial. For $w \in \mathrm{PP}$, we let $\partial_w(p)$ denote the partial derivative of $p(\mathbf{x})$ with respect to w. More precisely, if $w = x^e$ and $e = (e_1, \ldots, e_n)$ then

$$\partial_w(p) := \frac{\partial^{|e|} p}{\partial^{e_1} x_1 \partial^{e_2} x_2 \cdots \partial^{e_n} x_n}.$$

For instance, $w = x^3yz^3$ (we typically let $x = x_1, y = x_2, z = x_3$) then $\partial_w(p) := \frac{\partial^6 p}{(\partial x)^3(\partial y)(\partial z)^2}$. If $p(x, y, z) = 2x^2z^3 - 7xy + y^5 + 10$ then $\partial_x(p) = 4xz^3 - 7y$, $\partial_{xy}(p) = 7$, $\partial_{z^2}(p) = 24xz$, $\partial_{xyz}(p) = \partial_{yz^2}(p) = 0$.

Fix any admissible ordering $\leq_A$. We describe a blackbox sign function corresponding to $\leq_A$. The ordering $\leq_A$ gives an enumeration

$$(w_0, w_1, w_2, \ldots), \qquad w_i \leq_A w_{i+1}$$

of all power products in PP. Clearly, $w_0 = 1$. Define the sequence of polynomials

$$\partial(p) := (\partial_{w_0}(p), \partial_{w_1}(p), \cdots).$$

If $\mathbf{a} = (a_1, \ldots, a_n)$ then define

$$\partial(p; \mathbf{a}) := (\partial_{w_0}(p)(\mathbf{a}), \partial_{w_1}(p)(\mathbf{a}), \cdots).$$

So $\partial(p; \mathbf{a})$ is an infinite sequence of numbers obtained by evaluating each polynomial of $\partial(p)$ at $\mathbf{a}$. We finally define our "black box sign" $\mathtt{sign}_{\mathbf{a}}(p)$ to be the sign of the first non-zero value in $\partial(p; \mathbf{a})$. Note that if $p \neq 0$ then $\partial_{w_i}(p)$ is a non-zero constant for some a. Hence $\mathtt{sign}_{\mathbf{a}}(p)$ is well-defined for all non-zero p. By definition, $\mathtt{sign}_{\mathbf{a}}(p) = 0$ when p is the zero polynomial. Also write

$$\mathtt{sign}_{\leq_A, \mathbf{a}}(p) \tag{4}$$

for $\mathtt{sign}_{\mathbf{a}}(p)$ to indicate the dependence on $\leq_A$. We will call $\mathtt{sign}_{\leq_A, \mathbf{a}}$ an **admissible sign function**.

Let us first give two actual examples of perturbation that can be understood as special cases of our blackbox scheme.

¶8. Example: SoS Scheme of Edelsbrunner and Mücke. Suppose H is a set of n lines in the plane. Each $h_i \in H$ $(i = 1, \ldots, n)$ has the equation $y = a_ix + b_i$. Thus the input parameters are $(a_1, b_1, a_2, b_2, \ldots, b_n)$. The problem is to compute the arrangement of cells defined by these lines: each cells has dimensions 0, 1 or 2 where 0-cells are just points of intersection defined by the lines, 1-cells are open line segments bounded by 0-cells, and 2-cells are the open connected polygonal regions bounded by the lines. The algorithm for computing an arrangement is fairly easy in the generic case. But dealing with degeneracies is complicated, and it gets much worse in higher dimensions. Hence there is strong motivation to use a generic algorithm, and to perturb the input to ensure non-degeneracy. Suppose 3 distinct lines h_i, h_j, h_k are concurrent (passes through a common point). This is a degeneracy. It amounts to the vanishing of the following determinant:

$$\delta_{ijk} = \det \begin{bmatrix} a_i & b_i & 1 \\ a_j & b_j & 1 \\ a_k & b_k & 1 \end{bmatrix}. \tag{5}$$

We say H is **simple** if no 3 distinct lines are concurrent. We want to perturb the input parameters so that H is simple. For this purpose, let $\varepsilon > 0$ be a small value (to be clarified) and define $h_i(\varepsilon)$ with equation

$$\begin{aligned} y &= a_i(\varepsilon)x + b_i(\varepsilon), \\ a_i(\varepsilon) &= a_i + \varepsilon^{2^{2i}}, \\ b_i(\varepsilon) &= b_i + \varepsilon^{2^{2i-1}}. \end{aligned}$$

The corresponding determinant, expanded as a polynomial in ε is given by

$$\begin{aligned}
\delta_{ijk}(\varepsilon) &= \det \begin{bmatrix} a_i(\varepsilon) & b_i(\varepsilon) & 1 \\ a_j(\varepsilon) & b_j(\varepsilon) & 1 \\ a_k(\varepsilon) & b_k(\varepsilon) & 1 \end{bmatrix} \\
&= \det \begin{bmatrix} a_i & b_i & 1 \\ a_j & b_j & 1 \\ a_k & b_k & 1 \end{bmatrix} \\
&\quad -\varepsilon^{2^{2i-1}} \det \begin{bmatrix} a_j & 1 \\ a_k & 1 \end{bmatrix} \\
&\quad +\varepsilon^{2^{2i}} \det \begin{bmatrix} b_j & 1 \\ b_k & 1 \end{bmatrix} \\
&\quad +\varepsilon^{2^{2j-1}} \det \begin{bmatrix} a_i & 1 \\ a_k & 1 \end{bmatrix} \\
&\quad +\varepsilon^{2^{2j-1}+2^{2k-1}} \\
&\quad +\cdots .
\end{aligned}$$

The key observation is that each coefficient of this polynomial is a subdeterminant of Δ. Moreover, if ε is sufficiently small, and $i < j < k$, then the sign of $\delta_{ijk}(\varepsilon)$ is obtained by evaluating these coefficients in the order shown above (in order of increasing power of ε). The first term is therefore Δ. We stop at the first non-zero coefficient, as this is the sign of the overall polynomial.

This ε-scheme is called **Simulation of Simplicity** (SoS) and clearly works in any dimension. We leave it as an exercise to show that SoS amounts to an admissible sign function, based on a suitable lexicographic ordering $\leq_{\texttt{LEX}}$.

¶9. Example: Euclidean Maximum Spanning Tree. The second example arises in the computation of Euclidean maximum spanning tree [5]. Given n points $p_1, \ldots, p_n$ where $p_i = (x_i, y_i)$, we want to totally sort the set

$$\{D_{ij} : i, j = 1, \ldots, n, (i \neq j)\}$$

where $D_{ij} = \|p_i - p_j\|^2 = (x_i - x_j)^2 + (y_i - y_j)^2$. When $D_{ij} = D_{k\ell}$, Monma et al [5] break ties using the following rule: let $i_1 = \min\{i, j, k, \ell\}$ and relabel $\{i, j, k, \ell\} \setminus \{i_1\}$ as $\{j_1, i_2, j_2\}$ such that

$$\{\{i_1, j_1\}, \{i_2, j_2\}\} = \{\{i, j\}, \{k, \ell\}\}.$$

There are two cases:
(A) If $i_1 \notin \{i, j\} \cap \{k, \ell\}$ then make $D_{i_1,j_1} > D_{i_2,j_2}$ iff $x_{i_1} \geq x_{j_1}$.
(B) If $\{i_1\} = \{i, j\} \cap \{k, \ell\}$ then, without loss of generality, assume $i_1 = i_2$. Make $D_{i_1,j_1} > D_{i_2,j_2}$ iff $2x_{i_1} \geq x_{j_1} + x_{j_2}$.

Let us define the function σ which assigns to polynomials of the form $p = D_{ij} - D_{k\ell}$ a sign $\sigma(p) \in \{-1, +1\}$. The above scheme amounts to defining $\sigma(D_{ij} - D_{k\ell}) = +1$ iff $D_{ij} > D_{k\ell}$ (breaking ties as needed). An exercise below shows that σ arises from an admissible sign function.

¶10. Issues In the next sections, we address three key issues:

- The $\texttt{sign}(p(\mathbf{x}), \mathbf{b})$ must satisfy some notion of consistency. What is it?
- How should we interprete the output from the generic algorithm when the input is degenerate?
- How can we implement such a black box?

The answer to the first question is that, for any finite set of polynomials $A \subseteq \mathbb{R}[\mathbf{x}]$, and for any $\mathbf{b} \in \mathbb{R}^n$ and $\varepsilon > 0$, there exists $\mathbf{b}' \in \mathbb{R}^n$ such that $\|\mathbf{b} - \mathbf{b}'\| < \varepsilon$ such that $\texttt{sign}(p(\mathbf{x}), \mathbf{b}) = \texttt{sign}(p(\mathbf{b}'))$. This proves that the sign function $\texttt{sign}(p(\mathbf{x}), \mathbf{b})$ is "consistent".

Exercises

Exercise 3.1: Show that the sign given to $\Delta_{ijk}(\varepsilon)$ is exactly the sign given by a suitable admissible sign function. In fact, the admissible ordering is $\leq_{\texttt{LEX}}$ for a suitable ordering of the input parameters. ♢

Exercise 3.2: Generalize the SoS scheme to the computation of the sign of any $n \times n$ determinant. Show that this generalization of SoS is again case of our black-box sign, under a suitable $\leq_{\texttt{LEX}}$ ordering. ♢

Exercise 3.3: In the above analysis, we assumed that lines has the form $y = ax + b$. Suppose we now assume lines have equations $ax + by + c = 0$, with $a^2 + b^2 > 0$. What has to be modified in the SoS scheme? ♢

End Exercises

§4. Implementation Issues

We consider the implementation of Black Boxes. Such a scheme would be of general utility. For instance, it could have been used in place of SoS for perturbing sign of determinant, and in the Euclidean Maximum Spanning Tree application. An example from Paterson shows not all perturbation schemes arise from black boxes (Exercise).

One way to implement our black box is to choose a particular admissible ordering, and implement the corresponding black box. For instance, the SoS Scheme has been implemented by Mücke. If we choose the lexicographic ordering $\leq_{\texttt{LEX}}$, we would have a generalization of Mücke's implementation. However, we now want to motivate the need for a more general implementation. This example is suggested by the experience of F. Sullivan and I. Beichl (at the Center for Computing Sciences, Bowie, Maryland) with using the SoS scheme on points on a regular three dimensional grid. The symbolic perturbation may cause our triangulation algorithm to behave very badly. What do we mean by badly? Ideally, we want the triangles to have no sharp angles. For instance, any triangle formed by 3 points in a row of the grid is degenerate, and should be avoided. But our greedy algorithm (guided by any symbolic perturbation scheme) may not know how to avoid such situations.

We first need to fix the algorithm for triangulation. For the present study, assume the the **greedy algorithm**: given a set S of n points, form all $m = \binom{n}{2}$ edges from pairs of points in S. Sort these edges by non-decreasing order of their lengths:

$$e_1, e_2, \ldots, e_m$$

where $|e_i| \leq |e_j|$. We go through this list in order, and put e_i into our triangulation T as long as e_i does not intersect any of the previously picked edges. To break ties, we use our blackbox with pure lexicographic ordering. Consider the performance of the greedy algorithm on the following input sequence

$$(x_1, y_2, x_2, y_2, \ldots, x_{25}, y_{25})$$

representing the points $p_i = (x_i, y_i)$ $(i = 1, \ldots, 25)$. Suppose the points lie on the 5×5 grid:

$$\begin{array}{rrrr}
p_1 = (1,1), & p_2 = (1,2), & p_3 = (1,3), \; p_4 = (1,4), & p_5 = (1,5) \\
p_6 = (2,1), & p_7 = (2,2), & \cdots & p_{10} = (2,5) \\
p_{11} = (3,1), & p_{12} = (3,2), & \cdots & p_{15} = (3,5) \\
p_{16} = (4,1), & p_{17} = (4,2), & \cdots & p_{20} = (4,5) \\
p_{21} = (5,1), & p_{22} = (5,2), & \cdots & p_{25} = (5,5).
\end{array}$$

Figure 2 shows the result of applying the pure lexicographic perturbation. It is not a good triangulation by any measure.

THIS IS A PLACE HOLDER – NO FIGURE YET

Figure 2: Triangulation of points on a grid

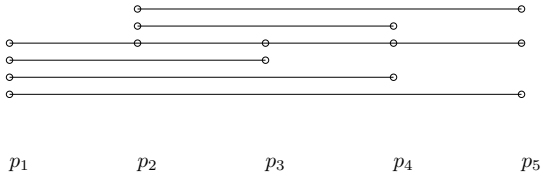


Figure 3: Schematic triangulation of a set of points on a line.

To understand analytically this behavior, let us analyze the output when there is only one row of points $p_1 = (1,1), p_2 = (1,2), \ldots, p_n = (1,n)$. Let d_{ij} be the distance between p_i and p_j. Clearly, $d_{ij} = |i-j|$. What is the result of comparing $d_{i,i+2} : d_{j,j+2}$?

So we would like to have general orderings at our disposal. It turns out that there is a beautiful theory of admissible orderings that can help us do this.

Another way to prevent triangulations such as Figure 2 is to use randomization. Suppose we use lexicographic ordering, but we first randomly reorder the variables first.

Exercises

Exercise 4.1: Assume the total degree ordering. How can the successive evaluations of a polynomial be sped up at the cost of some space (to store intermediate results)? Quantify this speedup. ♢

Exercise 4.2: (Paterson) Give an example of a perturbation scheme that does not arise from our black box. ♢

End Exercises

§5. Algebraic Consistency

We address the consistency of admissible sign functions. We will use an algebraic approach here.

Let R be any commutative ring with unit 1. For instance, $R = \mathbb{Z}$ or $R = \mathbb{R}$. A function $\sigma : R \to \{-1, 0, +1\}$ is called a **sign function** for R provided it satisfies these 4 axioms. For all $a, b \in R$:

(A0) $\sigma(a) = 0$ iff $a = 0$.

(A1) $\sigma(-a) = -\sigma(a)$.

(A2) $\sigma(a) > 0$ and $\sigma(b) > 0$ implies $\sigma(a+b) > 0$.

(A3) $\sigma(ab) = \sigma(a)\sigma(b)$.

If σ is a sign function for R, then it induces a total ordering $<_\sigma$ on R where

$$a <_\sigma b \Rightarrow \sigma(a-b) = -1.$$

It is easy to verify that $<_\sigma$ is indeed a total ordering: if a, b are distinct elements in R, then $a <_\sigma b$ or $b <_\sigma a$, by (A0). Moreover, (A1) says that only one of $a <_\sigma b$ or $b <_\sigma a$ must hold. Transitivity follows from (A2).

We need the following lemma about admissible orderings $\leq_A$:

Lemma 1. *Let $u, v, u', v' \in \mathrm{PP}$ and either $u \neq u'$ or $v \neq v'$. Then $uv \leq_A u'v'$ implies $u <_A u'$ or $v <_A v'$.*

See [8] for a proof. The next theorem constitute our "algebraic consistency" claim.

Theorem 2. *Let R be any ordered ring and $S = R[x_1, \ldots, x_n]$. Fix $\mathbf{a} = (a_1, \ldots, a_n) \in R^n$ and any admissible ordering $\leq_A$. Let $\sigma = \sigma_{\leq_A, \mathbf{a}}$ be the function that assigns to each polynomial $p \in S$ the sign $\mathtt{sign}_{\leq_A}(p; \mathbf{a})$ (see (4)). Then σ is a sign function for S.*

Proof. Recall that for a non-zero $p \in S$, our black box sign, $\sigma(p)$, is the first non-zero value in the sequence $\partial(p;\mathbf{a})$. Axiom (A0) is immediate. Axiom (A1) comes from the fact that $\partial(-p;\mathbf{a})$ is obtained from $\partial(p;\mathbf{a})$ by negating every value. Axiom (A2) comes from the fact that $\partial(p+q;\mathbf{a}) = \partial(p;\mathbf{a}) + \partial(q;\mathbf{a})$ (componentwise addition). To show (A3), let the first non-zero entry of $\partial(p;\mathbf{a})$ be $\partial_u(p)(\mathbf{a})$ and the first non-zero entry of $\partial(q;\mathbf{a})$ be $\partial_v(q)(\mathbf{a})$, for some $u, v \in \text{PP}$. Then

$$\partial_{uv}(pq) = \sum_{u'} \partial_{u'}(p)\partial_{v'}(q)$$

where $u' \in \text{PP}$ ranges over all divisors of uv and $v' = uv/u'$. But the previous lemma, unless $u = u'$ and $v = v'$, $u'v' \leq_A uv$ implies $u >_A u'$ or $v >_A v'$. Hence $\partial_{u'}(p)(\mathbf{a}) = 0$ or $\partial_{v'}(p)(\mathbf{a}) = 0$. This proves

$$\partial_{uv}(pq)(\mathbf{a}) = \partial_u(p)(\mathbf{a})\partial_v(q)(\mathbf{a}).$$

The same argument shows that for all $w <_A uv$,

$$\partial_w(pq)(\mathbf{a}) = 0.$$

Thus, the first non-zero entry of $\partial(pq;\mathbf{a})$ has the sign $\partial_u(p)(\mathbf{a})\partial_v(q)(\mathbf{a}) = \sigma(p)\sigma(q)$, as desired. **Q.E.D.**

¶11. Geometric Consistency. To motivate the need for a geometric notion of consistency, suppose that we have n collinear points, $q_1, \ldots, q_n \in \mathbb{R}^2$. Let Δ_{ijk} denote determinant whose sign is the $LeftTurn(q_i, q_j, q_k)$ predicate (Lecture II.1). Since the lines are collinear, $\Delta_{ijk} = 0$ for all i, j, k. On the other hand, our black box will assign a non-zero sign $s_{ijk} \in \{-1, +1\}$ to Δ_{ijk}. The geometric consistency question asks to know if there is an actual perturbation of $q_1, \ldots, q_n$ which achieves these s_{ijk}'s. This question is answered in the affirmative in [7].

It is a consequence of the next theorem. For $\delta > 0$ and $\mathbf{a}, \mathbf{a}' \in \mathbb{R}^n$, we say $\mathbf{a}'$ is a **δ-perturbation** of $\mathbf{a}$ if $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{a}' = (a'_1, \ldots, a'_n)$ and $|a'_i - a_i| \leq \delta$ for all $i = 1, \ldots, n$.

Theorem 3. *Let σ be any black box sign function for $\mathbb{R}[x_1, \ldots, x_n]$ (based on some admissible ordering and a choice of $\mathbf{a} \in \mathbb{R}^n$). For any finite set $P \subseteq \mathbb{R}[x_1, \ldots, x_n]$ of polynomials and any $\delta > 0$, there exists $\mathbf{a}' = (a'_1, a'_2, \ldots, a'_n)$, a δ-perturbation of $\mathbf{a}$, such that for all $p \in P$,*

$$\sigma(p) = \texttt{sign}(p(\mathbf{a}')).$$

In other words, the signs assigned by our black box to any finite set P of real polynomials can be achieved by an actual perturbation. Suppose A' is the transformed algorithm in figure 1, and $\mathbf{a}$ is any input for A'. Consider the branching (computation) path π through A' that is taken by input $\mathbf{a}$. This result shows that there is an arbitrarily small perturbation $\mathbf{a}'$ of $\mathbf{a}$ such that $\mathbf{a}'$ is non-degenerate for A' and $\mathbf{a}'$ would exact the same path π.

Exercises

Exercise 5.1: Show the following properties of $<_\sigma$ where σ is a sign function on R. For all $a, b, c, d \in R$:
(a) $a <_\sigma b$ implies $a + c <_\sigma b + c$.
(b) $a <_\sigma b$, $\sigma(c) > 0$ implies $ac <_\sigma bc$.
(c) If b^{-1}, a^{-1} exists, then $a <_\sigma b$ iff $b^{-1} <_\sigma a^{-1}$.
(d) $a + b <_\sigma c + d$ implies $a <_\sigma c$ or $b <_\sigma d$.
(e) $\sigma(a) = \sigma(b) = +1$ and $ab <_\sigma cd$ implies $a <_\sigma c$ or $b <_\sigma d$. ♢

Exercise 5.2: Show that the function σ defined by the Monma, et al, scheme arises from an admissible sign function. ♢

End Exercises

§6. Perturbation Framework

Following Seidel, we define a **problem** to be a partial function $f : X \to Y$ with suitable topologies on the input X and output Y spaces. We write $f(x)\downarrow$ and $f(x)\uparrow$ depending on whether $f(x)$ is defined or not. Typically, $X = \mathbb{R}^{nd}$ (a set of n points in d dimensions, with the usual Euclidean topology, and $Y = D \times R$ where D is a finite set with the discrete topology and R is a direct union of real spaces with the Euclidean topology. The domain of f is $dom(f) := \{x \in X : f(x)\downarrow\}$. We say f is **generic** if the closure $\overline{dom(f)} = X$.

Let us give two examples: (1) Convex hull volume and (2) Convex hull facet structure. In both cases, $X = \mathbb{R}^{nd}$, In (1), $Y = \mathbb{R}$. In (2), $Y = D \times R$ where D is the finite set of all the labeled combinatorial structures that may be output. Also, R can be empty, or for our purposes below, we can let R denote the k-dimensional volume of each k-dimensional face of the output. In this way, problem (1) is can be embedded in problem (2). It is imporant to note that the structures are labeled by the index of input points.

Let z be a set of nd variables (called input variables). A **branching straightline program** (BSLP) $A = A[z]$ is a rooted tree in which each non-branching node computes a value (using an input variable or previously computed values) or evaluates a predicate p on the input and then performs a three-way branch according to the sign of $p(x)$. Each non-branching node is associated with a (programming) variable. Note that we assume that predicates p directly operates on the input x. For any input x, the computation on x is a path that is either non-terminating or terminates at a leaf which by definition is a non-branching node. The output of A on x, denoted $A(x)$, is undefined if the path taken by x is non-terminating, and otherwise is the value computed at last node of the path. We say A is an algorithm for f if for all $x \in X$, $A(x) = f(x)$. An algorithm is **generic** if is an algorithm for a generic problem. It is **general** if it defines a total function on X.

If $x \in X$, a perturbed instance of x is a curve x^* rooted at x: $x^* : \mathbb{R}_{\geq 0} \to X$ such that $x^*(0) = x$. A perturbation scheme Q defines a perturbed instance for each $x \in X$. Once the perturbation is fixed, any problem f is transformed into a perturbed problem

$$f^* : X \to Y$$

where $f^*(x) = \lim_{n\to\infty} f(x(1/n))$ where $f^*(x)$ may be undefined if the limit does not exist. We say that the perturbation scheme Q is **valid** for f if $f^*(x)$ is defined for all $x \in X$ and is non-zero.

We similarly say Q is **valid** for A if $p^*(x) = \lim_{n\to\infty} p(x(1/n))$ is defined and non-zero for all predicates in A. If Q is valid for A, we can transform A to some A^* in the obvious way.

Theorem: Let A be an algorithm for f. Assume Q is valid for f and for algorithm A. Then A^* solves f^* and for each $x \in X$, if f is continuous at x then $A^*(x) = f(x)$. Moreover, A^* is a general algorithm.

Pf: Since Q is valid for f, the problem f^* is total. Also, f^* must be computed by A^*. If f

We can reformulate the above theorem in terms of $\mathbb{R}(\Omega)$ where $\mathbb{R}(\Omega) = \mathbb{R}(\omega)$, ω is an infimaximal.

§7. Numerical Perturbation

We present some efficient numerical schemes for perturbation. In particular, we are interested in numerical linear perturbations: $x \mapsto x + \varepsilon x^*$ where x^*

§8. Perturbing towards Non-Degenerate Instances

A fundamentally different idea for perturbation is proposed by Raimund Seidel: for any problem P and for each input size n, suppose we can find a non-degenerate instance G_n. Then for any input instance I of size n, we want to use the perturbed instance

$$I + \varepsilon G_n$$

where $\varepsilon > 0$ is a small constant.

Recall that one of our basic goals figure 1 is to transform a generic A to a general A'. In this case, we need to be able to generate G_n on the fly. This turns out to be easy in some problems. Consider the problem P of computing the convex hull of points in $\mathbb{R}^d$. For each n, there are well-known constructions for a set G_n of n points in general position. Basically, G_n can be any n distinct points chosen on the moment curve,

$$C = \{(t, t^2, \ldots, t^d) : t \in \mathbb{R}\}$$

Another direction in perturbation research is known as "controlled perturbation", as opposed to random perturbation (or perturbation where we don't care how the perturbation behaves as long as it is bounded). An example of why we want this is in the intersection of two convex polyhedron. We want the perturned polygons to grow outwardly, so that any tangential intersection between a vertex and a face can be detected. In random perturbation, we may not see such tangential intersections.

Removing Degeneracies: Gomez et al [4, 3] consider algorithms for finding orthogonal and perspective projections (respective) to remove degenaracies in point sets.

References

[1] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 16–23, 1995.

[2] A. R. Forrest. Foreword. *ACM Trans. on Graphics*, 3& 4, 1984. Two Special Issues on Computational Geometry. Guest Editors: R. Forrest, L.Guibas, J.Nievergelt.

[3] F. Gomez, F. Hurtado, T. Sellares, and G. Toussaint. On degeneracies removable by perspective projections. *Int. J. of Mathematical Algorithms*, 2:227–248, 2001.

[4] F. Gomez, S. Ramaswami, and G. Toussaint. On removing non-degeneracies assumptions in computational geometry. In *Proc. Italian Conf. on Algorithms*, pages 52–63, 1997. March 12-14, 1997, Rome, Italy.

[5] C. Monma, M. Paterson, S. Suri, and F. Yao. Computing Euclidean maximum spanning trees. *Algorithmica*, 5:407–419, 1990. Also: 4th ACM Symp.Comp.Geom. (1988).

[6] R. Sedgewick. *Algorithms in C: Part 5, Graph Algorithms.* Addison-Wesley, Boston, MA, 3rd edition edition, 2002.

[7] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. of Computer and System Sciences*, 40(1):2–18, 1990. Also: *Proc. 4th ACM Symp.Comp.Geom*, 1988, pp.134–142.

[8] C. K. Yap. Symbolic treatment of geometric degeneracies. *J. of Symbolic Computation*, 10:349–370, 1990. Also, *Proc. International IFIPS Conf. on System Modelling and Optimization,* Tokyo, 1987, Springer Lecture Notes in Control and Information Science, Vol.113, pp.348-358.

Lecture 10
NUMERICAL FILTERS

This theme of this chapter is the certification of approximate computation. This is best exemplified by a numerical program P that uses machine floating-point arithmetic. How do we certify that the output of P has certain properties? Conceptually, let us think of the certifier C as a program which, given the output of P, will always answer YES or MAYBE. We see that certification is a "partial predicate": it does not have to answer NO. So the MAYBE output could be produced even when the correct answer is YES. Certifiers are very important for the ultimate practical efficiency of EGC algorithms.

§1. Numerical Filters

An avenue to gain efficiency in EGC computation is to exploit machine floating-point arithmetic which is fast and highly optimized on current hardware. The idea is simply this: we must "check" or "certify" the output of machine evaluation of predicates, and only go for the slower exact methods when this fails.

In EGC, certifiers are usually **(numerical) filters**. These filters certify property of computed numerical values, typically its sign. This often amounts to computing some error bound, and comparing the computed value with this bound. When such filters aim to certifying machine floating-point arithmetic, we call them **floating-point filters**. We can also consider a cascade of certifiers of increasing effectivity for the problem. Such cascades can be quite effective [**?**, **?**, **?**]. There is an obvious connection between the notion of certifiers and the area of program checking [**?**, **?**]. It is also worth mentioning that similar techniques have later been used on large determinant sign evaluations based on distance to the nearest singularity [**?**].

There are two main classifications of numerical filters: static or dynamic. Static filters are those that can be computed at compile time for the most part, and they incur a low overhead at runtime. However, static error bounds may be overly pessimistic and thus less effective. Dynamic filters exhibit opposite characteristics: they have higher runtime cost but are much more effective (i.e., fewer false rejections). We can have semi-static filters which combine both features.

Certifiers can be used at different levels of granularity: from individual machine operations (e.g., arithmetic operations for dynamic filters), to subroutines (e.g., geometric predicates [**?**]), and to algorithms (e.g., [**?**]). See Funke et al. [**?**] for a general framework for filtering each "step" of an algorithm.

Computing upper bounds in machine arithmetic. In the implementation of numerical filters, we need to compute sharp upper bounds on numerical expressions. To be specific, suppose you have IEEE double values x and y. How can you compute an upper bound on $|z|$ where $z = xy$? We first compute

$$\widetilde{z} \leftarrow |x| \odot |y|. \tag{1}$$

Here, $|\cdot|$ is done exactly by the IEEE arithmetic, but the multiplication $\odot$ is not exact. One aspect of IEEE arithmetic is that we can change the rounding modes [**?**]. Thus changing the rounding mode to round towards $+\infty$, we will have $\widetilde{z} \geq |z|$. Otherwise, we only know that $\widetilde{z} = |z|(1+\delta)$ where $|\delta| \leq \mathbf{u}$. Here $\mathbf{u} = 2^{-53}$ is the "unit of rounding" for the arithmetic. We will describe the way to use the rounding modes later, in the interval arithmetic section. But here, instead to avoid rounding modes, we further compute $\widetilde{w}$ as follows:

$$\widetilde{w} \leftarrow \widetilde{z} \odot (1 + 4\mathbf{u}). \tag{2}$$

It is assumed that overflow and underflow do not occur during the computation of $\widetilde{w}$. Note that $1+4\mathbf{u} = 1+2^{-51}$ is exactly representable. Therefore, we know that $\widetilde{w} = \widetilde{z}(1+4\mathbf{u})(1+\delta')$ for some δ satisfying $|\delta'| \leq \mathbf{u}$. Hence,

$$\begin{aligned} \widetilde{w} &= z(1+\delta)(1+\delta')(1+4\mathbf{u}) \\ &\geq z(1-2\mathbf{u}+\mathbf{u}^2)(1+4\mathbf{u}) \\ &= z(1+2\mathbf{u}-7\mathbf{u}^2+4\mathbf{u}^3) \\ &> z \end{aligned}$$

Note that if any of the operations $\oplus$, $\ominus$ or $\oslash$ is used in place of $\odot$ in (1), the same argument still shows that $\widetilde{w}$ is an upper bound on the actual value. We summarize this result:

Lemma 1 *Let E be any rational numerical expression and let $\widetilde{E}$ be the approximation to E evaluated using IEEE double precision arithmetic. Assume the input numbers in E are IEEE doubles and E has $k \geq 1$ operations.*
(i) We can compute an IEEE double value $\texttt{MaxAbs}(E)$ *satisfying the inequality* $|E| \leq \texttt{MaxAbs}(E)$*, in $3k$ machine operations.*
(ii) If all the input values are positive, $2k$ machine operations suffice.
(iii) The value $\widetilde{E}$ is available as a side effect of computing $\texttt{MaxAbs}(E)$*, at the cost of storing the result.*

Proof. We simply replace each rational operation in E by at most 3 machine operations: we count 2 flops to compute $\widetilde{z}$ in equations (1), and 1 flop to compute $\widetilde{w}$ in (2). In case the input numbers are non-negative, $\widetilde{z}$ needs only 1 machine operation. Q.E.D.

0.1 Static Filters

Fortune and Van Wyk [**?**] were the first to implement and quantify the efficacy of filters for exact geometric computation. Their filter was implemented via the `LN` preprocessor system. Let us now look at the simple filter they implemented (which we dub the "FvW Filter"), and some of their experimental results.

The FvW filter. Static error bounds are easily maintained for a polynomial expression E with integer values at the leaves. Let $\widetilde{E}$ denote the IEEE double value obtained by direct evaluation of E using IEEE double operations. Fortune and Van Wyk compute a bound $\texttt{MaxErr}(E)$ on the absolute error,

$$|E - \widetilde{E}| \leq \texttt{MaxErr}(E). \tag{3}$$

It is easy to use this bound as a filter to certify the sign of $\widetilde{E}$: if $|\widetilde{E}| > \texttt{MaxErr}(E)$ then $\texttt{sign}\widetilde{E} = \texttt{sign}E$. Otherwise, we must resort to some fall back action. For simplicity, assume this action is to immediately use an infallible method, namely computing exactly using a Big Number package.

Expr E	$\texttt{MaxLen}(E)$	$\texttt{MaxErr}(E)$
Var x	$\texttt{MaxLen}(x)$ given	$\max\{0, 2^{\texttt{MaxLen}(E)-53}\}$
$F \pm G$	$1 + \max\{\texttt{MaxLen}(F), \texttt{MaxLen}(G)\}$	$\texttt{MaxErr}(F) + \texttt{MaxErr}(G) + +2^{\texttt{MaxLen}(F\pm G)-53}$
FG	$\texttt{MaxLen}(F) + \texttt{MaxLen}(G)$	$\texttt{MaxErr}(F)2^{\texttt{MaxLen}(G)} + \texttt{MaxErr}(G)2^{\texttt{MaxLen}(F)}$ $+2^{\texttt{MaxLen}(FG)-53}$

Table 1: Parameters for the FvW filter

Let us now see how to compute $\texttt{MaxErr}(E)$. It turns out that we also need the magnitude of E. The base-2 logarithm of the magnitude is bounded by $\texttt{MaxLen}(E)$. Thus, we say that the FvW filter has two **filter parameters**,

$$\texttt{MaxErr}(E), \quad \texttt{MaxLen}(E). \tag{4}$$

We assume that each input variable x is assigned an upper bound $\texttt{MaxLen}(x)$ on its bit length. Inductively, if F and G are polynomial expressions, then $\texttt{MaxLen}(E)$ and $\texttt{MaxErr}(E)$ are defined using the rules in Table 1.

Observe that the formulas in Table 1 assume exact arithmetic. In implementations, we compute upper bounds on these formulas. We assume that the filter has failed in case of an overflow; it is easy to see that no underflow occurs when evaluating these formulas. Checking for exceptions has an extra overhead. Since $\texttt{MaxLen}(E)$ is an integer, we can evaluate the corresponding formulas using IEEE arithmetic exactly. But the formulas for $\texttt{MaxErr}(E)$ will incur error, and we need to use some form of lemma 1.

Framework for measuring filter efficacy. We want to quantify the efficacy of the FvW Filter. Consider the primitive of determining the sign of a 4×4 integer determinant. First look at the unfiltered performance of this primitive. We use the IEEE machine double arithmetic evaluation of this determinant (with possibly incorrect sign) as the **base line** for speed; this is standard procedure. This base performance is then compared to the performance of some standard (off-the-shelf) Big Integer packages. This serves as the **top line** for speed. The numbers cited in the paper are for the Big Integer package in `LEDA` (circa 1995), but the general conclusion for other packages are apparently not much different. For random 31-bit integers, the top line time yields 60 time increase over the base line. We will say

$$\sigma = 60 \tag{5}$$

in this case; the symbol σ reminds us that this is the "slowdown" factor. Clearly, $\sigma = \sigma(L)$ is a function of the bit length L as well. For instance, with random 53-bit signed integers, the factor σ becomes 100. Next, still with $L = 31$, but using static filters implemented in `LN`, the factor σ ranges from 13.7 to 21.8, for various platforms and CPU speeds [**?**, Figure 14]. For simplicity, we say $\sigma = 20$, for some mythical combination of platforms and CPUs. Thus the static filters improve the performance of exact arithmetic by the factor

$$\phi = 60/20 = 3. \tag{6}$$

In general, using unfiltered exact integer arithmetic as base line, the symbol ϕ (or $\phi(L)$) denotes the "filtered improvement". We use it as a measure of the efficacy of filtering.

The above experimental framework is clearly quite general, and estimates the efficacy of a filter by a number ϕ. The framework requires the following choices: (1) a "test algorithm" (we picked one for 4×4 determinants), (2) the "base line" (the standard is IEEE double arithmetic), (3) the "top line" (we

picked LEDA's Big Integer), (4) the input data (we used random 31-bit integers). Another measure of efficacy is the fraction ρ of approximate values $\widetilde{E}$ which fail to pass the filter. In [**?**], a general technique for assessing the efficacy of an arithmetic filter is proposed based on an analysis which consists of evaluating both the threshold value and the probability of failure of the filter.

For a true complexity model, we need to introduce size parameters. In EGC, two size parameters are of interest: the combinatorial size n and the bit size L. Hence all these parameters ought to be written as $\sigma(n, L)$, $\phi(n, L)$, etc.

Realistic versus synthetic problems. Static filters have an efficacy factor $\phi = 3$ (see (6)) in evaluating the sign of randomly generated 4-dimensional matrices ($L = 31$). Such problems are called "synthetic benchmarks" in [**?**]. It would be interesting to see the performance of filters on **realistic benchmarks**, i.e., actual algorithms for natural problems that we want to solve. But even here, there are degrees of realism. Let us[1] equate realistic benchmarks with algorithms for problems such as convex hulls, triangulations, etc. The point of realistic benchmarks is that they will generally involve a significant amount of non-numeric computation. Hence the ϕ-factor in such settings ought to be different from (in fact, less than) the synthetic setting. To quantify this, suppose that a fraction

$$\beta \quad (0 \leq \beta \leq 1) \tag{7}$$

of the running time of the algorithm is attributable to numerical computation. After replacing the machine arithmetic with exact integer arithmetic, the overall time becomes $(1 - \beta) + \beta\sigma = 1 + (\sigma - 1)\beta$. With filtered arithmetic, the time becomes $1 + (\sigma - 1)\beta\phi^{-1}$. So "realistic" efficacy factor ϕ' for the algorithm is

$$\phi' := \frac{(1 + (\sigma - 1)\beta}{1 + (\sigma - 1)\beta/\phi}.$$

It is easy to verify that $\phi' < \phi$ (since $\phi > 1$). Note that our derivation assumes the original time is unit! This normalization is valid in our derivation because all the factors σ, ϕ that we use are ratios and are not affected by the normalization.

The factor β is empirical, of course. But even so, how can we estimate this? For instance, for 2- and 3-dimensional Delaunay triangulations, Fortune and Van Wyk [**?**] noted that $\beta \in [0.2, 0.5]$. Burnikel, et al. [**?**] suggest a simple method for obtaining β: simply execute the test program in which each arithmetic operation is repeated $c > 1$ times. This gives us a new timing for the test program,

$$T(c) = (1 - \beta) + c\beta.$$

Now, by plotting the running time $T(c)$ against c, we obtain β as the slope.

Some detailed experiments on 3D Delaunay triangulations have been made by Devillers and Pion [**?**], comparing different filtering strategies; they conclude that cascading predicates is the best scheme in practice. Other experiments on interval arithmetic have been done by Seshia, Blelloch and Harper [**?**].

0.2 Dynamic Filters

To improve the quality of the static filters, we can use runtime information about the actual values of the variables, and dynamically compute the error bounds. We can again use $\texttt{MaxErr}(E)$ and $\texttt{MaxLen}(E)$ as found in Table 1 for static error. The only difference lies in the base case: for each variable x, the $\texttt{MaxErr}(x)$ and $\texttt{MaxLen}(x)$ can be directly computed from the value of x. It is

[1] While holding on tightly to our theoretician's hat!

possible to make a dynamic version of the FvW filter, but we will not detail it here due to lack of space.

The BFS filter. This is a dynamic filter, but it can also be described as "semi-static" (or "semi-dynamic") because one of its two computed parameters is statically determined. Let E be a radical expression, *i.e.*, involving $+, -, \times, \div, \sqrt{\cdot}$. Again, let $\widetilde{E}$ be the machine IEEE double value computed from E in the straightforward manner (this time, with division and square-roots). In contrast to the FvW Filter, the filter parameters are now

$$\texttt{MaxAbs}(E), \quad \texttt{Ind}(E).$$

The first is easy to understand: $\texttt{MaxAbs}(E)$ is an upper bound on $|E|$. The second, called the **index** of E, is a natural number whose rough interpretation is that its base 2 logarithm is the number of bits of precision which are lost (i.e. which the filter cannot guarantee) in the evaluation of the expression. Together, they satisfy the following invariant:

$$|E - \widetilde{E}| \leq \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-53} \tag{8}$$

The value 2^{-53} may be replaced by the unit roundoff error $\mathbf{u}$ in general. Table 2 gives the recursive rules for maintaining $\texttt{MaxAbs}(E)$ and $\texttt{Ind}(E)$. The base case (E is a variable) is covered by the first two rows: notice that they distinguish between exact and rounded input variables. A variable x is **exact** if its value is representable without error by an IEEE double. In any case, x is assumed not to lie in the overflow range, so that the following holds

$$|\texttt{round}(x) - x| \leq |x| 2^{-53}.$$

The bounds are computed using IEEE machine arithmetic, denoted

$$\oplus, \quad \ominus, \quad \odot, \quad \oslash, \quad \tilde{\sqrt{\cdot}}.$$

The question arises: what happens when the operations lead to over- or underflow in computing the bound parameters? It can be shown that underflows for $\oplus$, $\ominus$ and $\tilde{\sqrt{\cdot}}$ can be ignored, and in the case of $\odot$ and $\oslash$, we just have to add a small constant $MinDbl = 10^{-1022}$ to $\texttt{MaxAbs}(E)$.

Expression E	$\texttt{MaxAbs}(E)$	$\texttt{Ind}(E)$
Exact var. x	x	0
Approx. var. x	$\texttt{round}(x)$	1
$E = F \pm G$	$\texttt{MaxAbs}(F) \oplus \texttt{MaxAbs}(G)$	$1 + \max\{\texttt{Ind}(F), \texttt{Ind}(G)\}$
$E = FG$	$\texttt{MaxAbs}(F) \odot \texttt{MaxAbs}(G)$	$1 + \texttt{Ind}(F) + \texttt{Ind}(G)$
$E = F/G$	$\frac{\lvert\widetilde{F}\rvert \oplus (\texttt{MaxAbs}(F) \oslash \texttt{MaxAbs}(G))}{(\lvert\widetilde{G}\rvert \oslash \texttt{MaxAbs}(G)) \ominus (\texttt{Ind}(G)+1) 2^{-53}}$	$1 + \max\{\texttt{Ind}(F), \texttt{Ind}(G) + 1\}$
$E = \sqrt{F}$	$\begin{cases} (\texttt{MaxAbs}(F) \oslash \widetilde{F}) \odot \widetilde{E} & \text{if } \widetilde{F} > 0 \\ \tilde{\sqrt{\texttt{MaxAbs}(F)}} \odot 2^{26} & \text{if } \widetilde{F} = 0 \end{cases}$	$1 + \texttt{Ind}(F)$

Table 2: Parameters of the BFS filter

Assuming (8), we have the following criteria for certifying the sign of $\widetilde{E}$:

$$|\widetilde{E}| > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-53} \tag{9}$$

Of course, this criteria should be implemented using machine arithmetic (see (2) and notes there). One can even certify the exactness of $\widetilde{E}$ under certain conditions. If E is a polynomial expression (*i.e.*, involving $+, -, \times$ only), then $E = \widetilde{E}$ provided

$$1 > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-52}. \tag{10}$$

Finally, we look at some experimental results. Table 3 shows the σ-factor (recall that this is a slowdown factor compared to IEEE machine arithmetic) for the unfiltered and filtered cases. In both cases, the underlying Big Integer package is from `LEDA`. The last column adds compilation to the filtered case. It is based on an expression compiler, EXPCOMP, somewhat in the spirit of `LN` (see Section 0.3). At $L = 32$, the ϕ-factor (recall this is the speedup due to filtering) is $65.7/2.9 = 22.7$. When compilation is used, it improves to $\phi = 65.7/1.9 = 34.6$. [Note: the reader might be tempted to deduce from these numbers that the BFS filter is more efficacious than the FvW Filter. But the use of different Big Integer packages, platforms and compilers, etc, does not justify this conclusion.]

BitLength L	Unfiltered σ	BFS Filter σ	BFS Compiled σ
8	42.8	2.9	1.9
16	46.2	2.9	1.9
32	65.7	2.9	1.9
40	123.3	2.9	1.8
48	125.1	2.9	1.8

Table 3: Random 3×3 determinants

While the above results look good, it is possible to create situations where filters are ineffective. Instead of using matrices with randomly generated integer entries, we can use degenerate determinants as input. The results recorded in Table 4 indicate that filters have very little effect. Indeed, we might have expected it to slow down the computation, since the filtering efforts are strictly extra overhead. In contrast, for random data, the filter is almost always effective in avoiding exact computation.

BitLength L	Unfiltered σ	BFS Filter σ	BFS Compiled σ
8	37.9	2.4	1.4
16	45.3	2.4	1.4
32	56.3	56.5	58.4
40	117.4	119.4	117.5
48	135.2	136.5	135.1

Table 4: Degenerate 3×3 determinants

The original paper [**?**] describes more experimental results, including the performance of the BFS filter in the context of algorithms for computing Voronoi diagrams and triangulation of simple polygons.

Dynamic filter using interval arithmetic. As mentioned, a simpler and more traditional way to control the error made by floating-point computations is to use interval arithmetic [**?**, **?**]. Some work has been done by Pion et al. [**?**, **?**, **?**] in this direction.

Interval arithmetic represents the error bound on an expression E at runtime by an interval $[E_m; E_p]$ where E_m, E_p are floating-point values and $E_m \leq E \leq E_p$. Interval operations such as $+, -, \times, \div, \sqrt{}$ can be implemented using IEEE arithmetic, exploiting its rounding modes. Changing the rounding mode has a certain cost (mostly due to flushing the pipeline of the FPU), but the remark has been made that it can usually be done only twice per predicate: at the beginning by setting the rounding mode towards $+\infty$, and at the end to reset it back to the default mode. This can be achieved by observing that computing $a+b$ rounded towards $-\infty$ can be emulated by computing $-((-a)-b)$ rounded towards $+\infty$. A similar remark can be done for $-, \times, \div$. Therefore it is possible to eliminate most rounding mode changes, which makes the approach much more efficient.

Most experimental studies (e.g. [**?**, **?**]) show that using interval arithmetic implemented this way usually induces a slowdown factor of 3 to 4 on algorithms, compared to floating-point. It is also noted that interval arithmetic is the most efficacious dynamic filter, failing rarely. This technique is available in the CGAL library, covering all predicates of the geometry kernel.

0.3 Tools for automatic generation of code for the filters

Given the algebraic formula for a predicate, it is tedious and error-prone to derive the filtered version of this predicate manually. Therefore tools have been developed to generating such codes.

We have already mentioned the first one, LN, which targets the FvW filter [**?**]. This tool does not address the needs of more complex predicates, which may contain divisions, square roots, branches or loops. Another attempt has been made by Funke et al. [**?**] with a tool called EXPCOMP (standing for expression compiler), which parses slightly modified `C++` code of the original predicate, and produces static and semi-static BFS filters for them.

The `CGAL` library implements filtering using interval arithmetic for all the predicates in its geometry kernel. The filtered versions of these predicates used to be generated by a Perl script [**?**, **?**], but the current approach uses template mechanisms to achieve this goal entirely within C++. The advantage of dynamic filters is that the code generator need not analyze the internal structure of the predicate.

Most recently, Nanevski, Blelloch and Harper [**?**] have proposed a tool that produces filters using Shewchuk's method [**?**], for the SML language, from an SML code of the predicate.Seeing these past and ongoing works, it seems important to have general software tools to generate such numerical code. Such work is connected to compiler technology and static code analysis.

§2. EXTRA NOTES for Filters

Model for certifying and checking. We give a simple formal model for certifying (filtering) and checking. Assume I and O are some sets called the **input and output spaces**. In numerical problems, it is often possible to identify I and O with suitable subsets of $\mathbb{R}^n$ or $\cup_{n\geq 1}\mathbb{R}^n$. A **computational problem** is simply a subset $\pi \subseteq I \times O$. For simplicity, we assume that for all $x \in I$, there exists y such that $(x, y) \in \pi$ (for most problems, this condition can be artificially enforced without changing the overall complexity). A **program** is a pair (P, C_P) where $P : I \to O$ is a partial function and $C_P : I \to \mathbb{R}_{\geq 0}$ is its complexity function. Thus P on input $x \in I$ takes time $C_P(x)$. Note that even if $P(x) \uparrow$, the time taken is $C_P(x)$. We usually refer to P as "the program" and leave its complexity function implicit. As a partial function, $P(x)$ may be undefined at x, written $P(x) \uparrow$; otherwise we write $P(x) \downarrow$. We say P is a **partial algorithm** for π if for all $x \in I$, if $P(x) \downarrow$ then $(x, P(x)) \in \pi$. An **algorithm** A for π is a partial algorithm that happens to be a total function. Informally, we say a partial program P is "efficacious" if for "most" $x \in X$, $P(x) \downarrow$.

A pair (P, A) is an **anchored algorithm** for π if P is partial algorithm for π and A is an algorithm for π. This pair (P, A) represents a new algorithm for π in which, for each input x, we first compute $P(x)$; if $P(x) \downarrow$ then output $P(x)$ and otherwise we compute and output $A(x)$. Logically, the algorithm (P, A) could be identical to A. It is the complexity considerations that motivates (P, A). We have

$$C_{(P,A)}(x) = C_P(x) + \delta(x)C_A(x)$$

where[2]

$$\delta(x) = \begin{cases} 0 & \text{if } P(x) \downarrow, \\ 1 & \text{else.} \end{cases}$$

Thus the algorithm (P, A) could be more efficient than A when P is efficacious and more efficient than A.

How do we obtain partial algorithms for π? Let P be an arbitrary total program. A **certifier** for P (relative to π) is a program that computes a total function $F : I \times O \to \{0, 1\}$ such that for all $x \in X$, if $F(x, P(x)) = 1$ then $(x, P(x)) \in \pi$. A **checker** C for P (relative to π) is a certifier for P (relative to π) such that if $C(x, P(x)) = 0$ then $(x, P(x)) \notin \pi$. Thus a checker is[3] a certifier, but not necessarily vice-versa. We call the above pair (P, F) a **certified program** for π. Thus (P, F) is seen as a new program P_F, such that on input x, we first compute $P(x)$ and certify that $F(x, P(x)) = 1$; if certified, we output $P(x)$ and otherwise $P_F(x) \uparrow$.

The main examples we have in mind is when P is the machine-arithmetic implementation of a mathematically valid algorithm for π, so the output of P is not always correct. In many situations, we can construct a certifier F that knows about the operations of P and can certify most outputs of P. An example is where $\pi = \pi_{asd}$ is the problem of "approximate signed determinant". For any input matrix x, and for any real number y, let $(x, y) \in \pi_{asd}$ iff $\texttt{sign}\det(x) = \texttt{sign}y$. Note that we do not care whether y is even close to the true determinant of x, as long as y has the right sign. Let P be some standard

[2]Unlike in recursive function theory, our model assumes that we can detect in a finite (namely, $C_P(x)$) amount of time whether $P(x) \uparrow$ In this example, after detecting $P(x) \uparrow$, we simply call $A(x)$.

[3]To be certified in our sense is similar to passing a driver's test. As in real life, having a driver's certificate meant that one can drive, but not having one does not mean one cannot drive.

algorithm to compute determinants, implemented in machine-arithmetic. In this case, a certifier $F(x, y)$ for P (relative to π_{asd}) can compute the standard error bound $b(x) > 0$ on the output $P(x)$, and output 1 iff $b(x) < |P(x)|$. The pair (P, F) is thus a certified program for π. It turns out that more efficacious certifiers F' based on distance to the nearest singularity can be constructed (Pan and Yu [**?**]). Note that we could insist that F be a checker, in order to increase the efficacy of (P, F). Unfortunately, this may be too strong a requirement. In the case of π_{asd}, we know of no non-trivial checker F that does amount to actually computing the determinant exactly. This would defeat the whole purpose of certifying.

The intuitive idea is easy: we must "verify", "check" or "certify" the output of useful programs. We will make a formal distinction among these three concepts. They are ordered in a hierarchy of decreasing difficulty, with verification having the most stringent requirement. The verification procedure takes as input a program P and some input-output specification S. The goal is to determine whether or not P has the behavior S. A checking procedure (say, specialized to a behavior S) takes an input-output pair (I, O) and will determine whether or not $(I, O) \in S$. This checker does not care how this (I, O) pair is produced (in practice it probably comes from running some program P). Finally, a certifying procedure (again, specialized to S) takes the same input pair (I, O) and again attempts to decide if $(I, O) \in S$. If it says "yes" then surely $(I, O) \in S$. If it says "no", it is allowed to be wrong (we call this **false rejections**). Of course, a trivial certifier is to say "no" to all its inputs. But useful certifiers should certainly do more than this. The main tradeoff for designing certifiers is between **efficacy** (the percentages of false rejections) and efficiency of the procedure.

The area of program checking [**?**, **?**] lends some perspective on how to do this. The related area of program verification requires all code to be proved correct. This requirement is onerous as it is, and is the reason why verification was abandoned. But imagine verifying numerical code. Presumably we must take into account the numerical accuracy issues. But if so, practically all numerical code will fail the program verification process! Program checking from the 1990s takes a more realistic approach: it does not verify programs, it basically checks specific input/output pairs. This ought to be called "problem checking". In program checking, one wants to do more than problem checking, by actually calling the program as a blackbox. This requires additional properties such as self-reducibility.

But our main emphasis in EGC is certifiers which turns out to be very useful in practice. The term **filters** here is used as an alternative terminology for certifiers. A filter not have to recognize that any particular input/output pair is correct – to be useful, it only needs to recognize this for a substantial number of such pairs. Thus, filters are even easier to construct than checkers. Ultimately, filters are used for efficiency purposes, not for correctness purposes.

§2.1. Static Filters for 4-Dimensional Convex Hulls

To see static filters in action in the context of an actual algorithm, we follow a study of Sugihara [**?**]. He uses the well-known beneath/beyond algorithm for computing the convex hull of a set of 4-dimensional points, augmented with symbolic perturbation techniques. By using static filters, Sugihara reported a 100-fold speedup for non-degenerate inputs, and 3-fold speedup for degenerate inputs (see table below).

Let us briefly review the method: let the input point set be S and suppose they are sorted by the x-coordinates: $P_1, \ldots, P_n$. Let S_i be the set of first i

points from this sorted list, and let $CH(S_i)$ denote their convex hull. In general, the facets of the convex hull are 3-dimensional tetrahedra. The complexity of $CH(S)$ can be $\Theta(n^2)$. Since this beneath/beyond algorithm is $O(n^2)$, it is optimal in the worst case. We begin by constructing $CH(S_5)$. In general, we show how to transform $CH(S_{i-1})$ to $CH(S_i)$. This amounts to deleting the boundary tetrahedra that are visible from P_i and addition of new tetrahedra with apexes at P_i.

The main primitive concerns 5 points A, B, C, D, E in $\mathbb{R}^4$. The sign of the following 5×5 determinant

$$D = \begin{bmatrix} 1 & A_x & A_y & A_z & A_u \\ 1 & B_x & B_y & B_z & B_u \\ 1 & C_x & C_y & C_z & C_u \\ 1 & D_x & D_y & D_z & D_u \\ 1 & E_x & E_y & E_z & E_u \end{bmatrix}$$

determines whether the tetrahedron (A, B, C, D) is visible from E. Of course, this is the primitive for all standard convex hull algorithms, not just the beneath/beyond method. If the coordinates are L bit integers, Hadamard bound says that the absolute value of the determinant is at most $25\sqrt{5}2^{4L}$, and thus a $6 + 4L$ bit integer. If $L = 28$, it suffices to use 4 computer words. Using IEEE 64-bit doubles, the mantissa is 53 bits, so δ the relative error in a number, is at most $2^{-53} < 4 \times 10^{-15}$.

The determinant can be evaluated as a 4×4 determinant $A = (a_{ij})$ with $4! = 24$ terms. Let us define

$$d := \max_{i,j,k,\ell} |a_{1i}a_{2j}a_{3k}a_{4\ell}|$$

where i, j, k, ℓ range over all permutations of $\{1, 2, 3, 4\}$. The computed determinant $\det(A')$ is only an approximation to the true determinant $\det(A)$, where A' is some perturbed version of A. It can be shown that if we define

$$\Delta := \frac{24}{10^{15}} d$$

then

$$|\det(A') - \det(A)| \leq \Delta.$$

Thus, if the absolute value of $\det(A')$ is greater than Δ, we have the correct sign.

[Display table of values from Sugihara here]

| ... |

[COMPUTE the PHI factor]

Exercise 2.1: Determine the value of β for any realistic algorithm of your choice. Does this β depend on the size of the input? ♢

0.4 Dynamic Filters

The following idea is important in practice: many numbers, at run time, have much smaller value than the worst case bounds. It is important to exploit this but it can only be done at run time.

Suppose we have a predicate $P(x_1, \ldots, x_n)$. Typically, we use a single parameter function $B(L)$ such that if each x_i has at most L bits then $|P(x_1, \ldots, x_n)| > B(L)$ when non-zero. If we have different estimates (or exact

bounds as in dynamic filters) for each of the x_i's then we would like some easy-to-compute function $B(L_1, \ldots, L_n)$ such that $|P(x_1, \ldots, x_n)| > B(L_1, \ldots, L_n)$ if non-zero. We have seen this in our static analysis of Fortune's Voronoi diagram algorithm.

Dynamic version of the FvW filter. It is easy enough to convert the FvW filter into a dynamic one: looking at Table 1, we see that the only modification is that in the base case, we can directly compute $\texttt{MaxLen}(x)$ and $\texttt{MaxErr}(x)$ for a variable x. Let us estimate the cost of this dynamic filter. We already note that $\texttt{MaxLen}(E)$ can be computed directly using the formula in Table 1 since they involve integer arithmetic. This takes 1 or 2 operations. But for $\texttt{MaxErr}(E)$, we need to appeal to lemma 1. It is easy to see that since the values are all non-negative, we at most double the operation counts in the formulas of Table 1. The worst case is the formula for $E = FG$:

$$\texttt{MaxErr}(E) = \texttt{MaxErr}(F)2^{\texttt{MaxLen}(G)} + \texttt{MaxErr}(G)2^{\texttt{MaxLen}(F)} + 2^{\texttt{MaxLen}(FG)-53}.$$

The value $2^{\texttt{MaxLen}(FG)-53}$ can be computed exactly in 2 flops. There remain 4 other exact operations, which require $2 \times 4 = 8$ flops. Hence the bound here is 10 flops. Added to the single operation to compute $\texttt{MaxLen}(F) + \texttt{MaxLen}(G)$, we obtain 11 flops. A similar analysis for $E = F + G$ yields 8 flops.

After computing these filter parameters, we need to check if the filter predicate is satisfied:

$$|\widetilde{E}| > \texttt{MaxErr}(E).$$

Assuming $\widetilde{E}$ is available (this may incur storage costs not counted above), this check requires up to 2 operations: to compute the absolute value of $\widetilde{E}$ and to perform the comparison. Alternatively, if $\widetilde{E}$ is not available, we can replace $\widetilde{E}$ by $2^{\texttt{MaxLen}(E)}$. In general, we also need to check for floating-point exceptions at the end of computing the filter parameters (the filter is assumed to have failed when an exception occurred). We may be able to avoid the exception handling, e.g., static analysis may tell us that no exceptions can occur.

Fortune and Van Wyk gave somewhat similar numbers, which we quote: let f_e count the extra runtime operations, including comparisons; let f_r count the runtime operations for storing intermediate results. In the `LN` implementation, $12 \le f_e \le 14$ and $24 \le f_r \le 33$. The $36 \le f_e + f_r \le 47$ overhead is needed even when the filter successfully certifies the approximate result $\widetilde{E}$; otherwise, we may have to add the cost of exact computation, etc. Hence $f_e + f_r$ is a lower bound on the σ-factor when using filtered arithmetic in `LN`. Roughly the same bound of $\sigma = 48$ was measured for `LEDA`'s system.

Other ideas can be brought into play: we need not immediately invoke the dynamic filter. We still maintain a static filter, and do the more expensive runtime filter only when the static filter fails. And when the dynamic filter fails, we may still resort to other less expensive computation instead of jumping immediately to some failsafe but expensive big Integer/Rational computation. The layering of these stages of computations is called **cascaded filtering** by Burnikel, Funke and Schirra (BFS) [**?**]. This technique seems to pay off especially in nearly degenerate situations. We next describe the BFS filter.

Exercise 2.2: Implement and perform a direct comparison of the FvW Filter and the BFS Filter to determine their efficacy (*i.e.*, their ϕ-factors) for sign of small determinants. ♢

0.5 Sign Determination in Modular Arithmetic

Modular arithmetic (i.e., residue number systems) is an attractive method of implementing arithmetic because it allows each digit to be computed independently of the others. That is, is avoids the problem of carries, and thus is amenable to parallel execution in machine precision (assuming each digit fits into a machine word). Unfortunately, determining the sign of a number in modular representation seems to be non-trivial. We describe a solution provided by Brönnimann et al. [**?**].

§2.2. Partial Compilation of Expressions

Sometimes, part of the arguments of an expression are fixed while others are vary. We can exploit this by constructing "partially compiled expressions". To see an example of this phenomenon, we look at the Beneath-Beyond Algorithm for convex hulls.

See [Page 147ff in Edelsbrunner].

Let S be a set of points in d dimensions. FOR SIMPLICITY, ASSUME GENERAL POSITION. We sort the points by their first coordinate, and add them in this order. The method (called Beneath-Beyond) begins with a $d+1$ simplex. At each step, we add a new point p to the current hull H:

Lecture 12
CONSTRUCTIVE ZERO BOUNDS

This chapter describes some effective methods for computing lower bound on algebraic expressions. Two particular methods will be developed in detail: the measure bound and BFMSS bound.

§1. An Approach to Algebraic Computation

Algebraic numbers that arise in practice are represented by expressions such as $\sqrt{2}+\sqrt{3}-\sqrt{5+2\sqrt{6}}$ or $1-100\sin(1/100)$. The critical question is to determine the sign of such expressions, or to detect when they are undefined. Assume we can approximate any well-defined expression e to any desired absolute precision, i.e., for all $p \in \mathbb{N}$, we can compute an approximate value $\widetilde{e}$ such that $|e-\widetilde{e}| \leq 2^{-p}$. If $e \neq 0$, then we can compute $\widetilde{e}$ for $p = 1, 2, 3, \ldots$ until $|\widetilde{e}| > 2^{-p}$. At this point, we know the sign of e is that of $\widetilde{e}$. The problem is that when $e = 0$, this iteration cannot halt. But suppose we can compute some bound $B(e) > 0$ with the property that if $e \neq 0$ then $|e| > B(e)$. In this case, we can halt our iteration when $p \geq 1 - \lg B(e)$, and declare that $e = 0$. In proof,

$$|e| \leq |\widetilde{e} + 2^{-p} \leq 2^{1-p} \leq B(e)$$

implies $e = 0$.

In general, an **expression** is a rooted DAG over some set Ω of real algebraic operators. A typical set is $\Omega = \{\pm, \times, \div\} \cup \mathbb{Z}$. Note that constants such as $n \in \mathbb{Z}$ are regarded as 0-ary operators, and these appear at the leaves of the DAGs. Let $Expr(\Omega)$ denote the set of expressions over Ω. There is a natural evaluation function $\texttt{val} : Expr(\Omega) \to \mathbb{R}$ such that $\texttt{val}(e)$ is the value denoted by e. In general, $\texttt{val}$ is a partial function, since some operators in Ω (like $\div$) may be partial. We write $\texttt{val}(e) = \uparrow$ in case $\texttt{val}(e)$ is undefined; otherwise we write $\texttt{val}(e) = \downarrow$. A function $B : Expr(\Omega) \to \mathbb{R}_{\geq 0}$ is called a **zero bound function** if for all $e \in Expr(\Omega)$, if $\texttt{val}(e) = \downarrow$ then $|\texttt{val}(e)| \geq B(e)$. Since the lower bound is only in effect when $\texttt{val}(e) = \downarrow$, we may call $B(e)$ a "conditional" lower bound.

We generalize the above observations to a general computational paradigm. This is basically the method encoded in the Core Library. Each algebraic operation in $\Omega = \{+, -, \times, \div, \sqrt{\cdot}, \ldots\}$ is regarded as the construction of a root of a DAG whose leaves are (say) integers. Thus, each node u of the DAG has an implicit real value $\texttt{val}(u)$ (which may be undefined). Moreover, assume that we store two quantities at every node u of the DAG: a precision parameter $p_u \in \mathbb{N}$ and a bigfloat interval $I_u \in \mathbb{Z}[\frac{1}{2}]$ (possibly I_u is undefined). Inductively assume that $\texttt{val}(u) \in I_u$ and $w(I_u) \leq 2^{-p_u}$. Moreover, we assume algorithms which can approximate each operation in Ω to whatever precision we wish. Suppose we want to approximate a given expression e to some absolute precision p. Assume $e = e' \diamond e''$ where $\diamond \in \Omega$. The lazy approach says that we just compute (using interval arithmetic) the value $I_e := I_{e'} \diamond I_{e''}$ and see if $w(I_e) \leq 2^{-p_e}$. If not, we refind the intervals $I_{e'}$ and $I_{e''}$ and repeat. But in Core Library, we do this iteration more actively, by computing the precision $p_{e'}, p_{e''}$ in e' and e'' that will ensure that I_e has precision p_e. This is called "precision-driven computation". All this computation is relatively straightforward in interval arithmetic. What makes our system unique is that we also compute a zero bound $B(e)$ for each expression, and this allows us to decide the sign of e. When $w(I_e) \leq B(e)/2$, and $0 \in I_e$, we conclude that e is zero.

We now focus on how to compute such $B(e)$ bounds. Using the theory of resultants, we can define a suitable zero bound function for expressions over $\Omega = \{\pm, \times, \div, \sqrt{\cdot}\} \cup \mathbb{Z}$. For instance, if $e = e_1 e_2$, then we know from resultants that a defining polynomial $A(X)$ for e can be obtained from the definition polynomials $A_i(X)$'s for e_i $(i = 1, 2)$. Moreover, if $\mathrm{ht(e_i) = h_i}$ then

$$\mathrm{ht(e) \leq h := h_1^{m_2} h_2^{m_1}}$$

where $m_i = \deg(A_i)$. From Cauchy's bound (see previous Chapter), it is clear that we can define $B(e) = (1+h)^{-1}$. There are similar relations for $e = e_1 \pm e_2$, $e = e_1/e_2$, $e = \sqrt[k]{e_1}$. Thus, if we maintain recursively,

for each node in e, an upper bound on the height and degree of the corresponding algebraic number, we can recursive compute $B(e)$ for any expression e. This is the **degree-height bound**, implemented in the first system for such kind of numerical algebraic computation, `Real/Expr` [16] (cf. [15, p. 177]).

In this chapter, we describe several other zero bound functions in detail. The general feature of such **constructive bounds** is illustrated by the degree-height bound. Two ingredients are needed: first, we need a set of recursive rules to maintain a set

$$p_1(e), \ldots, p_t(e)$$

of numerical parameters for an expression e. In the degree-height bound, $t = 2$ where $p_1(e)$ is a height bound and $p_2(e)$ is the degree bound. The second ingredient is a zero bound function $B(e)$ that is computed obtained as a function β of these parameters, $B(e) = \beta(p_1(e), \ldots, p_t(e))$. In the degree-height bound, $\beta(p_1, p_2) = (1 + p_1)^{-1}$.

Such recursive rules apply to a suitable set $Expr(\Omega)$ of expressions. For these notes, we mainly focus on the following set

$$\Omega_2 := \{\pm, \times, \div, \sqrt{\cdot}\} \cup \mathbb{Z}.$$

We can slightly extend Ω_2 to allow $\sqrt[k]{\cdot}$ for integers $k \geq 2$.

Let $B(e) = \beta(p_1(e), \ldots, p_t(e))$ be a constructive zero bound over a class K of expressions. Suppose we have another constructive zero bound $C(e) = \gamma(q_1(e), \ldots, q_u(e))$ over K which is based on a different set of parameters $q_1(e), \ldots, q_u(e)$ and bounding function $\gamma(q_1, \ldots, q_u)$. We would like to compare $C(e)$ and $B(e)$: we say B is as **efficient** than C if for every $e \in K$, the time complexity of maintaining the parameters $p_1(e), \ldots, p_t(e)$ and computing the function $b(p_1, \ldots, p_t)$ is order of the corresponding complexity for C. But there is another way to compare $B(e)$ and $C(e)$: we say $B(e)$ **dominates** $C(e)$ if for all $e \in K$, $B(e) \geq C(e)$. For instance, the degree-measure bound, to be described, dominates the degree-height bound. At least for known constructive bounds, the efficiency issue is less important than having as large a lower bound as possible. Hence we mainly compare zero bounds based on their domination relationship.

Degree Bound for Expressions. In all the zero bounds, we need to maintain an upper bound $D(e)$ on the degree of $\texttt{val}(e)$. Consider an expression e having r radical nodes with indices $k_1, k_2, \ldots, k_r$. Then we claim that the degree of e is at most

$$D(e) = \prod_{i=1}^{r} k_i. \tag{1}$$

In proof, suppose we topologically sort the nodes in e, beginning with a leaf node and ending in the node e. Let the sorted list be $(v_1, v_2, \ldots, v_s)$. Inductively, define $d_1, \ldots, d_s$ where $d_1 = 1$ and d_{i+1} is equal to kd_i if v_{i+1} is k-th root, and otherwise $d_{i+1} = d_i$. It is clear that $d_s = D(e)$. Now it is clear that, by induction, $\deg(v_i) \leq d_i$. This proves our claim. It is also not hard to compute $D(e)$. This method of bounding degree extends to the "RootOf" operator (below) which introduces arbitrary real algebraic numbers.

§1.1. The Mahler Measure bound

Every algebraic number α has a unique minimal polynomial $\mathrm{Irr}(\alpha) \in \mathbb{Z}[X]$. We can factor $\mathrm{Irr}(\alpha)$ over $\mathbb{C}$ as $\mathrm{Irr}(\alpha) = a\prod_{i=1}^{m}(X - \alpha_i)$ with a a positive integer. We may assume $\alpha = \alpha_1$; each α_i is called a **conjugate** of α. Mahler's **measure** of α is defined as

$$M(\alpha) := a \cdot \prod_{i=1}^{m} \max\{1, |\alpha_i|\}.$$

For instance, $M(\sqrt{2}) = 2$ because the minimal polynomial of $\sqrt{2}$ is $X^2 - 2 = (X - \sqrt{2})(X + \sqrt{2})$ and $a = 1$. On the other hand, $M(1 + \sqrt{2}) = 1 + \sqrt{2}$ which is irrational.

In the following, it is convenient to define for any complex z,

$$\max_1(z) := \max\{1, |z|\}. \tag{2}$$

In general, if $A(X) = \sum_{i=0}^{m} a_i X^i \in \mathbb{C}[X]$ is any polynomial, we define its measure as

$$M(A) := |a_m| \cdot \prod_{i=1}^{m} \max_1(\alpha_i),$$

where α_i's are the complex roots of $A(X)$. This definition might appear unnatural but in fact $M(\alpha)$ can also be defined by a natural integral (Exercise). In case $A(X) \in \mathbb{Z}[X]$, we see that $M(A) \geq 1$.

In particular, $M(\alpha) \geq 1$ for non-zero algebraic α. It is also easy to see that if $A(X), B(X) \in \mathbb{Z}[X]$ then

$$M(AB) = M(A)M(B) \tag{3}$$

and hence we conclude that

$$M(A) \leq M(AB).$$

The basis for the degree-measure bound is the following theorem:

Lemma 1.
(i) $|\alpha| \leq M(\alpha)$.
(ii) If $\alpha \neq 0$ *then* $M(1/\alpha) = M(\alpha)$.
(iii) $|\alpha| \geq \frac{1}{M(\alpha)}$.

Proof. (i) is immediate from the definition of measure. For (ii), we observe that if the minimal polynomial of α is $A(X) = \sum_{i=0}^{m} a_i X^i = a_m \prod_{i=1}^{m}(X - \alpha_i)$ then $a_0 \neq 0$ and

$$B(X) = X^m A(1/X) = a_0 \prod_{i=1}^{m}(X - 1/\alpha_i)$$

is (up to sign) the minimal polynomial of $1/\alpha$. Further,

$$a_0 = a_m \prod_{i=1}^{m} \alpha_i \tag{4}$$

Hence

$$\begin{aligned} M(1/\alpha) &= |a_0| \textstyle\prod_i \max_1(1/\alpha_i) \\ &= |a_m| \textstyle\prod_i |\alpha_i| \cdot \max_1(1/\alpha_i) \quad (\text{ by } (4)) \\ &= |a_m| \textstyle\prod_i \max_1(\alpha_i) \\ &= M(\alpha). \end{aligned}$$

Finally, (iii) follows from (i) and (ii): $1/|\alpha| \leq M(1/\alpha) = M(\alpha)$. **Q.E.D.**

In other words, if e is any expression, and we maintain an upper bound $m(e)$ on it's Mahler measure, then the $B(e) \geq 1/m(e)$. Let us now see how we can maintain such an upper bound. As in the case of height, we see that we need to also maintain an upper bound on the degree of e. Such bounds were first exploited by Mignotte [9] in the problem of identification of algebraic numbers.

In the following, we use the following elementary relations:

$$\max_1(\alpha\beta) \leq \max_1(\alpha)\max_1(\beta), \tag{5}$$

$$\max_1(\alpha \pm \beta) \leq 2\max_1(\alpha)\max_1(\beta). \tag{6}$$

The first inequality (5) is trivial in case $|\alpha\beta| \leq 1$. Otherwise,

$$\max_1(\alpha\beta) = |\alpha| \cdot |\beta| \leq \max_1(\alpha)\max_1(\beta).$$

The second inequality is trivial in case $|\alpha \pm \beta| \leq 2$. Otherwise, let $|\beta| \geq |\alpha|$ and

$$\max_1(\alpha \pm \beta) \leq |\alpha| + |\beta| \leq 2|\beta| \leq 2\max_1(\alpha)\max_1(\beta).$$

We have the following relations on measures [10]:

Lemma 2. *Let α and β be two nonzero algebraic numbers of degrees m and n respectively. Let $k \geq 1$ be an integer.*

$$\begin{array}{lll}
(o) & M(p/q) & \leq \max|p|,|q|, \quad p,q \in \mathbb{Z}, q \neq 0 \\
(i) & M(\alpha \times \beta) & \leq M(\alpha)^n M(\beta)^m \\
(ii) & M(\alpha/\beta) & \leq M(\alpha)^n M(\beta)^m \\
(iii) & M(\alpha \pm \beta) & \leq 2^d M(\alpha)^n M(\beta)^m, \quad d = \deg(\alpha \pm \beta) \\
(iv) & M(\alpha^{1/k}) & \leq M(\alpha) \\
(v) & M(\alpha^k) & \leq M(\alpha)^k
\end{array}$$

Proof. For (o), it sufficient to assume p, q are relatively prime so that the minimal polynomial of p/q is $qX - p$. Then $M(()p/q) = |q| \max_1(p/q)$, which we can verify is equal to $\max|p|,|q|$.

Let the minimal polynomials of α, β be $A(X) = a\prod_{i=0}^m (X-\alpha_i)$ and $B(X) = b\prod_{j=0}^m (X-\alpha_j)$, respectively. For (i), we use the fact that the minimal polynomial of $\alpha\beta$ divides $a^n b^m \prod_i \prod_j (X - \alpha_i\beta_j)$ which we saw is the resultant $\mathtt{res}_Y(A(Y), Y^n B(X/Y))$. Hence, by (3),

$$\begin{array}{rcl}
M(\alpha\beta) & \leq & a^n b^m \prod_i^m \prod_j^n \max_1(\alpha_i\beta_j) \\
& \leq & a^n \prod_i b \prod_j \max_1(\alpha_i)\max_1(\beta_j) \\
& = & a^n \prod_i \max_1(\alpha_i)^n \left[b \prod_j \max_1(\beta_j)\right] \\
& = & a^n \prod_i \max_1(\alpha_i)^n M(\beta) \\
& = & M(\beta)^m a^n \prod_i \max_1(\alpha_i)^n \\
& = & M(\beta)^m M(\alpha)^n.
\end{array}$$

Part (ii) follows from (i), using the fact that $M(1/\beta) = M(\beta)$ and $\deg(1/\beta) = n$.

For (iii), we use the fact that the minimal polynomial of $\alpha \pm \beta$ divides $a^n b^m \prod_i \prod_j (X - \alpha_i \mp \beta_j)$ which we saw is the resultant $\mathtt{res}_Y(A(Y), B(X \mp Y))$. Let $I \subseteq \{1,\ldots,m\} \times \{1,\ldots,n\}$ such that the conjugates of $\alpha \pm \beta$ is given by the set $\{\alpha_i \pm \beta_j : (i,j) \in I\}$. So $\deg(\alpha \pm \beta) = |I|$.

$$\begin{array}{rcl}
M(\alpha \pm \beta) & \leq & a^n b^m \prod_{(i,j)\in I} \max_1(\alpha_i \pm \beta_j) \\
& \leq & a^n b^m \prod_{(i,j)\in I} 2\max_1(\alpha_i)\max_1(\beta_j) \\
& = & 2^{|I|} a^n b^m \prod_{(i,j)\in I} \max_1(\alpha_i)\max_1(\beta_j) \\
& \leq & 2^{|I|} a^n b^m \prod_i \prod_j \max_1(\alpha_i)\max_1(\beta_j).
\end{array}$$

The rest of the derivation follows as in part (i).

For (iv) and (v), use the facts that the minimal polynomials of $\alpha^{1/k}$ and α^k (respectively) divide $A(X^k) = a\prod_i (X^k - \alpha_i)$ and $A(X)^k$. **Q.E.D.**

Using this lemma, we give a recursive definition of a function $m(e)$, as shown in Table 1 under the column with the heading "$m(e)$". Thus is an upper bound on measure

$$M(e) \leq m(e).$$

Hence we conclude from from Lemma 1 that the function

$$B(e) = 1/m(e)$$

is a root bound function.

Sekigawa [13] gave a refinement of these rules in the case an expression e is division-free. Let $M_0(A)$ denote the leading coefficient of A, and define $M_1(A)$ by the equation

$$M(A) = M_0(A)M_1(A).$$

In case e is an expression and A is the minimal polynomial of $\mathtt{val}(e)$, we write $M_0(e)$ and $M_1(e)$ for $M_0(A)$ and $M_1(A)$ (resp.). Sekigawa gave recursive definitions of two functions $m_0(e)$ and $m_1(e)$ which are upper

	e	$m(e)$	$m_1(e)$	$m_0(e)$
1.	rational p/q	$\max\{\|p\|, \|q\|\}$	$\max_1(p/q)$	$\|q\|$
2.	$e_1 \pm e_2$	$2^{d(e)} m(e_1)^n m(e_2)^m$	$m_1(e_1)^n m_1(e_2)^m$	$2^{d(e)} m_0(e_1)^n m_0(e_2)^m$
3.	$e_1 \times e_2$	$m(e_1)^n m(e_2)^m$	$m_1(e_1)^n m_1(e_2)^m$	$m_0(e_1)^n m_0(e_2)^m$
4.	$e_1 \div e_2$	$m(e_1)^n m(e_2)^m$	—	—
5.	$\sqrt[k]{e_1}$	$m(e_1)$	$m_1(e_1)$	$m_0(e_1)$

Table 1: Measure Bound Rules, including Sekigawa's refinement

bounds on $M_0(e)$ and $M_1(e)$, respectively. These definitions shown in the last two columns of Table 1. Note that we do not have rules for division. Thus, for division free expressions, the function

$$B(e) = \frac{1}{m_0(e)m_1(e)}$$

serves as a root bound function.

The rules shown here is actually a slightly simplified version of his rules.

An Example. Consider the expression

$$e = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}} \tag{7}$$

where $x = a/b, y = c/d$ and a, b, c, d are L-bit integers. Assume that $\sqrt{xy}$ is computed as $(\sqrt{x})(\sqrt{y})$. We will determine the Measure Bound on $-\lg|e|$ (expressed in terms of L). We call any upper bound for $-\lg|e|$ a **zero bit-bound** for e, because $-\lg|e|$ is the number bits of absolute precision that suffices to determine if $e = 0$.

We fill in the entries of the following table, using our rules for bounding measure. Ignore the last column for $\lg M_0(e)$ for now. It is usually simpler to maintain bounds on $\lg M(e)$ instead of $M(e)$ directly – so that is what we show in the table. The first entry for $\lg M(x)$ is justified in an exercise: $M(a/b) \leq \max\{|a|, |b|\}$ for $a, b \in \mathbb{Z}$.

No.	e	$d(e)$	$\lg M(e)$	$\lg M_0(e)$
1	x, y	1	L	L
2	$\sqrt{x}, \sqrt{y}$	2	L	L
3	$x + y$	1	$2L$	$2L$
4	$\sqrt{x}\sqrt{y}$	4	$4L$	$4L$
5	$\sqrt{x} + \sqrt{y}$	4	$4L + 4$	$4L$
6	$2\sqrt{xy}$	4	$4L + 4$	$4L$
7	$x + y + 2\sqrt{xy}$	4	$12L + 8$	$12L$
8	$\sqrt{x + y + 2\sqrt{xy}}$	8	$12L + 8$	$12L$
9	$\sqrt{x} + \sqrt{y} + \sqrt{x + y + 2\sqrt{xy}}$	8	$80L + 64$	$80L$

Since $|e| \geq 1/M(e)$, we conclude that $-\lg|e| \leq \lg M(e) \leq 80L + 64$. In line 4, the degree $\sqrt{x}\sqrt{y}$ of 4 is obtained from our rules, but it is clearly suboptimal.

Exercises

Exercise 1.1: (i) We have a rule for the measure of rational numbers p/q, and this is clearly tight in case p, q are relatively prime. But show that our measure bound (using the multiplication rule) is sub-optimal for rational input numbers.

(ii) Refine our measure rules for the special case of the division of two algebraic integers, similar to the rule (i). ♢

	e	$U(e)$	$L(e)$
1.	rational a/b	a	b
2.	$e_1 \pm e_2$	$U(e_1)L(e_2) + L(e_1)U(e_2)$	$L(e_1)L(e_2)$
3.	$e_1 \times e_2$	$U(e_1)U(e_2)$	$L(e_1)L(e_2)$
4.	$e_1 \div e_2$	$U(e_1)L(e_2)$	$L(e_1)U(e_2)$
5.	$\sqrt[k]{e_1}$	$\sqrt[k]{U(e_1)}$	$\sqrt[k]{L(e_1)}$

Table 2: BFMS Rules for $U(e)$ and $L(e)$

Exercise 1.2: Determine those algebraic numbers α whose Mahler measure $M(\alpha)$ are not natural numbers. ♢

Exercise 1.3: Determine the Mahler Bound for the expression in Equation (7) where x and y are L-bit rational numbers (i.e., the numerator and denominators are at most L-bit integers). ♢

Exercise 1.4: (i) Determine the Degree-Measure bound for the expression $(a + \sqrt{b})/d$ where a, b, d are (respectively) $3L$-bit, $6L$-bit and $2L$-bit integers.
(ii) Do the same for the difference of two such expressions. ♢

End Exercises

§1.2. The BFMS bound

One of the best constructive zero bounds for the class of radical expressions is from Burnikel et al [5]. We call this the **BFMSS Bound**. However, we begin with the presentation of the simpler version known as the **BFMS Bound** [4]. The bound depends on three parameters, $L(e), L(e), D(e)$ for an expression e. Since $D(e)$ is the usual degree bound, we only show the recursive rules for $L(e), L(e)$ in Table 2.

Conceptually the BFMS approach first transforms a radical expression $e \in Expr(\Omega_2)$ to a quotient of two division-free expressions $U(e)$ and $L(e)$.

If e is division-free, then $L(e) = 1$ and $\texttt{val}(e)$ is an algebraic integer (i.e., a root of some monic integer polynomial). The following lemma is immediate from Table 2:

Lemma 3. $\texttt{val}(e) = \texttt{val}(U(e))/\texttt{val}(L(e))$.

Table 2 should be viewed as transformation rules on expressions. We apply these rules recursive in a bottom-up fashion: suppose all the children v_i (say $i = 1, 2$) of a node v in the expression e has been transformed, and we now have the nodes $U(v_i), L(v_i)$ are available. Then we create the node $U(v), L(v)$ and construct the correspond subexpressions given by the table. The result is still a dag, but not rooted any more. See Figure 1 for an illustration in case the operation at v is $+$.

The transformation $e \Rightarrow (U(e), L(e))$ is only conceptual – we do not really need to compute it. What we do compute are two real parameters $u(e)$ and $l(e)$ are maintained by the recursive rules in Table 3. The entries in this table are "shadows" of the corresponding entries in Table 2. (Where are they different?)

To explain the significance of $u(e)$ and $l(e)$, we define two useful quantities. If α is an algebraic number, define

$$MC(\alpha) := \max_{i=1}^{m} |\alpha_i| \tag{8}$$

where $\alpha_1, \ldots, \alpha_m$ are the conjugates of α. Thus $MC(\alpha)$ is the "maximum conjugate size" of α. We extend this notation in two ways:
(i) If e is an expression, we write $MC(e)$ instead of $MC(\texttt{val}(e))$.

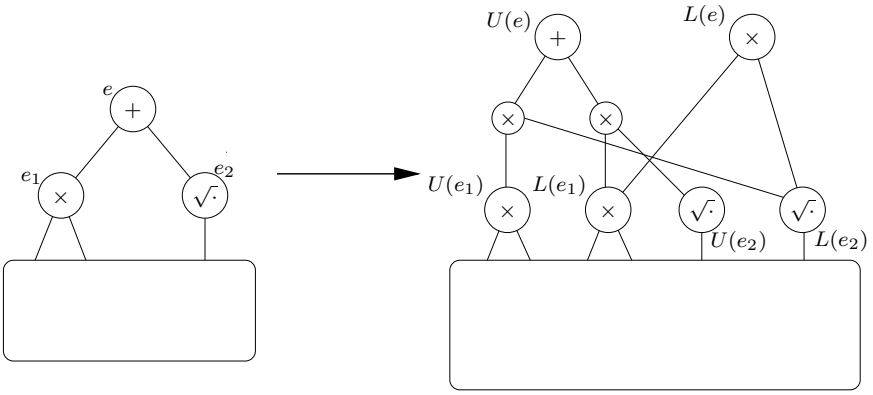


Figure 1: Transforming an expression e to $L(e), U(e)$.

	e	$u(e)$	$l(e)$
1.	rational a/b	$\|a\|$	$\|b\|$
2.	$e_1 \pm e_2$	$u(e_1)l(e_2) + l(e_1)u(e_2)$	$l(e_1)l(e_2)$
3.	$e_1 \times e_2$	$u(e_1)u(e_2)$	$l(e_1)l(e_2)$
4.	$e_1 \div e_2$	$u(e_1)l(e_2)$	$l(e_1)u(e_2)$
5.	$\sqrt[k]{e_1}$	$\sqrt[k]{u(e_1)}$	$\sqrt[k]{l(e_1)}$

Table 3: BFMS Rules for $u(e)$ and $l(e)$

(ii) If $A(X)$ is any polynomial, we write $MC(A(X))$ for the maximum of $|\alpha_i|$ where α_i range over the zeros of $A(X)$. For instance, we have this connection with Mahler measure:

$$M(\alpha) \leq M_0(\alpha) MC(\alpha)^d$$

where $d = \deg(\alpha)$. Using $MC(\alpha)$ and $M_0(\alpha)$, we obtain a basic approach for obtaining zero bounds:

Lemma 4. *If $\alpha \neq 0$ and then*

$$|\alpha| \geq M_0(\alpha)^{-1} MC(\alpha)^{-d+1}$$

where $d = \deg(\alpha)$.

Proof. Let $d = \deg(\alpha)$. If the minimal polynomial of α is $a\prod_{i=1}^{m}(X - \alpha_i)$ then we have $a\prod_i |\alpha_i| \geq 1$. Thus, assuming $\alpha = \alpha_1$,

$$|\alpha| \geq \frac{1}{a\prod_{i=2}^{d} |\alpha_i|} \geq \frac{1}{aMC(\alpha)^{d-1}}.$$

Q.E.D.

The following theorem shows the significance of $u(e), l(e)$.

Theorem 5. *Let $e \in Expr(\Omega_2)$. Then $u(e)$ and $l(e)$ are upper bounds on $MC(U(e))$ and $MC(L(e))$, respectively.*

Proof. The proof amounts to justifying Table 3 for computing $u(e), l(e)$. The base case where e is a rational number is clear. In general, $U(e)$ and $L(e)$ are formed by the rules in Table 2. Consider the case $e = e_1 \pm e_2$ so that

$$U(()e) = U(()e_1)L(()e_2) \pm L(()e_1)U(()e_2).$$

From the theory of resultants, we know that if $\alpha = \beta \circ \gamma$ ($\circ \in \{\pm, \times, \div\}$) then every conjugate of α has the form $\beta' \circ \gamma'$ where β', γ' are conjugates of β, γ (resp.). Thus,

$$MC(U(()e)) = MC(U(()e_1))MC(L(()e_2)) + MC(L(()e_1))MC(U(()e_2)).$$

By induction, $MC(U(()e_i)) \leq u(e_i)$ and $MC(L(()e_i)) \leq l(e_i)$ $(i = 1, 2)$. Hence, $MC(e) \leq u(e_1)l(e_2) + l(e_1)u(e_2) = u(e)$. This justifiies the entry for $u(e)$ on line 2 of Table 3. Similarly, we can justify each of the remaining entries of Table 3. **Q.E.D.**

Finally, we show how the BFMS Rules gives us a zero bound. It is rather similar to Lemma 4, except that we do not need to invoke $M_0(e)$.

Theorem 6. *Let $e \in Expr(\Omega_2)$ and $\mathtt{val}(e) \neq 0$. Then*

$$(u(e)^{D(e)^2-1}l(e))^{-1} \leq |\mathtt{val}(e)| \leq u(e)l(e)^{D(e)^2-1}. \tag{9}$$

If e is division-free,

$$(u(e)^{D(e)-1})^{-1} \leq |\mathtt{val}(e)| \leq u(e). \tag{10}$$

Proof. First consider the division-free case. In this case, $\mathtt{val}(e) = \mathtt{val}(U(e))$. Then $|\mathtt{val}(e)| \leq u(e)$ follows from Theorem 5. The lower bound on $|\mathtt{val}(e)|$ follows from lemma 4, since $M_0(e) = 1$ in the division-free case.

In the general case, we apply the division-free result to $U(e)$ and $L(e)$ separately. However, we need to estimate the degree of $U(e)$ and $L(e)$. We see that in the transformation from e to $U(e), L(e)$, the number of radical nodes in the dag doubles: each $\sqrt[k]{\cdot}$ is duplicated. This means that $\deg(U(e)) \leq \deg(e)^2$ and $\deg(L(e)) \leq \deg(e)^2$. From the division-free case, we conclude that

$$(u(e)^{D(e)^2-1})^{-1} \leq |\mathtt{val}(U(e))| \leq u(e).$$

and

$$(l(e)^{D(e)^2-1})^{-1} \leq |\mathtt{val}(L(e))| \leq l(e).$$

Thus $|\mathtt{val}(e)| = |\mathtt{val}(U(e))/\mathtt{val}(L(e))| \geq (l(e)u(e)^{D(e)^2-1})^{-1}$. The upper bound on $|\mathtt{val}(e)|$ is similarly shown. **Q.E.D.**

Example. Consider the expression $e_k \in Expr(\Omega_2)$ whose value is

$$\alpha_k = \mathtt{val}(e_k) = (2^{2^k} + 1)^{1/2^k} - 2. \tag{11}$$

Note that e_k is not literally the expression shown, since we do not have exponentiation in Ω_2. Instead, the expression begins with the constant 2, squaring k times, plus 1, then taking square-roots k times, and finally minus 2. Thus $u(e_k) = (2^{2^k} + 1)^{1/2^k} + 2 \leq 5$. The degree bound $D(e_k) = 2^k$. Hence the BFMS Bound says

$$|\alpha_k| \geq u(e_k)^{1-2^k} \geq 5^{1-2^k}.$$

How tight is this bound? We have

$$\begin{aligned}
(2^{2^k} + 1)^{1/2^k} - 2 &= 2\left(1 + 2^{-2^k}\right)^{1/2^k} - 2 \\
&= 2 \cdot e^{2^{-k}\ln(1+2^{-2^k})} - 2 \\
&\leq 2 \cdot e^{2^{-k}2^{-2^k}} - 2 \\
&\leq 2\left(1 + 2 \cdot 2^{-k}2^{-2^k}\right) - 2 \\
&= 2^{2-k-2^k}
\end{aligned}$$

	e	$mc(e)$	$m_0(e)$	REMARK
(i)	$a \in C$	$mc(a)$	$m_0(a)$	
(ii)	$e' \pm e''$	$mc(\alpha) + mc(\beta)$	$m_0(e')^{d''} m_0(e'')^{d'}$	$\deg(e') \le d'$
(iii)	$e' \times e''$	$mc(\alpha)mc(\beta)$	$m_0(e')^{d''} m_0(e'')^{d'}$	$\deg(e'') \le d''$
(iv)	$\sqrt[k]{e'}$	$\sqrt[k]{mc(e')}$	$m_0(e')$	

Table 4: Measure-BFMS Rules using $mc(e)$ and $m_0(e)$

using $\ln(1+x) \le x$ if $x > -1$ and $e^2 \le 1 + 2x$ if $0 \le x \le 1/2$. We also have

$$\begin{aligned}(2^{2^k}+1)^{1/2^k} - 2 &= 2 \cdot e^{2^{-k}\ln(1+2^{-2^k})} - 2 \\ &\ge 2 \cdot e^{2^{-k}2^{-2^k-1}} - 2 \\ &\ge 2\left(1 + 2^{-k}2^{-2^k-1}\right) - 2 \\ &\ge 2^{-k-2^k}\end{aligned}$$

using $e^x \ge 1 + x$. Hence $\alpha_k = \Theta(2^{-k-2^k})$. This example shows that the BFMS bound is, in a certain sense, asymptotically tight for the class of division-free expressions over Ω_2.

§1.3. Improvements on the BFMS bound

The root bit-bound in (9) is quadratic in $D(e)$, while in (10) it is linear in $D(e)$. This quadratic factor can become a serious efficiency issue. Consider a simple example: $e = (\sqrt{x} + \sqrt{y}) - \sqrt{x + y + 2\sqrt{xy}}$ where x, y are L-bit integers. Of course, this expression is identically 0 for any x, y. The BFMS bound yields a root bit-bound of $7.5L + \mathcal{O}(1)$ bits. But in case, x and y are viewed as rational numbers (with denominator 1), the bit-bound becomes $127.5L + \mathcal{O}(1)$. This example shows that introducing rational numbers at the leaves of expressions has a major impact on the BFMS bound. In this section, we introduce two techniques to overcome division.

The Measure-BFMS Bound. The first technique applies division-free expressions, but where the input numbers need not be algebraic integers (we can think of this as allowing division at the leaves). For instance, the input numbers can be rational numbers.

The basic idea is to exploit Lemma 4. Hence we would like to maintain upper bounds on $MC(\alpha)$ and $M_0(\alpha)$. Let

$$\Omega = \{\pm, \times, \sqrt[k]{\cdot}\} \cup C \tag{12}$$

where C is some set of algebraic numbers. Suppose e is an expression over Ω; so e is division-free. However C may contain rational numbers that implicitly introduce division. We define the numerical parameters $mc(e)$ and $m_0(e)$ according to the recursive rules in Table 4.

Table 4, gives the recursive rules for computing $mc(e), m_0(e)$. Note that as the base case, we assume the ability to compute upper bounds on $MC(a)$ and $M_0(a)$ for $a \in C$.

Lemma 7. *For any expression e over $\{\pm, \times, \sqrt[k]{\cdot}\} \cup C$, we have $MC(e) \le mc(e)$ and $M_0(e) \le m_0(e)$ where $mc(e)$ and $m_0(e)$ are given by Table 4.*

Proof. We justify each line of Table 4. Line (i) is immediate by definition.

(ii) Let the minimal polynomial for α, β be $a\prod_{i=1}^m (X - \alpha_i)$ and $b\prod_{j=1}^n (X - \beta_j)$, respectively. Then the minimal polynomial of $\alpha \pm \beta$ divides $R(X) = a^n b^m \prod_i \prod_j (X - \alpha_i \mp \beta_j)$. Thus each conjugate ξ of $\alpha \pm \beta$ has the form $\alpha_i \pm \beta_j$ for some i, j. Thus $|\xi| \le |\alpha_i| + |\beta_j|$. This proves $MC(\alpha \pm \beta) \le mc(\alpha) + mc(\beta)$. The

inequality $M_0(\alpha \pm \beta) \leq a^n b^m \leq m_0(\alpha)^n m_0(\beta)^m$ follows from the fact that the leading coefficient of the minimal polynomial divides the leading coefficient of $R(X)$.

The same proof applies for $e = e_1 e_2$. For (iii), if $A(X)$ is the minimal polynomial for α then the minimal polynomial for $\sqrt[k]{\alpha}$ divides $A(X^k)$, so $MC(\alpha) \leq MC(A(X^k))$. However, $MC(A(X^k)) = \sqrt[k]{MC(A(X))} = \sqrt[k]{MC(\alpha)}$. Finally, we have $M_0(\sqrt[k]{\alpha}) \leq a \leq M_0(\alpha)$. **Q.E.D.**

By Lemma 4, we conclude that

$$e \neq 0 \Rightarrow |e| \geq \frac{1}{M_0(e) mc(e)^{D(e)-1}}. \tag{13}$$

Such a bound has features of Measure Bound as well as of the BFMS Bound; so we call it the "Measure-BFMS Bound".

The BFMSS Bound. Returning to the case of radical expressions, we introduce another way to improve on BFMS. To avoid the doubling of radical nodes in the $e \mapsto (U(e), L(e))$ transformation, we change the rule in the last row of Table 3 as follows. When $e = \sqrt[k]{e_1}$, we use the alternative rule

$$u(e) = \sqrt[k]{u(e_1) l(e_1)^{k-1}}, \quad l(e) = l(e_1). \tag{14}$$

But one could equally use

$$u(e) = u(e_1), \quad l(e) = \sqrt[k]{u(e_1)^{k-1} l(e_1)}.$$

Yap noted that by using the symmetrized rule

$$u(e) = \min\{\sqrt[k]{u(e_1) l(e_1)^{k-1}}, u(e_1)\}, \qquad l(e) = \min\{l(e_1), \sqrt[k]{u(e_1)^{k-1} l(e_1)}\},$$

the new bound is provably never worse than the BFMS bound. The BFMSS Bound also extends the rules to support general algebraic expressions (Ω_4 expressions). NOTE: in the absence of division, the BFMSS and BFMS rules coincide.

Comparison. We consider expressions over a division-free set Ω of operators, as in (12). Two important examples are:

- Expressions can have rational input numbers (in particular binary floats or decimal numbers). See [Pion-Yap].
- Expressions where the leaves could have RootOf(P, i) operators. This is the case with Core Library Version 1.6.

It follows that Theorem 5 is still true. However, what is the replacement for Theorem 6? Using Lemma 4 and Lemma 7 we can obtain more effective bounds.

Let us see how BFMS, BFMSS and Measure-BFMS Bounds perform for the expression (7). In Table 3, we show the parameters $u(e), l(e)$ as defined for the BFMS Bound. In the next two columns, we show their variants (here denoted $uu(e)$ and $ll(e)$) as defined for the BFMSS Bound.

The BFMS Bound gives

$$-\lg|e| \leq (d(e)^2 - 1)\lg u(e) + \lg l(e) = 63(5L+4)/2 + 5L/2 < 160L + 126.$$

But the BFMSS Bound gives

$$-\lg|e| \leq (d(e) - 1)\lg uu(e) + \lg ll(e) \leq 7(4L+2) + 4L = 32L + 14.$$

No.	e	$d(e)$	$\lg u(e)$	$\lg l(e)$	$\lg uu(e)$	$\lg ll(e)$	$mc(e)$
1	x, y	1	L	L	L	L	L
2	$\sqrt{x}, \sqrt{y}$	2	$L/2$	$L/2$	L	L	$L/2$
3	$x + y$	1	$2L + 1$	$2L$	$2L + 1$	$2L$	$L + 1$
4	$\sqrt{x}\sqrt{y}$	4	L	L	$2L$	$2L$	L
5	$\sqrt{x} + \sqrt{y}$	4	$L + 1$	L	$2L + 1$	$2L$	$(L + 2)/2$
6	$2\sqrt{xy}$	4	$L + 1$	L	$2L + 1$	$2L$	$L + 1$
7	$x + y + 2\sqrt{xy}$	4	$3L + 2$	$3L$	$4L + 2$	$4L$	$L + 2$
8	$\sqrt{x + y + 2\sqrt{xy}}$	8	$(3L + 2)/2$	$3L/2$	$2L + 1$	$2L$	$(L + 2)/2$
9	$\sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$	8	$(5L + 4)/2$	$5L/2$	$4L + 2$	$4L$	$(L + 4)/2$

Table 5: BFMS and BFMSS on example

To apply the Measure-BFMS rule, we could use the fact that

$$MC(e) \leq u(e) \leq (5L + 2)/2$$

(by first column of Table

$$M_0(e) \leq 80L$$

(by last column of table in (i)). Hence

$$-\lg |e| \leq 7(5L + 2)/2 + 80L = 97.5L + 7.$$

But we can directly compute an upper bound on $MC(e)$ using the rules in Table 3. This is shown in the last column of Table . This gives $MC(e) \leq mc(e) \leq (L + 4)/2$. Then

$$-\lg |e| \leq 7(L + 4)/2 + 80L = 83.5L + 3.5.$$

In the next section, we consider the Conjugate Bound which is an extension of the Measure-BFMS approach: for this example, it yields $-\lg |e| \leq 28L + 60$.

Exercises

Exercise 1.5: Show an expression e involving L-bit integers where the application of Theorem 5 is asymptotically better than that of measure or BFMSS bounds. ♢

Exercise 1.6: Prove that the BFMSS Bound is never smaller than BFMS Bound. ♢

End Exercises

§1.4. Eigenvalue Bound

This bound adopts an interesting approach based on matrix eigenvalues [12]. Let $\Lambda(n, b)$ denote the set of eigenvalues of $n \times n$ matrices with integer entries with absolute value at most b. It is easy to see that $\Lambda(n, b)$ is a finite set of algebraic integers. Moreover, if $\alpha \in \Lambda(n, b)$ is non-zero then $|\alpha| \geq (nb)^{1-n}$. Scheinerman gives a constructive zero bound for division-free radical expressions e by maintaining two parameters, $n(e)$ and $b(e)$, satisfying the property that the value of e is in $\Lambda(n(e), b(e))$. These recursive rules are given by Table 6.

Note that the rule for $\sqrt{cd}$ is rather special, but it can be extremely useful. In Rule 6, the polynomial $\overline{P}(x)$ is given by $\sum_{i=0}^{d} |a_i| x^i$ when $P(x) = \sum_{i=0}^{d} a_i x^i$. This rule is not explicitly stated in [12], but can be deduced from an example he gave. An example given in [12] is to test whether $\alpha = \sqrt{2} + \sqrt{5 - 2\sqrt{6}} - \sqrt{3}$ is zero. Scheinerman's bound requires calculating α to 39 digits while the BFMS bound says 12 digits are enough.

	e	$n(e)$	$b(e)$
1.	integer a	1	$\|a\|$
2.	$\sqrt{cd}$	2	$\max\{\|c\|, \|d\|\}$
3.	$e_1 \pm e_2$	$n_1 n_2$	$b_1 + b_2$
4.	$e_1 \times e_2$	$n_1 n_2$	$b_1 b_2$
5.	$\sqrt[k]{e_1}$	kn_1	b_1
6.	$P(e_1)$	n_1	$\overline{P}(n_1 b_1)$

Table 6: Eigenvalue Rules

§1.5. Conjugate Bound

The "conjugate bound" [8] is an extension of the Measure-BFMS bound above expressions with division. This approach can give significantly better performance than BFMS in many expressions involving divisions and root extractions. Because of division, we also maintain upper bounds $\mathrm{tc}(e)$, $M(e)$ on the tail $\mathtt{tail}(e)$ and $M(e)$. Here the tail coefficient $\mathtt{tail}(e)$ is defined as the constant term of the irreducible polynomial $\mathrm{Irr}(e)$.

Table 7 gives the recursive rules to maintain $M_0(e), \mathrm{tc}(e)$ and $M(e)$.

	e	$\mathrm{lc}(e)$	$\mathrm{tc}(e)$	$M(e)$
1.	rational $\frac{a}{b}$	$\|b\|$	$\|a\|$	$\max\{\|a\|, \|b\|\}$
2.	$\mathrm{Root}(P)$	$\|\mathtt{lead}P\|$	$\|\mathtt{tail}P\|$	$\|P\|_2$
3.	$e_1 \pm e_2$	$\mathrm{lc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$M_1^{D_2} M_2^{D_1} 2^{D(e)}$	$M_1^{D_2} M_2^{D_1} 2^{D(e)}$
4.	$e_1 \times e_2$	$\mathrm{lc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$\mathrm{tc}_1^{D_2}\mathrm{tc}_2^{D_1}$	$M_1^{D_2} M_2^{D_1}$
5.	$e_1 \div e_2$	$\mathrm{lc}_1^{D_2}\mathrm{tc}_2^{D_1}$	$\mathrm{tc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$M_1^{D_2} M_2^{D_1}$
6.	$\sqrt[k]{e_1}$	lc_1	tc_1	M_1
7.	e_1^k	lc_1^k	tc_1^k	M_1^k

Table 7: Recursive rules for $\mathrm{lc}(e)$ (and associated $\mathrm{tc}(e)$ and $M(e)$)

The upper bounds on conjugates, $MC(e)$, are obtained through resultant calculus and standard interval arithmetic techniques. It turns out that it is necessary to maintain a lower bound $\underline{\nu}(e)$ on the conjugates at the same time. The recursive rules to maintain these two bounds are given in Table 8.

	e	MC(e)	$\underline{\nu}(e)$
1.	rational $\frac{a}{b}$	$\|\frac{a}{b}\|$	$\|\frac{a}{b}\|$
2.	$\mathrm{Root}(P)$	$1 + \|P\|_\infty$	$(1 + \|P\|_\infty)^{-1}$
3.	$e_1 \pm e_2$	$MC(e_1) + MC(e_2)$	$\max\{M(e)^{-1}, (MC(e)^{D(e)-1}\mathrm{lc}(e))^{-1}\}$
4.	$e_1 \times e_2$	$MC(e_1)MC(e_2)$	$\underline{\nu}(e_1)\underline{\nu}(e_2)$
5.	$e_1 \div e_2$	$MC(e_1)/\underline{\nu}(e_2)$	$\underline{\nu}(e_1)/MC(e_2)$
6.	$\sqrt[k]{e_1}$	$\sqrt[k]{MC(e_1)}$	$\sqrt[k]{\underline{\nu}(e_1)}$
7.	e_1^k	$MC(e_1)^k$	$\underline{\nu}(e_1)^k$

Table 8: Recursive rules for bounds on conjugates

Finally, we obtain the new zero bound as follows: Given an Ω_3-expression e, if $\mathtt{val}(e) \downarrow$ and $e \neq 0$, then we obtain the lower bound from Lemma 4. This bound was implemented the `Core Library` and experiments show that it can achieve significant speedup over previous bounds in the presence of division [8].

§1.6. The Factoring Method for Root Bounds

Pion and Yap [11] introduced a root-bound technique based on the following idea: maintain zero bounds for an expression e in the factored form, $b = b_1 b_2$ where b_i $(i = 1, 2)$ is a zero bound for e_i and $e = e_1 e_2$. If b_i is obtained using method X (X being one of the above methods), and the factorization is carefully chosen, then $b_1 b_2$ could be a better bound than what X would have produced for e directly. The catch is the method has no advantage unless such a factorization for e exists and can be easily found. Fortunately, there is a class of predicates for which this approach wins: division-free predicates (e.g., determinants) in which the input numbers are **k-ary rationals**, i.e., numbers of the form nk^m where $n, m \in \mathbb{Z}$. This is an important class as the majority of real world input numbers are k-ary rationals for $k = 2$ or $k = 10$. For such expressions, we maintain bounds for the factorization $e = e_1 e_2$ where e_1 is division-free (but may have square roots) and $e_2 = k^v$ for some $v \in \mathbb{Z}$. Our technique is orthogonal to the various zero bounds which we already discussed because each of the root bounds can, in principle, use this factorization technique. This has been implemented in the `Core Library` for the BFMSS Bound and the Measure Bound, resulting in significant improvement for zero bounds in, for instance, determinants with k-ary rational inputs.

§1.7. Comparison of Bounds

Comparisons between various constructive zero bounds can be found in [4, 8]. In general, a direct comparison of the above zero bounds is a difficult task because of the different choice of parameters and bounding functions used. Following the tact in [8], we compare their performance on various special subclasses of algebraic expressions. We should note that there are currently three zero bounds, the BFMSS, Li-Yap and Measure Bounds, that are not dominated by any other methods. In particular, these three are mutually incomparable.

1. For division-free radical expressions, the BFMS bound is never worse than all the other bounds. Moreover, for this special class of expressions, Li-Yap bound is identical to the BFMS bound.

2. For general algebraic expressions, in terms of root bit-bound, Li-Yap bound is at most $D \cdot M$ where D is the degree bound, and M is the root bit-bound from the degree-measure bound.

3. Considering the sum of square roots of rational numbers (a common problem in solving the shortest Euclidean path problem), it can be shown that each of Li-Yap bound and the degree-measure bound can be better than the other depending on different parameters about the expressions. But both of them are always better than the BFMS bound.

4. Given a radical expression e with rational values at the leaves, if e has no divisions and shared radical nodes, Li-Yap bound for e is never worse than the BFMS bound, and can be better in many cases.

5. A critical test in Fortune's sweepline algorithm is to determine the sign of the expression $e = \frac{a+\sqrt{b}}{d} - \frac{a'+\sqrt{b'}}{d'}$ where a's, b's and d's are $3L$-, $6L$- and $2L$-bit integers, respectively. The BFMS bound requires $(79L + 30)$ bits and the degree-measure (D-M) bound needs $(64L + 12)$ bits. Li-Yap root bit-bound improves them to $(19L+9)$ bits. We generate some random inputs with different L values which always make $e = 0$, and put the timings (in seconds) of the tests in Table 9. The experiments are performed on a Sun UltraSPARC with a 440 MHz CPU and 512MB main memory.

L	10	20	50	100	200
NEW	0.01	0.03	0.12	0.69	3.90
BFMS	0.03	0.24	1.63	11.69	79.43
D-M	0.03	0.22	1.62	10.99	84.54

Table 9: Timings for Fortune's expression

§1.8. Treatment of Special Cases

Root separation bounds. In both the `Core Library` and `LEDA`, the comparison of two expressions α and β is obtained by computing the zero bound of $\alpha - \beta$. However, more efficient techniques can be used. If $P(X) \in \mathbb{C}[X]$ is a non-zero polynomial polynomial, sep(P) denotes the minimum $|\alpha_i - \alpha_j|$ where $\alpha_i \neq \alpha_j$ range over all pairs of complex roots of P. When P has less than two distinct roots, define $\text{sep(P)} = \infty$. Suppose $A(X)$ and $B(X)$ are the minimal polynomials for α and β, then $|\alpha - \beta| \geq \text{sep(AB)}$. If we maintain upper bounds d, d' on the degrees of A and B, and upper bounds h, h' on the heights of A and B, a root separation bound for $A(X)B(X)$ (which need not be square-free) is given by

$$|\alpha - \beta| \geq \left[2^{(n+1)/2}(n+1)hh'\right]^{-2n} \tag{15}$$

where $n = d + d'$ (see Corollary 6.33 in [15, p. 176,173] and use the fact that $\|AB\|_2 \leq (n+1)hh'$). The advantage of using (15) is that the root bit bound here is linear in $d + d'$, and not dd', as would be the case if we use resultant calculus. We compute α and β to an absolute error $< \text{sep(AB)}/4$ each, then declare them to be equal iff their approximations differ by $\leq \text{sep(AB)}/2$ from each other. Otherwise, the approximations tell us which number is larger. Note that this approach not fit into our recursive zero bound framework (in particular, it does not generate bounds for a new minimal polynomial).

Zero test. Zero testing is the special case of sign determination in which we want to know whether an expression is zero or not. Many predicates in computational geometry programs are really zero tests (e.g. detection of degeneracy, checking if a point lies on a hyperplane). In other applications, even though we need general sign determination, the zero outcome is very common. For instance, in the application of EGC to theorem proving [14], true conjectures are equivalent to the zero outcome. In our numerical approach based on zero bounds, the complexity of sign determination is determined by the zero bound when the outcome is zero. Since zero bounds can be overly pessimistic, such tests can be extremely slow. Hence it is desirable to have an independent method of testing if an expression is zero. Such a zero test can be used as a filter for the sign determination algorithm. Only when the filter detects a non-zero do we call the iterative precision numerical method.

Yap and Blömer [3] observed that for expressions of the form $e = \sum_{i=1}^{n} a_i\sqrt{b_i}$ ($a_i \in \mathbb{Z}, b_i \in \mathbb{N}$), zero testing is deterministic polynomial time, while the sign determination problem is not known to be polynomial time. Blömer [3, 1] extended this to the case of general radicals using a theorem of Siegel; he also [2] gave a probabilistic algorithm for zero test. When the radicals are nested, we can apply denesting algorithms [6, 7]. Note that these methods are non-numerical.

Exercises

Exercise 1.7: Let $a > 0$. Show that $a\max\{1, 1/a\} = \max\{1, a\}$ and $(1/a)\max\{1, a\} = \max\{1, 1/a\}$ ♢

Exercise 1.8: Show the conjugates of α are distinct. ♢

Exercise 1.9: Show that $M(A) = \exp\left[\int_0^1 \log|A(e(\theta))|d\theta|\right]$. ♢

End Exercises

References

[1] J. Blömer. Computing sums of radicals in polynomial time. *IEEE Foundations of Computer Sci.*, 32:670–677, 1991.

[2] J. Blömer. A probabilistic zero-test for expressions involving roots of rational numbers. *Proc. of the Sixth Annual European Symposium on Algorithms*, pages 151–162, 1998. LNCS 1461.

[3] J. Blömer. *Simplifying Expressions Involving Radicals*. PhD thesis, Free University Berlin, Department of Mathematics, October, 1992.

[4] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica*, 27:87–99, 2000.

[5] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *9th ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 2001. To appear, Algorithmica.

[6] G. Horng and M. D. Huang. Simplifying nested radicals and solving polynomials by radicals in minimum depth. *Proc. 31st Symp. on Foundations of Computer Science*, pages 847–854, 1990.

[7] S. Landau. Simplification of nested radicals. *SIAM Journal of Computing*, 21(1):85–110, 1992.

[8] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *12th SODA*, pages 496–505, Jan. 2001.

[9] M. Mignotte. Identification of algebraic numbers. *J. of Algorithms*, 3:197–204, 1982.

[10] M. Mignotte and D. Ştefănescu. *Polynomials: An Algorithmic Approach*. Springer, Singapore, 1999.

[11] S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. In *19th SCG*, pages 256–263, San Diego, California., 2003. Accepted, Theoretical Computer Science (2006).

[12] E. R. Scheinerman. When close enough is close enough. *Amer. Math. Monthly*, 107:489–499, 2000.

[13] H. Sekigawa. Using interval computation with the Mahler measure for zero determination of algebraic numbers. *Josai Information Sciences Researches*, 9(1):83–99, 1998.

[14] D. Tulone, C. Yap, and C. Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. In J. Richter-Gebert and D. Wang, editors, *Proc. 3rd Int'l. Workshop on Automated Deduction in Geometry (ADG 2000)*, volume 2061 of *Lecture Notes in Artificial Intelligence*, pages 58–82. Springer, 2001. Zurich, Switzerland.

[15] C. K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.

[16] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.

Lecture 13
NUMERICAL ALGEBRAIC COMPUTATION

We introduce the **numerical algebraic** *mode of computation. This can be viewed as a third form of computing with algebraic numbers. The main features of this computation is that we avoid all explicit manipulation of the polynomials that underlie algebraic numbers. Instead, we only use a numerical approximation. To recover the algebraic properties of the exact numerical values, we only use the theory of zero bounds as developed in the previous chapter.*

In general, the numerical algebraic mode leads to more efficient and practical algorithms. The complexity of numerical algebraic computation is adaptive, making it useful for applications.

§1. What is Numerical Algebraic Computation?

We have described two standard views of algebraic computation: manipulating algebraic relations in $\mathbb{Q}(\alpha)$ and real algebraic computation via Sturm theory. We now describe the third approach, based on numerical approximations.

Although it has some similarities to isolating intervals, the key difference is that we never explicit compute defining polynomials. Instead, we maintain some suitable bounds on algebraic quantities related to our numbers, and remember the construction history (which is an expression). Using such bounds, we can carry out exact comparisons of our algebraic numbers.

What exactly do we need? Suppose we want to compare two numbers, e.g.,

$$\sqrt{2} + \sqrt{3} : \sqrt{5 + 2\sqrt{6}}.$$

These two values are the same, and so regardless of how accurately we approximate the quantities, we cannot decisively conclude that the numbers are equal!

Our premise for this mode of computation goes as follows. Like Kronecker[1], we assume that the original numerical inputs are $\mathbb{Z}$; all other numbers must be explicitly constructed by application of a suitable operator. This is formalized as follows.

Let Ω be a set of algebraic operators. We restrict ourselves to real operators, and so each $f \in \Omega$ is a partial function $f : \mathbb{R}^n \to \mathbb{R}$ for some arity $n = n(f)$. Let $\Omega^{(n)} \subseteq \Omega$ denote the set of operators of arity n. Thus $\Omega^{(0)}$ are the constant operators of Ω. representing the constant operators (i.e., f with $n(f) = 0$). Let $\Omega_0 = \mathbb{Z} \cup \{\pm, \times\}$. We assume that $\Omega_0 \subseteq \Omega$.

[1]We are misappropriating Leopold Kronecker's famous statement, "Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk" [God made the whole numbers, all else are man-made]. This statement appeared in H. Weber's memorial article (1892) on Kronecker, and which Weber attributes to a 1866 speech by Kronecker.

Remark: The choice of $\Omega^{(0)}$ can have significant impact on the efficiency of our algorithms. In the simplest case, we have $\Omega^{(0)} = \mathbb{Z}$. But other natural choices are $\Omega^{(0)} = \mathbb{Q}$ or $\Omega^{(0)} = \mathbb{F} = \{n2^m : n, m \in \mathbb{Z}\}$ (floats).

The class of algorithms we study is intuitively easy to understand, for it resembles programs in modern high-level computer languages. We have basic data types like Booleans, integers and also real numbers. Here "real" numbers really do refer to elements of $\mathbb{R}$. However, the input numbers are restricted to $\mathbb{Z}$ (or $\mathbb{Q}$ or $\mathbb{F}$). In computer languagues, these input numbers are sometimes called **literals**. You can perform basic arithmetic operations using the operators in Ω, and you can compare any two real numbers. Basic data structures like arrays and control structures for looping and branching will be available. The loops and branches are controlled by comparisons of real numbers. It is possible to formalize this class of algorithms as the **algebraic computation model** [3].

Our algorithm therefore manipulates real numbers which can be represented by expressions over Ω. Let $Expr(\Omega)$ denote the set of such expressions. Note that each $e \in Expr(\Omega)$ is either undefined or denotes a real algebraic number. We shall write $\texttt{val}(e)$ for the value of e ($\texttt{val}(e) = \uparrow$ when undefined). When we write expressions, we often view them as rooted oriented dags (directed acyclic graphs), with each node labeled by an operator in Ω of the appropriate arity. For instance, all the leaves of the dag must be elements of $\Omega^{(0)}$.

[FIGURE of an EXPRESSION]

CONVENTION: It is convenient to write a plain "e" instead of "$\texttt{val}(e)$" when the context is clear.

§1.1. Relative and Absolute Approximations

Since we will be using approximations extensively, it is good to introduce some convenient notations. We are interested in both relative and absolute approximations. Indeed, the interaction between these two concepts will be an important part of our algorithms.

For real numbers x and p, let $\texttt{Rel}(x, p)$ denote the set of numbers $\widetilde{x}$ such that $|x - \widetilde{x}| \leq 2^{-p}|x|$. Any value y in $\texttt{Rel}(x, p)$ is called a **relative p-bit approximation** of x. Similar, let $\texttt{Abs}(x, p)$ denote the set of all $\widetilde{x}$ such that $|x - \widetilde{x}| \leq 2^{-p}$, i.e., the set of **absolute p-bit approximations** to x. In practice, p will be an integer.

Example. Consider a binary floating point number, $\widetilde{x} = m2^n \in \mathbb{F}$. If $m \neq 0$ is a p-bit integer, then $\widetilde{x} = \pm(b_p b_{p-1}, \ldots, b_1)_2$ where $b_i \in \{0, 1\}$ and $b_p = 1$. Assume $\widetilde{x}$ is an approximation to some value x in the interval $[(m-1)2^n, (m+1)2^n]$. So $|x| \geq (m-1)2^n \geq (2^{p-1} - 1)2^n$. Assume $p \geq 2$. Then $|x| \geq 2^{p-2+n}$, or

$$|x - \widetilde{x}| \leq 2^n \leq |x|2^{p-2}.$$

Hence $\widetilde{x}$ has $p - 2$ relative bits of precision.

Instead of writing "$\widetilde{x} \in \texttt{Rel}(x, p)$", we also use the alternative notation,

$$\widetilde{x} \sim x \ (\texttt{Rel}[p]). \tag{1}$$

Similarly, we write $\widetilde{x} \sim x \ (\texttt{Abs}[p])$ instead of "$\widetilde{x} \in \texttt{Abs}(x, p)$". Note that $y \sim x \ (\texttt{Rel}[p])$ is not equivalent to $x \sim y \ (\texttt{Rel}[p])$, but $y \sim x \ (\texttt{Abs}[p])$ is equivalent to $x \sim y \ (\texttt{Abs}[p])$. Generally, we define concepts using relative approximations, leaving to the reader the task of extend the definition to the absolute case.

We note the following basic relationship:

Lemma 1 *If $\widetilde{x} \sim x \ (\texttt{Rel}[1])$ then $\texttt{sign}(\widetilde{x}) = \texttt{sign}(x)$. In particular, $x = 0$ iff $\widetilde{x} = 0$.*

The proof is left as an exercise.

We are interested in approximating numerical functions. If $f : \mathbb{R}^n \to \mathbb{R}$ is a numerical function, we call a function $\widetilde{f} : \mathbb{R}^{n+1} \to \mathbb{R}$ a **relative approximation** of f if for all $x_1, \ldots, x_n, p \in \mathbb{R}$, we have that

$$\widetilde{f}(x_1, \ldots, x_n, p) \in \mathtt{Rel}(f(x_1, \ldots, x_n), p).$$

Moreover, $\widetilde{f}(x_1, \ldots, x_n, p)$ is undefined iff $f(x_1, \ldots, x_n)$ is undefined. In general, we write $\mathcal{R}(f)$ or $\mathcal{R}f$ to denote any function that relative approximation of f. Similarly, we can define the notion of absolute approximation function, and use the notation $\mathcal{A}(f)$ or $\mathcal{A}f$ for any function that absolute approximation of f.

§2. Approximate Expression Evaluation

The central computational problem of numerical algebraic computation can be posed as follows: given an expression $e \in Expr(\Omega)$ and a p, to compute an approximate value $\widetilde{e} \sim e$ ($\mathtt{Rel}[p]$).

The basic method is to propagate the precision p to all the nodes in E using suitable inductive rules. In the simplest case, this propagation may proceed only in one direction from the root down to the leaves. Then, we evaluate the approximate values at the leaves and recursively apply the operations at each node from the bottom up. In implementations, the approximate values at each node is a float. Algorithms to propagate composite precision bounds are described in [2, 4, 1, 3].

Complications arise when the propagation of precision requires bounds on the magnitude of the values at some nodes. This is where the zero bounds become essential. This topic is taken up in the next lecture but for now, we make a definition:

A total function $B : Expr(\Omega) \to \mathbb{R}_{\geq 0}$ is called a **zero bound function** for expressions over Ω if for all $e \in Expr(\Omega)$, if $\mathtt{val}(e)$ is defined and non-zero, then

$$|\mathtt{val}(e)| > B(e).$$

For example if $B(e) = 0$ for all e, then we call $B(e)$ a trivial bound. Another equally useless zero bound function is $B(e) = |\mathtt{val}(e)|$ whenever $\mathtt{val}(e)$ is defined (and otherwise 0). It is useless because such a function cannot be effectively computed. In the following, we will assume the existence of some easy to compute zero bound function.

A well-known technique in the literature called "lazy evaluation" has similarities to precision-driven computation. The lazy evaluation typically "pumps" increasingly precise values from the leaves to the root, and simply tracks the forward error. If this error is larger than the desired precision at the root, the process is iterated. In other words, the lazy approach typically has only the upward phase of precision-driven computation. Such iterations alone is insufficient to guarantee the sign of an expression, which is another essential ingredient of the precision-driven approach.

Most Significant Bit (msb). Zero bounds aim to bound $|\mathtt{val}(e)|$ away from 0. We will also need to bound $|\mathtt{val}(e)|$ from above. If $\div \in \Omega$, these two types of bounds are essentially interchangeable. Since $|\mathtt{val}(e)|$ can be very large or very small, and these bounds need not be very accurate in practice, it is more efficient to bound only the bit size, i.e., $\lg |\mathtt{val}(e)|$. Define

$$\mu(e) := \lg |\mathtt{val}(e)|. \tag{2}$$

We write $\mu^+(e)$ and $\mu^-(e)$ for any upper and lower bound on $\mu(e)$. Note that $\mu^+(e), \mu^-(e)$ are not functional notations (the value the denote depends on the context). In a sense, relative approximation of x amounts to computing the $\mu(x)$, plus an absolute approximation of x. For, $\widetilde{x} \in \texttt{Rel}(x, p)$ iff $\widetilde{x} \in \texttt{Abs}(x, p - \mu(x))$.

Closely related to $\mu(e)$ is the **most significant bit function**: for any real number x, let $\texttt{msb}(x) := \lfloor \lg |x| \rfloor$. By definition, $\texttt{msb}(0) = -\infty$. Thus,

$$2^{\texttt{msb}(x)} \leq |x| < 2^{1+\texttt{msb}(x)}.$$

If x in binary notation is $\cdots b_2 b_1 b_0 . b_{-1} b_{-2} \cdots$ $(b_i = 0, 1)$ then $\texttt{msb}(x) = t$ iff $b_t = 1$ and $b_i = 0$ for all $i > t$.

§2.1. Propagating Precision in Basic Operators

Let Ω be a set of operators and $f \in \Omega$ where $f : \mathbb{R}^n \to \mathbb{R}$. We assume the existence of algorithms to compute $f(x_1, \ldots, x_n; \pi)$ where $\pi = \texttt{Rel}[p]$ or $\texttt{Abs}[p]$. Such algorithms are given and analyzed by Ouchi[2] for the rational operators and square root. The question we now address is the following: suppose we want to compute $f(x_1, \ldots, x_n; \texttt{Abs}[p])$. But x_i are not explicitly known, and our goal is to determine $n + 1$ parameters

$$q_1 = q_1(p), \ldots, q_n = q_n(p), q = q(p)$$

such that if

$$wtx_i \sim x_i \ (\texttt{Abs}[p]) \qquad (i = 1, \ldots, n) \tag{3}$$

then

$$f(\widetilde{x_1}, \ldots, \widetilde{x_n}; \texttt{Abs}[q]) \sim f(x_1, \ldots, x_n) \ (\texttt{Abs}[p]). \tag{4}$$

This relation for the operator + is visually illustrated in Figure 1

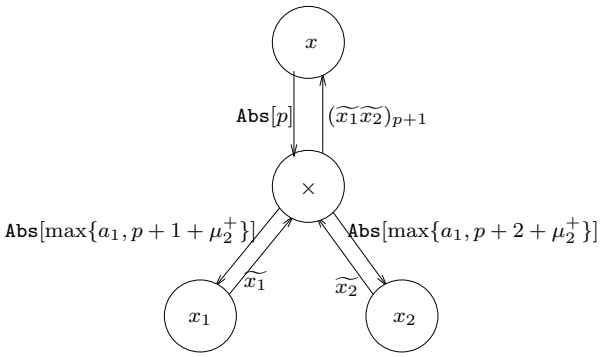


Figure 1: Propagation Rules for Absolute Precision Multiplication

Note that $q_1, \ldots, q_n, q$ can be negative. Also, the problem is trivial when $n = 0$ (just choose $q = p$). The following table summarizes the results:

The proofs can be found in [3]. We could have given this table in terms of relative error bounds although the entry for $\ln(e_1)$ cannot be filled. Even more generally, we can consider composite precision bounds [2]. But it seems that propagating absolute error bounds leads to a nice isolation of the "crucial issue" in approximation of expressions, as we will point out below.

The upward rules in Table 1 are straightforward and uniform: assuming each operator $f \in \Omega$ computes the value of $f(\widetilde{x_1}, \ldots, \widetilde{x_n})$ to at most $p + 1$ bits of absolute precsion. The downward rule can be view as

e	Downward Rules	Upward Rules
$e_1 \pm e_2$	$q_1 = p+2$, $q_2 = p+2$	$\widetilde{e} = (\widetilde{e_1} \pm \widetilde{e_2})_{p+1}$
$e_1 \times e_2$	$q_1 = \max\{a_1, p+1+\mu_2^+\}$, $q_2 = \max\{a_2, p+1+\mu_1^+\}$ where $a_1 + a + 2 = p+2$	$\widetilde{e} = (\widetilde{e_1} \times \widetilde{e_2})_{p+1}$
$e_1 \div e_2$	$q_1 = p+2-\mu_2^-$, $q_2 = \max\{1-\mu_2^-, p+2-2\mu_2^- + \mu_1^+\}$	$\widetilde{e} = (\widetilde{e_1}/\widetilde{e_2})_{p+1}$
$\sqrt[k]{e_1}$	$q_1 = \max\{p+1, 1-(\mu_2^-/2)\}$	$\widetilde{e} = (\sqrt[k]{\widetilde{e_1}})_{p+1}$
$\exp(e_1)$	$q_1 = \max\{1, p+2+2^{\mu_1^+ + 1}\}$	$\widetilde{e} = \exp(\widetilde{e_1}; \texttt{Abs}[p+1])$
$\ln(e_1)$	$q_1 = \max\{1-\mu_1^-, p+2-\mu_1^-\}$	$\widetilde{e} = \ln(\widetilde{e_1}; \texttt{Abs}[p+1])$

Table 1: Downward and Upward Rules for Absolute Precision

§2.2. Precision-Driven Approximation Algorithm

Let us consider the problem of guaranteeing absolute error bounds. We assume the availability of interval arithmetic for all our basic operators. That is, for any $f \in \Omega$, if interval arguments are given to f, then it can compute a reasonably tight interval output. To be specific, suppose f is a binary operator. Write $x \pm 2^p$ for the interval $[x - 2^p, x + 2^p]$. approximations of f, then $f(x\pi 2^{p_y}, y\pi 2^{p_y}) \in f(x,y)\pi 2^{p_z}$ where p_z is as small as possible.

The rules to propagate α_E to the children of E are presented in column 2 of Table 2. If E_1, E_2 are the children of E, we want to define α_{E_i} $(i = 1, 2)$ in such a way that if $\widetilde{E_i}$ satisfies $|E_i - \widetilde{E_i}| \leq \alpha_{E_i}$ then $\widetilde{E}$ satisfies $|E - \widetilde{E}| \leq \alpha_E$. For simplicity, we write α_{E_i} for α_i. Here, $\widetilde{E}$ is obtained by applying the operator at E to $\widetilde{E_1}, \widetilde{E_2}$, computed to some specified precision. The rule for computing this approximation is given in column 3 of Table 2. A notation used in column 3 is that, for any real X and $\alpha > 0$, $(X)_\alpha$ refers to any approximation $\widetilde{X}$ for X that satisfies the bound $|X - \widetilde{X}| \leq \alpha$.

In summary, column 2 tells us how to propagate absolute precision bounds downward towards the leaves, and column 3 tells us how to compute approximations from the leaves upward to the root. At the leaves, we assume an ability to generate numerical approximations to within the desired error bounds. At each node F, our rules guarantee that the approximate value at F satisfies the required absolute precision bound α_F. Since we only propagate absolute precision, and not composite precision, the underlying algorithms for each of the primitive operations $(+, -, \times, \div, \sqrt{}$, etc) is greatly simplified (cf. [2]).

Note that for the addition, subtraction and multiplication operations, the computation of $\widetilde{E}$ can be performed exactly (as in [1]). But the present rules no longer require exact computation, even in these cases. The new rule is never much worse than the old rule (at most one extra bit at each node), but is sensitive to the actual precision needed. In fact, for all operations, we now allow an absolute error of $\frac{\alpha_E}{2}$. Let us briefly justify the rule for the case $E = E_1 \times E_2$ in Table 2. The goal is to ensure $|E - \widetilde{E}| \leq \alpha_E$. Since $|\widetilde{E} - \widetilde{E_1}\widetilde{E_2}| \leq \alpha_E/2$ by our upward rules, it is sufficient to ensure that $|E - \widetilde{E_1}\widetilde{E_2}| \leq \alpha_E/2$. But $|E - \widetilde{E_1}\widetilde{E_2}| \leq \alpha_1|E_2| + \alpha_2|E_1| + \alpha_1\alpha_2 \leq \frac{\alpha_E}{c} + \frac{\alpha_E}{c} + \frac{\alpha_E^2}{c^2}$. So it is sufficient to ensure that $\frac{\alpha_E}{c} + \frac{\alpha_E}{c} + \frac{\alpha_E^2}{c^2} \leq \alpha_E/2$. Solving for c, we obtain $c \geq 2 + \sqrt{4 + 2\alpha_E}$. See [1] for justifications of the other entries.

Column 2 is not directly usable in implementation; the formulas for α_i $(i = 1, 2)$ are expressed in terms of $|E_1|$ and $|E_2|$ to make the formulas easier to understand. Since it is generally neither possible nor desirable to compute $|E_i|$

E	Downward Rules	Upward Rules
$E_1 \pm E_2$	$\alpha_1 = \alpha_2 = \frac{1}{4}\alpha_E$	$\widetilde{E} = (\widetilde{E_1} \pm \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$E_1 \times E_2$	Let $c \geq 2 + \sqrt{4 + 2\alpha_E}$, and $\alpha_1 = \frac{\alpha_E}{c}\min\{1, 1/\lvert E_2\rvert\}$, $\alpha_2 = \frac{\alpha_E}{c}\min\{1, 1/\lvert E_1\rvert\}$.	$\widetilde{E} = (\widetilde{E_1} \times \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$E_1 \div E_2$	$\alpha_1 = \alpha_E\lvert E_2\rvert/4$, $\alpha_2 = \frac{\alpha_E\lvert E_2\rvert}{4\lvert E\rvert + 2\alpha_E}$	$\widetilde{E} = (\widetilde{E_1} \oslash \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$\sqrt[k]{E_1}$	$\alpha_1 = \alpha_E \sqrt[k]{\lvert E_1\rvert}^{k-1}/2$	$\widetilde{E} = (\widetilde{\sqrt[k]{\widetilde{E_1}}})_{\frac{\alpha_E}{2}}$

Table 2: Rules for (1) Propagating absolute precision α_E and (2) Approximation $\widetilde{E}$

exactly, we will replace these by upper or lower bounds on $|E_i|$. We next address this issue.

If $\alpha_E = 0$, the rules in Table 2 propagates this zero value to all its children. This is not generally impossible in our implementation model, where $\widetilde{E}$ is assumed to be a big float value. Although it is possible to generalize this to allow exact representation at a hugh cost in efficiency, we prefer to simply require $\alpha_E > 0$.

§3. Bounds on the Magnitude of Expressions

In `Real/Expr` and `Core Library`, expression evaluation begins by computing bounds on the absolute value of each node (see [2]). We need such bounds for two purposes: (1) propagating absolute precision bounds using the rules in Table 2, and (2) translating a relative precision bound at root into an equivalent absolute one.

We review this magnitude bounds computation. For any expression E, we define its **most significant bit** $\texttt{msb}(E)$ to be $\lfloor \lg(|E|) \rfloor$. Intuitively, the `msb` of E tells us about the location of the most significant bit of E. By definition, the $\texttt{msb}(0) = -\infty$. For efficiency purpose in practice, we will compute a bounding interval $\left[\mu_E^-, \mu_E^+\right]$ that contains $\texttt{msb}(E)$, instead of computing its true value. The rules in Table 3 are used to maintain this interval.

E	μ_E^+	μ_E^-
rational $\frac{a}{b}$	$\lceil \lg(\frac{a}{b}) \rceil$	$\lfloor \lg(\frac{a}{b}) \rfloor$
$E_1 \pm E_2$	$\max\{\mu_{E_1}^+, \mu_{E_2}^+\} + 1$	$\lfloor \lg(\lvert E\rvert) \rfloor$
$E_1 \times E_2$	$\mu_{E_1}^+ + \mu_{E_2}^+$	$\mu_{E_1}^- + \mu_{E_2}^-$
$E_1 \div E_2$	$\mu_{E_1}^+ - \mu_{E_2}^-$	$\mu_{E_1}^- - \mu_{E_2}^+$
$\sqrt[k]{E_1}$	$\lceil \mu_{E_1}^+/k \rceil$	$\lfloor \mu_{E_1}^-/k \rfloor$

Table 3: Rules for upper and lower bounds on $\texttt{msb}(E)$

The main subtlety in this table is the entry for μ_E^- when $E = E_1 \pm E_2$. We call this the **special entry** of this table because, due to potential cancellation, we cannot derive a lower bound on $\texttt{msb}(E)$ in terms of the bounds on E_1 and E_2 only. If the MSB bounds cannot be obtained from the fast floating-point filtering techniques, there are two possible ways to determine this entry in practice. First, we could approximate E numerically to obtain its most significant bit, or to reach the root bound in case $E = 0$. Thus, this method really determines the true value of $\texttt{msb}(E)$, and provides the entry $\lfloor \lg |E| \rfloor$ shown in the table. This numerical approximation process can be conducted in a progressive and

adaptive way (e.g., doubling the absolute precision in each iteration, until it finds out the exact MSB, or reaches the root bit-bound). The second method is applicable only under certain conditions: either when E_1 and E_2 have the same (resp., opposite) sign in the addition (resp., subtraction) case, or their magnitudes are quite different (by looking at their `msb` bounds). In either case, we can deduce the μ_E^- from the bounds on E's children. For instance, if $\mu_{E_1}^- > \mu_{E_2}^+ + 1$ then $\mu_{E_1}^- - 1$ is a lower bound for μ_E^-. This approach could give better performance if the signs of both operands are relatively easier to be obtained.

In the current implementation, we assume μ_E^- and μ_E^+ are machine integers. Improved accuracy can be achieved by using machine doubles. Furthermore, we could re-interprete μ_E^- and μ_E^+ to be upper and lower bounds on $\lg|E|$ (and not $\lfloor \lg|E| \rfloor$). Note that $|\mu_E^+ - \mu_E^-| \leq 2^m$ where m is the number of operations in E.

§4. The Approximation and MSB Algorithms

Two key algorithms will be derived from the above tables: one is APPROX(E, α_E) which computes an approximation of E to within the absolute precision α_E. The other algorithm is ComputeMSB(E), which computes upper and/or lower bounds for `msb`(E), following the rules in Table 3. A third algorithm of interest is **sign determination** for an expression E. Although this can be reduced to APPROX$(E,\ \beta(E)/2)$ where $\beta(E)$ is the root bound, such bound can be overly pessimistic. Instead, we note below how to incorporate the sign determination algorithm into ComputeMSB. The original algorithms for APPROX and ComputeMSB in `Core Library` require the sign of every node in the expression to be determined first. This is no longer required in the algorithm to be presented.

It should be noted that APPROX and ComputeMSB are mutually recursive algorithms: this is because ComputeMSB will need to call APPROX to compute the special entry (for $\mu^-_{E_1 \pm E_2}$) in Table 3. Clearly, APPROX needs ComputeMSB for the downward rules (except for addition or subtraction) in Table 2. It is not hard to verify that this mutual recursion will not lead to infinite loops based on two facts: (1) the underlying graph of E is a DAG, and (2) for the addition/subtraction node, the ComputeMSB may need to call APPROX, while the APPROX will not call ComputeMSB at the same node again. Although it is conceivable to combine APPROX and ComputeMSB into one, it is better to keep them apart for clarity as well for as their distinct roles: ComputeMSB is viewed as a one-time pre-computation while APPROX can be repeatedly called.

APPROX The APPROX algorithm has two phases: (1) distributing the precision requirement down the DAG, and (2) calculating an approximate value from leaves up to the root. Step (2) is straightforward, as it just has to call the underlying algorithms for performing the desired operation ($+, -, \times$, etc) to compute the result to $\alpha_E/2$, the desired absolute precision. We refer the reader to Koji [2] for implementation of these underlying algorithms – it amounts to a multiprecision library with knowledge of precision bounds. As for step (1), we see from Table 2 that we need both upper bounds and lower bounds on `msb` . However, these are not required by every operation. Instead of computing both the upper and lower `msb` bounds at every node, we propose to compute them as needed. The following is immediate from column 2 of the Table 2:

- Addition and subtraction $E = E_1 \pm E_2$. No `msb` bounds on E_1, E_2 are needed to propagate precision bounds.

- Multiplication $E = E_1 \times E_2$. Only the μ^+'s of E_1, E_2 are needed.
- Division $E = E_1 \div E_2$. Only the bounds $\mu^+_{E_1}$ and $\mu^-_{E_2}$ are needed.
- Root extraction $E = \sqrt[k]{E_1}$. Only $\mu^-_{E_1}$ is needed.

Briefly, APPROX(E,α_E) proceeds as follows. It first checks to see if there is already a current approximation to E, and if so, whether this approximation has error less than α_E. If so, it immediately returns. Otherwise, depending on the nature of the operator at E, it checks to see if it already the current upper and/or lower bounds on $|E_i|$ (where E_i are the children of E). This follows the rules noted above. If any of these bounds are not known, it will call ComputeMSB($E, b, b', false$) where b, b' are the appropriate Boolean flags (see below). When this returns, it can compute α_i and recursively call APPROX(E_i, α_i) for the ith child. Finally, it takes the approximate values $\widetilde{E_i}$ available at its children and computes $\widetilde{E}$ from them using the upward rules.

ComputeMSB We describe a refined ComputeMSB algorithm that take three additional arguments: ComputeMSB(E, needUMSB, needLMSB, needSIGN) where the need-variables are Boolean flags. The needUMSB (resp., needLMSB) flag is set to true when an upper (resp., lower) bound on `msb`(E) is needed. The meaning of needSIGN is clear. We see that in all but one case, the sign of E is completely determined by the signs of E's children. Hence we just have to propagate needSIGN to the children. The exception is when E is a $\pm$-node. So we could now define `getSign(E)` $\equiv$ ComputeMSB`(E, false, false, true)`.

In the algorithm, we prevent re-computation of bounds or signs by first checking whether it has been computed before. This is important as nodes are shared. We simply present a self-explanatory ComputeMSB algorithm here with a `C++`-like syntax:

```
Algorithm
  ComputeMSB(E, needUMSB, needLMSB, needSIGN) {

  // Check if any requested computation is necessary:
    if (E.umsb was computed) needUMSB = false;
    if (E.lmsb was computed) needLMSB = false;
    if (E.sign was computed) needSIGN = false;
    if (needUMSB or needLMSB or needSIGN)=false return;

    switch (E.operation_type) {
    case 'constant':
        if (needUMSB) E.umsb = ceilLog(E.value));
                            // ceilLog is ceiling of log_2
        if (needLMSB) E.lmsb = floorLog(E.value));
        if (needSIGN) E.sign = sign(E.value));
        break;
    case '+' or '-':
        if (needLMSB or needSIGN) {     // some optimization omitted
            for (i=1; i<= ceilLog(E.root_bound) ; i++)  {
                APPROX(E, 2^{-i});      // adaptive precisio
                if (log_2 (|E.approx|)  < -i)
                        E.sign = sign(E.approx).
            }
            E.lmsb = floorLog(E.value);
```

```
            E.umsb = ceilLog(E.value);
            E.sign = sign(E.value);
        } else {
            ComputeMSB(E.first, true, false, false);
            ComputeMSB(E.second, true, false, false);
            E.umsb = max{E.first.umsb, E.second.umsb} + 1;
        }
        break;
    case '*':
        ComputeMSB(E.first,  needUMSB, needLMSB, needSIGN);
        ComputeMSB(E.second, needUMSB, needLMSB, needSIGN);
        if (needUMSB) E.umsb = E.first.umsb + E.second.umsb;
        if (needLMSB) E.lmsb = E.first.lmsb + E.second.lmsb;
        if (needSIGN) E.sign = E.first.sign * E.second.sign;
        break;
    case '/':
        ComputeMSB(E.first,  needUMSB, needLMSB, needSIGN);
        ComputeMSB(E.second, needLMSB, needUMSB, needSIGN);
        if (needUMSB) E.umsb = E.first.umsb - E.second.lmsb;
        if (needLMSB) E.lmsb = E.first.lmsb - E.second.umsb;
        if (needSIGN) {
            if (E.second.sign=0)  E.sign = undefined;
            else E.sign = E.first.sign * E.second.sign;
        }
        break;
    case 'k-th root extraction':
        ComputeMSB(E.first, needUMSB, needLMSB, needSIGN)
        if (needUMSB) E.umsb = E.first.umsb / k;
        if (needLMSB) E.lmsb = E.first.lmsb / k;
        if (needSIGN) E.sign = E.first.sign;
        if (needSIGN) {
            if (E.first.sign = -1)  E.sign = undefined;
            else E.sign = E.first.sign.
        }
        break;
     } //switch
} //ComputeMSB
```

This completes our description of a precision-driven evaluation. It should be noted that on top of this evaluation mechanism, the `Core Library` also has a floating-point filter. This, plus several other minor improvements, have been omitted for clarity. for instance, to get the sign of E_1E_2, we first determine the sign of E_1. If $E_1 = 0$, we do not need the sign of E_2. While our new design is an improvement over an older one [2], it still seems to be suboptimal. For instance, to determine the sign of E_1E_2, we always determine the sign of E_1. This is unnecessary if $E_2 = 0$. One possibility is to "simultaneously determine" the signs of E_1 and E_2, stopping this determination when either one returns a 0. The idea is to expend equal amounts of effort for the 2 children, but this may be complicated to implement in the presence of shared nodes.

In summary, we pose as a major open problem to design some precision-driven evaluation mechanism which is provably optimal, in some suitable sense.

Using the results there, we have a new table for downward propagation.

References

[1] C. Li. *Exact Geometric Computation: Theory and Applications*. Ph.d. thesis, New York University, Department of Computer Science, Courant Institute, Jan. 2001. Download from `http://cs.nyu.edu/exact/doc/`.

[2] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, Jan. 1997. Download from `http://cs.nyu.edu/exact/doc/`.

[3] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*. World Scientific Publishing Co., Singapore, 2004. To appear.

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

Lecture 20
CURVES

This chapter study algebraic curves.

§1. Plane Algebraic Curves

Curves are fairly complex mathematical objects. There are two main mathematical approaches to curves: from the viewpoint of differential geometry [3] or from the viewpoint of algebraic geometry [7]. The former view is more general but less constructive than the latter. We basically take the algebraic viewpoint, but we shall emphasize the computational aspects of curves. We focus mainly on plane algebraic curves. Bruce-Giblin [3] and Fulton [7] are excellent introduction to the differential and algebraic geometric viewpoints, respectively. Other references include [6, 2, 1].

What is a Curve? We all have intuitive ideas about curves because of their striking visual nature. Naively, a curve C is just a set $L(C)$ of points in some topological space S. We call $L(C)$ the **locus** of C. We are most interested in the case where S is the Euclidean plane, $L(C) \subseteq \mathbb{R}^2$. To say that the curve C is "algebraic" means $L(C)$ is the zero set of some non-zero polynomial

$$L(C) = \texttt{Zero}(A), \qquad A(X,Y) \in \mathbb{Z}[X,Y].$$

We also refer to C as "the curve $A(X,Y) = 0$" or simply, $C : A(X,Y) = 0$. We call $A(X,Y)$ the **defining polynomial** of the curve. Moreover, defining polynomials are considered equivalent up to multiplication by a non-zero scalar: $A(X,Y)$ and $cA(X,Y)$ are considered the same defining polynomial for $c \neq 0$. The equation $A(X,Y)$ contains more information than its locus $L(C)$. For instance, if $B(X,Y) = A(X,Y)^2$, then the curve $C' : B(X,Y) = 0$ and the curve $C : A(X,Y) = 0$ have the same locus. But algebraically, they are different: the curve C' is equal to two copies of the curve C. Equivalently, each point p in the locus $L(C')$ as multiplicity 2. These concepts will be made precise using algebraic tools. The choice of the space S is also important: thus equation $X^2 + Y^2 = 0$ gives a "curve" in $S = \mathbb{R}^2$ is just a single point at the origin, but in $S = \mathbb{C}^2$ consists of two lines ($Y = \pm \mathbf{i} X$). We prefer $\mathbb{R}^2$ whenever possible, but it is often convenient to consider $\mathbb{C}^2$. There is a natural notion of points in S satisfying a polynomial equation $A(X,Y) = 0$; the **zero set** $\texttt{Zero}_S(A(X,Y))$ is just the set of points in S which satisfy $A(X,Y) = 0$.

To summarize, a **plane algebraic curve** C is given by its defining polynomial $A(X,Y)$ and the underlying space S. Its locus $L(C)$ is the zero set $\texttt{Zero}_S(A(X,Y)) \subseteq S$. We often refer to C as "the curve $A(X,Y)$" when S is understood.

We see the first divergence between the algebraic viewpoint and the naive view of curves: most applications of curves are only interested in the locus $L(C)$,

and do not care about multiplicity of points in the locus. Fortunately, there is an easy way to remove multiplicities: if the polynomial $A(X,Y)$ is square-free, then all but finitely many points in $L(C)$ has multiplicity 1. Such curves are said to be **reduced**. Note that if the curve C intersects itself, then at the point p of self-intersection, p has multiplicity greater than 1. To admit such curves, we do not insist that *every* point in $L(C)$ have multiplicity 1.

We can simplify our study of curves by observing that if

$$A(X,Y) = A_1(X,Y)A_2(X,Y)$$

then the curve $A = 0$ is the union of two curves $A_1 = 0$ and $A_2 = 0$. We say the curve $A = 0$ is **reducible** if it is the union of two curves $A_1 = 0$ and $A_2 = 0$ where $\deg(A_1)\deg(A_2) > 0$; otherwise, the curve is **irreducible**. Irreducible curves are reduced, but not conversely. It is mathematically appealing to focus on irreducible curves, but this may not always be desirable. One problem is that factorization is quite non-trivial (even though it is in polynomial time). Second, factorization depends on the coefficient domain: even though $A(X,Y) \in \mathbb{Z}[X,Y]$, but we can consider its factors in $D[X,Y]$ for various D ($\mathbb{Z} \subseteq D \subseteq \mathbb{C}$). In contrast to factorization, the concept of square-freeness (i.e., of reduced curves) is intrinsic to $A(X,Y)$: A is square-free iff $\texttt{GCD}(A, A') = 1$, and the concept of `GCD` is defined in $\mathbb{Z}[X,Y]$.

Another area where the algebraic notion of curves diverge from practical applications is that in practice, we are only interested in some subset of $L(C)$ (usually a connected finite part of $L(C)$). The algebraic theory does not handle this refinement. We need semi-algebraic techniques or other methods which are common in geometric modeling. To see some issues arising from handling curve segments, consider the nodal cubic curve $Y^2 - X^2 - X^3 = 0$ illustrated in Figure 1. Consider the portion of the locus indicated by the thick curve: it is a connected set, but in the topology of the curve, the set is actually disconnected. This is related to the issue of curve tracing: we need to be able to "trace" a curve through singularities (such as self-intersection points) in order to properly order points along the curve. But how do we distinguish this computationally? One way is to introduce semi-algebraic geometry, namely geometry that are defined by polynomial inequalities as well as equalities. For example, in Figure 1, we can specify the desired portion by restricting the curve to the rectangular box indicated by dashed lines: i.e., (x,y) such that $a \le x \le b$ and $0 \le y \le c$. This introduces a new level of sophistication into the computational aspects of curves. In short, we need to be aware of gaps between the purely mathematical investigation of curves, and their computational feasibility.

Projective Space. We discuss the topological space S where $L(C)$ lives. We had defined $L(C)$ to be the set $\{(x,y) \in \mathbb{R}^2 : A(x,y) = 0\}$, i.e., we assumed $S = \mathbb{R}^2$. This is the **real affine locus** of the curve $C : A(X,Y) = 0$, and $\mathbb{R}^2$ is the **real affine plane**. If we consider the **complex affine space** $S = \mathbb{C}^2$, then the corresponding **complex affine locus** is given by $\{(x,y) \in \mathbb{C}^2 : A(x,y) = 0\}$. Unless otherwise stated, let $L(C)$ refer to the complex affine locus; the real affine locus is just $L(C) \cap \mathbb{R}^2$.

In general, for any field K, we can denote the affine field K^n by the more fancy notation, $\mathbb{A}^n(K)$. To understand the behaviour of curves at infinity, we must go to projective spaces $\mathbb{P}^n(K)$. In general, the points of $\mathbb{P}^n(K)$ are the equivalence classes over the set $K^{n+1} \setminus \{\mathbf{0}\}$ where $\mathbf{0} = (0,0,\ldots,0)$ and the equivalence relations have the form $(x_0,x_1,\ldots,x_n) \equiv (cx_0,cx_1,\ldots,cx_n)$, for all $c \in K \setminus \{0\}$. Thus a **projective point** is the equivalence class of $(x_0,x_1,\ldots,x_n) \in K^{n+1} \setminus \{0\}$, and we denote this point by $(x_0 : x_1 : \cdots : x_n)$. The tuple $(x_0,x_1,\ldots,x_n)$ is called a **set of homogenous coordinates** for

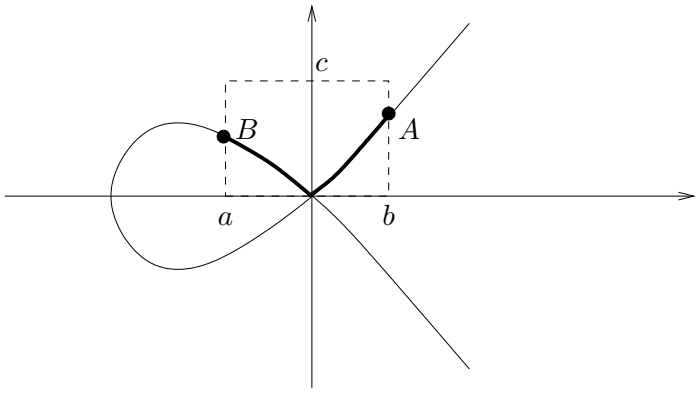


Figure 1: Nodal Cubic curve $Y^2 - X^2 - X^3 = 0$.

$(x_0 : x_1 : \cdots : x_n)$. In case $n = 2$, we have the **projective plane** $\mathbb{P}^2(K)$ over K. We use $K = \mathbb{R}$ and $K = \mathbb{C}$.

Given $A(X, Y) \in \mathbb{Z}[X, Y]$, there is a standard way to define a homogeneous polynomial $\widehat{A}(W, X, Y) \in \mathbb{Z}[W, X, Y]$ of the same degree. The homogeneous polynomial $\widehat{A}(W, X, Y)$ is characterized by the equation

$$\widehat{A}(W, X, Y) = W^d A(X/W, Y/W)$$

where $d = \deg(A)$. For instance, if $A(X, Y) = X^2Y^2 + X^2 + XY - 3Y + 5$ then $\widehat{A}(W, X, Y) = X^2Y^2 + X^2W^2 + XYW^2 - 3YW^3 + 5W^4$. Thus W is the **homogenization variable**. It follows that

$$\widehat{A}(1, X, Y) = A(X, Y). \tag{1}$$

If $H(W, X, Y)$ is any homogeneous polynomial, we can define the polynomial $H^v(X, Y)$ to be $H(1, X, Y)$. Thus, $(\widehat{A}(X, Y))^v = A(X, Y)$ for all $A(X, Y)$.

We define the **projective locus** of $A(X, Y)$ in $\mathbb{P}^2(K)$ to be

$$\{(w : x : y) \in \mathbb{P}^2(K) : \widehat{A}(w, x, y) = 0\}.$$

This definition requires justification since we have used an arbitrary set of homogeneous coordinates (w, x, y) from projective point $(w : x : y)$. However, the justification is easy because a polynomial $H(W, X, Y)$ is homogeneous iff $H(w, x, y) = 0$ iff $H(cx, cy, cw) = 0$ for all $c \neq 0$. Hence, the locus of $A(X, Y)$ in $\mathbb{P}^2(K)$ is well-defined.

The Four Loci for a Curve. We are interested in the projective locus of the curve $A(X, Y) = 0$ in two cases: $K = \mathbb{R}$ and $K = \mathbb{C}$ (real and complex projective loci, respectively). Thus we have define four loci for the curve $A(X, Y)$. How are they related? The relationship between

$$\mathbb{A}^n(\mathbb{R}) \subseteq \mathbb{A}^n(\mathbb{C})$$

is immediate since $\mathbb{R} \subseteq \mathbb{C}$. We also have the relationship between

$$\mathbb{A}^n(K) \subseteq \mathbb{P}^n(K) \tag{2}$$

is achieved via the identification $(x_1, \ldots, x_n) \in \mathbb{A}^n(K)$ iff $(1 : x_1 : \cdots : x_n) \in \mathbb{P}^n(K)$. We call x_0 the homogenization coordinate. We call points of $\mathbb{A}^n(K)$ the

finite points of $\mathbb{P}^n(K)$, and the rest are the **infinite points**, having the form $(0 : x_1 : \cdots : x_n)$. The set of these infinite points corresponds to the hyperplane defined by the equation $x_0 = 0$. Thus $\mathbb{P}^n(K)$ is simply $\mathbb{A}^n(K)$ adjoined with the hyperplane $x_0 = 0$. Summarizing, we have the following of lattice of inclusions:

$$\begin{array}{ccc} \mathbb{A}^n(\mathbb{R}) & \subset & \mathbb{A}^n(\mathbb{C}) \\ \cap & & \cap \\ \mathbb{P}^n(\mathbb{R}) & \subset & \mathbb{P}^n(\mathbb{C}). \end{array} \tag{3}$$

Focusing on $n = 2$ again, consider the affine and projective loci of a curve $C : A(X, Y) = 0$. Let $L(C) \subseteq K^2$ and $\widehat{L}(C) \subseteq \mathbb{P}^2(K)$. We characterize the extra points in $\widehat{L}(C) \setminus L(C)$ as follows: for any polynomial $A(X, Y)$, let $A_i(X, Y)$ denote the homogeneous part of $A(X, Y)$ of degree i. Then, if $d = \deg(A)$, we have

$$A = A_d + A_{d-1} + \cdots + A_0. \tag{4}$$

For instance, if $A(X, Y) = X^2Y^2 + X^2 + XY - 3Y + 5$ then $A_0 = 5$, $A_1 = -3Y$, $A_2 = X^2 + XY$, $A_3 = 0$, $A_4 = X^2Y^2$ and $A_i = 0$ for $i \geq 5$.

Lemma 1 *If $d = \deg(A(X, Y) \geq 1$ then*

$$\widehat{L}(C) \setminus L(C) = \{(x : y : 0) : A_d(x, y) = 0\}.$$

If case $\widehat{L}(C) \subseteq \mathbb{P}^2(\mathbb{C})$, it has exactly d infinite points.

Proof. In proof, from (4) we have

$$\widehat{A} = A_d + WA_{d-1} + \cdots + W^dA_0. \tag{5}$$

But $(x : y : 0)$ is a point of $\widehat{L}(C)$ iff $\widehat{A}(x, y, 0) = 0$. But (5) shows that $\widehat{A}(x, y, 0) = 0$ iff $A_d(x, y) = 0$, as claimed. In case we consider the complex projective plane, $A_d(X, Y)$ factors into exactly d factors,

$$A_d(X, Y) = c\prod_{i=1}^{d}(a_iX - b_iY)$$

and so $(b_i : a_i : 0)$ (for $i = 1, \ldots, d$) are the points at infinity in this locus. Of course, this means that there are at most d such points in $\mathbb{P}^2(\mathbb{R})$. **Q.E.D.**

Ultimately, we are interested in real affine locus but the other loci often yield additional information and permits more elegant mathematical treatment. We shall freely move among the four spaces of (3) as convenient.

Although we focus on plane curves, in general, we can study 1-dimensional varieties in n-dimensional space. These might be called "space curves", to emphasize that they are not necessarily in the plane. It is known that space curves are birationally equivalent to a plane curve – so, up to birational equivalence, the restriction to plane curves is justified.

Exercises

Exercise 1.1: What are the points at infinity of the curve $F(X, Y) = 0$?
(i) $F(X, Y) = X^2 + Y^2 - 1$.
(ii) $F(X, Y) = X^{k+1} - Y^k$ $(k \geq 1)$.
(iii) $F(X, Y) = XY - 1$.
(iv) $F(X, Y) = X^k + bY^k - 1$ ($b > 0$ and k even) ♢

End Exercises

§2. Singular Points and Multiplicity

Consider the curve $C : A(X, Y) = 0$ in some affine space S. For any point $p = (a, b) \in S$, consider the Taylor expansion of $A(X, Y)$ at p:

$$\begin{aligned} A(X,Y) &= \sum_{i\geq 0}\sum_{j\geq 0}\binom{i+j}{i}A_{i,j}(a,b)(X-a)^i(Y-b)^j \\ &= A(a,b) + A_{1,0}(a,b) \end{aligned}$$

where $A_{i,j} = \frac{\partial^{i+j}A}{\partial X^i \partial Y^j}$. We say p has **multiplicity** m if the partial derivatives $A_{i+j}(a, b)$ vanish for all $i + j \leq m - 1$ and some $A_{i+j}(a, b)$ does not vanish for some $i + j = m$. For instance, if $m = 0$ it means that $A(a, b) \neq 0$ and so p is not in the locus of C. If $m = 1$, then p is a **simple point** of C. If $m \geq 2$, then p is a **singular point**. Alternatively, p is a singular point iff $A_{1,0}(p) = A_{0,1}(p) = 0$. Alternatively,

$$\frac{\partial A}{\partial X} = \frac{\partial A}{\partial Y} = 0.$$

Write $\text{mult}_p(C)$ for the multiplicity of p in the curve C. Suppose $\text{mult}_p(C) = m$. Then consider the homogeneous function,

$$\sum_{i\geq 0}^{r}\binom{r}{i}A_{i,r-i}(a,b)(X-a)^i(Y-b)^{r-i}.$$

This equation, in $S = \mathbb{C}^2$, factors into m linear factors. Clearly, the lines passes p, and are called the **tangent lines** at p. If there are m distinct lines, we call this an **ordinary point**, otherwise an **non-ordinary point**.

Example: Consider $A(X, Y) = Y^2 - X^2 - X^3 = 0$ (see Figure 1). We have $A_{1,0}(X, Y) = -2X - 3X^2$ and $A_{0,1}(X, Y) = 2Y$. The origin $\mathbf{0}$ is a singular point since $A_{1,0}(0, 0) = A_{0,1}(0, 0) = 0$. Since $A_{0,2}(X, Y) = 2$, the origin have multiplicity 2. The tangent lines are the linear factors of

$$A_{2,0}(0,0)X^2 + 2A_{1,1}(0,0)XY + A_{0,2}(0,0)Y^2 = -2X^2 + 2Y^2 = 2(Y-X)(Y+X).$$

These are the lines $Y = X$ and $Y = -X$. Hence the origin is an ordinary singularity.

A basic inequality on multiplicity is this: for a projective irreducible curve C,

$$\sum_p mult_p(C)(mult_p(C) - 1) \leq (d-1)(d-2). \tag{6}$$

or if C is a reduced curve, then

$$\sum_p mult_p(C)(mult_p(C) - 1) \leq d(d-1). \tag{7}$$

The proof goes as follows...

§3. Curve Transformation

A transformation is an invertible function between two spaces, S and S'. In the following, assume $S = S' = K^2$ (e.g., $K = \mathbb{R}$ or $K = \mathbb{C}$). Call S the (X, Y)-space and S' the (U, V)-space, where X, Y, U, V are indeterminates. This naming convention allows us to distinguish points in S from S': a point $(a, b) \in K^2$ will be called an (X, Y)-point when viewed as a member of S, and a (U, V)-point when viewed as a member of S'.

A transformation takes each point $(a, b) \in S$ to some point $(a', b') \in S' \cup \{\infty\}$ using some rule. We need ∞ in the range in case the transformation is undefined at (a, b). The simplest transformations are linear transformations. More generally, consider transformations specified by a pair of rational functions, $U(X, Y), V(X, Y) \in K(X, Y)$. Thus,

$$T : (a, b) \mapsto (a', b') = (U(a, b), V(a, b)).$$

Such transformation T is called a **birational map** if it is invertible and the inverse T^{-1} is also given by a pair of rational functions, $X(U, V), Y(U, V) \in K(U, V)$. Thus,

$$T^{-1} : (a', b') \mapsto (a, b) = (X(a', b'), Y(a', b'))$$

where

$$X(U(a, b), V(a, b)) = a, \qquad Y(U(a, b), V(a, b)) = b \tag{8}$$

provided each function is defined at their arguments.

Suppose $F(X, Y) = 0$ defines a curve C in S. Applying the birational transformation to each point (a, b) on $F(X, Y) = 0$, we obtain the collection of points $(a', b') \subseteq S'$. Do these points define the locus of some curve in S'? The answer is yes: (a', b') are points on the curve $C' : F'(U, V) = 0$ where

$$F'(U, V) = F(X(U, V), Y(U, V)).$$

To see this, if $(a', b') \in T(\texttt{Zero}(F))$, then

$$\begin{aligned} F'(a', b') &= F(X(a', b'), Y(a', b')) \\ &= F(X(U(a, b), V(a, b)), Y(U(a, b), V(a, b))) \\ &= F(a, b) \\ &= 0 \end{aligned}$$

i.e., T maps each point in $\texttt{Zero}(F)$ into a point in $\texttt{Zero}(F')$. Conversely, Moreover, every point (a', b') on $F' = 0$ is the image of a point $(X(a', b'), Y(a', b'))$ on $F = 0$. We say that the map $(U(X, Y), V(X, Y))$ **transforms** the curve $F = 0$ into the $F' = 0$. Two curves are **birationally equivalent** if there is a birational map that transforms one to the other.

Example: consider the nodal cubic $F(X, Y) = Y^2 - X^2 - X^3$. Let $U(X, Y) = X, V(X, Y) = Y/X$. Thus, $V(a, b)$ is undefined iff $a = 0$. This specifies a birational map since it is invertible and its inverse transform is $X(U, V) = U$ and $Y(U, V) = UV$. Note that

$$\begin{aligned} F'(U, V) &= F(X(U, V), Y(U, V)) \\ &= F(U, UV) \\ &= (UV)^2 - U^2 - U^3 \\ &= U^2(V^2 - 1 - U). \end{aligned}$$

The curve $F' = 0$ is the union of the line $U = 0$ and the parabola $U = V^2 - 1$. Thus, by a suitable transformation, we "reveal" the true nature of the original curve; it is a revelation in the sense that we have transformed it into familiar or "canonical" curves we already know. This idea is expressed in the theory of canonical forms for curves.

§4. Curve Parametrization

One extremely useful property for curves is parametrization. Sendra [5] is a reference for this.

A real affine curve $F(X,Y) = 0$ is said to be **parametrized** by the functions $X = X(t)$ and $Y = Y(t)$ if (1) for almost all $t \in \mathbb{R}$, $F(X(t), Y(t)) = 0$. and (2) for almost all points (x, y) in the locus, there exists some $t \in \mathbb{R}$ such that $(x, y) = (X(t), Y(t))$. Here "almost all" means with finitely many exceptions. In case $X(T), Y(T) \in \mathbb{R}(T)$ for some indeterminate T, we say $(X(T), Y(T))$ is a **rational parameterization** of F. Only irreducible curves are rationally parametrizable.

Equivalent formulations (see Sendra).

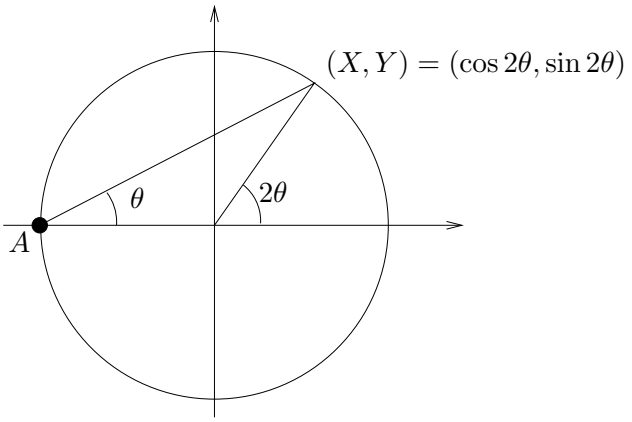


Figure 2: Parametrization of the unit circle

Example: the parabola $F(X,Y) = Y - X^2$ is rationally parametrizable because if $X(t) = t$ and $Y(t) = t^2$ then $F(X(t), Y(t)) = 0$ holds for all t. In fact, we see that the parabola is even **polynomially parametrizable**. On the other hand, the circle $X^2 + Y^2 = 1$ cannot be polynomially parametrized. It has the rational parametrization

$$X(t) = \frac{1 - 2t^2}{1 + t^2}, Y(t) = \frac{2t}{1 + t^2}.$$

Referring to Figure 2, this parametrization is easily understood as followings. Originating from the point $A = (-1, 0)$, we have rays $R(\theta)$ making angle θ with the positive X-axis. For $\theta \in (-\pi/2, \pi/2]$, the ray $R(\theta)$ intersects the circle at a unique point $(\cos 2\theta, \sin 2\theta)$. Recall the half-angle formulas, we have $\cos 2\theta = \frac{1-2t^2}{1+t^2}$ and $\sin 2\theta = \frac{2t}{1+t^2}$ where $t = \tan \theta$.

Not all curves are rational. E.g., for instance the cubic $Y^2 - X(X^2 - 1) = 0$ has no singular points, and hence is not rational.

On the other hand, at any non-singular point (x, y) of F, the curve $F = 0$ can be locally parameterized in the following sense:

Application. Parametrizable curves simplify many computational tasks. For instance, consider the problem of **curve display**: given $F(X,Y)$, a rectangle $R \subseteq \mathbb{R}^2$, and $\varepsilon > 0$, to produce a finite set $T \subseteq \mathbb{R}^2$ that is an **ε-cover** for the curve $F(X,Y) = 0$ restricted to R. I.e., for every point $p \in R \cap \texttt{Zero}(F)$, every point of the curve either lies outside R or else lies within distance ε from some point of T. To solve this problem, we proceed as follows: assume that we can determine an interval $[t_0, t_1]$ such that for t outside this interval, $(X(t), Y(t))$ lies outside R. Write $p(t)$ for the point $(X(t_0), Y(t_0))$. Initially, assume that

the points $p(t_0)$ and $p(t_1)$ are already added to our set T. In general, we have an interval $[t', t'']$ and $p(t'), p(t'') \in T$. We perform a **distance test** on an interval $[t', t'']$: that is, we "certify" whether for all t in $[t', t'']$, $p(t)$ is within ε distance from $p(t')$ or $p(t'')$. If this test succeeds, we are done. If not, we add into T the point $p(t''')$ where $t''' = (t' + t'')/2$, and recursively perform distance tests for the intervals $[t', t''']$ and $[t''', t'']$. This test is called a **certification**. Alternatively, it is called a "conservative predicate" or a "one-sided predicate". With reasonable implementation, this procedure should terminate.

1. So we would like to know when a curve is rationally parametrizable. We say $F(X, Y)$ is said to be **X-regular** if X^d appears in $F(X, Y)$ and $d = \deg F$. Y-regular is similarly defined. Note that the equation $F(X, Y)$ is not Y-regular. For any degree 2 curve $F = 0$ that is not regular in one of its variables, we can apply the following general parametrization: without loss of generality, say F is a second degree equation that is not Y-regular. Then $F(X, Y) = Y \cdot (aX + b) + (cX^2 + dX + e)$ for some $a, b, c, d, e \in \mathbb{Z}$. Then we choose the rational parametrization

$$X = t, \qquad Y = (ct^2 + dt + e)/(at + b).$$

2. What if $F(X, Y)$ is regular in both X and Y? In this case, we would like to use a "shear transformation", that is, replace X by $X + \alpha Y$ for some $\alpha \in \mathbb{R}$. We just have to see what happens to the homogeneous part $F_d(X, Y)$ where $d = \deg F$. $F_d(X, Y) = aX^2 + bXY + cY^2$. Then $F_d(X + \alpha Y, Y) = Y^2(a\alpha^2 + b\alpha + c) + lot(Y)$. So we only need to choose α so that $a\alpha^2 + b\alpha + c = 0$. This amounts to solving a quadratic equation.

3. What if α turns out to be complex? This happens when $b^2 < 4ac$. For instance, for the circle $F(X, Y) = X^2 + Y^2 - 1$, we get $\alpha^2 + 1 = 0$ or $\alpha = \pm\sqrt{-1}$. This forces us to consider a more general transformation than shears: linear fractional transformations.

UNCLEAR FROM ACCOUNT OF ABHYANKAR...

Remark: shears and linear fractional transformations are special cases of the transformation

$$(X', Y', Z') = (X, Y, Z) \cdot A + (a, b, c)$$

where A is an invertible 3×3 matrix and $(a, b, c) \in \mathbb{R}^3$.

The above outline can be turned into an algorithm for computing a rational parametrization of a conic or a cubic: we leave this

§5. Linear Systems of Curves

We consider the set of all curves of a fixed degree $n \geq 1$. A bivariate polynomial $A(X, Y) \in K[X, Y]$ of degree n has up to $\binom{n+2}{2}$ coefficients in some field $K \subseteq \mathbb{C}$. As these coefficients are distinguished up to multiplication by a non-zero constant, we can view such a curve C as a point in N-dimensional projective space $\mathbb{P}^N(K)$ where

$$N = N(n) = \binom{n+2}{2} - 1 = n(n+3)/2.$$

For instance, $N(2) = 5$ and $N(3) = 9$. Thus the space of conics and cubics are 5- and 9-dimensional, respectively.

Corresponding to a curve $C \in \mathbb{P}^N(K)$, let $A(X, Y) \in K[X, Y]$ be its defining polynomial. There is also the homogeneous polynomial $\widehat{A}(X, Y, W] \in K[X, Y, W]$. For $q \in \mathbb{P}^2(\mathbb{C})$, we will write "$C(q)$" to denote the evaluation of the defining polynomial $\widehat{A}(X, Y, W)$ at the point q. The locus $L(C)$ of C is then

$$L(C) = \{q \in \mathbb{P}^2(\mathbb{C}) : C(q) = 0\}.$$

If λ is a variable λ then the linear polynomial

$$P(\lambda) = \lambda C + (1-\lambda)C'$$

is called the **pencil** generated by C, C'. For each $a \in K$, $P(a)$ represents a curve. We usually prefer to write P_a instead of $P(a)$. The family is $\{P_a : a \in K\}$ is called a **curve pencil**. The pencil is **trivial** when[1] $C = C'$, viewed as projective points in $\mathbb{P}^N(K)$. Alternatively, triviality happens exactly when $P(\lambda)$ can be factored as $(a + b\lambda)C$ for some $a, b \in K$. Pencils are useful for studying the intersection properties of the curves. Call $L(C) \cap L(C')$ the **base** of this pencil. The reason is the following observation:

Lemma 2 *Let $P(\lambda)$ be a non-trivial pencil. Let $a, b \in K$, $a \neq b$.*
(i) We have $P_a \neq P_b$. Thus, there are infinitely many curves in the curve pencil.
(ii) Also $L(P_a) \cap L(P_b)$ is the base of the pencil $P(\lambda)$.

Proof.
(i) If $P_a = P_b$ this means $aC + (1-a)C' = bC + (1-b)C'$. This means $C = C'$, contradiction.
(ii) If $q \in L(P_a) \cap L(P_b)$ then $aC(q) + (1-a)C'(q) = 0 = bC(q) + (1-b)C'(q)$. This easily implies $C(q) = C'(q) = 0$. **Q.E.D.**

Here is an interesting theorem which can be proved using pencils:

Theorem 3 (Generalized Pascal Theorem) *et C, C' be projective plane curves of degree n. If C, C' intersect in n^2 distinct points and there is another curve D of degree $m \leq n$ that passes through nm of these points, then there is third curve D' of degree $n - m$ that is disjoint from D and passes through the remaining $n(n-m)$ of these points.*

Proof. Let I be the set of mn points in $L(C) \cap L(C')$. Choose any point q on $L(D) \setminus I$. We claim that q lies in some curve $P_a = aC + (1-a)C'$ of the pencil $P(\lambda) = \lambda C + (1-\lambda)C'$. This amounts to choosing a so that $aC(q) + (1-a)C'(q) = 0$. By the previous lemma $L(P_a)$ contains the I. Thus $L(P_a) \cap L(D) \geq mn + 1$ since we also have $q \in L(P_a) \cap D$. By Bezout's theorem, P_a and D must share a common component (otherwise their intersection must have at most mn points). Since D is assumed to be irreducible, we must have $L(D) \subseteq L(P_a)$, or viewing D and P_a as polynomials,

$$D | P_a.$$

Alternatively, there is another curve E of degree $n - m$ such that $P_a = DE$. Clearly, $L(E)$ contains all the remaining $n(n-m)$ points of I. **Q.E.D.**

A special case of this theorem is the following: let $(p_0, q_0, p_1, q_1, p_2, q_2)$ be any sequence of 6 distinct points on an ellipse, and consider the three lines $L_i : \overline{p_i q_i}$, $i = 1, 2, 3$. **Pascal's theorem** says that the three points

$$r_0 = L_1 \cap L_2, r_1 = L_2 \cap L_0, r_2 = L_0 \cap L_1$$

are collinear. To deduce this from the generalized Pascal's theorem, we consider the cubic curve C comprising the three lines $\overline{p_i q_i}$ for $i = 0, 1, 2$ and the cubic curve C' comprising the three lines $\overline{q_i p_{i+1}}$. They intersect in the 9 points $\{p_i, q_i, r_i : i = 0, 1, 2\}$. Now, 6 of these points lies on a conic. Hence the remaining three lies on a line.

Exercises

[1] When C, C' are represented by two sets of homogeneous coordinates, then the equality "$C = C'$" amounts to $aC = C'$ for some non-zero $a \in K$.

Exercise 5.1: Any second degree curve is birationally equivalent to a line. Conclude that any two second degree curve is birationally equivalent. ♢

Exercise 5.2: Give another concrete application of the Generalized Pascal theorem. ♢

Exercise 5.3: (Abhyankar) Let C be a rationally parametrizable curve. Then C is polynomially parametrizable iff it has one place at infinity. NOTE: "places" are generalizations of points. ♢

End Exercises

§6. Intersection of Curves

We have already defined the multiplicity of a point P on a curve $f = 0$, $\text{mult}_P(f)$.

We now define the multiplicity $\text{mult}_P(f, g)$ of a point $P = (P_x, P_y)$ in the intersection of two curves $f = 0, g = 0$. There are several ways to do this, but computationally, the best approach is to use the resultant formulation: the naive idea is to simply define $\text{mult}_P(f, g)$ as the multiplicity of P_x as a root of $R(X) = \texttt{res}_Y(f, g)$. The main problem is that we have not used the other component P_y in this definition; this definition may give a larger multiplicity than we want when there is another point Q with the same X-coordinate as P. To get around this, we consider the (horizontal) shear transform,

$$(X, Y) \mapsto (X + tY, Y).$$

Let $R_t(X) = \texttt{res}_Y(f(X + tY, Y), g(X + tY, Y))$ and let $m(t)$ is the multiplicity of 0 as a root of $R_t(X)$. Let $P = (0, 0)$ be the origin. If $f(0, 0) = g(0, 0) = 0$, then the multiplicity of intersection $\text{mult}_P(f, g)$ is defined to be $\min_t m(t)$.

Theorem 4 *he definition of* $\text{mult}_P(f, g)$ *is invariant.*

Theorem 5 *(Bezout) Two projectives curves* f, g *of degrees* m *and* n *intersect in exactly* mn *points, counting multiplicities. More precisely,* $\sum_P \text{mult}_P(f, g) = mn$.

§7. Rational Curves

Do the basic idea about base points of curves and parametrizability of curves...

§8. Properties of Curves in Parametric Form

We now describe the basic properties of curves which are given in the parametric form.

REFERENCES: The following is from Myung-Soo's paper with In-Kwon Lee [4], and also Notes of Jan Stevens at Chalmers (see /prob/curves).

1. Difference between parametric representation and implicit representation:

E.g. The tangent to $A(x, y) = 0$ at point $p(x, y)$ is ... But the tangent to $C(t)$ at point $p = C(t_0)$ is...

The following is from Myung-Soo's paper with In-Kwong Lee (from postech):

Lemma 6 *Let $C(t) = (c_1(t), c_2(t))$ be a parametric curve and $p = C(t)$.*
(i) p is a non-self-intersecting singular point iff $c_1'(t) = c_2'(t) = 0$
(ii) p is a non-singular x-extreme point iff $c_2'(t) = 0$ and $c_1'(t) \neq 0$.
(iii) p is a non-singular y-extreme point iff $c_2'(t) \neq 0$ and $c_1'(t) = 0$.
(iv) p is an inflexion point iff $c_1'(t) \cdot c_2''(t) - c_2'(t) \cdot c_1''(t) = 0$
(v) convex...
(vi) concave...

Lemma 7 *Let C be the implicit curve $f(x, y) = 0$ and $f(p) = 0$.*
(i) p is a singular point iff $f = f_x = f_y = 0$
(ii) p is a non-singular x-extreme point iff $f = f_x = 0$ and $f_y \neq 0$
(iii) p is a non-singular y-extreme point iff $f = f_y = 0$ and $f_x \neq 0$
(iv) p is an inflexion point iff $f = f_{xx} \cdot f_y^2 - 2f_{xy} \cdot f_x f_y + f_{yy} \cdot f_x^2 = 0$.
(v) If C has no inflexion point then C is convex iff $f_{xx} \cdot f_y^2 - 2f_{xy} \cdot f_x f_y + f_{yy} \cdot f_x^2 > 0$.
(vi) If C has no inflexion point then C is concave iff $f_{xx} \cdot f_y^2 - 2f_{xy} \cdot f_x f_y + f_{yy} \cdot f_x^2 < 0$.

Tangent, Hessian of a curve Intersection Multiplicity: we define this so that Bezout's theorem is a trivial consequence of the definition!

Following Jan Stevens:

Assume $F(X, Y, Z), G(X, Y, Z)$ have no common component. Let $R(Y, Z)$ be their resultant w.r.t. X. This is non-zero and homogeneous of degree mn.

Choose a point P not on $F = 0$ or $G = 0$, and outside all lines connecting two common zeros of F and G. Taking new coords, $P = (1 : 0 : 0)$.

Let $Q = (\alpha : \beta : \gamma)$ be a common zero of F, G. By construction, this is the only point on the line $\gamma Y - \beta Z = 0$. Define the **intersection multiplicity** $\text{mult}_Q(F, G)$ of F and G at Q to be the multiplicity of $(\beta : \gamma)$ as a zero of $R(Y, Z)$.

THEOREM: Bezout is true, trivially (?)

Clean up operation:

Tangent, Hessian of a curve Let $F_d(X_0, X_1, X_2)$ be the equation of a curve C and $P = (p_0 : p_1 : p_2)$ be a point on C. The **tangent line** at P is given by the equation

$$\frac{\partial F}{\partial X_0}(P)X_0 + \frac{\partial F}{\partial X_1}(P)X_1 + \frac{\partial F}{\partial X_2}(P)X_2 = 0, \tag{9}$$

provided $\frac{\partial F}{\partial X_i}(P) \neq 0$ for some $i = 0, 1, 2$. I.e., provided P is not a singular point.

There are several ways to understand (9):

- In affine coordinates, $f(x_1, x_2) = F(1, X_1, X_2)$, and $p = (p_1, p_2)$ this equation becomes $\frac{\partial f}{\partial x_1}(p)(x_1 - p_1) + \frac{\partial f}{\partial x_2}(p)(x_2 - p_2) = 0$.

  By Euler's formula, etc (see Jan Steven Notes)

Let L be a tangent line L to a curve C at the point P. We call L an **inflexional tangent** (or a **flex** for short), and P an **inflexion point**, if the intersection multiplicity of L and C at P is at least 3.

Definition: The **Hessian** H_F of F is the curve with equation

$$\det\left(\frac{\partial^2 F}{\partial X_i \partial X_j}\right) = 0.$$

§9. Jacobi Curves and Generalizations

The following is adapted from Nicola Wolpert's thesis.

In this section, assume $f, g \in \mathbb{Z}[X, Y]$ are both normal with respect to X and Y. Suppose $p = (a, b) \in \texttt{ZERO}(f, g)$ where p is not a singularity of f or g. Let $B = [x_{\min}, x_{max}] \times [y_{\min}, y_{\max}]$ be an "isolating box" meaning that $[x_{\min}, x_{\max}]$ is an isolating interval for $\texttt{res}_Y(f, g)$ and $[y_{\min}, y_{\max}]$ is an isolating interval for $\texttt{res}_X(f, g)$. We are interested in methods for detecting (f, g) intersection inside B. Note that such a procedure will output either YES or NO, since there can be at most one intersection inside B.

[FIGURE: isolating box]

In case p is a transversal intersection, there is a simple "box hitting" method to detect p. So assume $p = (a, b)$ is a tangential intersection, i.e., the Jacobi curve

$$h = g_X f_Y - g_Y f_X \tag{10}$$

passes through p. Now, if h intersects f (and hence g) transversally, we can reduce the problem to the detection of (f, h) intersection inside B, and also of (g, h) intersection inside B.

Exercise: device a detection method in case where, if there is an intersection point p, then either the (f, g) or (f, h) intersection is transversal.

[EXPAND: Simple Hitting Box Method]

But if h and f intersect at p tangentially, we can try to see if the "next" Jacobi curve $h_X f_Y - h_Y f_X$ passes through p transversally. This process can be repeated several time. Our goal is to how to do this decisively.

From now on, assume that

$$f_Y(p) \neq 0. \tag{11}$$

If this is false, we can carry out the analogous argument with $f_X(p) \neq 0$ (recall that p is non-singular point of $f = 0$.) We also assume

$$h(p) = 0, \tag{12}$$

i.e., f and g intersect tangentially at p. Then we claim that

$$g_Y(p) \neq 0.$$

To see this, suppose $g_Y(p) = 0$. Then $g_X(p) \neq 0$. But $0 = h(p) = g_X(p) f_Y(p)$ implies $f_Y(p) = 0$, contradiction. Wow inductively define $h_0 = g$ and for $i \geq 0$,

$$h_{i+1} = (h_i)_X - (h_i)_Y \frac{f_X}{f_Y}. \tag{13}$$

Note that $h_i(p)$ is well-defined since $f_Y(p) \neq 0$. In fact, $f_Y h_1$ is the Jacobi curve h, and we also have $h_0(p) = 0$ and $h_1(p) = 0$.

Since $f_Y(p) g_Y(p) \neq 0$, by the Implicit Function Theorem, there are analytic functions $F(X), G(X)$ such that in a neighborhood of $X = a$, we have

$$f(X, F(X)) = 0, \qquad g(X, G(X)) = 0.$$

By differentiating $f(X, F(X))$, we get

$$f_X(X, F(X)) + f_Y(X, F(X)) F'(X) = 0$$

and hence

$$F'(X) = -\frac{f_X(X, F(X))}{f_Y(X, F(X))}$$

Similarly, $G'(X) = -\frac{g_X(X,G(X))}{g_Y(X,G(X))}$.

To understand the behavior of the curves $h_i(X,Y)$ at the point (a,b), we also define for $i \geq 0$,

$$H_i(X) = h_i(X, F(X)). \tag{14}$$

We already know that $H_0(a) = 0$ and $H_1(a) = 0$. Differentiating with respect to X,

$$\begin{aligned}
(H_i(X))' &= (h_i)_X(X,F(X)) + (h_i)_Y(X,F(X))F'(X) \\
&= (h_i)_X(X,F(X)) - (h_i)_Y(X,F(X))\frac{f_X(X,F(X))}{f_Y(X,F(X))} \\
&= h_{i+1}(X,F(X)) \\
&= H_{i+1}(X).
\end{aligned}$$

References

[1] S. S. Abhyankar. *Algebraic Geometry for Scientists and Engineers*. Americal Mathematical Society, Providence, Rhode Island, 1990.

[2] E. Brieskorn and H. Knörrer. *Plane Algebraic Curves*. Birkhäuser Verlag, Berlin, 1986.

[3] J. Bruce and P. Giblin. *Curves and Singularities*. Cambridge University Press, Cambridge, second edition, 1992.

[4] M.-S. Kim and I.-K. Lee. Gaussian approximations of objects bounded by algebraic curves. In *Proc. 1990 IEEE Int'l, Conf. on Robotics and Automation*, pages 322–326, 1990. May 13–18, Cincinnati ,U.S.A.

[5] J. R. Sendra. Rational curves and surfaces: Algorithms and some applications. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 3. World Scientific Publishing Co., Singapore, 2004. To appear.

[6] R. J. Walker. *Algebraic Curves*. Springer Verlag, Berlin-New York, 1978.

[7] W. Walker. *Algebraic Curves*. The Benjamin/Cummings Pub.Co., Inc, Reading, Massachusetts, 1969.

Lecture 21
SURFACES

This chapter focus on two important and practical classes of surfaces: quadric surfaces and NURB surfaces.

§1. Quadric Surfaces

The zero set of a polynomial $P(\mathbf{X}) = P(X_1, \ldots, X_n) \in \mathbb{Q}[X_1, \ldots, X_n]$ is a **hypersurface** in n-dimensional affine space (either real or complex). When $P(\mathbf{X})$ is homogeneous, it is called a **form** and defines a hypersurface in $(n-1)$-dimensional projective space. When $\deg(P) = 2$, the polynomial is **quadratic** and the corresponding hypersurface is **quadric**. We shall reserve the term "surface" for 3-dimensional affine or projective hypersurfaces. When the context is clear, we will say "quadric" for quadric surfaces. Quadric surfaces are thus the simplest nonlinear non-planar geometry we can study.

The intersection of two quadric surfaces is a key problem in computer graphics and geometric modeling. Levin [7] initiated a general approach of reducing such intersection to finding a ruled quadric in the pencil of the input quadrics. Many recent papers have expanded upon this work, e.g., [6, 10, 4]. A general reference is Wenping's survey [5, Chap. 31].

Consider the quadratic polynomial

$$P(X, Y, Z) = aX^2 + bY^2 + cZ^2 + 2fXY + 2gYZ + 2hZX + 2pX + 2qY + 2rZ + d.$$

This polynomial can be written as a matrix product:

$$P(X, Y, Z) = \mathbf{X}^T \cdot A \cdot \mathbf{X} = 0 \tag{1}$$

where $\mathbf{X} = (X, Y, Z, 1)^T$ and A is the symmetric 4×4 matrix,

$$A = \left[\begin{array}{ccc|c} a & f & h & p \\ f & b & g & q \\ h & g & c & r \\ \hline p & q & r & d \end{array}\right]. \tag{2}$$

The principal 3×3 submatrix of A which is indicated in (2) is conventionally denoted A_u, and called the **upper submatrix of** A.

NOTATION: For any $n \times n$ matrix A, and $\mathbf{X} = (X_1, \ldots, X_n)^T$, let[1] $Q_A = Q_A(\mathbf{X})$ denote the **quadratic form**

$$Q_A(\mathbf{X}) = \mathbf{X}^T A \mathbf{X} \tag{3}$$

associated with A. For $n = 4$, we usually write $\mathbf{X} = (X, Y, Z, W)^T$. The notation in (3) is also used in the affine case, in which case $\mathbf{X} = (X_1, \ldots, X_{n-1}, 1)^T$ and $Q_A(\mathbf{X}) = Q_A(X_1, \ldots, X_{n-1})$. The corresponding quadric surface (affine or projective) is denoted $Q_A(\mathbf{X}) = 0$.

Classification of Quadrics in $\mathbb{A}^n(\mathbb{R})$. Classification of quadric surfaces is a highly classical subject (see [2]). There are four versions of this problem, depending on whether we consider the real $\mathbb{R}$ or complex $\mathbb{C}$ field, and whether we consider the affine $\mathbb{A}^n$ or projective $\mathbb{P}^n$ space. We focus on the real affine case. Our approach follows Burington [3] For instance, for $n = 2$, there is the well-known classification of conics. The classification for $n = 3$ is shown in Table 1. The main thrust of this section is to prove the correctness of this table.

Various invariants of the matrix A can be used in making such classifications. Some candidates to consider are:

[1] More generally, if $\mathbf{Y} = (Y_1, \ldots, Y_n)$ then $Q_A(\mathbf{X}, \mathbf{Y}) = \mathbf{X}^T A \mathbf{Y}$ is a bilinear form.

ρ	rank of A
ρ_u	rank of A_u
Δ	determinant of A
Δ_u	determinant of A_u
$\mathtt{sign}(\Delta)$	sign of Δ
$\mathtt{sign}(\Delta_u)$	sign of Δ_u

Δ and Δ_u are called the **discriminant** and **subdiscriminant** of the quadric. But there are other possibilities. For instance Levin [7] uses an oft-cited table from Breyer [1, p. 210–211], based on a different set of invariants: $\rho, \rho_u, \mathtt{sign}(\Delta), k$. Here $k = 1$ if the non-zero eigenvalues of A_u have the same sign, and $k = 0$ otherwise. The exact constitution of such a classification varies, depending on which "degenerate" cases are omitted (published classifications have 17, 18 or 19 classes, and if the imaginary ones are omitted, correspondingly smaller number is obtained). Our classification in Table 1 do not omit degenerate classes and has 20 classes.

The notion of invariance depends on the choice of the class of transformations, and whether we consider projective or affine space, and whether we consider the complex or real field. The most general transformation considered in this chapter are those defined by an arbitrary non-singular $n \times n$ real matrix M. These are called (real) **projective transformations**, and they transform space under the action

$$\mathbf{X} \mapsto M\mathbf{X}.$$

We also say that they **transform polynomials** under the action

$$P(\mathbf{X}) \mapsto P(M\mathbf{X}). \tag{4}$$

Suppose a quadratic polynomial $Q_A(\mathbf{X}) = \mathbf{X}^T A\mathbf{X}$ is transformed to $Q_B(\mathbf{X}) = Q_A(M\mathbf{X})$, for some matrix B. Thus $\mathbf{X}^T B\mathbf{X} = (M\mathbf{X})^T A(M\mathbf{X}) = \mathbf{X}^T(M^T AM)\mathbf{X}$. Hence

$$B = M^T AM. \tag{5}$$

Using the same terminology, we say that M **transforms** A to $M^T AM$. Two matrices A and B related as in (5) via some non-singular matrix M are said to be **congruent**, or $A \mapsto M^T AM$ is a **congruence transformation**. Further, when $A = A^T$, we have

$$(M^T AM)^T = M^T A^T M = M^T AM$$

This proves that symmetric matrices are closed under congruence transformation. It is easy to see that matrices, including symmetric ones, are thereby partitioned into congruence classes. If M_i transforms A_i to A_{i+1} for $i \geq 1$, then $M_1 M_2 \cdots M_i$ transforms A_1 to A_{i+1}. Thus the composition of transformations amounts to matrix multiplication.

The ranks ρ, ρ_u are clearly invariants of A under congruence transformations. Also, $\mathtt{sign}(\Delta)$ is an invariant since $\det(A) = \det(B)\det(M^T M) = \det(B)\det(M)^2$. But $\Delta = \det(A)$ is not invariant unless $|\det(M)| = 1$.

The (real) **affine transformations**, are those projective transformation represented by real matrices M of the form

$$M = \left[\begin{array}{c|c} M_u & \mathbf{t} \\ \hline \mathbf{0} & d \end{array}\right] \tag{6}$$

where $d \neq 0$, M_u is any invertible matrix and $\mathbf{t}$ is a $(n-1)$-column vector. Two matrices A, B are **affinely congruent** if there is an affine transformation M such that $A = M^T BM$. Affine transformations treats the last (homogenization) coordinate as special. The **rigid transformation** are affine transformations with $\det(M_u) = \pm 1$ and $d = \pm 1$. Rigid transformations preserve distances between pairs of points, and are exemplified by rotation, translation or reflection. Thus Δ and Δ_u are invariants under rigid transformations.

	Name	Canonical Equation	σ	σ_u	Notes
	Nonsingular Surfaces				
1	Imaginary Ellipsoid	$X^2+Y^2+Z^2=-1$	(4,0)	(3,0)	(**)
2	Ellipsoid	$X^2+Y^2+Z^2=1$	(3,1)	(3,0)	R_0, C
3	Hyperboloid of 2 sheet	$X^2+Y^2-Z^2=-1$	(3,1)	(2,1)	R_0, C
4	Hyperboloid of 1 sheet	$X^2+Y^2-Z^2=1$	(2,2)	(2,1)	R_2, C
5	Elliptic Paraboloid	$X^2+Y^2-Z=0$	(3,1)	(2,0)	R_0
6	Hyperbolic Paraboloid	$X^2-Y^2-Z=0$	(2,2)	(1,1)	R_2, L
	Singular Nonplanar Surfaces				
7	Imag. Elliptic Cone (Point)	$X^2+Y^2+Z^2=0$	(3,0)	(3,0)	(*)
8	Elliptic Cone	$X^2+Y^2-Z^2=0$	(2,1)	(2,1)	R_1
9	Elliptic Cylinder	$X^2+Y^2=1$	(2,1)	(2,0)	R_1
10	Imag. Elliptic Cylinder	$X^2+Y^2=-1$	(3,0)	(2,0)	(**)
11	Hyperbolic Cylinder	$X^2-Y^2=-1$	(2,1)	(1,1)	R_1, L
12	Parabolic Cylinder	$X^2=Z$	(2,1)	(1,0)	R_1, L
	Singular Planar Surfaces				
13	Intersecting Planes	$X^2-Y^2=0$	(1,1)	(1,1)	
14	Intersect. Imag. Planes (Line)	$X^2+Y^2=0$	(2,0)	(2,0)	(*)
15	Parallel Planes	$X^2-1=0$	(1,1)	(1,0)	
16	Imag. Parallel Planes	$X^2+1=0$	(2,0)	(1,0)	(**)
17	Single Plane	$X=0$	(1,1)	(0,0)	
18	Coincident Planes	$X^2=0$	(1,0)	(1,0)	
19	Invalid	$1=0$	(1,0)	(0,0)	(**)
20	Trivial	$0=0$	(0,0)	(0,0)	(*)

Table 1: Classification of Quadric Surfaces in $\mathbb{A}^3(\mathbb{R})$

We emphasize that Table 1 is a classification for *affine* and *real* space. For instance, the elliptic cylinder and elliptic cone has, as special case, the usual cylinder and cone. In projective space, there would be no difference between a cone and a cylinder (e.g., Rows 8 and 9 would be projectively the same). Similarly, there would be no distinction between an imaginary and a real ellipsoid in complex space (i.e., Rows 1 and 2 would collapse).

Under "Notes" (last column) in Table 1, we indicate by $(**)$ those surfaces with no real components, and by $(*)$ those "surfaces" that degenerate to a point or a line or fills the entire space. It is convenient[2] to call those quadrics in Table 1 that are starred, either (*) or (**), as **improper**; otherwise they are **proper**. We indicate by R_1 the singly ruled quadric surfaces, and by R_2 the doubly ruled quadric surfaces. Also R_0 is for the non-ruled quadric surfaces. The planar surfaces are automatically ruled, so we omit any indication. The letter L are the parameterization surfaces used in Levin's algorithm. The letter C indicate the so-called **central quadrics**.

Non-singular Quadrics. To gain some insight into Table 1, we consider the non-singular quadrics. In general, a surface $P(\mathbf{X}) = P(X_1, \ldots, X_n) = 0$ is **singular** if it contains a singular point $\mathbf{p}$, i.e., one in which $P(\mathbf{p}) = 0$ and $(\partial P/\partial X_i)(\mathbf{p}) = 0$ for $i = 1, \ldots, n$. Otherwise the surface is **non-singular**. Non-singular quadrics are those whose matrix A have rank 4 (Exercise). In Table 1, these are represented by the first 6 rows. These can further be grouped into three pairs: 2 ellipsoids, 2 hyperboloids and 2 paraboloids. In the following discussion, we assume the reader is familiar with the basic properties of conics (ellipses, hyperbolas and parabolas).

1. There is nothing to say for the imaginary ellipsoids. The real ellipsoids is possibly the easiest quadric to understand, perhaps because it is the only bounded surface in our shortlist. Ellipsoids are basically squashed spheres. But in our canonical equation $X^2 + Y^2 + Z^2 = 1$, it is just a regular sphere.

2. The hyperboloids (either 1- or 2-sheeted) have equations $X^2 + Y^2 - Z^2 = \pm 1$. The Z-axis is an axis of symmetry in this canonical form: for any value of $Z = z_0$, the (X, Y)-values lies in the circle[3] $X^2 + Y^2 = z_0^2 \pm 1$. In $\mathbb{A}^3(\mathbb{R})$, the circle $X^2 + Y^2 = Z^2 - 1$ would be purely imaginary for $|Z| < 1$. Hence this quadric surface is clearly divided into two connected components (separated by the plane $Z = 0$). Thus

[2]This is not standard terminology. See, e.g., [5, p.778, Chap. 31].

[3]In general, we would have an ellipse $(aX)^2 + (bY)^2 = (cz_0)^2 \pm 1$, and hence these hyperboloids are also known as *elliptic hyperboloids*.

it is "two sheeted". On the other hand, $X^2 + Y^2 = Z^2 + 1$ has solutions for all values of Z and it is clearly seen to be a connected set, or "one sheeted".

3. Finally, consider the paraboloids: in the elliptic case $X^2 + Y^2 = Z$ is clearly confined to the upper half-space $Z \geq 0$. In the hyperbolic case, we see a hyperbola $X^2 - Y^2 = z_0$ in every plane $Z = z_0$. When $z_0 > 0$, the hyperbola lies in a pair of opposite quadrants defined by the pair of lines $X + Y = 0$ and $X - Y = 0$. When $z_0 < 0$, the hyperbola lies in the other pair of opposite quadrants. At $z_0 = 0$, the hyperbola degenerates into the pair of lines $X + Y = 0, X - Y = 0$. It is easy to see a saddle point at the origin.

The Inertia Invariant. The reader will note that Table 1 did not use the invariants $\rho, \rho_u, \mathtt{sign}(\Delta), \mathtt{sign}(\Delta_u)$. These information are implicit in σ, σ_u, defined to be the **inertia** of A and A_u. We now explain this concept.

Let A be any real symmetric matrix A of order n. It is not hard to see that rank and sign of determinants are invariants of the congruence relation. But they are only partial invariants in that they do not fully characterize congruence. Instead, we let us look at the eigenvalues of A. The eigenvalues of A are all real. It is easy to see that these eigenvalues (like the determinant) are not preserved by congruence transformations. But the number of positive and negative eigenvalues turns out to be invariant. This is the substance of Sylvester's law of inertia.

Let $\sigma^+ = \sigma^+(A)$ and $\sigma^- = \sigma^-(A)$ denote the numbers of positive and negative eigenvalues of A. The pair

$$\sigma = \sigma(A) = (\sigma^+(A), \sigma^-(A))$$

is called the **inertia** of A. Note that rank of A is given by $\rho = \sigma^+ + \sigma^-$.

Theorem 1 (Sylvester's Law of Inertia). *Let A, A' be real symmetric matrices. Then A and A' are congruent iff $\sigma(A) = \sigma(A')$.*

We shall prove a generalization of this theorem. Note that the diagonal sign matrix

$$\mathtt{diag}(\underbrace{1, \ldots, 1}_{\sigma^+}, \underbrace{-1, \ldots, -1}_{\sigma^-}, 0, \ldots, 0)$$

has inertia (σ^+, σ^-). Hence the inertia theorem implies that every real symmetric matrix is congruent to such a sign diagonal matrix. Note that this theorem applies only to symmetric matrices. For instance, the matrix $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ is not congruent to any diagonal matrix (Exercise).

On the other hand:

Corollary 2. *Inertia is a complete set of invariants for congruence: A, B are congruent iff they have the same inertia.*

Proof. If they are congruent, then they have the same inertia. Conversely, suppose A, B have the same inertia (s, t). Then it is easy to see that they are each congruent to $\mathtt{diag}(a_1, \ldots, a_s, b_1, \ldots, b_t, 0, \ldots, 0)$ where $a_i = 1$ and $b_j = -1$ for all i and j. Since congruence is an equivalence relation, this shows that A and B are congruent. **Q.E.D.**

Since our surface is unaffected if A is multiplied by any non-zero constant, and as the eigenvalues of $-A$ are just the negative of the eigenvalues of A, we regard the inertia values (s, s') and (s', s) as equivalent. By convention, we choose $\sigma^+ \geq \sigma^-$. For instance, for rank 4 matrices, we have the following possible inertia: $(4, 0), (3, 1), (2, 2)$.

We should note that, in fact, A is never congruent to $-A$ except for the trivial case $A = \mathbf{0}$. This remark is depends on our assumption that only real matrices are used in the definition of congruence. If we allow complex matrices then $A = M^T(-A)M$ with the choice $M = \mathbf{i}I$.

Canonical Equation. In Column 3 of Table 1, we see the canonical equation for each type of surface. We shall show that these are canonical forms for affine transformations. Most, but not all, canonical equations are **diagonal**, i.e., have the form

$$Q_A = aX^2 + bY^2 + cZ^2 + d = 0. \tag{7}$$

It is well-known that quadratic equation can be made diagonal by applying a projective transformations (this is shown in the next section). If we require $|\det(M)| = 1$ for our transformation matrices, then a, b, c, d would be general real numbers. For instance, if $a, b, c, -d$ are all positive, then we often see this in the equivalent form $(X/\alpha)^2 + (Y/\beta)^2 + (Z/\gamma)^2 = r^2$, representing an ellipsoid. But with general projective transformations, we can further ensure that $a, b, c, d \in \{-1, 0, +1\}$. Such are the canonical equations in Table 1.

From the canonical equations in Column 3, we can almost read off the inertia values σ, σ_u. There is an observation that will help our reading. Consider the hyperbolic paraboloid $X^2 - Y^2 - Z = 0$ in Row 6. Its equation is not diagonal (7). The matrix for this quadric is A_0, implicitly defined by the following equation. The matricies M_0 and A_1 are also implicitly defined:

$$M_0^T A_0 M_0 = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & e & -e \\ & & e & e \end{bmatrix} \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & & -1 \\ & & -1 & \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & e & e \\ & & -e & e \end{bmatrix} \tag{8}$$

$$= \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & 1 & \\ & & & -1 \end{bmatrix} = A_1, \qquad e = \frac{1}{\sqrt{2}}. \tag{9}$$

Since A_0 is the matrix of the hyperbolic paraboloid, and A_1 has inertia $(2, 2)$, this proves that inertia σ is $(2, 2)$ as given in Row 6. The same trick enables us to determine σ for all the other cases. Note the inverse of M_0 is its transpose: $M_0^{-1} = M_0^T$ and M_0 is not an affine transformation.

Projective Classification. The classification of surfaces in projective space $\mathbb{P}^3(K)$ (for $K = \mathbb{R}$ or $K = \mathbb{C}$) is relatively simple. In the projective case, we replace the equation in (7) by

$$aX^2 + bY^2 + cZ^2 + dW^2 = 0 \tag{10}$$

where W is the homogenizing variable. In projective space, the W-coordinate is no different than the other three coordinates. If $K = \mathbb{C}$, then we can choose complex transformations so that each a, b, c, d are either 0 or 1. Hence, *the rank ρ of the matrix A is a complete set invariant for complex projective surfaces.* Next consider the case $K = \mathbb{R}$. In this case, we can make $a, b, c, d \in \{0, \pm 1\}$. Consider the classification for non-singular surfaces under general projective transformations. Non-singular means $abcd \neq 0$. Then there are three distinct values for the inertia σ:

- $\sigma = (4, 0)$. This is purely imaginary.
- $\sigma = (3, 1)$. One has opposite sign than the others. Wlog, let $d = -1$ and the others positive. We get the equation of the ellipsoid
$$X^2 + Y^2 + Z^2 = W^2.$$
- $\sigma = (2, 2)$. Say $a = d = +1$ and $b = c = -1$. Then the canonical equation becomes $X^2 - Y^2 = Z^2 - W^2$ or
$$(X + Y)(X - Y) = (Z + W)(Z - W). \tag{11}$$
Note that (11) gives rise to an alternative canonical form for such surfaces: $X'Y' - Z'W' = 0$.

In contrast to the projective classification, the affine classification in Table 1 makes a distinction between the X, Y, Z-coordinates and the W-coordinate. This introduces, in addition to σ, another invariant σ_u. In analogy to Corollary 2, we have:

Theorem 3.
(i) The pair (σ, σ_u) is a complete set of invariants for real affine transformations.
(ii) This set of four numbers $\sigma^+, \sigma^-, \sigma_u^+, \sigma_u^-$ is minimal complete: if any of the numbers is omitted, then the resulting set is not complete.

Proof. (Partial) (i) One direction is trivial: if A, B are affine congruent, then they clearly have the same inertia pair (σ, σ_u). We will defer the proof of the other direction until we have have introduced the affine diagonalization algorithm.

(ii) It is enough to look at Table 1. If any of the four numbers are omitted, we can find two rows in the table which agree on the remaining three numbers but differ on the omitted number. We can even restrict our rows to the proper quadrics for this purpose. **Q.E.D.**

We are now ready to prove the main classification theorem.

Theorem 4. *The classification of quadrics under real affine transformations in Table 1 is complete and non-redundant.*

Proof. In view of Theorem 3, this amounts to saying that the list of inertia pairs (σ, σ_u) in Table 1 is complete and non-redundant. Non-redundancy amounts to a visual inspection that the inertia pairs in the table are distinct. As for completeness, we need to show that there are no other possibilities beyond the 20 rows displayed. For instance, the pair $(\sigma, \sigma_u) = ((3,1),(1,0))$ is absent from the table, and we have to account for such absences. Our plan is to examine the possible σ_u associated with each value of σ. Assume that our quadratic polynomial $Q_A(X, Y, Z, W)$ is equivalent to (10) under some projective transformation.

- $\sigma = (4, 0)$: clearly, $\sigma_u = (3, 0)$ is the only possibility.
- $\sigma = (3, 1)$: In this case, we can have $d > 0$ or $d < 0$. This corresponds to $\sigma_u = (2, 1)$ or $\sigma_u = (3, 0)$. But in any case, we can apply the transformation matrix M_0 in (8) to create the matrix with $\sigma_u = (2, 0)$, illustrated as:

$$\begin{aligned} M_0^T A M_0 &= \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & e & -e \\ & & e & e \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & e & e \\ & & -e & e \end{bmatrix} \\ &= \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & -1 & \\ & & & 1 \end{bmatrix}, \qquad e = \frac{1}{\sqrt{2}}. \end{aligned}$$

  Such application of the matrix M_0 has the effect of annhilating one of the diagonal entries of A_u. All three values of σ_u appear in Table 1 (Rows 2,3,5).
- $\sigma = (2, 2)$: Note that if the inertia of A is $(2, 2)$ then the inertia of A_u must be $(2, 1)$ (Row 4). But we can also apply M_0 to annihilate one diagonal entry of A_u, giving $\sigma_u = (1, 1)$ (Row 6).
- $\sigma = (3, 0)$: The inertia of A_u is either $(3, 0)$ or $(2, 0)$ when inertia of A is $(3, 0)$. This is shown in Rows 7, 10. In this case, no application of M_0 is possible.
- $\sigma = (2, 1)$: Without applying the M_0, we get $\sigma_u = (2, 1), (2, 0)$ or $(1, 1)$ (Rows 8,9,11). Using M_0, we get $\sigma_u = (1, 0)$ (Row 12).

- $\sigma = (2,0)$: The only possibilities are $\sigma_u = (2,0), (1,0)$, (Rows 14,16).
- $\sigma = (1,1)$: The possibilities are $\sigma_u = (1,1), (1,0), (0,0)$ (Rows 13,15,17).
- $\sigma = (1,0)$: The possibilities are $\sigma_u = (1,0), (0,0)$ (Rows 18,19).
- $\sigma = (0,0)$: The possibilities are $\sigma_u = (0,0)$ (Row 20).

Q.E.D.

Examples. Let us see some simple techniques to recognize the type (i.e., classification) of a quadric.

- Consider the quadric surface whose equation is

$$Q_A(X,Y,Z) = X^2 + YZ = 0. \tag{12}$$

  What is the type of this surface? Similar to the transformation (8), if we set

$$X = X', \quad Y = Y' - Z', \quad Z = Y' + Z' \tag{13}$$

  the equation is transformed to

$$Q_{A'}(X',Y',Z') = X'^2 + Y'^2 - Z'^2 = 0. \tag{14}$$

  Thus $\sigma = \sigma_u = (2,1)$, and so our surface is an elliptic cone (row 8, Table 1). Actually, we ought to verify that (13) is invertible. See Exercise.

- Consider next the quadric $X^2 + Y^2 + YZ = 0$. To determine its type, use the substitution

$$X = X', \quad Y = Y', \quad Z = Z' - Y'$$

  to transformed this quadric to the form $X'^2 + Y'Z' = 0$. The later has already been shown to be an elliptic cone.

- What if there are linear terms? Consider $X^2 + Y + Z = 0$. Linear terms do not affect σ_u (since they live outside the upper submatrix), and so we can read off $\sigma_u = (1,0)$. To determine σ, we view the equation projectively as $X^2 + (Y+Z)W = 0$. This can first be transformed to $X^2 + Y'W = 0$ for some new variable Y'. Then, using the projective transformation as in (8), the equation is further transformed to to $X^2 + Z'^2 - W'^2 = 0$. Thus $\sigma = (2,1)$. Thus the surface is a parabolic cylinder.

- A key trick is to "complete the square" to kill off off-diagonal terms. Consider $X^2 + 2XY - 2XZ + Y^2 - 2Z^2 = 0$. Here, we have a square term (X^2) as well as linear terms ($2XY - 2XZ$) in the variable X. Collecting these terms in X together, we rewrite it as

$$X^2 + 2XY - 2XZ = (X + (Y-Z))^2 - (Y-Z)^2 = X'^2 - (Y-Z)^2.$$

  We rename $X + (Y - Z)$ as X'. This is called **completing the square for** X. The original equation becomes

$$X'^2 - (Y-Z)^2 + Y^2 - 2Z^2 = X'^2 - 2YZ - 3Z^2 = 0.$$

  Completing the square for Z,

$$3Z^2 + 2YZ = 3[Z^2 + (2/3)YZ] = 3[(Z + Y/3)^2 - Y^2/9] = 3(Z + Y/3)^2 - Y^2/3$$

  The original equation now becomes $X'^2 - 3Z'^2 + Y^2/3 = 0$. Since we only used affine transformations, we see that $\sigma = \sigma_u = (2,1)$, and the surface is an elliptic cone.

- What if there constant terms? Consider $X^2+Y^2-2Z=1$. As far σ_u is concerned, the constant term has no effect. In this case, we can just read off $\sigma_u=(2,0)$ immediately. To compute σ, the constant term amounts to the introduction of W^2. The equation becomes $X^2+Y^2-2ZW-W^2=0$. But we can complete the square for W to get $X^2+Y^2-(W+Z)^2+Z^2=0$. Thus $\sigma=(3,1)$. We conclude that the surface is an elliptic paraboloid.

Exercises

Exercise 1.1: Prove that $A=\begin{bmatrix}1&2\\0&1\end{bmatrix}$ is not congruent to a diagonal matrix. Suppose $A=M^TBM$ where $M=\begin{bmatrix}a&b\\c&d\end{bmatrix}$ and B is diagonal. Then, by suitable modification of M, we can assume that $B=\texttt{diag}(1,\pm1)$. But in fact, $B=\texttt{diag}(1,-1)$ leads to a contradiction: for any non-zero vector $v=(x,y)$, we have $f(x,y)=v^TAv=(x+y)^2>0$. But we can choose v such that $Mv=(0,1)$. Then $f(x,y)=(Mv)^TB(Mv)=-1<0$. So we may assume $B=\texttt{diag}(1,1)=I$. Then $M^TBM=M^TM=$ is a symmetric matrix, and this cannot be equal to A which is non-symmetric. ♢

Exercise 1.2: (a) Verify the transformation (13) is invertible.
(b) Show that the following transformations $(X,Y,Z)\mapsto(X,X,Z)$ and $(X,Y,Z)\mapsto(X+Y+Z,X-Y-Z,Y+Z)$ are not invertible.
(c) Show that $X_i\mapsto cX_i$ and $X_i\mapsto X_i+cX_j$ (for $c\neq0$ and $i\neq j$) are invertible. In this notation, it is assumed that X_k $(k\neq i)$ is unchanged. Call these transformations **elementary**.
(d) Show that every invertible transformations is a composition of elementary operations. HINT: view transformations as matrices, and give a normal form for such matrices under elementary transformations. ♢

Exercise 1.3: What is the real affine classification of the following surfaces?
(a) $X^2+Y^2-2X=0$
(b) $X^2+Y^2-2X+XZ=0$
(c) $X^2+Y^2-2X+XY=0$
(d) $X^2+Y^2-2X+XY=1$
(e) $X^2+Y^2-2X+XY+YZ=0$
(f) $X^2-Y^2-2X+Y+Z=1$
(g) $X^2-Y^2-2X+XY+Z=0$ ♢

Exercise 1.4: In the previous exercise, construct the transformation matrices that bring each of quadric into its normal form. ♢

Exercise 1.5: Let $Q(\mathbf{X})=\mathbf{X}^TA\mathbf{X}=0$ be a quadric hypersurface where $\mathbf{X}=(X_1,\ldots,X_{n-1},1)$ and $A=(a_{ij})_{i,j=1}^n$ is symmetric. Prove that the hypersurface is singular iff $\det(A)=0$. ♢

Exercise 1.6: Show that a non-singular quadric surface $Q_A(X,Y,Z)=0$ is elliptic, hyperbolic, parabolic iff $\det(A_u)>0,<0,=0$ (respectively). ♢

Exercise 1.7: Let $A=\begin{bmatrix}A_u&t\\t^T&c\end{bmatrix}$ be the matrix for a central quadric. Show that the eigenvectors of A_u gives the three axes of the quadric. Also, the center of the quadric is $-(A_u)^{-1}t$. ♢

Exercise 1.8: Extend our classification of quadrics in $\mathbb{A}^3(\mathbb{R})$ to $\mathbb{A}^4(\mathbb{R})$. ♢

End Exercises

§2. Projective Diagonalization

The last section gave some ad hoc tricks for recognizing the type of a quadric. To do this systematically, we need an algorithm to diagonalize quadratic polynomials. Note that it is a small step to move from a diagonal form to the canonical quadrics. In some applications, we want something more: not only do we want to know the type, we actually need to know the transformation matrix M that transforms a quadric $Q_A(\mathbf{X})$ into a diagonal form $Q_A(M\mathbf{X})$. For instance, it is easy to parametrize the canonical quadrics, and knowing M, we can convert this into a parametrization of the original quadric. We now provide such an algorithm.

In fact, we shall describe a diagonalization algorithm for a quadratic form in any dimension n,

$$Q_A(\mathbf{X}) = Q_A(X_1, \ldots, X_n) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} X_i X_j$$

where $a_{ij} = a_{ji}$ and $\mathbf{X} = (X_1, \ldots, X_n)^T$. Although we may think of X_n as the homogenization variable, but the present algorithm makes no such distinction. There are two simple ideas in this algorithm:

- Completing the Square for a Diagonal Term: if $Q_A(\mathbf{X})$ contains a **diagonal term** $a_{11}X_1^2$ ($a_{11} \neq 0$) then we can transformation $Q_A(\mathbf{X})$ to $Q_A(M\mathbf{X}) = cX_1^2 + Q_B(X_2, \ldots, X_n)$ for some $c \neq 0$ and Q_B.
- Elimination of a Hyperbolic Term: if $Q_A(\mathbf{X})$ contains a **hyperbolic term** $a_{1j}X_1X_j$ ($1 \neq j$, $a_{1j} \neq 0$) then we can transform $Q_A(\mathbf{X})$ to $Q_A(M\mathbf{X}) = cX_1^2 + Q_C(X_1, X_2, \ldots, X_n)$ for some $c \neq 0$ and Q_C. In this case, Q_C may contain terms in X_1. But, if $a_{11} = 0$ in the first place, then Q_C has only terms that are linear in X_1. Therefore, we have introduced a diagonal term in X_1, which can be completed by the previous method.

The input of our algorithm is the real symmetric matrix A. The output is a pair $(M, \mathtt{diag}(a_1, \ldots, a_n))$ of real matrices where M is invertible satisfying

$$A = M^T \mathtt{diag}(a_1, \ldots, a_n) M \tag{15}$$

and all elements in M and the a_i's belong to the rational field generated by elements of A. This field will be denoted $\mathbb{Q}(A)$. We shall use induction on n to do this work. All the computation can be confined to a subroutine which takes an input matrix A, and outputs a pair (M, B) of real matrices such that $A = M^T B M$ where

$$\begin{aligned} Q_B(\mathbf{X}) &= Q_A(M\mathbf{X}) \\ &= a_1 X_1^2 + Q_{B'}(X_2, \ldots, X_n), \end{aligned} \tag{16}$$

for some real a_1. Here, as a general notation, B' is the submatrix of B obtained by deleting the first row and first column. Call this subroutine **Diagonalization Step**. Using such a subroutine, we construct the recursive diagonalization algorithm:

Projective Diagonalization Algorithm
Input: A real symmetric matrix A of order n
Output: Pair of matrices (M, B) where $B = \mathtt{diag}(a_1, \ldots, a_n)$ and $A = M^T B M$.
1. Call Diagonalization Step on input A.
   This returns a pair (M_1, B_1).
2. If $n = 1$, we return (M_1, B_1).
3. Recursively call Projective Diagonalization with matrix B_1'.
   This returns a pair (M_2, B_2) of $(n-1) \times (n-1)$ matrices.
   Suppose the top-left entry of B_1 is a.
   Let $B_3 = \left[\begin{array}{c|c} a & \\ \hline & B_2 \end{array}\right]$, and $M_3 = \left[\begin{array}{c|c} 1 & \\ \hline & M_2 \end{array}\right]$.
4. Return $(M, B) = (M_3 M_1, B_3)$.

The correctness of this algorithm is straightforward. We only verify that step 4 returns the correct value (M_3M_1, B_3):

$$\begin{aligned} B_1 &= \left[\begin{array}{c|c} a & \\ \hline & B_1' \end{array}\right] \\ &= \left[\begin{array}{c|c} 1 & \\ \hline & M_2^T \end{array}\right]\left[\begin{array}{c|c} a & \\ \hline & B_2 \end{array}\right]\left[\begin{array}{c|c} 1 & \\ \hline & M_2 \end{array}\right]. \\ &= M_3^T B_3 M_3 \\ A &= M_1^T B_1 M_1 \\ &= (M_3M_1)^T B_3 (M_3M_1) \end{aligned}$$

This last equation is just the specification for diagonalizing A.

Diagonalization Step. This is a non-recursive algorithm. It maintains a pair of two matrices (M, B) which satisfies the invariant $A = M^T BM$. We initialize $M = \mathbf{1}$ (identity matrix) and $B = A$. Thus, (M, B) satisfies the invariant $A = M^T BM$ initially. Our algorithm consists of applying at most three transformations of the pair (M, B) by some invertible N as[4] follows:

$$(M, B) \mapsto (NM, N^{-T} BN^{-1}) \tag{17}$$

It is easy to verify that our invariant is preserved by such updates. The transformation (17) amounts to transforming the quadric $Q_B(\mathbf{X})$ to $Q_C(\mathbf{X})$ where $Q_C(N\mathbf{X}) = Q_B(\mathbf{X})$ or $N^T CN = B$. Thus $Q_A(\mathbf{X}) = Q_B(M\mathbf{X}) = Q_C(NM\mathbf{X})$. Let $B = (b_{ij})_{i,j=1}^n$.

The algorithm consists of the following sequence of steps, which we have organized so that it can be directly implemented in the order described.

STEP 0 If $n = 1$ or if $b_{12} = b_{13} = \cdots = b_{1n} = 0$, we can return $(M, B) = (\mathbf{1}, A)$ immediately. So assume otherwise. If $b_{11} = 0$, go to STEP 1; else, go to STEP 3.

STEP 1 We now know $b_{11} = 0$. First the first $k > 1$ such that $b_{kk} \neq 0$. If such a k is not found, we go to STEP 2. Otherwise, we transpose the variables X_1 and X_k. This amounts to applying the matrix $P_k = (p_{ij})_{i,j=1}^n$ which is the identity matrix except that $p_{1i} = p_{i1} = 1$ and also $p_{11} = p_{ii} = 0$:

$$P_k = \begin{bmatrix} 0 & & \cdots & & 1 & & & \\ & 1 & & & & & & \\ \vdots & & \ddots & & \vdots & & & \\ & & & 1 & & & & \\ 1 & & \cdots & & 0 & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}. \tag{18}$$

Note that P_k is its own transpose and its own inverse. Thus P_k transforms $Q_B(\mathbf{X})$ to $Q_C(\mathbf{X}) = Q_B(P_k\mathbf{X})$ whose associated matrix

$$C = P_k \cdot B \cdot P_k$$

simply interchanges the 1st and kth rows, and also the 1st and kth columns. We update the pair (M, B) by the matrix $N = P_k$ as in (17). Go to STEP 3.

[4]Note that we write N^{-T} as shorthand for $(N^{-1})^T = (N^T)^{-1}$.

STEP 2 We now know that $b_{1k} \neq 0$ for some $k \geq 2$, and moreover $b_{11} = b_{kk} = 0$. Our goal is to create a non-zero diagonal term from a hyperbolic term $b_{1k}X_1X_k$. Putting

$$Y_1 = (X_1 + X_k)/2, \qquad Y_k = (X_1 - X_k)/2, \tag{19}$$

we have $2X_1X_k = Y_1^2 - Y_k^2$, $X_1 = (Y_1 + Y_k)$ and $X_2 = (Y_1 - Y_k)$. We use the convention that "$(X_1 \ldots \widehat{X_k} \ldots X_n)$" indicates the absence of X_k in the sequence $(X_1, \ldots, X_n)$. Then

$$\begin{aligned} Q_B(\mathbf{X}) &= 2b_{1k}X_1X_k + 2X_1\left(\sum_{j=2;j\neq k}^{n} b_{1j}X_j\right) + 2X_k\left(\sum_{j=2;j\neq k}^{n} b_{kj}X_j\right) + Q_{B''}(X_2, \ldots, \widehat{X_k}, \ldots, X_n) \\ &= b_{1k}(Y_1^2 - Y_k^2) + 2(Y_1 + Y_k)\left(\sum_{j=2;j\neq k}^{n} b_{1j}X_j\right) + 2(Y_1 - Y_k)\left(\sum_{j=2;j\neq k}^{n} b_{kj}X_j\right) + Q_{B''} \\ &= Q_C(Y_1, X_2, \ldots, X_{k-1}, Y_k, X_{k+1}, \ldots, X_n) \qquad \text{(implicit definition of } Q_C\text{)} \end{aligned}$$

The coefficients of Y_1^2 and Y_k^2 are non-zero in $Q_C(Y_1, X_2, \ldots, X_{k-1}, Y_k, X_{k+1}, \ldots, X_n)$. The transformation (19) is given by the matrix $R_k = (r_{ij})_{i,j=1}^n$ which is similar to the identity matrix except that $r_{11} = r_{1k} = r_{k1} = e$ and $r_{kk} = -e$ where $e = 1/2$:

$$R_k = \begin{bmatrix} e & & \cdots & & e & & & \\ & 1 & & & & & & \\ \vdots & & \ddots & & \vdots & & & \\ & & & 1 & & & & \\ e & & \cdots & & -e & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}. \tag{20}$$

The inverse R_k^{-1} is obtained from R_k by replacing each occurence of e by 1 (and $-e$ becomes -1). Also, the R_k and R_k^{-1} are their own transposes. Observe that $R_k\mathbf{X} = (\frac{X_1+X_2}{2}, X_2, \ldots, X_{k-1}, \frac{X_1-X_2}{2}, X_{k+1}, \ldots, X_n)^T$ and $R_k^{-1}\mathbf{X} = ((X_1 + X_2), X_2, \ldots, X_{k-1}, (X_1 - X_2), X_{k+1}, \ldots, X_n)^T$. Hence $Q_B(\mathbf{X}) = Q_C(R_k\mathbf{X})$, or $Q_B(R_k^{-1}\mathbf{X}) = Q_C(\mathbf{X})$. We may therefore update (M, B) with the matrix $N = R_1$ as in (17). Go to STEP 3.

STEP 3 At this point $b_{11} \neq 0$. All the previous steps are preparations to arrive at this point. The basic idea is to "complete the square" for X_1. We have

$$\begin{aligned} Q_B(\mathbf{X}) &= b_{11}X_1^2 + 2X_1\left(\sum_{j=2}^{n} b_{1j}X_j\right) + Q_{B'}(X_2, \ldots, X_n) \\ &= \frac{1}{b_{11}}\left(b_{11}X_1 + \left(\sum_{j=2}^{n} b_{1j}X_j\right)\right)^2 - \frac{1}{b_{11}}\left(\sum_{j=2}^{n} b_{1j}X_j\right)^2 + Q_{B'}(X_2, \ldots, X_n) \\ &= \frac{1}{b_{11}}Y_1^2 + Q_{C'}(X_2, \ldots, X_n) \qquad (C' \text{ implicitly defined here with } Y_1 = \textstyle\sum_{j=2}^{n} b_{1j}X_j \text{ }) \\ &= Q_C(Y_1, X_2, \ldots, X_n) \qquad (Q_C \text{ implicitly defined here}). \end{aligned}$$

Thus, with the matrix

$$T = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}. \tag{21}$$

then $Q_C(N\mathbf{X}) = Q_B(\mathbf{X})$. Hence $Q_C(\mathbf{X}) = Q_B(N^{-1}\mathbf{X}) = X_1^2 + Q_C(X_2, \ldots, X_n)$. The inverse N^{-1} is easy to describe:

$$N^{-1} = \begin{bmatrix} 1/b_{11} & -b_{12}/b_{11} & \cdots & -b_{1n}/b_{11} \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}.$$

So we update (M, B) using the matrix N as in (17). This concludes the description of Diagonalization Step.

Sign Diagonalization. In general, let us call a matrix a **sign matrix** if every entry of the matrix is a sign (i.e., $0, \pm 1$). In particular, the matrix $\texttt{diag}(a_1, \ldots, a_n)$ is a **sign diagonal matrix** if each $a_i \in \{0, \pm 1\}$. Let (M, B) be the output of our diagonalization algorithm where $B = \texttt{diag}(a_1, \ldots, a_n)$. Thus, $A = M^T B M$ where the entries of B and M belongs to $\mathbb{Q}(A)$ (rational in entries of A).

We can modify this output so that B is a sign diagonal matrix as follows: let $c_i = \sqrt{|a_i|}$ for $i = 1, \ldots, n$. Define $S = \texttt{diag}(1/c_1, \ldots, 1/c_n)$ and $S^{-1} = \texttt{diag}(c_1, \ldots, c_n)$. Then the output $(S^{-1}M, SBS)$ would have the desired property. Let $\sqrt{\mathbb{Q}(A)}$ denote those elements that are square roots of elements in $\mathbb{Q}(A)$. Thus the elements of S belong to $\sqrt{\mathbb{Q}(A)}$. We do not build this "sign diagonalization" into the main algorithm because of its need for square roots. Further more, if we allow $\sqrt{-1}$, then we can make all the signs to be positive or 0. Thus, we have given a constructive proof of the following theorem.

Theorem 5. *Let A be a symmetric real matrix.*
(i) There exists an invertible matrix M whose elements belong to $\mathbb{Q}(A)$ such that $M^T AM$ is diagonal.
(ii) There exists an invertible matrix M whose elements belong to $\sqrt{\mathbb{Q}(A)}$ such that $M^T AM$ is sign diagonal.
(iii) There exists an invertible matrix M whose elements belong to $\mathbb{C}$ such that $M^T AM$ is non-negative sign diagonal. In particular if A is non-singular, $M^T AM = \mathbf{1}$.

§3. Affine Diagonalization

The canonical form for symmetric matrices under projective transformations is the diagonal matrix. What is the corresponding canonical form under affine transformations? We observe that diagonal matrices may not be achievable: e.g., the canonical hyperbolic paraboloid $X^2 - Y^2 - Z = 0$ cannot be diagonalized using affine transformations. Part of our task is to determine what the "affine canonical form" take.

We shall describe the diagonalization process for a quadratic polynomial in $n-1$ dimensions,

$$Q_A(X_1, \ldots, X_{n-1}, 1) = \mathbf{X}^T A \mathbf{X}$$

where A is $n \times n$ and $\mathbf{X} = (X_1, \ldots, X_{n-1}, 1)^T$, $n \geq 2$. As usual, the (i,j)th entry of A is denoted a_{ij}. The two techniques of projective diagonalization still works, but a third technique is needed.

- Completing the square can be extended to the affine case. Let

$$Q_A(\mathbf{X}) = a_{11}X_1^2 + 2X_1\left(\sum_{i=2}^{n-1} a_{1i}X_i\right) + 2X_1 a_{1n} + Q_B(X_2, \ldots, X_{n-1})$$

Completing the square for X_1, we get

$$Q_A(\mathbf{X}) = \frac{1}{a_{11}}\left(a_{11}X_1 + \left(\sum_{i=2}^{n-1} a_{1i}X_i\right) + a_{1n}\right)^2 + Q_C(X_2, \ldots, X_{n-1})$$

where

$$Q_C(X_2, \ldots, X_{n-1}) = -\frac{1}{a_{11}}\left(a_{1n} + \sum_{i=2}^{n-1} a_{1i}X_i\right)^2 + Q_C(X_2, \ldots, X_{n-1})$$

- Hyperbolic terms can again be eliminated by reduction to square terms: assuming $a_{11} = a_{22} = 0$ and $a_{12} \neq 0$, we have

$$Q_A(\mathbf{X}) = 2a_{12}X_1X_2 + 2X_1\left(\sum_{i=3}^{n-1} a_{1i}X_i\right) + 2X_2\left(\sum_{i=3}^{n-1} a_{2i}X_i\right) + Q_B(X_3, \ldots, X_{n-1}).$$

  Then putting $X_1 = (X_1' + X_2')$ and $X_2 = (X_1' - X_2')$, we get

$$\begin{aligned} Q_A(\mathbf{X}) &= 2a_{12}(X_1'^2 - X_2'^2) \\ &\quad +2(X_1' + X_2')\left(\sum_{i=3}^{n-1} a_{1i}X_i\right) + 2(X_1' - X_2')\left(\sum_{i=3}^{n-1} a_{2i}X_i\right) + Q_B(X_3, \ldots, X_{n-1}). \end{aligned}$$

  Thus we can now complete the square for either X_1' or X_2'.

- Consolidation of Linear and Constant Terms: if $a_{11}, a_{12}, \ldots, a_{1,n-1} = 0$ and $a_{1n} \neq 0$, then we have no technique for eliminating the linear term $a_{1n}X_1$. We shall live with such terms. The key observation is that *we do not more than one such term*. Suppose that we can no longer apply any of the above transformations. Then $Q_A(\mathbf{X})$ is equal to a linear term in some variables (say) $X_1, \ldots, X_k$, plus the sum of squares in the remaining variables, $X_{k+1}, \ldots, X_{n-1}$:

$$Q_A(\mathbf{X}) = a_{nn} + 2\left(\sum_{i=1}^{k} a_{in}X_i\right) + \sum_{i=k+1}^{n} a_{ii}X_i^2$$

  where a_{nn} is arbitrary but $a_{in} \neq 0$ for each $i = 1, \ldots, k$. There are two possibilities: if $k = 0$ then there are no linear terms. This gives the **affine diagonal form**,

$$Q_A(\mathbf{X}) = a_{nn} + \sum_{i=1}^{n} a_{ii}X_i^2$$

  Otherwise, suppose $k \geq 1$. Then by the substitution

$$X_1 = \frac{1}{2a_{1n}}\left(X_1' - a_{nn} - 2\sum_{i=2}^{k} a_{in}X_i\right)$$

  we obtain the **almost diagonal form**,

$$Q_A(\mathbf{X}) = X_1' + \sum_{i=k+1}^{n} a_{ii}X_i^2$$

Note that the above transformations are affine. We have thus proved:

Theorem 6. *Every quadratic polynomial $Q_A(X_1, \ldots, X_{n-1})$ can be transformed by a rational affine transformation to an affine diagonal form*

$$Q_B(X_1, \ldots, X_{n-1}) = a_n + \sum_{i=1}^{n-1} a_i X_i^2 \tag{22}$$

or an almost diagonal form

$$Q_C(X_1, \ldots, X_{n-1}) = X_{n-1} + \sum_{i=1}^{n-2} a_i X_i^2. \tag{23}$$

Using transformation matrices whose elements come from $\sqrt{\mathbb{Q}(A)}$, we can make the a_i's to be signs.

The matrices B and C corresponding to the quadratic polynomials (22) and (23) are

$$B = \left[\begin{array}{ccc|c} a_1 & & & \\ & \ddots & & \\ & & a_{n-1} & \\ \hline & & & a_n \end{array}\right] \tag{24}$$

and

$$C = \left[\begin{array}{ccc|cc} a_1 & & & & \\ & \ddots & & & \\ & & a_{n-2} & & \\ \hline & & & & \frac{1}{2} \\ & & & \frac{1}{2} & \end{array}\right], \tag{25}$$

respectively. Also, a quadratic polynomial is said to be in **canonical affine form** if it is in either of these forms. If we are willing to use square-roots, then we can make each a_i into a sign $(0, \pm 1)$; such are the canonical equations shown in Table 1.

Lemma 7. *Let A and A_u have rank r and r_u, respectively. The following are equivalent:*
(i) A is affine congruent to an almost diagonal matrix.
(ii) $r - r_u = 2$.
Similarly, the following are equivalent:
(i)′ A is affine congruent to a diagonal matrix.
(ii)′ $r - r_u \leq 1$.

Proof. By Theorem 6, A is affine congruent to B where B is either diagonal or almost diagonal.
(a) (i) implies (ii): Suppose B is almost diagonal. Then $r_u = \texttt{rank}(A_u) = \texttt{rank}(B_u) = \texttt{rank}(B_{uu})$ where B_{uu} is the upper submatrix of B_u. Further $r = \texttt{rank}(A) = \texttt{rank}(B) = \texttt{rank}(B_{uu}) + 2$. This shows $r - r_u = 2$. This proves (i) implies (ii).
(b) (i)′ implies (ii)′: If B is diagonal, then $r = \texttt{rank}(B) \leq \texttt{rank}(B_u) + 1 = r_u + 1$. Thus $r - r_u \leq 1$.

To show the converse of (a), suppose suppose (ii) holds. So (ii)′ does not hold. Then (b) says that (i)′ cannot hold. Hence (i) must hold. A symmetrical argument shows the converse of (b). **Q.E.D.**

The conditions (ii) $r - r_u \leq 1$ and (ii)′ $r - r_u = 2$ characterize the diagonal and almost diagonal matrices. Since these two conditions are mutually exclusive, we have shown:

Corollary 8. *A diagonal matrix and an almost diagonal matrix are not affine congruent.*

We can now complete a deferred proof. Our proof of Theorem 3 was incomplete because we did not prove the following result:

Lemma 9. *If A, B have the same inertia pair (σ, σ_u), then A and B are affine congruent.*

Proof. We may assume that A, B are already in affine canonical form. By Lemma 7, either A and B are both diagonal, or they are both almost diagonal. By a further transformatiom, we can assume A and B are sign matrices. Let $\sigma = (\sigma^+, \sigma^-)$ and $\sigma_u = (\sigma_u^+, \sigma_u^-)$. Recall that we distinguish matrices up to multiplication by a non-zero constant.

CLAIM: There is a choice of $B \in \{A, -A\}$ such that (a) B has exactly σ^+ positive entries, and (b) B_u has exactly σ_u^+ positive entries. *Proof*: If A is diagonal, and $\sigma^+ > \sigma^-$, then we choose B such that it has exactly σ^+ positive diagonal entries. If $\sigma^+ = \sigma^-$, then we pick B such that the last diagonal entry in B is negative or zero. If B is almost diagonal, then we pick B such that that B has σ^+ positive diagonal entries. This proves our claim.

First assume A, B are diagonal. Since B_u and A_u have the same number of positive and same number of negative entries, and they are sign matrices, we can find an affine permutation transformation M such that

$(B)_u = (M^T AM)_u$. But since A and B have the same rank, we conclude that, in fact, $B = M^T AM$. This proves that A and B are affine congruent.

If A, B are almost diagonal, the proof is similar. **Q.E.D.**

Affine Diagonalization Algorithm. We consider the affine analogue of the projective diagonalization algorithm. The state of our algorithm is captured, as before, by the pair of matrices (M, B) that will eventually represent our output. Again (M, B) is initially $(\mathbf{1}, A)$ and we successively transform (M, B) by "updating with an invertible matrix N", as in (17).

In addition, we keep track of a pair of integers (s, t) initialized to $(1, n)$ and satisfying $1 \le s \le t \le n$. The shape of the matrix B has the following structure:

$$B = \left[\begin{array}{c|c|c||c} D & 0 & 0 & 0 \\ \hline 0 & C & 0 & c \\ \hline 0 & 0 & 0 & d \\ \hline\hline 0 & c^T & d^T & e \end{array}\right], \qquad \text{where } \begin{cases} D = \mathtt{diag}(a_1, \ldots, a_{s-1}), a_i\text{'s arbitrary} \\ C = (t-s) \times (t-s), \\ c = (t-s) \times 1, \\ d = (d_t, \ldots, d_{n-1})^T, \text{each } d_j \neq 0 \\ e \in \mathbb{R}. \end{cases} \tag{26}$$

Intuitively, the first $s-1$ variables $(X_1, \ldots, X_{s-1})$ have already been put in diagonal form; last $n-t$ variables $(X_t, \ldots, X_{n-1})$ are known to be linear. The variable X_s is the current focus of our transformation.

Note that when $s = t$, the matrix C and vector c are empty. In that case

$$Q_B(\mathbf{X}) = \sum_{i=1}^{t-1} a_i X_i^2 + 2 \sum_{j=t}^{n-1} d_j X_j + e$$

where $d = (d_t, d_{t+1}, \ldots, d_{n-1})^T$. With at most one more transformation, we would be done. More precisely:
(a) If $t = n$, B is diagonal.
(b) If $t \le n-1$ then we apply the transformation $X_{n-1} \mapsto \frac{1}{2d_{n-1}} \left(2X'_{n-1} - e - 2\sum_{j=t}^{n-2} d_j X_j\right)$ or $2X'_{n-1} \mapsto e + 2\sum_{j=t}^{n-1} d_j X_j$. This transforms

$$Q_B(\mathbf{X}) = \sum_{i=1}^{t-1} a_i X_i^2 + 2 \sum_{j=t}^{n-1} d_j X_j + e \quad \mapsto \quad \sum_{i=1}^{t-1} a_i X_i^2 + 2X'_{n-1} = Q_C(X_1, \ldots, X_{n-2}, X'_{n-1}).$$

Denoting the transformation matrix by

$$T = \left[\begin{array}{c|ccc|c|c} \mathbf{1}_{t-1} & & & & & \\ \hline & & \mathbf{1}_{n-t-1} & & & \\ \hline & d_t & d_{t+1} \quad \cdots & d_{n-2} & d_{n-1} & e/2 \\ \hline & & & & & 1 \end{array}\right] \tag{27}$$

where $\mathbf{1}_m$ is the $m \times m$ identity matrix. Thus $Q_B(\mathbf{X}) = Q_C(T\mathbf{X})$ where $Q_C(X_1, \ldots, X_{n-1})$ is almost diagonal.

Affine Diagonalization Algorithm
Input: A real symmetric matrix A of order n
Output: Pair of matrices (M, B) where $B = \mathtt{diag}(a_1, \ldots, a_n)$
and $A = M^T B M$.
INITIALIZATION:
$(M, B) \leftarrow (\mathbf{1}, A)$ and $(s, t) \leftarrow (1, n)$.
– so B initially has the form of (26) –
MAIN LOOP:
while $(s < t)$ do
1. If $b_{ss} \neq 0$, complete the square for X_s.
Update (M, B) (as in Equation (21)).
$s \leftarrow s + 1$. Break.
2. Else if $(\exists k = s+1, \ldots, t-1)$ with $b_{kk} \neq 0$ then
Apply the matrix $P_{s,k}$ (as in (18)) to exchange the X_s and X_k. Break.
3. Else if $(\exists k = s+1, \ldots, t-1)$ with $b_{sk} \neq 0$ then
Apply the matrix $R_{s,k}$ (as in (20)) to remove the cross term $X_s X_k$. Break.
4. Else (we know $b_{si} = 0$ for $i = s, \ldots, n-1$)
Set $t := t - 1$, and if $s < t$ then apply the matrix $P_{s,t}$ to exchange X_s and X_t. Break.
CONSOLIDATING LINEAR AND CONSTANT TERMS:
If $(t = n$ or $(t = n - 1$ and $e = 0))$ then return (M, B).
Else apply the matrix (27).

Example: Find the affine canonical form for the following quadratic polynomial:

$$Q(W, X, Y, Z) = WX - 2W + 2Y - 2Z + 1. \tag{28}$$

For simplicity, we omit the tracking of the matrix M. Eliminating the hyperbolic term WX gives

$$(W'^2 - X'^2) - 2(W' + X') + 2Y - 2Z + 1.$$

Completing the square for W' gives

$$W''^2 - X'^2 - 2X' + 2Y - 2Z.$$

Completing the square for X' gives

$$W''^2 - X''^2 + 2Y - 2Z + 1.$$

Simplifying linear and constant terms,

$$W''^2 - X''^2 + 2Y' = 0.$$

REMARKS:
1. Note that we classify only *real* matrices A since we use inertia of A in our classification. But we could generalize A to be Hermitian, i.e., complex matrices satisfying $A = A^*$ (its conjugate transpose). In this case, the eigenvalues of A are still real, and we again have the law of inertia.
2. The affine diagonalization algorithm contains the projective diagonalization algorithm as a subalgorithm: all one has to do is to ignore the part that has to do with the last homogenization component.

Exercises

Exercise 3.1: Give the matrix M that transforms $Q_A(W, X, Y, Z)$ in (28) to $Q_B(W, X, Y, Z) = W^2 - X^2 + 2Y$. ♢

Exercise 3.2: Find the affine canonical form of $XY + YZ + 2X + 2Y + 2Z = 1$. Also determine the matrix which achieves this transformation. ♢

Exercise 3.3: Refine our algorithm so that we keep track of three integers, $1 \leq r \leq s \leq t \leq n$. With s, t as before, we want $r-1$ to be the number of identically zero rows in B. Moreover, we assume that these corresponds to the variables $X_1, \ldots, X_{r-1}$. Thus, the diagonal entries $a_{rr}, \ldots, a_{s-1,s-1}$ are non-zero. ♢

Exercise 3.4: Extend the affine diagonalization process to Hermitian matrices. ♢

Exercise 3.5: Suppose our transformations M are restricted to be upper block diagonal of the form

$$\left[\begin{array}{c|c|c|c} M_{11} & M_{12} & \cdots & M_{1k} \\ \hline & M_{22} & \cdots & M_{2k} \\ \hline & & \ddots & \\ \hline & & & M_{kk} \end{array}\right]$$

where each M_{ii} is an invertible $n_i \times n_i$ matrix. Give a complete set of invariants under such congruence. ♢

End Exercises

§4. Parametrized and Ruled Quadric Surfaces

A surface S with locus $L(S) \subseteq \mathbb{R}^3$ is **rational** if there exists a function

$$\mathbf{q} : \mathbb{R}^2 \to L(S) \cup \{\infty\}, \qquad \mathbf{q}(U, V) = (X(U, V), Y(U, V), Z(U, V))^T \tag{29}$$

where $X(U, V), Y(U, V), Z(U, V) \in \mathbb{R}(U, V)$ are rational functions, and with finitely many exceptions, each point in $L(S)$ is given by $\mathbf{q}(U, V)$ for some U, V. We call $\mathbf{q}$ a **rational parametrization** of S. We are concerned here with a special kind of rational surface, one that is swept by a line moving in space. More precisely, a **ruled surface** is one with a rational parametrization of the form

$$\mathbf{q}(U, V) = \mathbf{b}(U) + V\mathbf{d}(U). \tag{30}$$

Here, $\mathbf{b} \in \mathbb{R}(U, V)^3$ is called the **base curve** and $\mathbf{d} \in \mathbb{R}(U, V)^3$ is called the **director curve**. Observe that for each $u \in \mathbb{R}$, we have a line $\{\mathbf{b}(u) + v\mathbf{d}(u) : v \in \mathbb{R}\}$, called a **ruling**. In general, a ruling for a surface is a family of lines whose union is the locus of the surface, and with finitely point exceptions, each point in the surface lies on a unique line. The Gaussian curvature[5] is everywhere non-positive on such a surface. Two special cases may be mentioned:
(1) When the base curve degenerates to a single point $\mathbf{b}_0 \in \mathbb{R}^3$: $\mathbf{b}(u) = \mathbf{b}_0$ for all $u \in \mathbb{R}$. The ruled surface is a cone pointed at $\mathbf{b}_0$.
(2) When the director curve degenerates to a single point $\mathbf{d}_0 \in \mathbb{R}^3$: $\mathbf{d}(u) = \mathbf{d}_0 \neq \mathbf{0}$ for all $u \in \mathbb{R}$. The ruled surface is a cylinder whose axis is parallel to $\mathbf{d}_0$.

A surface is **singly-ruled** (resp., **doubly-ruled**) if there is a unique set (resp., exactly two sets) of rulings. For instance, the cylinder is singly-ruled. But a plane has infinitely many sets of rulings. The 1-sheeted hyperboloid $X^2 + Y^2 - Z^2 = 1$ has 2 ruled parametrizations,

$$\mathbf{q}(\theta, V) = \begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix} + V \begin{bmatrix} \pm\sin\theta \\ \mp\cos\theta \\ 1 \end{bmatrix}. \tag{31}$$

[5]The Gaussian curvature at a point p of a differentiable surface is given by $\kappa_1\kappa_2$ where κ_1, κ_2 are the principal curvatures at p.

It has no other sets of rulings (Exercise), so it is doubly-ruled. To see that equation (31) is a rational parametrization, we replace $\cos\theta$ and $\sin\theta$ by the rational functions,

$$\cos\theta = \frac{1-U^2}{1+U^2}, \qquad \sin\theta = \frac{2U}{1+U^2}, \qquad (U \in \mathbb{R}). \tag{32}$$

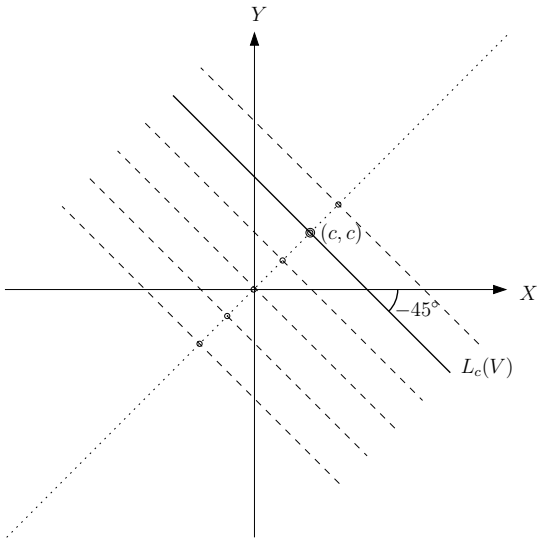


Figure 1: Rulings of the Hyperbolic Paraboloid projected to the (X, Y)-plane.

The hyperbolic paraboloid $X^2 - Y^2 - Z = 0$ also has 2 ruled parametrizations

$$\mathbf{q}(U,V) = \begin{bmatrix} U \\ \pm U \\ 0 \end{bmatrix} + V \begin{bmatrix} 1 \\ \mp 1 \\ 4U \end{bmatrix}. \tag{33}$$

This case is important in Levin's algorithm, so let us understand the parametrization. Assume $\mathbf{q}(U,V) = (U+V, U-V, 4UV)$ (the other parametrization $\mathbf{q}(U,V) = (U+V, -U+V, 4UV)$ is similarly treated). Observe that for each $c \in \mathbb{R}$, the projection of the ruling $\{\mathbf{q}(c,V) : V \in \mathbb{R}\}$ into the (X,Y)-plane is a line $L_c(V)$ passing through the point (c,c) with a constant slope $-45°$. Since the lines $L_c(V)$ are parallel and pairwise disjoint, we conclude that the rulings are pairwise disjoint.

Table 2 gives a list of 10 quadric surfaces. These are all ruled quadrics, as demonstrated by the ruled parameterization given in the last column. The proper quadrics which are not accounted for by Table 2 are the ellipsoid, 2-sheeted hyperboloid and the elliptic paraboloid.

Let us prove that these 3 quadrics are non-ruled. The ellipsoid is clearly non-ruled, since it is bounded and cannot contain any line. The 2-sheeted hyperboloid $X^2 + Y^2 - Z^2 = -1$ has parametrization

$$[\sinh U \cos V, \sinh U \sin V, 1 \pm \cosh U]$$

but this is not a ruled parametrization. In fact, this surface also cannot contain any line: This surface is disjoint from the plane $Z = 0$. So if it contain any line at all, the line must lie is some plane $Z = c$ ($|c| \geq 1$). But our hyperboloid intersects such a plane in an ellipse, which also cannot contain any line. A similar argument proves that the elliptic paraboloid $X^2 + Y^2 - Z = 0$ (which has a parametrization $[V\cos U, V \sin U, V^2]$) cannot contain any line. Combined with the parametrized quadrics listed in Table 2, we have proved:

	Name	Equation	σ	σ_u	Ruled Parametrization
	Nonsingular Surfaces				
1	Hyperboloid of 1 sheet	$X^2+Y^2-Z^2=1$	(2,2)	(2,1)	$[\cos\theta \pm V\sin\theta, \sin\theta \mp V\cos\theta, V]$
2	Hyperbolic Paraboloid	$X^2-Y^2-Z=0$	(2,2)	(1,1)	$[U+V, \pm(U-V), 4UV]$
	Singular Nonplanar Surfaces				
3	Elliptic Cone	$X^2+Y^2-Z^2=0$	(2,1)	(2,1)	$[V\cos\theta, V\sin\theta, V]$
4	Elliptic Cylinder	$X^2+Y^2=1$	(2,1)	(2,0)	$[\cos\theta, \sin\theta, V]$
5	Hyperbolic Cylinder	$X^2-Y^2=-1$	(2,1)	(1,1)	$[\frac{1}{2}(U-\frac{1}{U}), \frac{1}{2}(U+\frac{1}{U}), V]$
6	Parabolic Cylinder	$X^2=Z$	(2,1)	(1,0)	$[U, V, U^2]$
	Singular Planar Surfaces				
7	Intersecting Planes	$X^2-Y^2=0$	(1,1)	(1,1)	$[U, U, V]$
8	Parallel Planes	$X^2-1=0$	(1,1)	(1,0)	$[1, U, V]$
9	Coincident Planes	$X^2=0$	(1,0)	(1,0)	$[0, U, V]$
10	Single Plane	$X=0$	(1,1)	(0,0)	$[0, U, V]$

Table 2: Ruled Quadric Surfaces

Theorem 10.
(i) A proper quadric is non-ruled iff it does not not contain a real line, and these are the ellipsoid, 2-sheeted hyperboloid and elliptic paraboloid.
(ii) Table 2 is a complete list of all proper ruled quadrics.

We now address the form of the ruled parametrizations (30). In particular, what kind of functions arise in the definition of the base curve $\mathbf{b}(U)$ and director curve $\mathbf{d}(U)$? Each is a vector function, e.g., $\mathbf{b}(U) = (X_1(U), X_2(U), X_3(U))$ where the $X_i(U)$'s are rational functions, i.e., $X_i(U) = P_i(U)/Q_i(U)$ where $P_i(U), Q_i(U)$ are relatively prime polynomials. The degree of $\mathbf{b}(U)$ is the maximum of the degrees of the polynomials $P_i(U), Q_i(U)$, $(i = 1, 2, 3)$. Similarly for the degree of $\mathbf{d}(U)$. There are three steps in this determination:
1. By an examination of Table 2, we see that in the base curve $\mathbf{b}(U)$ and director curve $\mathbf{d}(U)$ of canonical quadrics are **quadratic** rational functions of U. Quadratic means the degree is at most 2. Note that the appearance of $\cos\theta, \sin\theta$ in the table should be interpreted as the rational functions in (32).
2. We now examine the transformation from $Q_A(\mathbf{X})$ to the canonical form $Q_C(\mathbf{X})$ where C is a sign matrix that is either diagonal or almost diagonal. We want to determine the transformation matrix N such that

$$Q_A(N\mathbf{X}) = Q_C(\mathbf{X}). \tag{34}$$

By applying our algorithm to A, we obtain (M, B) such that

$$A = M^T BM, \qquad B = \mathtt{diag}(a_1, \ldots, a_n) \quad \text{or} \quad \mathtt{diag}(a_1, \ldots, a_{n-2}; d).$$

Let

$$S = \mathtt{diag}(c_1, \ldots, c_n)$$

where $c_i = \sqrt{|a_i|}$ for $i = 1, \ldots, n$. In case of the almost diagonal matrix, we use $c_{n-1} = c_n = \sqrt{|d|}$ instead. Then we have $B = SCS$ where $C = \mathtt{diag}(s_1, \ldots, s_n)$ or $C = \mathtt{diag}(s_1, \ldots, s_{n-2}; s)$ is a sign matrix. Thus $A = (SM)^T C(SM)$ and we can choose $N = SM$ in (34).
3. Suppose that $\mathbf{x} = (x(U,V), y(U,V), z(U,V))^T$ is a parametization of the canonical quadric $Q_C(X, Y, Z) = 0$. From $C = S^{-1}(M^{-1})^T AM^{-1}S^{-1}$, we see that

$$\mathbf{q}(U,V) = M^{-1}S^{-1} \cdot \mathbf{x}$$

is a parametrization for the quadric $Q_A(X, Y, Z) = 0$. Thus:

Theorem 11. *Every ruled quadric $Q_A(X, Y, Z)$ in real affine space has a rational quadratic parametrization*

$$\mathbf{q}(U,V) = (X(U,V), Y(U,V), Z(U,V))^T$$

of degree 2.

EXAMPLE: Give the ruling for the quadric

$$Q_A : XY + 2YZ + Z - 2 = 0. \tag{35}$$

We first analyze Q_A: with

$$(X, Y) \mapsto (X' + Y', X' - Y') \tag{36}$$

we obtain

$$X'^2 - Y'^2 + 2X'Z - 2Y'Z + Z - 2 = 0.$$

With

$$(X', Y') \mapsto (X'' - Z, Y'' - Z) \tag{37}$$

we obtain

$$X''^2 - Y''^2 + Z - 2 = 0.$$

With

$$Z \mapsto -Z' + 2 \tag{38}$$

we obtain

$$X''^2 - Y''^2 - Z' = 0.$$

Hence Q_A is a hyperbolic paraboloid. One ruled parametrization of the canonical hyperbolic paraboloid is

$$\begin{bmatrix} X'' \\ Y'' \\ Z' \end{bmatrix} = \mathbf{q}(U, V) = \begin{bmatrix} U \\ U \\ 0 \end{bmatrix} + V \begin{bmatrix} 1 \\ -1 \\ 4U \end{bmatrix}. \tag{39}$$

From (36), (37) and (38), we see that

$$\begin{aligned} X'' &= (X + Y + 2Z)/2 \\ Y'' &= (X - Y + 2Z)/2 \\ Z' &= -Z + 2 \end{aligned}$$

Thus

$$\begin{aligned} U + V &= (X + Y + 2Z)/2 \\ U - V &= (X - Y + 2Z)/2 \\ 4UV &= -Z + 2 \end{aligned}$$

or

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 2U + 8UV - 4 \\ 2V \\ 2 - 4UV \end{bmatrix} \tag{40}$$

Exercises

Exercise 4.1: Consider the surface $X^2 + Y + Z = 1$. Determine the ruling of this surface. ♢

Exercise 4.2:

(i) Show that ruled parameterizations (31) are the only rulings of the canonical hyperboloid of 1-sheet. (ii) Do the same for ruled parameterizations (33) of the canonical hyperbolic paraboloid. (i) To see that this is indeed a ruling, we note that for each V $X = \cos U - V \sin U$, $Y = \sin U + V \sin U$ lies on a circle of radius $1 + V^2$ in the plane $Z = U$. So if $V \neq V'$ then the points $\mathbf{q}(U, V)$ and $\mathbf{q}(U, V')$ lies on different circles. For fixed V, we also see that (X, Y) corresponds to distinct points on the circle. (ii) We must show that if L is a line in the hyperboloid, then L is one of the rulings in our two families. ♢

Exercise 4.3: The following is another parametrization of the hyperboloid of 1 sheet: $[U+V, UV-1, U-V, UV+1]$. What is the relation to the ruled parameterization in our table? ♢

Exercise 4.4: Consider the intersection of two orthogonal right-circular cylinders along the Y- and Z-axes:

$$Q_0 = X^2 - Y^2 - 1, \quad Q_1 = Z^2 + X^2 - 1$$

Compute the projecting cone of their intersection. ♢

Exercise 4.5: Write a display program to visualize parametric space curves. For instance, display the curve given by (40) and (44). ♢

End Exercises

§5. Parametrization of Quadric Space Intersection Curves

Consider the intersection of two quadrics:

$$Q_0 = \mathbf{X}^T A_0 \mathbf{X} = 0, \qquad Q_1 = \mathbf{X}^T A_1 \mathbf{X} = 0, \tag{41}$$

with $\mathbf{X} = (X, Y, Z, 1)^T$. They intersect in a space curve that is known[6] as a **quadric space intersection curve** (QSIC). This is a **quartic**, i.e., degree 4 curve. By definitely, the degree of a space curve is the maximum number of intersection points with any plane. In this section, we describe a well-known method of parametrizing a QSIC from Levin (1976). There are some subtleties in this parametrization, however, which we will point out.

The classification of QSIC is a classic problem. We have three general binary criteria for this classification:

- A QSIC is **reducible** if it contains a component of degree 3 or less; otherwise it is **irreducible**. The linear or quadratic components can be complex, but cubic components are always real.
- A QSIC can be **planar** or **nonplanar**. A planar QSIC lies in one or two planes, and such QSIC's are necessarily reducible.
- A QSIC is **singular** if it contains a singular point. Reducible QSIC's must be singular. An irreducible and singular QSIC has one double singular point that may be classified as acnode, crunode or cusp. Such a curve is rationally parametrizable. An acnode may be the only real point of the QSIC.

A QSIC that is non-singular and irreducible is **non-degenerate** and otherwise **degenerate**. A complete classification for $\mathbb{P}^3(\mathbb{C})$ may be based on the **Segre characteristic** [6]. This classification scheme does not differentiate important morphological features in $\mathbb{P}^3(\mathbb{R})$. This is addressed in [11, 6]. Tu et al [9] considered the classification in $\mathbb{P}^3(\mathbb{R})$ by analysing the Jordan Canonical form of the pencil matrix.

If λ is a real variable, a **matrix pencil** has the form

$$R(\lambda) := (1-\lambda)A_0 + \lambda A_1 \tag{42}$$

where A_0, A_1 are symmetric real matrices. Thus, $R(\lambda)$ is a symmetric matrix whose elements are linear in λ. The parametrized family $\{\mathbf{X}^T R(\lambda)\mathbf{X} = 0 : \lambda \in \mathbb{R}\}$ of surfaces is called a **surface pencil**. We say the pencil is **trivial** if there is a constant matrix R_0 such that $R(\lambda) = \lambda R_0$. When $R(\lambda)$ is trivial, its surface pencil has only one surface, $\mathbf{X}^T R_0 \mathbf{X} = 0$.

Let C be the QSIC of the two surfaces (41). In $\mathbb{A}^3(\mathbb{R})$, C may have empty locus. When it is non-empty, it also known as the **base curve** of the corresponding pencil. Every member of the surface pencil contains the base curve: for if $\mathbf{X} \in C$ then $\mathbf{X}^T A_i \mathbf{X} = 0$ for $i = 0, 1$ and so $\mathbf{X}^T((1-\lambda)A_0 + \lambda A_1)\mathbf{X} = 0$. The intersection of any two distinct surfaces in the pencil is equal to C (Exercise).

[6] It seems as reasonable to read "QSIC" as quadric *surface* intersection curve.

Can C be empty in $\mathbb{A}^3(\mathbb{C})$? To understand this, we consider a new diagonalization problem. We say that a pair (A_0, A_1) of symmetric matrices is **simultaneously diagonalizable** if there exists an invertible matrix M such that

$$M^T A_0 M = \mathbf{1}, \qquad M^T B_0 M = \mathtt{diag}(\lambda_1, \ldots, \lambda_n) \tag{43}$$

for some $\lambda_1, \ldots, \lambda_n$.

Theorem 12. *Let A_0, A_1 be symmetric matrices of order n. If A_0 is non-singular and $\det(zA_0 - A_1)$ has n distinct roots $\lambda_1, \ldots, \lambda_n$, then (A_0, A_1) is simultaneously diagonalizable.*

Proof. By Theorem 5, we can choose M_0 such that $M_0^T A_0 M_0 = \mathbf{1}$. The matrix M_0 may be complex. Then $M_0^T(zA_0 - A_1)M_0 = z\mathbf{1} - A_1'$ where $A_1' = M_0^T A_1 M_0$. But

$$\det(z\mathbf{1} - A_1') = \det(M_0^T(zA_0 - A_1)M_0) = \det(M_0)^2 \det(zA_0 - A_1)$$

and thus the eigenvalues of A_1' are the roots of $\det(zA_0 - A_1)$. These are, by assumption, are distinct and equal to $\lambda_1, \ldots, \lambda_n$. Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be the associated eigenvectors: $A_1'\mathbf{x}_i = \lambda_i \mathbf{x}_i$ and we may assume the scalar product $\mathbf{x}_i^T \mathbf{x}_i = 1$ for all i. Putting $M_1 = [\mathbf{x}_1 | \cdots | \mathbf{x}_n]$, we obtain $A_1' M_1 = M_1 \mathtt{diag}(\lambda_1, \ldots, \lambda_n)$. Note that

$$\begin{aligned}
\lambda_i \mathbf{x}_i^T \mathbf{x}_j &= \mathbf{x}_j^T(\lambda_i \mathbf{x}_i) \\
&= \mathbf{x}_j^T A_1' \mathbf{x}_i \\
&= (\mathbf{x}_j^T A_1' \mathbf{x}_i)^T \\
&= \mathbf{x}_i^T (A_1')^T \mathbf{x}_j \\
&= \mathbf{x}_i^T (A_1' \mathbf{x}_j) \\
&= \lambda_j \mathbf{x}_i^T \mathbf{x}_j.
\end{aligned}$$

If $i \neq j$, the equality $\lambda_i \mathbf{x}_i^T \mathbf{x}_j = \lambda_j \mathbf{x}_i^T \mathbf{x}_j$ implies that $\mathbf{x}_i^T \mathbf{x}_j = 0$. Thus $M_1^T M_1 = \mathbf{1}$, and $M_1^T A_1' M_1 = \mathtt{diag}(\lambda_1, \ldots, \lambda_n)$.

$$\begin{aligned}
M_1^T M_0^T (zA_0 - A_1) M_0 M_1 &= M_1^T (z\mathbf{1} - A_1') M_1 \\
&= z\mathbf{1} - \mathtt{diag}(\lambda_1, \ldots, \lambda_n).
\end{aligned}$$

The matrix $M = M_0 M_1$ achieves the desired simultaneous diagonalization of (A_0, A_1). **Q.E.D.**

The approach of Levin can be broken into two observations:

- The first observation is that it is easy to intersect a ruled quadric with a general quadric. We just plug in a ruling $\mathbf{b}(U) + V\mathbf{d}(U)$ of the ruled quadric into the general quadric $Q(X, Y, Z) = 0$ to obtain $Q'(U, V)$. Since Q' is of degree 2 in V, we can solve for V in terms of U. Thus the QSIC is parametrized by U.
- The second observation is that in any non-trivial pencil there is a ruled quadric surface.

EXAMPLE. To illustrate the first observation, recall the hyperboloid paraboloid (35) whose ruled parametrization is given by (40). Suppose we want to intersect the ruled surface with the sphere,

$$Q_B : X^2 + Y^2 + Z^2 - 3 = 0.$$

We might observe that this intersection is non-empty since $(-1, 1, 1)$ lies on both quadrics. Plugging the ruled parametrization into the second quadric:

$$\begin{aligned}
X^2 + Y^2 + Z^2 - 3 &= (2U + 8UV - 4)^2 + (2V)^2 + (2 - 4UV)^2 - 3 \\
&= 4U^2 + 64U^2V^2 + 16 + 32U^2V - 16U - 64UV + 4V^2 + 4 - 16UV + 16U^2V^2 - 3 \\
&= (80U^2 + 4)V^2 + (32U^2 - 80U)V + (4U^2 - 16U + 17) \\
&= 4(20U^2 + 1)V^2 + 16(2U^2 - 5U)V + (4U^2 - 16U + 17) \\
&= 0.
\end{aligned}$$

Solving for V:

$$\begin{aligned} V &= \frac{1}{4(20U^2+1)}\left(8(5U-2U^2) \pm \sqrt{8^2(2U^2-5U)^2 - 4(20U^2+1)(4U^2-16U+17)}\right) \\ &= \frac{1}{2(20U^2+1)}\left(4(5U-2U^2) \pm \sqrt{16(2U^2-5U)^2 - (20U^2+1)(4U^2-16U+17)}\right) \\ &= \frac{1}{2(20U^2+1)}\left(4(5U-2U^2) \pm \sqrt{-16U^4+56U^2-16U+17}\right) \end{aligned} \tag{44}$$

To prove the second observation, we follow the "geometric argument" in [11, Appendix]. In fact, this argument is quite general and applies to a quadric in any dimension. We first do this argument in projective space.

There is a preliminary concept. By definition, the locus of a line L in $\mathbb{P}^n(\mathbb{C})$ is a set of points of the form $\{a\mathbf{X}_0 + b\mathbf{X}_1 : (a,b) \in \mathbb{P}^1(\mathbb{C})\}$ where $\mathbf{X}_0, \mathbf{X}_1 \in \mathbb{P}^n(\mathbb{C})$ are distinct points. We say the line is **real** if the restriction of its locus to $\mathbb{A}^n(\mathbb{R})$ is a line in the usual sense. For instance, if $\mathbf{X}_0 = (\mathbf{i}, 0, 1, 1)^T$ and $\mathbf{X}_1 = (1, 1, 0, 1)^T$ then L is not a real line: we may verify that $a\mathbf{X}_0 + b\mathbf{X}_1$ is non-real for all $a, b \in \mathbb{C} \setminus \{0\}$.

Theorem 13. *If $R(\lambda) = \lambda A_0 + (1-\lambda)A_1$ is a non-trivial pencil, there exists a $\lambda_0 \in \mathbb{R}$ such that the $R(\lambda_0)$ is a ruled quadric surface.*

Proof. Our first task is to find a real line L that passes through two points $\mathbf{X}_0, \mathbf{X}_1$ (not necessarily real) in the intersection of the quadrics. If the intersection has two real points then the line passing through them will be our L. Suppose the intersection does not have two real points. Pick a complex point $\mathbf{X}_0 = \mathbf{U} + \mathbf{i}\mathbf{V}$ in the intersection such that $\mathbf{U}, \mathbf{V}$ are distinct real vectors. Then $\mathbf{X}_1 = \mathbf{U} - \mathbf{i}\mathbf{V}$ is also in the intersection (using the fact that for any real polynomial $Q(X, Y, Z)$, we have $Q(x, y, z) = 0$ iff $Q(\overline{x}, \overline{y}, \overline{z}) = 0$ where $x, y, z \in \mathbb{C}$ and $\overline{x} = u - \mathbf{i}v$ is the complex conjugate of $x = u + \mathbf{i}v$. But note that the line L through $\mathbf{X}_0, \mathbf{X}_1$ contains two real points $\mathbf{U} = (\mathbf{X}_0 + \mathbf{X}_1)/2$ and $\mathbf{V} = (\mathbf{X}_0 - \mathbf{X}_1)/(2\mathbf{i})$. We conclude L is a real line.

Next pick a real point $\mathbf{X}^* = a\mathbf{X}_0 + b\mathbf{X}_1$ on L that is distinct from $\mathbf{U}, \mathbf{V}$. We may assume $a = b = 1$. Our goal is to show that $\mathbf{X}^*$ lies on some surface $Q_S = 0$ where $S = \lambda_0 A_0 + \lambda_1 A_1$ and $\lambda_0, \lambda_1 \in \mathbb{R}$. Observe that

$$\begin{aligned} \mathbf{X}^{*T}(\lambda_0 A_0 + \lambda_1 A_1)\mathbf{X}^* &= \mathbf{X}_0^T(\lambda_0 A_0 + \lambda_1 A_1)\mathbf{X}_1 + \mathbf{X}_1^T(\lambda_0 A_0 + \lambda_1 A_1)\mathbf{X}_0 \\ &= \lambda_0(\mathbf{X}_0^T A_0 \mathbf{X}_1 + \mathbf{X}_1^T A_0 \mathbf{X}_0) + \lambda_1(\mathbf{X}_0^T A_1 \mathbf{X}_1 + \mathbf{X}_1^T A_1 \mathbf{X}_0) \\ &= \lambda_0 d_0 + \lambda_1 d_1. \end{aligned}$$

If $d_0 = d_1 = 0$ then $\mathbf{X}^*$ lies on the surface $Q_S = 0$ for for all $\lambda_0, \lambda_1 \in \mathbb{R}$. Otherwise, say $d_1 \neq 0$, and we may choose

$$(\lambda_0, \lambda_1) = (1, -d_0/d_1).$$

We need to verify that λ_0, λ_1 are real. It suffices to show that d_0, d_1 are real. This is clear when $\mathbf{X}_0, \mathbf{X}_1$ are real; otherwise, it follows from

$$d_i = (\mathbf{X}_0 + \mathbf{X}_1)^T A_i (\mathbf{X}_0 + \mathbf{X}_1) = (2\mathbf{U})^T A_i (2\mathbf{U}).$$

The surface $Q_S = 0$ thus contains three points X_0, X_1, X^* of the line L. The intersection of any quadric with a line has at most two points, unless the line is contained in the quadric. We conclude that the surface R contains L. By Theorem 10, Q_S is ruled. **Q.E.D.**

This general theorem gives us more than we need: in case the two surfaces has no real intersection, our application do not really care to find a ruled surface in the pencil. But we can obtain more restrictive conditions on the nature of the ruled quadric in the pencil, essentially in the form that Levin actually proved [7, Appendix]:

Theorem 14. *The intersection of two real affine quadric surfaces lies in a hyperbolic paraboloid, a cylinder (either hyperbolic or parabolic), or a degenerate planar surface.*

In outline, Levin's method first finds a ruled quadric S in the pencil of A_0, A_1. The points of S are rationally parametrized by $\mathbf{q}(U,V) \in \mathbb{R}^3$. For any fixed u_0, $\mathbf{q}(u_0,V)$ is a line parametrized by V. Plugging $\mathbf{q}(u_0,V)$ into one of the input quadrics, say $Q_0(X,Y,Z)$, we obtain a quadratic equation in V. Solving this equation, we obtain $v_0 = V(u_0)$ as an closed expression involving a square root. Thus we have found one point $\mathbf{q}(u_0, V(u_0))$ (possibly an imaginary one) on the intersection curve. Note that this method is suitable for display of the intersection curve. Thus, the QSIC is parametrized by the U-parameter. We shall show that the parametrization of the QSIC takes the form:

$$\mathbf{p}(U) = \mathbf{a}(U) \pm \mathbf{d}(U)\sqrt{s(U)} \tag{45}$$

where the bold face letters indicate vector valued polynomials and $s(U)$ is a degree 4 polynomial.

Any ruled surface S in a pencil is called **parametrization surface** of the pencil. It has the parameterization $S = \{\mathbf{q}(u,v) : u, v \in \mathbb{R}\}$ given by

$$\mathbf{q}(u,v) = \mathbf{b}(u) + v\mathbf{d}(u)$$

where $\mathbf{b}(u)$ is the base curve and $\mathbf{d}(u)$ the director curve. Recall that we showed for a ruled quadric, $\mathbf{b}(u)$ and $\mathbf{d}(u)$ are rational functions of degree ≤ 2 in u. Levin calls u the primary parameter and v the secondary parameter. For a fixed u_0, $\mathbf{q}(u_0,v)$ is a line in S. Thus the intersection of S with the surface $Q_0 = 0$ is reduced to the intersection of a ruling $\mathbf{q}(u_0,v)$ with the surface $Q_0 = 0$. [If S coincides with Q_0, then use Q_1 instead.] As noted in the outline above, this intersection can be solved numerically by plugging $\mathbf{q}(u_0,v)$ into the quadratic equation $Q_0 = 0$, yielding a quadratic equation in the secondary parameter. Solving, we obtain a value $v = v(u_0)$. But we can also find v symbolically, simply by plugging $\mathbf{q}(u,v)$ into the equation $Q_0 = 0$. This gives

$$c_2 v^2 + 2c_1 v + c_0 = 0 \tag{46}$$

where

$$c_2 = c_2(u) = \mathbf{d}(u)^T A_0 \mathbf{d}(u), \quad c_1 = c_1(u) = \mathbf{b}(u)^T A_0 \mathbf{d}(u), \quad c_0 = c_0(u) = \mathbf{b}(u)^T A_0 \mathbf{b}(u)$$

Let

$$v_0(u) = \frac{-c_1 \pm \sqrt{s}}{c_2}$$

where $s = c_1^2 - c_2 c_0$. Thus we obtain

$$\mathbf{p}(u) = \mathbf{q}(u, v_0(u))$$

which has the form (45) above. This point $\mathbf{p}(u)$ might be imaginary and there are also two possible values for $v(u_0)$.

The algorithm [7, p. 560] goes as follows: first compute the real roots λ_i $(i = 1,2,3,4)$ of the equation $R(\lambda) = \det(A_0 - \lambda A_1) = 0$. If for any i, $R(\lambda_i)$ is improper, then there is no intersection. Otherwise, $R(\lambda_i)$ corresponds to a ruled surface for some i. We may now proceed to intersect this surface with one of the original surface. Since λ_i is an algebraic number, this computation will require careful approximation in the sense of Exact Geometric Computation. Details of this process has not been analyzed.

Exercises

Exercise 5.1: If $\lambda \neq \lambda'$ then the intersection of the surfaces $\mathbf{X}^T R(\lambda)\mathbf{X} = 0$ and $\mathbf{X}^T R(\lambda')\mathbf{X} = 0$ is equal to the base curve. ♢

Exercise 5.2: Let C be the intersection of the quadrics in (41). Assume A_0, A_1 have rank 3 or 4. For each of the spaces $S = \mathbb{P}^3(\mathbb{C}), \mathbb{P}^3(\mathbb{R}), \mathbb{A}^3(\mathbb{C}), \mathbb{A}^3(\mathbb{R})$, either show that C is non-empty, or give an instance where C is empty. ♢

Exercise 5.3: Show that the line L with locus $\{a\mathbf{X}_0 + b\mathbf{X}_1 : a, b \in \mathbb{P}^1(\mathbb{C})\}$ where $\mathbf{X}_0 = (\mathbf{i}, 0, 1)^T$ and $\mathbf{X}_1 = (1,1,0)^T$ is not a real line. ♢

Exercise 5.4: Give an algebraic version and proof of Theorem 13. HINT: The roots of the pencil $R(\lambda)$ are continuous function of λ. ♢

End Exercises

§6. Morphology of Degenerate QSIC

We now describe the approach of Farouki, Neff and O'Connor [6]. Their focus is on how to parametrize the degenerate QSIC. This decision is reasonable because in the non-degenerate case, Levin's method is adequate.

If the QSIC is a non-singular irreducible QSIC, we say it is **non-degenerate**; all other cases are **degenerate**. The degenerate QSIC's are of 5 types:

1. a singular quartic space curve
2. a cubic space curve and a straight line
3. two irreducible conic curves
4. an irreducible conic curve and two straight lines
5. four straight lines

The first case is irreducible, but the rest are reducible. In any case, we have

Theorem 15. *A QSIC is rationally parametrizable iff it is degenerate.*

Hence, it is useful to detect degeneracy and to exploit this parametrization property.

For any pencil $R(\lambda) = \lambda A_0 + (1-\lambda)A_1$, let

$$\Delta(\lambda) = \det(R(\lambda)) = \sum_{i=0}^{4} \Delta_i \lambda^i$$

be the **determinant** of the pencil $R(\lambda)$. Recall that the discriminant of any univariate polynomial $p(X)$ is denoted $\mathrm{disc}(p)$ and it defined as a resultant

$$\mathrm{disc}(p) := \mathtt{res}(p(X), p'(X))$$

where $p'(X)$ indicates differentiation by X. The **discriminant** of the pencil $R(\lambda)$ is defined to be $\mathrm{disc}(\Delta(\lambda))$.

Lemma 16. *A QSIC is degenerate iff* $\mathrm{disc}(\Delta(\lambda))$ *vanishes.*

Proof. **Q.E.D.**

For a generic point $\mathbf{X} = (X, Y, Z)^T$, and real variable t, consider the parametric line $t\mathbf{X} = (tX, tY, tZ)^T$ that passes through the origin and $\mathbf{X}$. If $t\mathbf{X}$ lies on the quadric Q_i, then we have $Q_i(tX, tY, tZ) = 0$. Now consider the surface defined by

$$R(X, Y, Z) = \mathtt{res}_t(Q_0(t\mathbf{X}), Q_1(t\mathbf{X})) = 0. \tag{47}$$

By the usual interpretation of resultants, this means for any point $P = (X, Y, Z)^T$ in the surface $R(X, Y, Z) = 0$, there exists a $t \in \mathbb{R}$ such that tP lies in the QSIC. In view of this, $R(X, Y, Z)$ is called the **projection cone** of the QSIC.

EXAMPLE. Compute the projection cone in the case of a sphere

$$Q_0 : X^2 + Y^2 + Z^2 = 4$$

and a cylinder

$$Q_1 : X^2 + Y^2 = 2X.$$

$$\begin{aligned} R(X,Y,Z) &= \det \begin{bmatrix} X^2+Y^2+Z^2 & & -4 & \\ & X^2+Y^2+Z^2 & & -4 \\ X^2+Y^2 & -2X & & \\ & X^2+Y^2 & -2X & \end{bmatrix} \\ &= -4 \det \begin{bmatrix} X^2+Y^2+Z^2 & & -4 \\ X^2+Y^2 & -2X & \\ & X^2+Y^2 & -2X \end{bmatrix} \\ &= -4[4X^2(X^2+Y^2+Z^2) - 4(X^2+Y^2)^2] \\ &= 16[Y^4 + X^2Y^2 - X^2Z^2]. \end{aligned} \tag{48}$$

In this example from Farouki et al [6], the QSIC is a figure-of-8, which is singular at the point $(2,0,0)$ where the two loops of the "8" meet.

Lemma 17. *$R(X,Y,Z) \in \mathbb{R}[X,Y,Z]$ is homogeneous of degree 4.*

Proof. The proof is like in the proof of Bezout's theorem. Introduce a new variable s as follows:

$$\begin{aligned} R(sX,sY,sZ) &= \det \begin{bmatrix} s^2a_2 & sa_1 & a_0 & \\ & s^2a_2 & sa_1 & a_0 \\ s^2b_2 & sb_1 & b_0 & \\ & s^2b_2 & sb_1 & b_0 \end{bmatrix} \\ s^2R(sX,sY,sZ) &= \det \begin{bmatrix} s^3a_2 & s^2a_1 & sa_0 & \\ & s^2a_2 & sa_1 & a_0 \\ s^3b_2 & s^2b_1 & sb_0 & \\ & s^2b_2 & sb_1 & b_0 \end{bmatrix} \\ &= s^6 \det \begin{bmatrix} a_2 & a_1 & a_0 & \\ & a_2 & a_1 & a_0 \\ b_2 & b_1 & b_0 & \\ & b_2 & b_1 & b_0 \end{bmatrix}. \end{aligned}$$

From $R(sX,sY,sZ) = s^4R(X,Y,Z)$, we conclude that R is homogeneous of degree 4. **Q.E.D.**

Suppose the plane curve $R(X,Y,1) = 0$ has a rational parametrization. If $(X(U), Y(U))$ is a parametrization of the plane curve $R(X,Y,1)$, then we can obtain a parametrization of the QSIC by plugging $(tX(U), tY(U), t)^T$ into any one of the original curve, say $Q_A(\mathbf{X}) = 0$. This gives a quadratic polynomial in t, which we can solve, say $t = t(U)$. This gives a parametrization of the QSIC of the form

$$(t(U)X(U), t(U)Y(U), t(U))^T.$$

We should also find a parametrization of the plane curve $R(X,Y,0)$ and repeat this procedure to get a complete parametrization of the QSIC.

EXAMPLE (contd). In the projection cone $R(X,Y,Z) = Y^4 + X^2Y^2 - X^2Z^2 = 0$ from (48), we prefer to set $Y = 0$ and $Y = 1$. The curve $R(X,1,Z)$ has parametrization $Z = \pm X\sqrt{X^2+1}$) I.e., we plug $(tU, t, \pm U\sqrt{U^2+1})^T$ into the cylinder $X^2 + Y^2 = 2X$ to get

Errors! Also
set $Z = 1$ i
OK

$$t^2U^2 + t^2 = 2tU$$

thus $t = 2U/(1+U^2)$. Thus the QSIC becomes

$$(2U^2/(1+U^2), 2U/(1+U^2), \pm U\sqrt{U^2+1})^T$$

Similarly, the curve $R(X, 0, Z)$ has parametrization $Z = \pm X\sqrt{X^2 - 1)}$. Then $t = 2/U$, and the QSIC is

$$(2, 2/U, \pm U\sqrt{U^2 - 1})^T$$

ALTERNATIVE APPROACH: The sphere is

$$Q_A : X^2 + Y^2 + Z^2 = 4$$

but we may reexpress the cylinder $Q_B : X^2 + Y^2 = 2X$ as

$$Q_B : (X - 1)^2 + Y^2 = 1$$

The ruled parametrization of Q_B is $(X, Y, Z) = (1+\cos\theta, \sin\theta, V)$. So plugging into Q_A, we get $2X + Z^2 = 4$ or $V = \sqrt{2 - 2\cos\theta}$. Let

$$P(\theta) = (1 + \cos\theta, \sin\theta, \pm\sqrt{2 - 2\cos\theta})^T.$$

E.g. $P(0) = (2, 0, 0)$. This is a singular point of the QSIC.

Since R is homogeneous, it is basically equivalent to the bivariate polynomial

$$f(X, Y) = \Phi(X, Y, 1).$$

We can now factor $f(X, Y)$ using elementary techniques. First consider what the linear factors of $f(X, Y)$ give us: ...

§7. Approach of Dupont, Lazard, et al

The ruled quadrics are given one of the canonical forms:

	Equation	Parametrization
(2,2)	$aX^2 + bY^2 - cZ^2 - dW^2 = 0$	$\left[\frac{u+av}{a}, \frac{uv-b}{b}, \frac{u-av}{\sqrt{ac}}, \frac{uv+b}{\sqrt{bd}}\right]$
(2,1)	$aX^2 + bY^2 - cZ^2 = 0$	$\left[uv, \frac{u^2-abv^2}{2b}, \frac{u^2+abv^2}{2\sqrt{bc}}, s\right]$
(2,0)	$aX^2 + bY^2 = 0$	$[0, 0, u, v]$
(1,1)	$aX^2 - bY^2 = 0$	$\left[u, \pm\frac{ab}{b}u, v, s\right]$
(1,0)	$X^2 = 0$	$[0, u, v, s]$

REMARKS: An different approach to intersecting quadrics is proposed in [8]: by the simultaneous transformation of two quadrics into a canonical form in which their intersection is easily computed. See [5, Chapter 31] for discussion of this approach.

§8. Steiner Surfaces

A **rational parametrization** of a surface S is a map $\mathbf{q} : \mathbb{R}^2 \to \mathbb{R}^3$ of the form

$$\mathbf{q}(U, V) = (x(U, V), y(U, V), z(U, V))^T$$

where x, y, z rational functions in U, V such that the locus of S is, with finitely many exceptions, equal to $\{\mathbf{q}(U, V) : U, V \in \mathbb{R}\}$. The **degree** of this rational parametrization is the maximum degree of the rational functions. The rational parametrization is **faithful** if, with finitely many exceptions, $\mathbf{q}(U, V)$ is 1-1 on the surface.

A surface with rational parametrization of degree 2 is called a **Steiner surface**. Such surfaces are a generalization of quadrics, since we know that every quadric has a rational paramerization of degree 2. It is to see that Steiner surfaces have algebraic degree at most 4: consider the polynomials,

$$x_1(U, V)X - x_2(U, V), y_1(U, V)Y - y_2(U, V), z_1(U, V)Z - z_2(U, V)$$

where $x(U, V) = x_1(U, V)/x_2(U, V)$, etc, and the $x_i(U, V)$, etc, are polynomials of degree ≤ 2.

Exercises

Exercise 8.1: The Segre Characteristic is based on the theory of invariants. Suppose λ_i $(i = 1, \ldots, k)$ are the distinct complex eigenvalues of a matrix Q. Let λ_i have multiplicity $\mu_i \geq 1$. We can refine this notion of multiplicity as follows: let λ_i have multiplicity $\mu_{i,j}$ in... ♢

Exercise 8.2: A set of real symmetric matrices $\{A_i\}_i$ is simultaneously diagonalizable by an orthogonal matrix iff they are pairwise commutative, $A_i A_j = A_j A_i$. ♢

End Exercises

§9. Cut Curves of Quadric Surfaces

Consider the surfaces defined by $p, q \in \mathbb{Z}[X, Y, Z]$ where

$$\begin{aligned} p &= Z^2 + p_1 Z + p_0 \\ q &= Z^2 + q_1 Z + q_0 \end{aligned}$$

where $p_i, q_i \in \mathbb{Z}[X, Y]$. In case of quadric surfaces, $\deg(p_i) = 2 - i$, but part of this analysis does not require this restriction.

The **cut curve** of these two surfaces is the plane curve defined by the polynomial

$$\begin{aligned} f &= \mathtt{res}_Z(p, q) && (49)\\ &= \begin{bmatrix} 1 & p_1 & p_0 & 0 \\ 0 & 1 & p_1 & p_0 \\ 1 & q_1 & q_0 & 0 \\ 0 & 1 & q_1 & q_0 \end{bmatrix} && (50)\\ &= (p_0 q_1 - p_1 q_0) \cdot (q_1 - p_1) + (p_0 - q_0)^2 && (51)\\ &= (p_0 q_1 - p_1 q_0) \cdot \mathtt{psc}_1(p, q; Z) + (p_0 - q_0)^2 && (52) \end{aligned}$$

Our main goal is to analyze the singular points of this curve.

Let $(a, b) \in \mathtt{Zero}(f)$. We call this a **top/bot point** if there exist $c \neq c'$ such that $\{(a, b, c), (a, b, c')\} \subseteq \mathtt{Zero}(p, q)$. We call it a **genuine point** if there exist c such that (a, b, c) is a tangential intersection of $p = 0$ and $q = 0$.

Theorem 18. *(i) Every singularity of $f = 0$ is either a top/bot point or a genuine point.*
(ii) Conversely, every top/bot point or genuine point is a singularity of $f = 0$.

Proof. (i) Without loss of generality, assume the singularity of f is $\mathbf{0} = (0, 0)$.

Note that if $\mathtt{psc}_1(p, q; Z)(\mathbf{0}) = 0$, then $p(0, 0, Z)$ and $q(0, 0, Z)$ has at least two roots in common, and so $\mathbf{0}$ is a top/bot singularity. Hence assume $\mathtt{psc}_1(p, q; Z) = (q_1 - p_1)$ does not vanish at $\mathbf{0}$.

To show that $\mathbf{0}$ is genuine amounts to showing that the tangent planes

$$p_X(\mathbf{0})X + p_Y(\mathbf{0})Y + p_Z(\mathbf{0})Z = 0, \qquad q_X(\mathbf{0})X + q_Y(\mathbf{0})Y + q_Z(\mathbf{0})Z = 0$$

are the same. [Note $\mathbf{0}$ here is $(0, 0, 0)$.] This amounts to

$$0 = (p_Y q_Z - p_Z q_Y)(\mathbf{0}) = (p_X q_Z - p_Z q_X)(\mathbf{0}) = (p_Y q_X - p_X q_Y)(\mathbf{0}). \tag{53}$$

We check that

$$\begin{aligned} p_0 &= p|_{Z=0}, \qquad & p_1 &= p_Z|_{Z=0}, \\ q_0 &= q|_{Z=0}, \qquad & q_1 &= q_Z|_{Z=0} \end{aligned}$$

Therefore

$$\begin{aligned} f &= \mathtt{res}_Z(p,q) \\ &= (pq_Z - p_Z q)|_{Z=0}\mathtt{psc}_1(p,q;Z) + (p-q)^2|_{Z=0} \end{aligned}$$

Now, using the fact that $(p_X)|_{Z=0} = (p|_{Z=0})_X$, we can compute

$$\begin{aligned} f_X &= ((pq_Z - p_Z q)_X)|_{Z=0}\mathtt{psc}_1(p,q;Z) + (pq_Z - p_Z q)|_{Z=0}(\mathtt{psc}_1(p,q;Z))_X + (2(p-q)\cdot(p_X - q_X)|_{Z=0} \\ &= ((pq_Z - p_Z q)_X)|_{Z=0}\mathtt{psc}_1(p,q;Z) \\ &= (p_X q_Z - p_Z q_X)|_{Z=0}\mathtt{psc}_1(p,q;Z). \end{aligned}$$

By assumption, $f_X(\mathbf{0}) = 0$ and so we conclude that

$$(p_X q_Z - p_Z q_X)|_{Z=0}(\mathbf{0}) = 0.$$

This is the same as

$$(p_X q_Z - p_Z q_X)(\mathbf{0}) = 0,$$

as required by (53). In the same way, we show

$$(p_Y q_Z - p_Z q_Y)(\mathbf{0}) = 0.$$

There is one more case in (53), namely

$$(p_Y q_X - p_X q_Y)(\mathbf{0}) = 0.$$

This requires a slightly different argument, left as an exercise.

(ii) Now we show that if p is a top/bot point, then it is a singularity of $f = 0$. Again we assume $p = (0,0)$.

Q.E.D.

Singularities of Projection of QSIC Note that none of the arguments so far required any bounds on the degrees of p_i, q_i's. The following arguments will now use the fact that p and q are quadratic. Again, let f be their cut curve.

Assume f is square free and generally aligned (i.e., its leading coefficients in X and in Y are constants). Then we claim that the cutcurve $f = 0$ has at most 2 top/bot points and at most 4 genuine points.

Theorem 19. *There are at most two top/bot points. We can compute two quadratic polynomials $t \in \mathbb{Z}[X]$ and $b \in \mathbb{Z}[Y]$ such that the top/bot points lies in the grid of zeros of t, b.*

References

[1] W. H. Breyer. *CRC Standard Mathematical Tables*. CRC Press, Boca Raton, Florida, 28 edition, 1987.

[2] T. J. I. Bromwich. The classification of quadrics. *Transactions of the American Mathematical Society*, 6(3):275–285, 1905.

[3] R. S. Burington. A classification of quadrics in affine n-space by means of arithmetic invariants. *Amer. Math. Monthly*, 39(9):527–532, 1932.

[4] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics. In *ACM Symp. on Comp. Geometry*, volume 19, pages 246 – 255, 2003.

[5] G. Farin, J. Hoschek, and M.-S. Kim, editors. *Handbook of Computer Aided Geometric Design*. North-Holland, Amsterdam, 2002.

[6] R. Farouki, C. Neff, and M. O'Connor. Automatic parsing of degenerate quadric-surface intersections. *ACM Trans. on Graphics*, 8(3):174–203, 1989.

[7] J. Z. Levin. A parametric algorithm for drawing pictures of solid objects composed of quadric surfaces. *Comm. of the ACM*, 19(10):555–563, 1976.

[8] S. Ocken, J. T. Schwartz, and M. Sharir. Precise implementation of CAD primitives using rational parameterization of standard surfaces. In J. Hopcroft, J. Schwartz, and M. Sharir, editors, *Planning, Geometry, and Complexity of Robot Motion*, pages 245–266. Ablex Pub. Corp., Norwood, NJ, 1987.

[9] C. Tu, W. Wang, and J. Wang. Classifying the intersection curve of two quadric surfaces, 2004. To appear.

[10] W. Wang, R. Goldman, and C. Tu. Computing quadric surface intersections based on an analysis of plane cubic curves. *Graphical Models*, 64:335–367, 2003.

[11] W. Wang, R. Goldman, and C. Tu. Enhancing Levin's method for computing quadric-surface intersection. *Computer Aided Geom. Design*, 20:401–422, 2003.