Self-Validated Numerical Methods and Applications

Jorge Stolfi
Instituto de Computação, Universidade Estadual de Campinas (UNICAMP)

Luiz Henrique de Figueiredo
Laboratório Nacional de Computação Científica (LNCC)

Preface

This monograph is a set of course notes written for the 21^{st} Brazilian Mathematics Colloquium held at IMPA in July 1997. It gives an overview of the field of *self-validated numerics*—computation models in which approximate results are automatically provided with guaranteed error bounds. We focus, in particular, on two such models: *interval arithmetic* and *affine arithmetic*.

Interval arithmetic (IA) was developed in the 1960's by Ramon E. Moore [**42**]. IA is the simplest and most efficient of all validated numerics models, and, not surprisingly, the most widely known and used. After two decades of relative neglect, IA has been enjoying a strong and steady resurgence, driven largely by its successful use in all kinds of practical applications. We are confident that many readers of this monograph will find IA to be a useful tool in their own work as well.

Affine arithmetic (AA) is a more complex and expensive computation model, designed to give tighter and more informative bounds than IA in certain situations where the latter is known to perform poorly. The AA model was proposed and developed recently by the authors [**10–12**], although a similar model had been developed in 1975 by E. R. Hansen [**20**]. Apart from its usefulness for certain special applications, AA is being presented here as an example of the many topics for research that are still unexplored in the field of self-validated numerical methods.

We apologize to the reader for the length and verbosity of these notes but, like Pascal,[1] we didn't have the time to make them shorter.

[1] "Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte." —Blaise Pascal, *Lettres Provinciales*, XVI (1657).

Acknowledgements

We thank the Organizing Committee of the 21st Brazilian Mathematics Colloquium for the opportunity to present this course.

We wish to thank João Comba, who helped implement a prototype affine arithmetic package in Modula-3, and Marcus Vinicius Andrade, who helped debug the C version and wrote an implicit surface ray-tracer based on it. Ronald van Iwaarden contributed an independent implementation of AA, and investigated its performance on branch-and-bound global optimization algorithms. Douglas Priest and Helmut Jarausch provided code and advice for rounding mode control.

We wish to thank also Sergey P. Shary and Lyle Ramshaw for valuable comments and references, and Paulo Correa de Mello for the use of his computing equipment.

The authors' research has been supported over the years partly by grants from Brazilian research funding agencies (CNPq, CAPES, FAPESP, FAPERJ), and by the State University of Campinas (UNICAMP), the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), the Institute for Pure and Applied Mathematics (IMPA), the National Laboratory for Scientific Computation (LNCC), and Digital's Systems Research Center (DEC SRC) in Palo Alto.

Figures 4.7 and 4.9 were generated with *Geomview* [**17**]. Figure 4.1 was generated with *xfarbe* [**48**]. Figures 4.2 and 4.3 were kindly provided by Luiz Velho. The other figures were generated with software written by the authors.

Jorge Stolfi
Instituto de Computação, UNICAMP
Caixa Postal 6176
13083-970 Campinas, SP, Brazil
`stolfi@dcc.unicamp.br`

Luiz Henrique de Figueiredo
Lab. Nacional de Computação Científica
Rua Lauro Müller 455
22290-160 Rio de Janeiro, RJ, Brazil
`lhf@lncc.br`

May 1997

Contents

Chapter 1

Introduction

1.1 Approximate computations

Many numerical computations, especially those concerned with modeling physical phenomena, are inherently *approximate*: they will not deliver the "true" exact values of the target quantities, but only some values that are in some sense "near" the true ones.

Approximate numerical computation has been an essential tool of science and technology for several centuries. The history of the field is actually as long as that of science itself: the ancient Babylonians were already computing reciprocals and square roots as truncated sexagesimal fractions [**36, 45**]. One of Archimedes's best-known endeavors was a numerical approximation algorithm for π. The greatest mathematicians of modern history, such as Newton and Gauss, were deeply concerned with this field. The first electronic computers were expressly designed for numerical computing, and that application is still an overriding concern in the design of modern CPU chips.

1.1.1 Error analysis

The difference between a computed value and the "true" value of the corresponding quantity is commonly called the *error* of that computed value.

Some sources of error are external to the computation: the inputs may have been contaminated by measurement error or missing data, or the computation may be based on a simplified mathematical model that later proves to be inadequate.

Other sources of error are internal, due to the discrete nature of digital computing, to resource limitations (on computing time, storage capacity, or program complexity), or to compatibility constraints (such as hardware floating-point formats and decimal input/output conversion). These factors usually force the original mathematical model to be replaced by a discrete approximation, with finite steps, truncated series, rounded arithmetic, etc.

The accuracy of numeric algorithms is notoriously hard to analyze. In practice, it is often impossible or unfeasible to predict mathematically the magnitude of the roundoff and truncation errors hidden in the output of a numeric program.

In order to be truly useful, every approximate numerical procedure must be accompanied by an *accuracy specification*: a statement that defines the magnitude of the errors in the output values (usually, as a function of the input values and their errors). The accuracy specification must be supported by an *error analysis*: a mathematical proof that the output errors obey the specifications.

Unfortunately, even for relatively simple algorithms, a rigorous error analysis is often prohibitively long, or difficult, or both **[62]**. Moreover, a useful accuracy specification often requires that the inputs satisfy a host of prerequisites—this matrix must be well-conditioned, that function must have bounded derivatives, these formulas should not overflow, etc. In practice, these prerequisites are often impossible to guarantee, or even to check.

As a consequence, numerical algorithms are often put to use without accuracy specifications, much less a proper error analysis. Interpretation of the results is left to the user, who must rely on his intuition, crude tests, or pure luck.

This unfortunate state of affairs has even led to prejudice against approximate numerical computing in contemporary computer science, despite its importance and evident success in applications. The field is generally perceived by outsiders as “sloppy” and “fuzzy”; and hence not really precise and scientific; and hence not a respectable part computer science—where, as in mathematics, there is no place for things that are “99% correct”.

A simple and common approach for estimating the error in a floating-point computation is to repeat it using more precision, and compare the results. If the results agree in many decimal places, then the computation is assumed to be correct, at least in the common part. However, this common practice can be seriously misleading, as the following simple example by Rump [**53**] shows. Consider the evaluation of

$$f = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y),$$

for $x = 77617$ and $y = 33096$. Note that x, y, and all coefficients in f are exactly representable in floating-point (be it binary or decimal). Rump [**53**] reports that computing f in FORTRAN on a IBM S/370 mainframe yield:

$f = 1.172603\ldots$	using single precision;
$f = 1.1726039400531\ldots$	using double precision;
$f = 1.172603940053178\ldots$	using extended precision.

Since these three values agree in the first seven places, common practice would accept the computation as correct. However, the true value is $f = -0.8273960599\ldots$; not even the *sign* is right in the computed results!

Similar results can be obtained with Maple:[1]

```
x:=77617; y:=33096;
# evaluate in floating point
f:=333.75*y^6+x^2*(11*x^2*y^2-y^6-121*y^4-2)+5.5*y^8+x/(2*y);
                                f := 1.172603940
# evaluate in exact rational arithmetic
f:=33375/100*y^6+x^2*(11*x^2*y^2-y^6-121*y^4-2)+55/10*y^8+x/(2*y);
                                      54767
                                f := - -----
                                      66192
# show decimal equivalent
evalf(f,10);
                                -.8273960599
```

1.1.2 Self-validated numerics

In response to this problem, there have arisen several models for *self-validated computation* (SVC) [**2**], also called *automatic result verification* [**37**], in which the computer itself keeps track of the accuracy of

[1] Maple is a registered trademark of 766884 Ontario Inc.

computed quantities, as part of the process of computing them.[2] So, if the magnitude of the error cannot be predicted, at least it can be known *a posteriori.*

The simplest and most popular of these models is R. Moore's *interval arithmetic* [**42**], which we study at length in Chapter 2. There are many other models, however, such as E. Hansen's *generalized interval arithmetic* [**20**], and the *ellipsoid calculus* of Chernousko, Kurzhanski, and Ovseevich [**8, 38**]. In Chapter 3, we describe our own model, similar to Hansen's, which we call *affine arithmetic* [**10**].

A basic limitation of self-validated numerical models is that they can only determine the output errors *a posteriori.* Still, this capability is quite sufficient in certain applications, especially those where the errors are mostly due to external causes. If the output errors, as computed by the model, are deemed too large, then the response must involve some external action—acquire more data, stop the process, alert a human operator, etc.

In cases where the output errors are mainly due to internal approximations and rounding, a self-validated model makes it possible to automatically redo or refine the computation, with increased precision, until the output errors are acceptable. This approach has been turned into a fundamental principle in *lazy real arithmetic* [**39**].

Of course, there are many applications, such as real-time process control, where one needs *a priori* guarantees on the accuracy and timeliness of the results. In such cases, the self-validated approach is not of much help; one still needs to perform a rigorous error analysis, prior to implementation, in order to guarantee that the output errors will be always acceptable.

1.2 Error models

For this monograph, we will start from a very general model of self-validated computation. We assume that the original goal was to evaluate $z \leftarrow f(x)$ for some mathematical function $f\colon \mathbf{R}^m \to \mathbf{R}^n$; but we actually had to implement a discrete computation $Z \leftarrow F(X)$, where X and Z are *approximate values*—discrete mathematical objects that carry only

[2] This would allow the correct result to be obtained in Rump's example, using only floating-point arithmetic.

partial information about the values of the corresponding continuous quantities x and z.

There are many different self-validated computation models that fit this pattern. They are distinguished by the nature of the approximate values they use: probability distributions, intervals, boxes, ellipsoids, polytopes, confidence intervals, lazy digital expansions, interval bags, and many more.

1.2.1 Probabilistic error models

In the natural sciences and engineering, it is customary to view approximate values in a statistical sense: the computed result Z is seen to define a probability distribution for the (unknown) true quantities $z_1, .. z_n$.

Typically, the errors are assumed to follow a Gaussian (normal) distribution. The computed result Z should then specify the mean and variance of each true quantity z_i, and possibly the full covariance matrix for the z_i, that is, the joint (Gaussian) probability distribution of the vector $(z_1, .. z_n)$.

In this framework, a self-validated computing model should automatically compute the statistical parameters for the output quantities z_i, given those for the inputs x_j.

Unfortunately, this model is limited to relatively simple situations, where the measurement errors can be modeled by a Gaussian distributions, and the computations use only linear formulas with negligible roundoff errors. When these conditions do not hold, computing the probability distribution of the error, or even its mean and variance, appears to be an intractable mathematical problem.

1.2.2 Range-based models

To avoid the apparent limitations of the probabilistic model, most self-validated numerical models are based on *range analysis*, i.e., use ranges, rather than distributions, as approximate values.

Specifically, the approximate value Z defines a *range* $[Z]$ for the quantity z, i.e., a set of real values that is *guaranteed* to contain the true value of z — provided that the input quantities x lie in the range $[X]$ specified by the input value X. We refer to this property as the *fundamental invariant of range analysis* (see Section 1.2.3).

In the simplest range analysis models, such as interval arithmetic, each component Z_i of the computed value defines a range of values $[Z_i]$ for the corresponding real quantity z_i. The output Z does not include any constraint relating two or more quantities z_i; in other words, the range $[Z]$ for the output vector $z = (z_1, .. z_n)$ is merely the Cartesian products of those individual ranges: $[Z] = [Z_1] \times \cdot\cdot [Z_n]$. All combinations of $z_1, .. z_n$ in the box $Z_1 \times \cdot\cdot Z_n$ are in principle allowed by this model.

In more sophisticated models, such as affine arithmetic and the ellipsoid calculus, the output Z includes also some information on partial dependencies between the quantities z_i and the inputs x_j. Thus, the computed result Z may give more information about the vector z than one would get by considering its meaning for each z_i independently. Therefore, the Z_i together define a joint range for the vector $(z_1, .. z_n; x_1, .. x_m)$, which is generally a proper subset of the Cartesian product of the individual ranges.

Some models fall halfway between these two extremes. Hansens's generalized interval arithmetic **[20]**, for example, records only correlations between the output quantities z_i and the inputs x_j, but not among the inputs, or among the outputs (see Section 3.19).

For efficiency reasons, each range-based SVC model restricts its ranges to a specific family $\mathcal{R}_{n+m}$ of subsets of $\mathbf{R}^{n+m}$, whose members can be efficiently represented, handled, and combined: boxes, ellipsoids, polytopes, etc.

1.2.3 The fundamental invariant of range analysis

Whatever the shape of the allowed ranges, all range-based SVC models provide, for every function $f: \mathbf{R}^m \to \mathbf{R}^n$, a *range extension* $F: \mathcal{R}_m \to \mathcal{R}_n$, with the following property, which we shall call the *fundamental invariant of range analysis*:

> *If the input vector $(x_1, .. x_m)$ lies in the range jointly determined by the given approximate values $X_1, .. X_m$, then the quantities $(z_1, .. z_n) = f(x_1, .. x_m)$ are guaranteed to lie in the range jointly defined by the approximate values $(Z_1, .. Z_n) = F(X_1, .. X_m)$.*

Ideally, the joint range determined by the outputs Z_i should be as tight as possible, namely the set of all vectors $f(x_1, .. x_m)$ such that $x_j \in X_j$.

In practice, however, a range-based numerical routine is allowed to err on the conservative side, if that is necessary to keep the output ranges representable, or desirable for efficiency reasons. We shall see plenty of examples in the following chapters.

1.2.4 Relative accuracy

As we shall see, a major problem in all forms of range-based computation is the excessive conservativism of the results—the computed ranges are often much wider than necessary. Therefore, in order to effectively compare different algorithms and approaches, we must develop some quantitative measure of this conservativism.

Let $Z \leftarrow F(X)$ be a range computation that purports to represent the mathematical computation $z \leftarrow f(x)$, where $f: \mathbf{R}^m \rightarrow \mathbf{R}^n$. Its *relative accuracy* is, by definition, the ratio between the size of the ideal range $Y = \{ f(x) : x \in [X] \}$ and that of the computed range $[Z]$. By "size" we mean the measure appropriate to the space in question: length for one dimension, area for two, and so on.

Because of the fundamental invariant of range analysis, the relative accuracy is therefore a number between zero (meaning that the output range is infinitely wider than the ideal range) and one (meaning that the two ranges are essentially the same).

Note that if the input range has zero measure, then the relative accuracy is likely to be zero, because of roundoff errors. Therefore, the concept is useful only when X is large enough to make roundoff errors irrelevant.

1.2.5 Confidence-range models

The fundamental invariant stated above requires that the input values lie in the range described by the X_i. This requirement is too strict for scientific and engineering applications, where the input quantities are obtained by physical measurement, and hence may be affected by essentially unbounded error.

In order to use range analysis in such applications, we must replace absolute guarantees by probability statements. That is, each approximate value Z specifies a *confidence range* for the corresponding quantity z: a range of values $[Z]$, as before, and also a real number p_Z, the probability or *confidence level* of z lying in that range.

More generally, an ensemble of approximate values $Y = (Y_1, .. Y_k)$ for quantities $(y_1, .. y_k)$ specifies a subset of $\mathbf{R}^k$, and an associated confidence level p_Y, the probability of the vector $(y_1, .. y_k)$ lying in that set. Note that this approach is quite different from the Gaussian-error model: here, no assumption is made about the shape or variance of the probability distribution, except that its integral inside the range $[Y]$ is at least p_Y.

In this confidence-interval framework, an SVC model should tell us how to compute the joint range and confidence level for the outputs of a formula, given the same data about the inputs. Again, the model is allowed to err on the conservative side, when computing ranges and probabilities.

Most of the ordinary (i.e., non-probabilistic) range-based SVC models can be easily adapted to the confidence-range interpretation. Specifically, one should compute the ranges as in the non-probabilistic model, and then evaluate the confidence level according to the laws of probability. For example, if x lies in the interval $[-1 \_\_ +1]$ with probability 0.95, and y lies in $[4 \_\_ 6]$ with probability 0.80, then $x+y$ lies in $[3 \_\_ 7]$ with probability at least $0.95 + 0.80 - 1 = 0.75$. In general, we have $p_{f(x,y)} \geq p_x + p_y - 1$.

1.3 Floating-point number systems

A *floating-point number system* is a scheme for representing real numbers in discrete machines **[19]**. To allow a wide range of real numbers to be represented, floating-point number systems encode a fraction part, called *mantissa*, and a scale part, called *exponent*. More precisely, a floating-point number system has a *base* β, usually 2 or 10, and encodes real numbers as β-adic fractions of the form:

$$\pm(0.d_1 \ldots d_p)_\beta\, \beta^e = \pm\left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_p}{\beta^p}\right)\beta^e,$$

where the mantissa $m = (0.d_1 \ldots d_p)_\beta$ is written in base β, and e is the exponent. A floating-point number system is characterized by the base β, the *precision* p, and the *exponent range*, $e_{\min} \leq e \leq e_{\max}$. Most computers use base 2, whereas most hand calculators use base 10.

Figure 1.1 shows a floating-point system with $\beta = 2$, $p = 4$, $e_{\min} = -1$, and $e_{\max} = 3$. Note that floating-point numbers are not uniformly spaced, but instead display "logarithmic clustering".

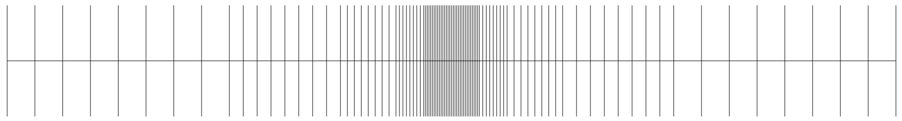

Figure 1.1: Non-uniform distribution of floating-point numbers.

There are two sources of errors when representing real numbers in floating-point: First, a floating-point number system only represents a finite set of numbers. Thus, most real numbers will be either too large in absolute value to be represented, resulting in *overflow*, or too small in absolute value, resulting in *underflow*, Second, not every real number is a β-adic fraction, and so most real numbers in the range of the floating-point number system will fall between two floating-point numbers, and one of them has to be chosen to represent it. This choice is called *rounding*, and the error committed is called the *roundoff error*.

For example, decimal fractions such as 0.1 or 0.001 are very popular as a step sizes in numerical computation, but have no exact representation in binary floating-point systems. Thus, in binary floating-point arithmetic, $10 \times 0.1 \neq 1$ and $1000 \times 0.001 \neq 1$, because of roundoff errors. It would be much safer to use diadic fractions instead, such as 0.125 or 0.0009765625, specially in long computations.

1.4 The IEEE floating-point standard

The IEEE floating-point standard [**3**] is arguably one of the most significant developments in numerical computing since the advent of FORTRAN. Until the widespread adoption of the IEEE standard in the early 1990's, every computer manufacturer, and often every computer model, had its own floating-point number system, with its own base, precision, range; and its own semantics for rounding, commutativity, overflow, underflow, division by zero, etc. Moreover, those rules were usually illogical and poorly documented, being generally the consequence of decisions made by the hardware designers, who were more concerned with speed and cost than with mathematical precision.

The IEEE standard ended this era of confusion. The standard postulates quite rigid formats for single-precision (32-bit) and double-precision (64-bit) floating-point formats. To be precise, there are two IEEE floating-point standards: ANSI/IEEE Std 754-1985 and IEEE Std 854-1987. The first one deals specifically with 32-bit and 64-bit binary formats, whereas the second covers floating-point systems of any base and precision. Since most machines follow both standards, we can safely view them as one.

The standard also ties down precisely the semantics of the four basic arithmetic operations, and of certain common transcendental functions, by requiring that they be as correct as logically possible. That is, hardware conforming to the IEEE standard must interpret the operands as rational numbers, compute the exact result, as in mathematics, and then round it to the nearest representable number, in a specific direction. Moreover, the standard provides control over rounding, a feature that is essential to SVC (see Section 1.4.3). Finally, the standard specifies precisely the results of exceptional operations, such as division by zero, overflow, and underflow. To this end, it reserves certain bit patterns to denote two "infinite" values ($+\infty$ and $-\infty$), and a series of error codes or "not-a-numbers" (`NaN`); and extends the semantics of all operations to accept and return these special values.

Essentially, the standard does not leave unspecified a single significant bit of the floating-point model. Thanks to this unforgiving strictness, every floating-point number that can be represented in the IEEE standard format can be stored in any standard-compliant machine, without loss of precision. Moreover, any standard-compliant processor that performs the same sequence of floating-point operations on the same data will return precisely the same result, down to the last bit. This provided a much welcome portability of numerical programs and data.

The IEEE standard is so thorough, and filled such a need, that it was quickly adopted by most computer manufacturers, down to the last bit. Like every standard, this one has several technical flaws, which are all the more irritating for having been cast in stone for decades to come. Still, as in most other fields, a bad standard is better than no standard.

Besides all its practical contributions to portability, robustness, and documentation, the IEEE standard has had an enormous psychological impact on the programming community. Suddenly, it became worthwhile to worry about roundoff errors in a precise way. It became possible, at least in principle, to design truly robust numerical algorithms, and give rigorous proofs of their correctness, even in the presence of underflow and overflow. Thus, the IEEE standard prepared the way for self-validated computation.

1.4.1 Special floating-point values

As mentioned above, the IEEE floating-point standard defines, in addition to ordinary "finite" numbers, certain special values:

- the *infinities* $+\infty$ and $-\infty$, whose meaning and properties are for the most part obvious;
- the *not-a-number* values, collectively denoted by `NaN`, which are the conventional result of indeterminate operations like $0/0$, $0 \cdot \infty$, and $\infty - \infty$;
- the *negative zero* $\bar{0}$, which by definition is the reciprocal of $-\infty$ (and vice-versa).

As we shall see, the infinities $+\infty$ and $-\infty$ are very useful for self-validated computation, since they allow us to represent the notion of "no lower bound" and "no upper bound", respectively.

The `NaN` values behave rather peculiarly in comparisons: a `NaN` is neither greater than, equal to, nor less than any other value—including itself! A `NaN` may signify either as "no value", "more than one value", or "any real value", depending on the context. Since many SVC models have other ways of representing these concepts, `NaN` values tend to be little used.

We shall use the term "float" for any IEEE floating-point value—finite, infinite, or `NaN`. We shall denote by $\mathbf{F}$ the set of all finite floats, which we shall consider as a subset of the real line $\mathbf{R}$; and by $\mathbf{F}^*$ the set of *numeric floats*: $\mathbf{F}^* = \mathbf{F} \cup \{-\infty, +\infty\}$. Note that `NaN` does not belong to either $\mathbf{F}$ or $\mathbf{F}^*$.

1.4.2 Negative zero

One of the most controversial features of the IEEE standard is the existence of a negative zero, $\bar{0} = 1/(-\infty)$. While it is possible to concoct examples where this feature saves an instruction or two, in the vast majority of applications this value is an annoying distraction, and a possible source of subtle bugs.

Unlike infinite values, which merely extend the domain of arithmetic operations without changing their semantics for ordinary numbers, the introduction of negative zero actually affects the semantics of many operations in non-obvious and mathematically inconsistent ways. For instance, the square root of negative zero is defined to be negative zero.

Fortunately, negative zero behaves like ordinary zero in many respects. In particular, in numeric comparisons negative zero turns out to be equal to (and not less than) ordinary zero. Thus, we can usually pretend that ordinary zero and minus zero are the same value; and we shall adopt this viewpoint here. However, one must watch out for occasional pitfalls: for instance, I/O routines will usually print negative zero as `-0`, and its sign comes out as -1 instead of $+1$.

1.4.3 Rounding mode control

A feature of the IEEE standard that is highly relevant to SVC is its provision for rounding control. A standard-compliant processor must allow the programmer to specify the direction in which computed results are rounded to representable numbers. As we shall see, this feature is essential for the efficient implementation of interval arithmetic and other self-validated computation models.

In this monograph, we use the notation $\langle\mathcal{E}\rangle$ for the value of expression $\mathcal{E}$ evaluated in IEEE floating-point arithmetic, with the default rounding mode ("to nearest or to even"). We also write $\uparrow\mathcal{E}\uparrow$ for a numeric float (possibly $+\infty$) that is greater than or equal to the value of a formula $\mathcal{E}$; that is, the value of $\mathcal{E}$ rounded up to a representable number (not necessarily the smallest one). Similarly, we write $\downarrow\mathcal{E}\downarrow$ for the value of $\mathcal{E}$ rounded down to a representable value.

In the special case when $\mathcal{E}$ consists of a single arithmetic operation, $\uparrow\mathcal{E}\uparrow$ and $\downarrow\mathcal{E}\downarrow$ are by definition the result of computing $\mathcal{E}$ on an IEEE-compliant processor with the rounding direction set as specified. If $\mathcal{E}$ contains two or more operations, then each must be rounded in the appropriate direction so as to ensure that the final result is rounded as specified. For example, when evaluating $\uparrow x^2/(x^2+1)\uparrow$, the denominator must be rounded towards $-\infty$, whereas the numerator and the quotient must be rounded towards $+\infty$.

Sometimes, the correct rounding mode to use when evaluating a sub-expression depends on the values of other sub-expressions. For example, in $\uparrow x(y+1)\uparrow$, the result of $y+1$ must be rounded up if x is positive, and down if x is negative. We shall generally avoid such complicated situations, and use external tests to ensure that each operation inside a $\uparrow\cdot\uparrow$ or $\downarrow\cdot\downarrow$ can be evaluated with a single, statically determined rounding mode.

Unfortunately, the rounding-mode controls of most processors are extremely inconvenient to use, so that changing the rounding mode is expensive—typically a dozen machine instructions. To minimize such changes, we can use the identities

$$\uparrow -\mathcal{E}\uparrow = -\downarrow\mathcal{E}\downarrow, \qquad \downarrow -\mathcal{E}\downarrow = -\uparrow\mathcal{E}\uparrow.$$

These formulas work for all float values, because, according to the IEEE standard, sign negation is involutory, exact, and never overflows.

1.4.4 Single vs. double precision

When implementing an SVC model, one may have to choose between using single precision (32-bit) or double precision (64-bit). Three decades ago, the difference in speed and storage space dictated that most numerical computing should be performed in single precision, with double precision being used only when really necessary. Nowadays, storage is rarely a limiting factor, and most floating-point processors will be just as fast on 64-bit operations as on 32-bit ones; so the advantages of single precision have all but disappeared, and double precision is increasingly being seen as the default. (For instance, the original definition of the C programming language stated that all floating-point arithmetic was to be done in double precision [**32**, p. 41].)

Moreover, many mathematical libraries and programming languages will automatically convert arguments and results to the double-precision format. Now, another questionable "feature" of IEEE (denormalized numbers) has the unfortunate consequence of making conversion between single and double precision *very* expensive on certain machines, because it has to be done by software. Therefore, it is quite possible (in the authors' own experience!) for a numerical computation to become *much* faster when converted from single to double precision.

Chapter 2

Interval arithmetic

In this chapter, we describe interval arithmetic, the simplest and most efficient of all validated numerics models. We also discuss how to write procedures for most elementary operations and functions.

2.1 Introduction

Interval arithmetic (IA), also known as *interval analysis*, is a range-based model for numerical computation where each real quantity x is represented by an interval $\bar{x}$ of floating-point numbers. Those intervals are added, subtracted, multiplied, etc., in such a way that each computed interval $\bar{x}$ is guaranteed to contain the (unknown) value of the corresponding real quantity x.

Interval arithmetic was invented in the 1960's by Ramon E. Moore [**42, 43**], then a Ph. D. student at Stanford University. Its prestige and popularity among the numerical analysis community has been somewhat of a roller-coaster ride. Interest in IA was quite high for a few years after Moore's thesis, at which time it seem to have been oversold as a panacea for all numerical computation problems. As the euphoria subsided, a reaction set in, and for the next two decades IA was viewed very negatively—to such an extent that authors who needed to use intervals in their algorithms reportedly had to call them "segments" or "ranges" in order to get their papers published.

However, in recent years there has been a strong and steady resurgence of interest in IA. Practitioners and researchers in the most varied fields—from pure mathematics to computer graphics to economics—have found that IA provides a simple and relatively efficient solution to computational problems that were intractable under the "classical" approach. IA is now appreciated for its ability to manipulate imprecise data, keep track automatically of truncation and round-off errors, and probe the behavior of functions efficiently and reliably over whole sets of arguments at once.

Successful applications of IA include, for example, robust root finders for ray tracing [**7, 40**], domain enumeration for solid modeling, [**14, 44, 54, 58, 59**], surface intersection [**18**], global optimization [**21–23, 27, 30, 41, 50, 51, 61**] We will discuss some important applications of IA in Chapter 4. It is also noteworthy that interval arithmetic recently played a key role in settling the *double bubble conjecture* [**24**], a longstanding open problem in the theory of minimal surfaces.

This revival of IA was greatly helped by the publishing and universal acceptance of the IEEE floating-point standard [**3**]. The standard mandated the implementation of directed rounding, which is indispensable for practical implementations of IA. Moreover, adoption of the standard by all major computer manufacturers encouraged the development of portable IA packages [**31, 33–35**]. Finally, the high visibility of the standard undoubtedly made programmers more aware of the floating-point roundoff problem, and hence interested in self-validated numeric computation.

Interval arithmetic and related techniques now have a dedicated journal (*Reliable Computing*, formerly *Interval Computations*, published by Kluwer Academic Publishers), a central web site containing a wealth of information and links [**28**], and several established conferences.

2.2 Intervals

In interval arithmetic, each quantity x is represented by an interval $\bar{x} = [\bar{x}.lo \mathbin{\_\_} \bar{x}.hi]$ of real numbers, meaning that the "true" value of x is known to satisfy $\bar{x}.lo \le x \le \bar{x}.hi$.

Those intervals are added, subtracted, multiplied, etc., in such a way that each computed interval is guaranteed to contain the (unknown) value of the quantity it represents. Thus, for example, the sum and difference of two intervals $\bar{x}$ and $\bar{y}$ is computed as

$$\begin{aligned}\bar{x}+\bar{y} &= [\bar{x}.lo+\bar{y}.lo \;\_\_\; \bar{x}.hi+\bar{y}.hi] \\ \bar{x}-\bar{y} &= [\bar{x}.lo-\bar{y}.hi \;\_\_\; \bar{x}.hi-\bar{y}.lo].\end{aligned}$$

(These formulas ignore roundoff errors, overflow, and other details, which we address in Section 2.3.)

The reader may check that these are the smallest intervals that contain $x+y$ and $x-y$, respectively, for all possible pairs $x \in \bar{x}$ and $y \in \bar{y}$. Analogous formulas can be devised for multiplication, division, square root, and all common mathematical functions (see Section 2.4).

2.2.1 Precise definition of intervals

In the spirit of SVC, before proceeding any further we must define very precisely the representation and semantics of intervals in the IA model.

Accordingly, we define a *non-empty interval* as a set of the form

$$[\bar{x}.lo \;\_\_\; \bar{x}.hi] = \{\, x \in \mathbf{R} \,:\, \bar{x}.lo \le x \le \bar{x}.hi \,\},$$

where $\bar{x}.lo$ (the *lower bound* of the interval) is in $\mathbf{F} \cup \{-\infty\}$, and $\bar{x}.hi$ (the *upper bound*) is in $\mathbf{F} \cup \{+\infty\}$.

We also define the *empty interval* $[\,]$ as synonymous of the empty set. The upper and lower bounds of $[\,]$ are not defined.

Note that every *finite* float x can be represented as an interval $[x \;\_\_\; x]$. Every other real number can be approximated by an interval $[a \;\_\_\; b]$, where a and b are consecutive float values, possibly infinite.

Note that the *bounds* of an interval are *float values*, possibly infinite, but its *elements* are drawn from the finite *real numbers* $\mathbf{R}$. Thus, for example, the interval $[1 \;\_\_\; +\infty]$ includes 1 and $\sqrt{2}$ and 10^{1000}, but not $+\infty$. In particular, $[-\infty \;\_\_\; +\infty]$ is the same as $\mathbf{R}$, the set of all (finite) real numbers.

The obvious way to represent a non-empty interval $\bar{x}$ in the computer is by record with two float components, $\bar{x}.lo$ and $\bar{x}.hi$. The empty interval could then be represented by any such pair with $\bar{x}.lo > \bar{x}.hi$. We shall use $[+\infty \;\_\_\; -\infty]$, specifically, because this choice simplifies the implementation in some cases.

Since $-\infty$ and $+\infty$ are not real numbers, the pairs with $\bar{x}.lo = \bar{x}.hi = -\infty$ or $\bar{x}.lo = \bar{x}.hi = +\infty$ would denote the empty set, too. However, it is advisable to outlaw these two pairs altogether, so that the common test $x = [\,]$ can be consistently implemented as $\bar{x}.lo > \bar{x}.hi$.

In summary, a pair $(\bar{x}.lo, \bar{x}.hi)$ represents a non-empty interval if $\bar{x}.lo \in \mathbf{F} \cup \{-\infty\}$, $\bar{x}.hi \in \mathbf{F} \cup \{+\infty\}$, and $\bar{x}.lo \leq \bar{x}.hi$; or the empty interval, if $\bar{x}.lo > \bar{x}.hi$. The pairs

$$[+\infty \mathrel{\_\_} +\infty], [-\infty \mathrel{\_\_} -\infty], [\mathtt{NaN} \mathrel{\_\_} \mathtt{NaN}], [a \mathrel{\_\_} \mathtt{NaN}], [\mathtt{NaN} \mathrel{\_\_} a],$$

are not valid intervals, for any float a.

We say that an interval $\bar{x}$ *straddles* a real number z when $\bar{x}.lo < z < \bar{x}.hi$. When $\bar{x}.lo = z$ or $\bar{x}.hi = z$, and the interval is not empty, we say that $\bar{x}$ merely *touches* z.

2.3 Computing with IA

For every operation $f(x, y, \ldots)$ from reals to reals (such as sum, product, square root, etc.), the interval arithmetic model defines a corresponding *interval extension* $\bar{f}(\bar{x}, \bar{y}, \ldots)$. This operation returns some interval—preferably the smallest one—that contains all values of $f(x, y, \ldots)$, where the variables $x, y, \ldots$ range independently over the given intervals $\bar{x}, \bar{y}, \ldots$

For elementary operations, implementing these interval extensions is generally straightforward: we need only devise formulas for the maximum and minimum values of f when the arguments $x, y, \ldots$ vary independently over specified intervals. Often, a case analysis is required.

For certain functions, determining the *exact* maxima and minima may be too difficult. In such cases, it is acceptable to return any computable interval that contains the theoretical range of the function, not necessarily the smallest one. That is, we are allowed to increase the upper bound, and decrease the lower bound; doing so does not violate the fundamental invariant of range analysis (Section 1.2.3).

Once we have implemented interval extensions for all elementary operations and functions, interval extensions for a complicated function can be computed by composing these primitive formulas in the same way the primitive operations are composed to compute the function itself. In other words, any algorithm for computing a function using primitive operations can be readily (and automatically) interpreted as an algorithm for computing an interval extension for the same function. (This is specially elegant to implement with programming languages that support operator overloading, such as Ada, C++, Fortran-90, and Pascal-XSC, but can be easily implemented in any programming language, either manually or with the aid of a pre-compiler.) Thus, the class of functions for which interval extensions can be easily (and automatically) computed is much larger than the class of rational polynomial functions. This proves the fundamental invariant of range analysis for IA.

2.3.1 Handling roundoff errors

A common reason for widening the result interval is the need to represent the endpoints as floating-point values. In order to preserve the fundamental invariant, we must be careful to round each bound in the proper direction: namely, the lower bound must be rounded towards $-\infty$, and the upper bound towards $+\infty$.

This concern also applies to any intermediate values that may affect the computed bounds. Such values must always be rounded in the most conservative direction—the one which leads the resulting interval to be widened, rather than narrowed. In particular, we can always replace the *input* intervals by wider ones. (See Section 2.4.12 for an example where this action is necessary.)

2.3.2 Handling overflow

In some operations, we must also worry about the possibility of overflow when computing the extrema of f in the given interval. Fortunately, in IEEE-compliant floating-point arithmetic, overflow generally produces a special infinity value with the appropriate sign, so that we do not need to handle those cases explicitly. Thus, if overflow occurs, the resulting interval will automatically extend to infinity, in either or both directions, and the fundamental invariant will be preserved.

However, the IEEE standard also specifies that certain operations, such as $0/0$ or $0 \cdot \infty$, and $(-\infty) + (+\infty)$, result in the special "not-a-number" value `NaN`. Recall that we decided (in Section 2.2.1) to forbid intervals with `NaN` endpoints, because of its ambiguous meaning and bizarre properties. Therefore, whenever an operation might return `NaN`, we must test for that event, and return either $\mathbf{R} = [-\infty \mathbin{\_\_} +\infty]$ or $[\,]$, as appropriate. The reason why we outlawed the intervals $[+\infty \mathbin{\_\_} +\infty]$ and $[-\infty \mathbin{\_\_} -\infty]$ is precisely to reduce the need for such tests.

2.3.3 Handling domain violations

When implementing an elementary function, such as square root and logarithm, which is defined only for a proper subset of the real line, one should simply ignore any part of the argument interval that is outside domain of definition.

Thus, for example, the square root routine, when given $[-3 \mathrel{\_\_} +4]$, should simply return $[0 \mathrel{\_\_} 2]$. It would not be appropriate to signal an error in this case, because the argument interval is merely a conservative estimate of the true range of the corresponding quantity. The fact that the interval extends into the negative values does not imply that the quantity may be negative.

This "soft" policy towards undefined values may seem to violate the fundamental invariant of range analysis. After all, $[0 \mathrel{\_\_} 2]$ does *not* contain all possible values of $\sqrt{x}$ when x ranges over $[-3 \mathrel{\_\_} +4]$; some of those "values" are undefined (or imaginary). However, if an algorithm says to compute $\sqrt{x}$ at some point, and expects a real result, then the square-root routine must assume that the true value of x would always be positive in any exact evaluation of the algorithm. If the true value of x could be negative at that point, then the algorithm would be logically incorrect. Now, the IA model cannot guarantee that the final intervals are correct if the exact algorithm is not correct (otherwise, $[-\infty \mathrel{\_\_} +\infty]$ would be the only valid output!).

On the other hand, if the argument interval to an IA operation is entirely outside the domain of definition of the corresponding function, then something is clearly wrong with the program. In that case, the IA routine should probably signal an error.

Another alternative in such cases is to use a "super-soft" policy, and return the empty interval $[\,]$. It is then the programmer's responsibility to test whether the result is $[\,]$, and take action if necessary. In that case, for consistency, every IA operation should return $[\,]$ whenever one or more operands are $[\,]$.

Early detection vs. soft failure

The choice between signalling an error and returning $[\,]$ is a special case of a classical dilemma of software engineering, the tension between *early detection* versus *soft failure*.

The early detection policy requires that "exceptional" conditions, such as end-of-file or division by zero, be treated as immediate breaks in the control flow, which require explicit handling. One advantage of this policy is that programming errors that derive from or cause such conditions tend to be detected sooner, and hence are easier to debug. It also has the merit of forcing the programmer to be aware and handle all exceptional cases.

The soft failure policy, in contrast, implies that "exceptional" conditions should be handled as "ordinarily" as possible, namely by encoding them as distinguished values that can be returned and assigned like any other value. This approach usually simplifies the code that follows procedure calls, since it is not necessary to handle the exceptional results explicitly. On the other hand, under this policy one often needs extra tests at procedure entry to recognize and handle the exceptional values.

Software engineering experts seem still divided on this issue. The designers of the IEEE standard avoided taking a stand on this matter: they provided infinities and `NaN`s, in accordance to the soft-failure approach, but also allowed the programmer to specify whether the creation of such values should cause an error trap, as required for early detection.

2.4 Specific operations

We now describe in detail how to compute interval extensions for the elementary operations and functions.

We shall use a Pascal-like syntax: Iteration and branching will be specified with `for`, `while`, and `if` commands, with obvious semantics. However, command grouping will be indicated by indentation alone, without the `begin` ... `end` brackets of Pascal. Comments appear in italics guarded by `|`. Assignment statements will be written *variable* $\leftarrow$ *value*. Variables will be declared by `var` *name*: *Type*, as in Pascal; but declarations may appear at the beginning of any compound statement, as in C or Algol 60. The type `Float` stands for any IEEE floating-point value except `NaN`, and `Finite` is `Float` $\setminus \{-\infty, +\infty\}$.

For actual implementations of interval operations, see the public domain libraries [**31, 33–35**].

2.4.1 Negation

We begin with negation, because of its simplicity:

```
IA.neg(x̄: Interval): Interval ≡
| Computes −x̄.
  return [−x̄.hi ‥ −x̄.lo]
```

Note the reversal of the upper and lower bounds. Note also that negation of intervals, unlike almost any other operation, is not affected by roundoff or overflow. Note, finally, that the code above works even for the special intervals **R** and [ ].

2.4.2 Addition

The code for addition is straightforward, except that we must explicitly return [ ] if either argument is [ ]:

```
IA.add(x̄, ȳ: Interval): Interval ≡
| Computes x̄ + ȳ.
  if x̄ = [ ] or ȳ = [ ] then
    return [ ]
  else
    return [↓x̄.lo + ȳ.lo↓ ‥ ↑x̄.hi + ȳ.hi↑]
```

The reader may want to check that this algorithm works even for intervals with one or two infinite bounds. Note that $a \neq +\infty$ and $b \neq +\infty$ imply $\downarrow a + b \downarrow \neq +\infty$, even when $a + b$ overflows the finite floating-point range; and similarly for $\uparrow a + b \uparrow$ and $-\infty$. Therefore, the algorithm above cannot return the "forbidden" intervals $[+\infty \ldots +\infty]$ and $[-\infty \ldots -\infty]$, as long as they are not given as arguments.

One might think that the initial tests for [ ] could be avoided if [ ] were consistently represented by $[+\infty \ldots -\infty]$, since the general formula would then give

$$[\,] + \bar{y} = [(+\infty) + \bar{y}.lo \ldots (-\infty) + \bar{y}.hi] = [+\infty \ldots -\infty] = [\,],$$

as desired. Unfortunately, this simplification would fail when adding [ ] to $\mathbf{R} = [-\infty \ldots +\infty]$, because $(-\infty) + (+\infty)$ is `NaN`. So the tests seem unavoidable.

2.4.3 Translation

A common special case of addition is translation of an interval by a finite float c (which may be regarded as an interval of zero width). There is no reasonable way to extend this operation for infinite values of c, because $[a + \infty \,\_\_\, b + \infty]$ is $[+\infty \,\_\_\, +\infty]$, which is not a valid interval.

```
IA.shift(x̄: Interval; c: Finite): Interval ≡
| Computes x̄ + c.
  if x̄ = [] then
    return []
  else
    return [↓x̄.lo + c↓ __ ↑x̄.hi + c↑]
```

2.4.4 Subtraction

To subtract two intervals, we merely add the first to the negation of the second. Combining the two operations into a single procedure, we get

```
IA.sub(x̄, ȳ: Interval): Interval ≡
| Computes x̄ − ȳ.
  if x̄ = [] or ȳ = [] then
    return []
  else
    return [↓x̄.lo − ȳ.hi↓ __ ↑x̄.hi − ȳ.lo↑]
```

2.4.5 Scaling

Scaling an interval by a positive factor is straightforward; scaling by a negative factor requires swapping the bounds. Scaling by 0, of course, should result in the degenerate interval $[0 \,\_\_\, 0]$. We must handle this case explicitly, in case the interval has infinite bounds, because $0 \cdot \infty =$ NaN in IEEE arithmetic. As in the case of translation, there is no reasonable way to define scaling when c is infinite.

```
IA.scale(x̄: Interval; c: Finite): Interval ≡
| Computes c · x̄.
  if x̄ = [] then
    return []
  else if c > 0 then
    return [↓c · x̄.lo↓ __ ↑c · x̄.hi↑]
  else if c < 0 then
    return [↓c · x̄.hi↓ __ ↑c · x̄.lo↑]
  else
    return [0 __ 0]
```

2.4.6 Multiplication

For the multiplication routine, we need formulas for the maximum and minimum of xy when the pair (x, y) ranges over a rectangle $[\bar{x}.lo \mathbin{\_\_} \bar{x}.hi] \times [\bar{y}.lo \mathbin{\_\_} \bar{y}.hi]$. The key observation here is that, for a fixed value of x, the product xy is linear (hence monotonic) in y; and vice-versa. It follows that the extrema must occur at corners of the rectangle.

The simplest implementation is thus:

```
IA.mul(x̄, ȳ: Interval): Interval ≡
| Computes x̄ · ȳ – naive version.
  if x̄ = [] or ȳ = [] then
    return []
  else if x̄ = [0 __ 0] or ȳ = [0 __ 0] then
    return [0 __ 0]
  else
   a ← min {↓x̄.lo · ȳ.lo↓ , ↓x̄.lo · ȳ.hi↓ , ↓x̄.hi · ȳ.lo↓ , ↓x̄.hi · ȳ.hi↓}
   b ← max {↑x̄.lo · ȳ.lo↑ , ↑x̄.lo · ȳ.hi↑ , ↑x̄.hi · ȳ.lo↑ , ↑x̄.hi · ȳ.hi↑}
   return [a __ b]
```

Note that we must handle separately the cases where one of the operands is $[0 \mathbin{\_\_} 0]$, in case the other one has infinite bounds (which would lead to `NaN` bounds in the result).

This routine requires eight multiplications in all non-trivial cases. However, we can reduce this to only two multiplications in most cases, and four in only one case, by testing the signs of the operands in order to determine which corners yield the maximum and minimum product.

There are nine main cases to consider, depending on whether each interval is entirely non-negative, entirely non-positive, or straddles zero (see Figure 2.1). If either x or y has consistent sign, then the two extremal corners are determined by the sign combinations alone. If both intervals straddle zero, then the signs alone do not suffice: we must evaluate the product at all four corners, and compare the results.

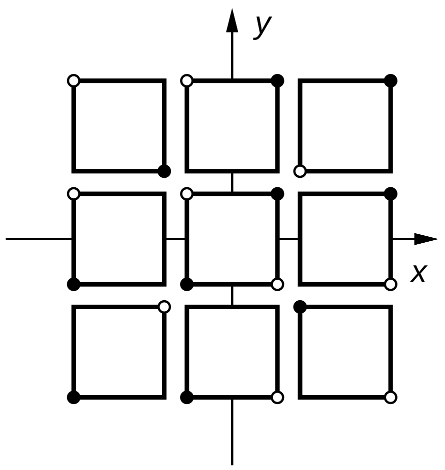

Figure 2.1: The nine cases for multiplication ($\bullet$ = possible maximum, $\circ$ = possible minimum).

```
IA.mul(x̄, ȳ: Interval): Interval ≡
| Computes x̄ · ȳ.
  if x̄ = [] or ȳ = [] then
    return []
  else if x̄ = [0 __ 0] or ȳ = [0 __ 0] then
    return [0 __ 0]
  else if x̄.lo ≥ 0 then
    if ȳ.lo ≥ 0 then
      return [↓x̄.lo · ȳ.lo↓ __ ↑x̄.hi · ȳ.hi↑]
    else if ȳ.hi ≤ 0 then
      return [↓x̄.hi · ȳ.lo↓ __ ↑x̄.lo · ȳ.hi↑]
    else
      return [↓x̄.hi · ȳ.hi↓ __ ↑x̄.hi · ȳ.lo↑]
  else if x̄.hi ≤ 0 then
    if ȳ.lo ≥ 0 then
      return [↓x̄.lo · ȳ.hi↓ __ ↑x̄.hi · ȳ.lo↑]
    else if ȳ.hi ≤ 0 then
      return [↓x̄.hi · ȳ.hi↓ __ ↑x̄.lo · ȳ.lo↑]
    else
      return [↓x̄.lo · ȳ.hi↓ __ ↑x̄.lo · ȳ.lo↑]
  else
    if ȳ.lo ≥ 0 then
      return [↓x̄.lo · ȳ.hi↓ __ ↑x̄.hi · ȳ.hi↑]
    else if ȳ.hi ≤ 0 then
      return [↓x̄.hi · ȳ.lo↓ __ ↑x̄.lo · ȳ.lo↑]
    else
      a ← min {↓x̄.lo · ȳ.hi↓ , ↓x̄.hi · ȳ.lo↓}
      b ← max {↑x̄.lo · ȳ.lo↑ , ↑x̄.hi · ȳ.hi↑}
      return [a __ b]
```

2.4.7 Reciprocal

The reciprocal function $1/x$ is not defined for $x = 0$. Hence (as discussed in Section 2.3.3), the IA implementation must implicitly exclude that argument value from the input interval $\bar{x}$.

The algorithm has two main cases, depending on whether the input interval $\bar{x}$ straddles zero or not. In the first case, the reciprocal may assume arbitrarily large and arbitrarily small real values; hence the correct interval result must be $\mathbf{R} = [-\infty \,..\, +\infty]$. In the second case, the reciprocal is monotonic, so we only need to evaluate it at the interval endpoints.

However, input intervals with zero bounds must be handled separately. The IEEE standard defines $1/0$ as $+\infty$, and $1/(-0) = -\infty$; but we cannot trust that zero bounds in the argument will have the "right" sign, due to the existence of negative zero.

`IA.inv(`$\bar{x}$`: Interval): Interval` $\equiv$
| *Computes* $1/\bar{x}$.
  `if` $\bar{x} = [\,]$ `then`
    `return` $[\,]$
  `else if` $\bar{x}.lo < 0$ `and` $\bar{x}.hi > 0$ `then`
    `return` $[-\infty \,..\, +\infty]$
  `else if` $\bar{x} = [0 \,..\, 0]$ `then`
    `return` $[\,]$
  `else`
    `if` $\bar{x}.hi = 0$ `then` $a \leftarrow -\infty$ `else` $a \leftarrow {\downarrow}1/\bar{x}.hi{\downarrow}$
    `if` $\bar{x}.lo = 0$ `then` $b \leftarrow +\infty$ `else` $b \leftarrow {\uparrow}1/\bar{x}.lo{\uparrow}$
    `return` $[a \,..\, b]$

It should be noted that ${\downarrow}1/\bar{x}.hi{\downarrow}$ may be $-\infty$ and ${\uparrow}1/\bar{x}.lo{\uparrow}$ may be $+\infty$, even when the denominators are finite. Fortunately, these possible overflows will not affect the correctness of the result.

2.4.8 Division

The division of $\bar{x}$ by $\bar{y}$ could be implemented in IA as the product of $\bar{x}$ by $1/\bar{y}$. However, we may gain a couple of bits of accuracy by coding a special routine for division. For example, the latter should be able to evaluate $[3 \,..\, 3]/[3 \,..\, 3] = [1 \,..\, 1]$ without any roundoff error; whereas the reciprocal of $[3 \,..\, 3]$ would introduce some rounding error.

In fact, if the bounds of $\bar{y}$ are too close to zero, then their reciprocals may overflow, leading to an infinite range for $\bar{x}(1/\bar{y})$, even when the quotient may still be finite.

Like multiplication and reciprocal, the division algorithm must be broken down into several distinct cases, depending on the signs of the operands. The code structure is a bit simpler than multiplication, however, because in the three cases where $\bar{y}$ straddles zero the result is simply the entire real line.

```
IA.div(x̄, ȳ: Interval): Interval ≡
| Computes x̄/ȳ.
  if x̄ = [] or ȳ = [] or ȳ = [0 __ 0] then
    return []
  else if x̄ = [0 __ 0] then
    return [0 __ 0]
  else if ȳ.lo ≥ 0 then
    if x̄.lo ≥ 0 then
      return [↓x̄.lo/ȳ.hi↓ __ ↑x̄.hi/ȳ.lo↑]
    else if x̄.hi ≤ 0 then
      return [↓x̄.lo/ȳ.lo↓ __ ↑x̄.hi/ȳ.hi↑]
    else
      return [↓x̄.lo/ȳ.lo↓ __ ↑x̄.hi/ȳ.lo↑]
  else if ȳ.hi ≤ 0 then
    if x̄.lo ≥ 0 then
      return [↓x̄.hi/ȳ.hi↓ __ ↑x̄.lo/ȳ.lo↑]
    else if x̄.hi ≤ 0 then
      return [↓x̄.hi/ȳ.lo↓ __ ↑x̄.lo/ȳ.hi↑]
    else
      return [↓x̄.hi/ȳ.hi↓ __ ↑x̄.lo/ȳ.hi↑]
  else
    return [−∞ __ +∞]
```

2.4.9 Square root

The IA version of square root is extremely simple, because the function is monotonic and is not liable to overflow or underflow for any finite arguments. The only special precaution we must take is to remove the negative part of the input range, if any (using the "super-soft" policy described in Section 2.3.3):

```
IA.sqrt(x̄: Interval): Interval ≡
| Computes √x̄.
  if x̄ = [] or x̄.hi < 0 then
    return []
  else if x̄.lo ≤ 0 then
    return [0 __ ↑√(x̄.hi)↑]
  else
    return [↓√(x̄.lo)↓ __ ↑√(x̄.hi)↑]
```

2.4.10 Logarithm

The code for logarithm is quite similar to that of square root; except that log is not defined at zero, and tends to $-\infty$ at its right.

```
IA.log(x̄: Interval): Interval ≡
| Computes log x̄.
  if x̄ = [] or x̄.hi ≤ 0 then
    return []
  else if x̄.lo ≤ 0 then
    return [−∞ __ ↑log(x̄.hi)↑]
  else
    return [↓log(x̄.lo)↓ __ ↑log(x̄.hi)↑]
```

This code can be used for computing logarithms in any base. Note that we do not need to handle the case $\bar{x}.lo = 0$ separately, because $\downarrow\log 0\downarrow$ is $-\infty$ in IEEE-compliant platforms.

2.4.11 Exponential

The exponential function $\exp(x) = e^x$ is also monotonic, and defined everywhere:

```
IA.exp(x̄: Interval): Interval ≡
| Computes exp x̄.
  if x̄ = [] then
    return []
  else
    return [↓exp(x̄.lo)↓ __ ↑exp(x̄.hi)↑]
```

The floating-point evaluation of $\exp x$ will overflow for values of x above a few hundred. Ideally, these overflows should not require special handling in the code: $\uparrow\exp(\bar{x}.hi)\uparrow$ should be $+\infty$, and $\downarrow\exp(\bar{x}.lo)\downarrow$ should be the maximum finite value, in which case the interval result would be correct.

2.4.12 Sine and co-sine

There are several features of the trigonometric functions sin and cos that require special attention. For one thing, they are non-monotonic; therefore, when computing their extremal values in some interval, we must consider their local maxima and minima, as well as the interval endpoints.

The local extrema of cos occur at integer multiples of π. Thus, after disposing of the empty case, our first step is to scale the input interval $\bar{x}$ by $1/\pi$. If the resulting range straddles an even integer, then the interval $\bar{x}$ contains a maximum, and we can set the upper bound of the result to 1. Symmetrically, if the scaled range straddles an odd integer, then the lower bound is -1. If only one of these conditions holds, then we must compare the values of cos at the endpoints of $\bar{x}$ in order to determine the other bound for the result. Finally, if the scaled interval straddles no integers, then cos is monotonic (increasing or decreasing) in the original interval $\bar{x}$, and the extremal values occur at $\bar{x}.lo$ and $\bar{x}.hi$.

Here, as always, we must take into account all roundoff errors. In particular, since π is not exactly representable as a float value, we must treat it as an interval $[\bar{\pi}.lo \;\_\_\; \bar{\pi}.hi]$ of non-zero width, where $\bar{\pi}.lo$ and $\bar{\pi}.hi$ are consecutive floats.

```
IA.cos(x̄: Interval): Interval ≡
| Computes cos x̄.
  if x̄ = [] then
    return []
  else
    | Scale x̄ by 1/π:
    if x̄.lo > 0 then a ← ↓x̄.lo/π̄.hi↓ else a ← ↓x̄.lo/π̄.lo↓
    if x̄.hi > 0 then b ← ↑x̄.hi/π̄.lo↑ else b ← ↑x̄.hi/π̄.hi↑
    | Check for odd and even integers in [a __ b]:
    m ← ⌊a⌋
    n ← ⌈b⌉
    if ↑n − m↑ < 2 then
      | There are no extremal values in (x̄.lo __ x̄.hi):
      if even(m) then
        u ← ↓cos x̄.lo↓;    v ← ↑cos x̄.hi↑
      else
        u ← ↓cos x̄.hi↓;    v ← ↑cos x̄.lo↑
    else if ↑n − m↑ = 2 then
      | At most one extremal value in (x̄.lo __ x̄.hi):
      if even(m) then
        u ← −1;    v ← max {↑cos x̄.lo↑, ↑cos x̄.hi↑}
      else
        u ← min {↓cos x̄.lo↓, ↓cos x̄.hi↓};    v ← +1
    else
      | There seem to be maxima and minima in (x̄.lo __ x̄.hi):
      u ← −1;    v ← +1
    return [u __ v]
```

The notation $\lfloor a \rfloor$ means, as usual, the greatest integer not greater than a. In order to avoid integer overflow (which, in most machines, is either fatal or hard to detect), this quantity must be computed entirely in floating-point, with the `floor` function from the standard C `math` library, or equivalent. That way, the operation $m \leftarrow \lfloor a \rfloor$ will return the correct result, without rounding, even if $|a|$ is very large, or the exponent of $\lfloor a \rfloor$ is greater than that of a. Similar remarks apply to $\lceil b \rceil$.

The logic behind this algorithm is somewhat subtle. Note that a and b may be affected by roundoff errors, and so $\uparrow n - m \uparrow = 2$ does not imply that the original interval $\bar{x}$ actually straddles any local maxima or minima. Nevertheless, we can safely assume that it does, and use the parity of m to decide whether this presumed extremum is $+1$ or -1. A similar observation applies to the case $\uparrow n - m\uparrow > 2$. Here we are relying on an important principle of interval arithmetic: in a correctly implemented IA operation, replacing any argument $\bar{x}$ by another interval $\bar{x}'$ that contains $\bar{x}$ will not affect the validity of the result.

The code works even when $\bar{x}$ is infinite in one or both directions. The code for sin is entirely analogous, except that the local extrema are shifted by $\pi/2$. Thus, we must subtract 0.5 from a (rounding down) and from b (rounding up).

2.4.13 Other elementary functions

The preceding examples should offer sufficient guidance for the reader to implement any other elementary function f in IA, assuming the availability of a routine that computes f in floating-point with directed rounding.

When such a routine is not available, however, the task becomes much harder. Implementing, say, $\arctan x$ with directed-rounding and last-bit accuracy requires far more work than most numerical programmers can spare.

Still, if one has an algorithm $F(x)$ that computes $f(x)$ with known error bound $\delta(x)$, then one can simulate (crudely) the desired directed-rounding procedure by computing $\downarrow F(x) - \delta(x)\downarrow$ or $\uparrow F(x) + \delta(x)\uparrow$, as appropriate.

2.5 Utility operations

We will now describe some useful IA operations that are specific to intervals, rather than mere interval extensions of ordinary operations.

2.5.1 Midpoint

The *midpoint* of an interval is a finite float value contained in the interval, and as close as possible to its center $(\bar{x}.lo + \bar{x}.hi)/2$. By definition, the midpoint of a semi-infinite interval is either $-M$ or $+M$, where $M =$ `MaxFloat` is the maximum finite floating-point number; and the midpoint of $\mathbf{R}$ is 0 by convention.

Because of possible overflow, we must divide each endpoint by 2 before adding them. Because of possible underflow in the division, we must round each term in a different direction.

```
IA.mid(x̄: Interval): Float ≡
| Computes the approximate midpoint of x̄. Assumes x̄ ≠ R.
  if x̄ = [] or x̄ = R then
    return 0
  else if x̄.lo = x̄.hi then
    return x̄.lo
  else if x̄.lo = −∞ then
    return −MaxFloat
  else if x̄.hi = +∞ then
    return +MaxFloat
  else
    return ⟨↑x̄.lo/2↑ + ↓hix/2↓⟩
```

The divisions by 2 are exact, except for numbers of very small magnitude, when the quotient may be rounded by half a unit in the last bit. At worst, the sum $\uparrow\bar{x}.lo/2\uparrow + \downarrow\bar{x}.hi/2\downarrow$ will differ from the exact midpoint by half a unit in the last bit, and therefore will lie in the original interval $\bar{x}$.

The final rounding of the sum may increase the error to 3/4 of the last bit, so the returned result may not be the most accurate answer possible. In any case, since rounding cannot cross over a representable float, the result will still be inside $\bar{x}$.

2.5.2 Radius

The *half-width* or *radius* of an interval is half of the difference between the upper and lower endpoints, rounded upwards. As special cases, the half-width of unbounded intervals is $+\infty$, and that of $[\,]$ is zero.

The half-width of every bounded interval is representable as a finite float. This property makes the half-width more useful than the total width $\uparrow\bar{x}.hi - \bar{x}.lo\uparrow$, which may overflow for some bounded intervals. Another useful property of the half-width is that it is zero if and only if the interval is empty, or contains a single point.

In order to maximize the usefulness of the half-width r, we must round it in such a way that the given interval $\bar{x}$ is contained in the interval $[m - r \mathrel{..} m + r]$, where m is the midpoint of $\bar{x}$ (as computed by `IA.mid`). The simplest way to achieve this goal is to derive the half-width from the midpoint:

```
IA.rad(x̄: Interval): Float ≡
| Computes the half-width of x̄.
  if x̄ = [] or x̄.lo = x̄.hi then
    return 0
  else
    m ← IA.mid(x̄)
    return max {↑m − x̄.lo↑ .. ↑x̄.hi − m↑}
```

Thanks to the rules of IEEE arithmetic, this code will return $+\infty$ whenever $\bar{x}.lo = -\infty$ or $\bar{x}.hi = +\infty$. (Recall that $[+\infty \mathrel{..} +\infty]$ and $[-\infty \mathrel{..} -\infty]$ are not valid intervals.)

In practice, since `IA.rad` and `IA.mid` are often used together, it may be convenient to combine them into a single procedure that returns both parameters.

2.5.3 Meet (intersection)

The set-theoretical intersection, or *meet*, of two intervals $\bar{x}$ and $\bar{y}$ is another interval (possibly empty) denoted by $\bar{x} \cap \bar{y}$ or $\bar{x} \wedge \bar{y}$. The intersection is trivial to compute:

```
IA.meet(x̄, ȳ: Interval): Interval ≡
| Returns x̄ ∧ ȳ, i.e. x̄ ∩ ȳ.
  if x̄ = [] or ȳ = [] or x̄.lo > ȳ.hi or x̄.hi < ȳ.lo then
    return []
  else
    return [max {x̄.lo, ȳ.lo} .. min {x̄.hi, ȳ.hi}]
```

If the internal representation of $[\,]$ is any pair $[a \mathrel{..} b]$ with $a > b$, then the `if` test can be eliminated—the max / min formula will automatically return $[\,]$ when appropriate.

This operation is typically used when we obtain, by different lines of reasoning or computation, two ranges $\bar{x}'$ and $\bar{x}''$ that are both known to contain some quantity x. The interval $\bar{x}' \cap \bar{x}''$ condenses that information into a single interval.

2.5.4 Join (convex hull)

The union of two intervals $\bar{x}$ and $\bar{y}$ may not be a single interval. However, we can easily compute their *join*, or *convex hull*, which is the smallest interval $\bar{x} \vee \bar{y}$ that contains both:

```
IA.join(x̄, ȳ: Interval): Interval ≡
| Returns x̄ ∨ ȳ.
  if x̄ = [] then
    return ȳ
  else if ȳ = []
    return x̄
  else
    return [min {x̄.lo, ȳ.lo} ‥ max {x̄.hi, ȳ.hi}]
```

The tests for $[\,]$ can be omitted only if the empty interval is consistently represented as $[+\infty \ldots -\infty]$. If pairs like $[1 \ldots 0]$ are also allowed to represent $[\,]$, then they must be handled as special cases, as shown above.

The join operation is often used when coding the interval version of a conditional algorithm. If x is variable, then a test like

```
if x > 0
  then y ← f(x, ...)
  else y ← g(x, ...)
```

can often be translated into

$$\bar{u} \leftarrow \bar{f}(\bar{x} \wedge [0 \ldots +\infty], \ldots)$$
$$\bar{v} \leftarrow \bar{g}(\bar{x} \wedge [-\infty \ldots 0], \ldots)$$
$$\bar{y} \leftarrow \bar{u} \vee \bar{v}$$

where $\bar{f}$ and $\bar{g}$ are the interval extensions of f and g.

Note, in this example, that the case $x = 0$ is incorrectly included in both branches. The first statement should have been

$$\bar{u} \leftarrow \bar{f}(\bar{x} \wedge (0 \mathrel{..} +\infty]),$$

but standard IA only allows for closed intervals. However, as discussed in Section 2.3, the computation $\bar{f}$ must tolerate the widening of $(0 \mathrel{..} +\infty]$ to $[0 \mathrel{..} +\infty]$, even if f is undefined for $x = 0$.

2.6 The error explosion problem

The main weakness of IA is that it tends to be too conservative: the computed interval for a quantity may be much wider than the exact range of that quantity, often to the point of uselessness. This problem is particularly severe in long computation chains, where the intervals computed at one stage are inputs for the next stage. Unfortunately, such "deep" computations are not uncommon in practical applications.

This over-conservatism is mainly due to the assumption that the (unknown) values of the arguments to primitive operations may vary *independently* over the given interval. If this assumption is not valid — that is, if there are any mathematical constraints between those quantities — then not all combinations of values in the given intervals will be valid. In that case, the result interval computed by IA may be much wider than the exact range of the result quantity.

As an extreme example, when we evaluate the expression $x - x$ with IA, given the interval $\bar{x} = [2 \mathrel{..} 5]$ for x, we get $[2-3 \mathrel{..} 5-2] = [-3 \mathrel{..} +3]$ — instead of $[0 \mathrel{..} 0]$, which is the true range of the expression. The IA subtraction routine cannot tell that the two given intervals actually denote the same quantity, since they could also denote two independent quantities that just happen to have the same range.

For a less extreme (and more typical) example, consider evaluating $x(10 - x)$, where x is known to lie in the interval $\bar{x} = [4 \mathrel{..} 6]$. Applying the formulas of Section 2.4 blindly, we get

$$\begin{aligned} 10 - \bar{x} &= [10 \mathrel{..} 10] - [4 \mathrel{..} 6] = [4 \mathrel{..} 6] \\ \bar{x}(10 - \bar{x}) &= [4 \mathrel{..} 6] \cdot [4 \mathrel{..} 6] = [16 \mathrel{..} 36]. \end{aligned}$$

On the other hand, a trivial analysis shows that the true range of $x(10 - x)$ is $[24 \mathrel{..} 25]$. The relative accuracy of the IA computation is thus $(25-24)/(36-16) = 0.05$, meaning the resulting interval was 20 times wider than what it should be.

The large discrepancy between the two intervals is due to the inverse relation between the quantities x and $10-x$, which is not known to the interval multiplication algorithm. This problem affects all operations with two or more arguments: if the corresponding quantities are not independent, and are correlated in the "wrong" way, then the result interval may be much wider than necessary.

2.6.1 Error explosion

The over-conservatism of IA is particularly bad in a long computation chain, because the overall relative accuracy of the chain tends to be the product of the relative accuracies of the individual stages. In such cases, one often observes an "error explosion": as the evaluation advances down the chain, the relative accuracy of the computed intervals decreases at an exponential rate. Thus, after a few such stages the intervals may easily be too wide to be useful, by many orders of magnitude.

For an example of this phenomenon, consider the function $g(x) = \sqrt{x^2-x+1/2}/\sqrt{x^2+1/2}$. Figure 2.2a shows the graph of $g(x)$ (black curve) and the result of evaluating $g(\bar{x})$ with standard IA, for 16 consecutive equal intervals $\bar{x}$ in $[-2 \mathrel{..} +2]$. Figure 2.2b shows the same data for the second iterate $h(x) = g(g(x))$ of the same function. Although the iterates g^k converge to a constant function, the intervals $\bar{g}^k(\bar{x})$ computed by standard IA diverge.

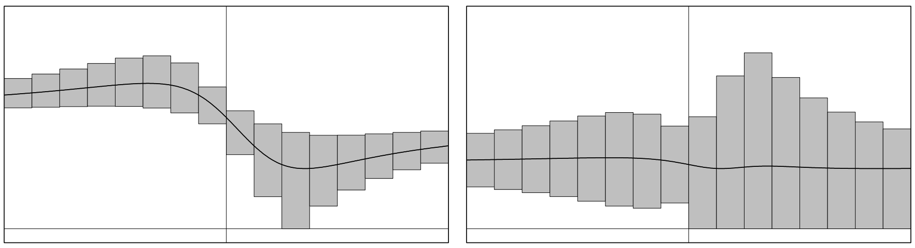

Figure 2.2: Error explosion in IA estimates for iterated functions.

2.6.2 Error explosion and subdivision depth

When the IA evaluation of $\bar{f}(\bar{x}, \bar{y}, \ldots)$ produces an interval that is too wide for the purpose at hand, we can often improve matters by partitioning the argument range $\bar{x} \times \bar{y} \times \cdots$ into two or more sub-ranges, evaluating f on each of these, and combining the results into a single interval. However, this technique is not very effective against error explosion, because the relative accuracy of an IA operation is generally independent of the width of the input intervals. (That is, IA has basically first-order approximation error.) So, if the relative accuracy of a computation is too small to be useful by a factor of 1000, then we will probably have to split the domain into 1000 sub-intervals to obtain a useful result.

2.7 Avoiding error explosion

In order to avoid this error explosion problem, we should try to arrange the computation in such a way as to avoid unfavorable correlations between the arguments of the IA operations. In particular, minimizing the number of occurrences of a variable in a formula usually results in tighter range estimates; if each variable occurs only once, then the range estimates produced by IA are exact.

Another general technique is to lump several arithmetic operations into a single "macro operation", and write a special-purpose IA routine for it. Since the routine can take into account the correlation between shared sub-expressions, it may be able to compute a tighter range for the result than which could be given by using IA at each step.

However, these remedies can only be applied to relatively simple computations over restricted domains. When the expression to be computed is determined only at run-time, or involves dozens of variables and operations, avoiding bad correlations is almost impossible. In such cases, one should consider using more sophisticated SVC models, such as those described in Chapter 3.

2.7.1 Powers

A trivial but important example of avoidable error explosion is the evaluation of powers $z \leftarrow x^n$. The naive IA implementation, based on repeated multiplication, will show poor accuracy for intervals that straddle zero. In particular, for $\bar{x} = [-2 \,\_\_\, +2]$, the evaluation of $z \leftarrow x^2$ in IA as $x \cdot x$ will give $\bar{z} = [-4 \,\_\_\, +4]$, even though x^2 cannot be negative.

For this reason, IA libraries should always include special routines for squares and other integer powers. Here is a typical example:

```
IA.sqr(x̄: Interval): Interval ≡
| Computes x̄².
  if x̄ = [] then
    return []
  else if x̄.lo ≥ 0 then
    return [⌊x̄.lo²⌋ __ ⌈x̄.hi²⌉]
  else if x̄.hi ≤ 0 then
    return [⌊x̄.hi²⌋ __ ⌈x̄.lo²⌉]
  else if x̄.hi > −x̄.lo then
    return [0 __ ⌈x̄.hi²⌉]
  else
    return [0 __ ⌈x̄.lo²⌉]
```

2.7.2 Bézier bounding for polynomials

Another important example is the evaluation of a polynomial $h(x)$ over an interval $\bar{x}$. A naive IA evaluation, either as a sum of powers or through Horner's rule, is likely to be affected by negative correlation among its terms. We can obtain a tighter range for $h(x)$ by computing the Bézier-Bernstein coefficients **[15]** of h over $\bar{x}$, and returning the smallest interval that contains them all.

2.7.3 Alternating series

As another example, consider the problem of evaluating a series $f(x) = \sum_{i=0}^{\infty} a_i x^i$. If the argument x or some of the coefficients a_i are negative, then there will often be adverse correlation between the various terms. In that case, the series computed with IA will have poor relative accuracy, even if the series itself is strongly convergent.

However, if we happen to know that the terms have alternating signs and non-increasing magnitude, then we can usually improve the relative accuracy by evaluating two terms at a time, and estimating the maximum range of the pair analytically.

As a concrete example, suppose we want to evaluate $\sin x$ by the Taylor series

$$z \leftarrow \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \tag{2.1}$$

for $\bar{x} = [0.6250 \mathrel{\_\_} 0.8750]$. (This is merely an example of series manipulation, and not necessarily the best way to compute $\sin x$.) Evaluating each term separately, we get

$$\begin{array}{l} [+0.6250 \mathrel{\_\_} +0.8750] + \\ [-0.1117 \mathrel{\_\_} -0.0406] + \\ [+0.0007 \mathrel{\_\_} +0.0043] + \\ [-0.0001 \mathrel{\_\_} 0.0000] + \\ \qquad\qquad\qquad\quad \cdots \end{array}$$

The sum of the first three intervals is $[0.5140 \mathrel{\_\_} 0.8387]$, and the sum of the first four is $[0.5139 \mathrel{\_\_} 0.8387]$. Since the terms have alternating signs, we know that the infinite series lies between these two partial sums; thus we can safely set $z \leftarrow [0.5139 \mathrel{\_\_} 0.8387]$.

Now, the true range of $\sin x$ in that interval is $[0.5850 \mathrel{\_\_} 0.7676]$. The relative accuracy is thus only 0.5622. If we continued adding more terms, the total interval would not shrink any further—in fact, it would slowly grow wider, because of accumulated roundoff error.

On the other hand, we can rewrite the series as

$$\begin{aligned} z &= \sum_{k=0}^{\infty} \left(\frac{x^{4k+1}}{(4k+1)!} - \frac{x^{4k+3}}{(4k+3)!} \right) \\ &= \sum_{k=0}^{\infty} \frac{x^{4k+1}}{(4k+1)!} \left(1 - \frac{x^2}{(4k+3)(4k+2)} \right). \end{aligned} \tag{2.2}$$

If x lies in $[0 \mathrel{\_\_} 1]$, then each term of this series is non-negative and monotonically increasing with x. So, for argument ranges $\bar{x}$ contained in that interval, a tight range for each term will be

$$r_k = [\downarrow f_k(\bar{x}.lo) \downarrow \mathrel{\_\_} \uparrow f_k(\bar{x}.hi) \uparrow],$$

where

$$f_k(x) = \frac{x^{4k+1}}{(4k+1)!}\left(1 - \frac{x^2}{(4k+3)(4k+2)}\right).$$

In our case, the first few terms are

$$\begin{aligned}
&[+0.5843 \mathrel{..} +0.7634] + \\
&[+0.0007 \mathrel{..} +0.0042] + \\
&[+0.0000 \mathrel{..} +0.0001] + \\
&[+0.0000 \mathrel{..} +0.0001] + \\
&\qquad\qquad\qquad\cdots
\end{aligned}$$

Since the original series (2.1) is alternating, and x lies in $[0 \mathrel{..} 1]$, the sum of all the terms with $k \geq 2$ in the series (2.2) lies between 0 and $1/9! < 0.0001$. Thus, we can safely replace those terms by the interval $[00.0000 \mathrel{..} +0.0001]$, and return $z \leftarrow [+0.5850 \mathrel{..} +0.7677]$. The relative accuracy is now 0.9995.

Unfortunately, these remedies are hardly applicable when the expression to be computed is determined only at run-time, or involves more than a few variables and operations.

Chapter 3

Affine arithmetic

In this chapter, we describe another method for range analysis, which we call *affine arithmetic* (AA). This model is similar to standard interval arithmetic, to the extent that it automatically keeps track of rounding and truncation errors for each computed quantity. In addition, AA keeps track of *correlations* between those quantities.

Thanks to this extra information, AA is able to provide much tighter bounds for the computed quantities, with errors that are approximately quadratic in the uncertainty of the input variables. This advantage of AA is especially noticeable in computations of great arithmetic depth or subject to cancellation errors.

As one may expect, the AA model is more complex and expensive than ordinary interval arithmetic. However, we believe that its higher accuracy will be worth the extra cost in many applications, as indicated by the examples given in Chapter 4.

3.1 Affine forms

In affine arithmetic , a partially unknown quantity x is represented by an *affine form* $\hat{x}$, which is a first-degree polynomial:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n.$$

The coefficients x_i are finite floating-point numbers, and the ε_i are symbolic real variables whose values are unknown but assumed to lie in the interval $\mathbf{U} = [-1 \mathrel{..} +1]$. We call x_0 the *central value* of the affine form $\hat{x}$; the coefficients x_i are the *partial deviations*, and the ε_i are the *noise symbols*.

Each noise symbol ε_i stands for an independent component of the total uncertainty of the quantity x; the corresponding coefficient x_i gives the magnitude of that component. The source of the uncertainty may be either "external" (due to original measurement error, indeterminacy, or numerical approximation affecting some input quantity), or "internal" (due to arithmetic roundoff, series truncation, function approximation, and other numerical errors committed in the computation of $\hat{x}$).

In particular, the internal sources of error include the need to cast the results of non-linear operations as affine forms. As we shall see in Section 3.7, this casting requires approximating a non-linear function of the noise symbols ε_i by an affine function. The error of this approximation will be represented in the result by a new noise symbol ε_k.

3.1.1 The fundamental invariant of affine arithmetic

The semantics of affine forms is formalized by the *fundamental invariant of affine arithmetic*:

> *At any stable instant in an AA computation, there is a single assignment of values from* **U** *to each of the noise variables in use at the time that makes the value of every affine form equal to the value of the corresponding quantity in the ideal computation.*

By *stable instant* we mean any time when the algorithm is not performing an AA operation.

3.2 Joint range of affine forms

The key feature of the AA model is that the same noise symbol may contribute to the uncertainty of two or more quantities (inputs, outputs, or intermediate results) arising in the evaluation of an expression.

The sharing of a noise symbol ε_i by two affine forms $\hat{x}$ and $\hat{y}$ indicates some partial dependency between the underlying quantities x and y. The magnitude and sign of the dependency is determined by the corresponding coefficients x_i and y_i. Note that the signs of the coefficients are not significant in themselves, but the relative sign of x_i and y_i defines the direction of the correlation.

For example, suppose that the quantities x and y are represented by the affine forms

$$\begin{aligned}\hat{x} &= 10 + 2\varepsilon_1 + 1\varepsilon_2 \qquad\quad - 1\varepsilon_4 \\ \hat{y} &= 20 - 3\varepsilon_1 \qquad\quad + 1\varepsilon_3 + 4\varepsilon_4.\end{aligned}$$

From this data, we can tell that x lies in the interval $\bar{x} = [6 \mathrel{\_\_} 14]$ and y lies in $\bar{y} = [12 \mathrel{\_\_} 28]$. However, since they both include the same noise variables ε_1 and ε_4 with non-zero coefficients, they are not entirely independent of each other. In fact, the pair (x, y) is constrained to lie in the dark grey region of $\mathbf{R}^2$ depicted in Figure 3.1.

Obviously, this dependency information would be lost if we were to replace $\hat{x}$ and $\hat{y}$ by the intervals $\bar{x}$ and $\bar{y}$. Taken individually, these intervals encode precisely the same ranges of values as the affine forms. Taken jointly, however, they only tell us that the pair (x, y) lies in the rectangle $\bar{x} \times \bar{y} = [6 \mathrel{\_\_} 14] \times [12 \mathrel{\_\_} 28]$, shown in light grey in Figure 3.1.

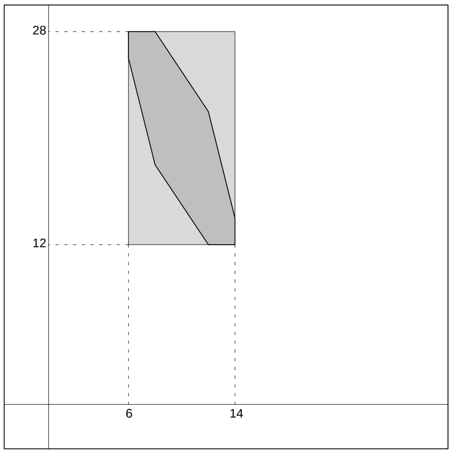

Figure 3.1: Joint range of two partially dependent quantities in AA.

3.2.1 The shape of joint ranges

In Figure 3.1, observe that the joint range of x and y is a convex polygon, symmetric around the central point (x_0, y_0). Each pair of parallel sides corresponds to a noise variable ε_i appearing in $\hat{x}$ or $\hat{y}$. The coefficients of ε_i determine the length and direction of those two sides; the corresponding plane vectors are $(2x_i, 2y_i)$ and $(-2x_i, -2y_i)$. The value pairs (x, y) lying on those sides are obtained from the affine forms by varying ε_i over $\mathbf{U}$ while all other noise variables ε_j are fixed at -1 or $+1$, in some specific pattern.

In general, if we have m affine forms depending on n noise symbols, then the set of possible joint values for the corresponding m quantities will be a center-symmetric convex polytope in $\mathbf{R}^m$. That polytope is the parallel projection on $\mathbf{R}^m$ of the hypercube $\mathbf{U}^n$ by the affine map consisting of the m affine forms.

Each k-dimensional face of this polytope corresponds to a subset $\mathcal{E}$ of k noise variables appearing in the affine forms. The points on that face are obtained by ranging the variables in $\mathcal{E}$ over $\mathbf{U}^k$, while fixing the remaining variables at some specific combination of -1 and $+1$.

3.3 Special affine forms

As in IA, it is convenient to have special affine forms [], meaning "no value", and $\mathbf{R}$, meaning "any real value".

Note that the set of values described by an ordinary affine form is necessarily bounded (because all coefficients are finite floats), and nonempty (because the set always contains the center value). Therefore, the precise computer representations of [] and $\mathbf{R}$ must be established by convention. For details over possible representations, see Section 3.17.

Note also that the special form $\mathbf{R}$ does not record any dependency information. That is, if $\hat{x} = \hat{y} = \mathbf{R}$, then we cannot infer any constraint or relationship between quantities x and y. The range of the point (x, y), as implied by those affine forms, is the whole plane $\mathbf{R}^2$.

Prospective AA implementors may be tempted to allow infinite floats as coefficients, as in $\hat{x} = 0 + \infty\,\varepsilon_k$, in order to represent unbounded quantities within the general AA model. Unfortunately, such infinite forms cannot convey the relationship between, say, x and $2x$, when x has unbounded range. Moreover, in order to avoid `NaN`s, the affine arithmetic routines would have to test for such forms, and handle them separately. Therefore, such infinite forms would be merely equivalent representations of the single special value $\mathbf{R}$, as defined above.

3.3.1 Computer representation

Computer representations of affine forms will be discussed in Section 3.17. Until then, we will describe AA algorithms in a representation-independent manner, based on the following conventions:

- We assume an unlimited supply of noise variables ε_i, each identified by its *index* i (a positive integer).
- We denote by $\mathcal{E}(\hat{x})$ the set of indices of all noise variables that appear in the affine form $\hat{x}$.
- We denote by x_i the coefficient of the noise variable ε_i in the affine form $\hat{x}$. In particular, $x_i = 0$ if $i \notin \mathcal{E}(\hat{x})$.
- The central value of $\hat{x}$ will be denoted by x_0.
- The procedure `newsym()` is assumed to return the index of a "new" noise variable, not used in any affine form computed so far.
- For any finite float value c, we denote by $\hat{c}$ its AA representation, namely the affine form $\hat{x}$ with center value $x_0 = c$ and $\mathcal{E}(\hat{x}) = \{\}$.

3.4 Conversions between IA and AA

Conversion between affine forms and ordinary intervals is often required, especially in the input and output of numerical programs. Although simple in principle, the conversions requires some care in the handling of roundoff errors. Also, special intervals such as [] and unbounded intervals must be handled separately.

Because of the unavoidable roundoff errors and overflows that may occur in the conversion between intervals and affine forms, the two models are not exactly equivalent.

In particular, some finite affine forms must be converted to infinite or semi-infinite intervals. Conversely, all semi-infinite intervals, and some finite ones, must be converted to $\mathbf{R}$.

3.4.1 Conversion from AA to IA

If a quantity x is described by the affine form $\hat{x} = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$, then its value is guaranteed to be in the interval

$$[\hat{x}] = [x_0 - \mathrm{rad}(\hat{x}) \mathrel{..} x_0 + \mathrm{rad}(\hat{x})],$$

where

$$\mathrm{rad}(\hat{x}) = \sum_{i=1}^{n} |x_i|. \tag{3.1}$$

Note that $[\hat{x}]$ is the smallest interval that contains all possible values of $\hat{x}$, assuming that each ε_i ranges independently over the interval $\mathbf{U} = [-1 \mathrel{..} +1]$.

Obviously, this conversion discards all knowledge of constraints between the computed quantities that was preserved in their affine forms.

The quantity $\mathrm{rad}(\hat{x})$ defined by (3.1) plays an important role in arithmetic operations (see Section 3.5). We call it the *total deviation* of $\hat{x}$.

```
AA.rad(x: AA.Form): Float ≡
| Computes rad(x̂).
  if x = [] then
    return 0
  else if x = R then
    return +∞
  else
    return ↑∑{|x_i| : i ∈ ℰ(x̂)}↑
```

```
IA.from.AA(x: AA.Form): Interval ≡
| Converts x to interval.
  if x = [] then
    return []
  if x = R then
    return R
  else
    r ← AA.rad(x)
    return [↓x_0 − r↓ .. ↑x_0 + r↑]
```

3.4.2 Conversion from IA to AA

Given an interval $\bar{x} = [a \mathrel{..} b]$ representing some quantity x in IA, an equivalent affine form for the same quantity is given by $\hat{x} = x_0 + x_k\varepsilon_k$, where x_0 is the midpoint of the interval, and x_k is its half-width:

$$x_0 = \frac{b+a}{2}, \quad \text{and} \quad x_k = \frac{b-a}{2}.$$

The noise symbol ε_k represents the uncertainty in the value of x that is implicit in its interval representation $\bar{x}$. Since the interval tells us nothing about possible constraints between the value of x and that of other variables, ε_k must be distinct from all other noise symbols used so far in the same computation.

```
AA.from.IA(x̄: Interval): AA.Form ≡
| Converts x̄ to affine form.
  if x̄ = [] then
    return []
  if x̄.lo = −∞ or x̄.hi = +∞ then
    return R
  else
    r ← IA.rad(x̄)
    if r = +∞ then return R
    m ← IA.mid(x̄)
    k ← newsym()
    return m + rε_k
```

Note that the correctness of this code depends on `IA.rad(x̄)` being large enough to compensate any rounding of `IA.mid(x̄)`, as explained in Section 2.5.

3.5 Computing with AA

In order to evaluate a formula with AA, we must replace each elementary operation on real quantities by a corresponding operation on their affine forms, returning an affine form.

Let's consider specifically a binary operation $z \leftarrow f(x, y)$. The corresponding AA operation $\hat{z} \leftarrow \hat{f}(\hat{x}, \hat{y})$ is a procedure that computes an affine form for $z = f(x, y)$ that is consistent with affine forms $\hat{x}, \hat{y}$.

By definition,

$$\begin{aligned} x &= x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n && (3.2)\\ y &= y_0 + y_1\varepsilon_1 + \cdot\cdot\, y_n\varepsilon_n && (3.3) \end{aligned}$$

for some (unknown) values of $\varepsilon_1, ..\, \varepsilon_n \in \mathbf{U}^n$. Therefore, the quantity z is a function of the ε_i, namely

$$\begin{aligned} z &= f(x, y) \\ &= f(x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n,\ y_0 + y_1\varepsilon_1 + \cdot\cdot\, y_n\varepsilon_n.) && (3.4) \end{aligned}$$

The challenge now is to replace $f(x, y)$ by an affine form

$$\hat{z} = z_0 + z_1\varepsilon_1 + \cdot\cdot\, z_n\varepsilon_n$$

that preserves as much information as possible about the constraints between x, y, and z that are implied by (3.2–3.4), but without implying any other constraints that cannot be deduced from the given data.

3.6 Affine operations

If the operation f itself is an affine function of its arguments x and y, then formula (3.4) can be expanded and rearranged into an affine combination of the noise symbols ε_i. Except for roundoff errors and overflows, this affine combination describes all the information about the quantities x, y, and z that can be deduced from the given affine forms $\hat{x}$ and $\hat{y}$, and the operation f. In particular, for any $\alpha, \zeta \in \mathbf{R}$,

$$\begin{aligned} \hat{x} \pm \hat{y} &= (x_0 \pm y_0) + (x_1 \pm y_1)\varepsilon_1 + \cdots + (x_n \pm y_n)\varepsilon_n \\ \alpha\hat{x} &= (\alpha x_0) + (\alpha x_1)\varepsilon_1 + \cdots + (\alpha x_n)\varepsilon_n \\ \hat{x} \pm \zeta &= (x_0 \pm \zeta) + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n. \end{aligned}$$

Note that, according to the formulas above, the difference $\hat{x} - \hat{x}$ between an affine form and itself is identically zero. The subtraction formula “knows” that, in this case, the operands are actually the same quantity, and not just two quantities that happen to have the same range of possible values, from the fact that they share the same noise symbols with the same coefficients.

By the same token, linear identities such as $(\hat{x} + \hat{y}) - \hat{x} = \hat{y}$ or $(3\hat{x}) - \hat{x} = 2\hat{x}$, which do not hold in IA, do hold in AA (except for floating-point roundoff errors). More generally, computations that consist only of affine operations with numeric coefficients will have relative accuracy near unity.

3.6.1 Negation

Negation is one of the few exact AA operations:

```
AA.neg(x̂: AA.Form): AA.Form ≡
| Computes −x.
  if x̂ = [] or x̂ = R then
    return x̂
  else
    var ẑ: AA.Form ← 0̂
    z₀ ← −x₀
    for each i in ℰ(x̂) do
      zᵢ ← −xᵢ
    return ẑ
```

3.6.2 Handling roundoff errors

For operations other than negation, we must take into account the floating-point roundoff errors that may occur when computing the coefficients of the result.

One might think that (as in IA) it suffices to round each coefficient z_i in the "safe" direction, namely away from zero. However, in AA there is no "safe" direction for rounding a partial deviation z_i. If the noise variable ε_i occurs in some other affine form $\hat{w}$, then any change in z_i — in either direction — would imply a different correlation between the quantities z and w, and would falsify the fundamental invariant of affine arithmetic.

In order to preserve the fundamental invariant, whenever a computed coefficient z_i differs from its correct value by some amount d, we must account for this error by adding an extra term $d\varepsilon_r$, where ε_r is a noise symbol that does not occur in any other affine form.

3.6.3 General affine operations

An AA operation must also handle the special forms $[\,]$ and $\mathbf{R}$. Moreover, if any coefficient of the result overflows, then the operation must return $\mathbf{R}$ as the result.

All these cases are taken into account by the following code, which computes the general affine operation in two variables: $\alpha x + \beta y + \zeta$. The routine also accepts an extra uncertainly coefficient δ, to be added to the result. This uncertainty is combined with the rounding error term.

AA.affine($\hat{x}, \hat{y}$: AA.Form; α, β, ζ: Finite; δ: Float):
AA.Form $\equiv$
| *Computes* $\alpha\hat{x} + \beta\hat{y} + \zeta \pm \delta$.
| *Assumes* $\delta > 0$.
  if $\hat{x} = [\,]$ or $\hat{y} = [\,]$ then
    return $[\,]$
  else if $\hat{x} = \mathbf{R}$ or $\hat{y} = \mathbf{R}$ or $\delta = +\infty$ then
    return $\mathbf{R}$
  else
    var $\hat{z}$: AA.Form $\leftarrow \hat{0}$
    $z_0 \leftarrow \langle \alpha x_0 + \beta y_0 + \zeta \rangle$
    if $|z_0| = +\infty$ then return $\mathbf{R}$
    $a \leftarrow {\downarrow}\alpha x_0 + \beta y_0 + \zeta{\downarrow}$
    $b \leftarrow {\uparrow}\alpha x_0 + \beta y_0 + \zeta{\uparrow}$
    $\delta \leftarrow {\uparrow}\max\{b - z_0, z_0 - a\}{\uparrow}$
    for each i in $\mathcal{E}(\hat{x}) \cup \mathcal{E}(\hat{y})$ do
      $z_i \leftarrow \langle \alpha x_i + \beta y_i \rangle$
      $a \leftarrow {\downarrow}\alpha x_i + \beta y_i{\downarrow}$
      $b \leftarrow {\uparrow}\alpha x_i + \beta y_i{\uparrow}$
      $\delta \leftarrow {\uparrow}\delta + \max\{b - z_i, z_i - a\}{\uparrow}$
    if $\delta = +\infty$ then return $\mathbf{R}$
    $k \leftarrow$ newsym(); $z_k \leftarrow \delta$
    return $\hat{z}$

Note that we are allowed to round δ away from zero only because the noise variable ε_k is not yet shared by any other affine form.

In practical implementations, it is worth having several versions of this code, specialized for addition, subtraction, scaling and translation. The inner loop can be significantly simplified in these cases.

It is quite annoying that we have to evaluate the coefficient $\alpha x_i + \beta y_i$ three times, with different rounding modes, only to obtain an upper bound on its error. We can save two multiplications, at the cost of losing one bit of precision, by computing just a and b, and then setting $z \leftarrow a$, $\delta \leftarrow \uparrow\delta + (b-a)\uparrow$.

Here is a place where a trivial redesign of the floating-point processor interface would make a significant difference in computing time. For example, suppose the processor returned in a special register `fperr` some upper bound to the roundoff error of the most recent operation performed. Then we could simplify the body of the `for` loop above by

$$
\begin{aligned}
&u \leftarrow \langle \alpha x_i \rangle\,\texttt{;}\;\; \delta \leftarrow \uparrow\delta + \texttt{fperr}\uparrow \\
&v \leftarrow \langle \beta y_i \rangle\,\texttt{;}\;\; \delta \leftarrow \uparrow\delta + \texttt{fperr}\uparrow \\
&z_i \leftarrow \langle u + v \rangle\,\texttt{;}\;\; \delta \leftarrow \uparrow\delta + \texttt{fperr}\uparrow
\end{aligned}
$$

thus saving four multiplications and two additions per coefficient.

3.7 Non-affine operations

Let's now consider the case of a non-affine operation $z \leftarrow f(x, y)$. If x and y are described by the affine forms $\hat{x}$ and $\hat{y}$, then z is described by the formula

$$
\begin{aligned}
z &= f(x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n,\; y_0 + y_1\varepsilon_1 + \cdot\cdot\, y_n\varepsilon_n) \\
&= f^*(\varepsilon_1, \ldots \varepsilon_n),
\end{aligned}
\tag{3.5}
$$

where f^* is a function from $\mathbf{U}^n$ to $\mathbf{R}$. If f^* is not affine, then z cannot be expressed exactly as an affine combination of the noise symbols ε_i. In that case, we must pick some affine function of the ε_i,

$$
f^{\mathrm{a}}(\varepsilon_1, \ldots \varepsilon_n) = z_0 + z_1\varepsilon_1 + \cdots + z_n\varepsilon_n \tag{3.6}
$$

that approximates $f^*(\varepsilon_1, \ldots \varepsilon_n)$ reasonably well over its domain $\mathbf{U}^n$, and then add to it an extra term $z_k\varepsilon_k$ to represent the error introduced by this approximation. That is, we return

$$
\begin{aligned}
\hat{z} &= f^{\mathrm{a}}(\varepsilon_1, \ldots \varepsilon_n) + z_k\varepsilon_k \\
&= z_0 + z_1\varepsilon_1 + \cdots + z_n\varepsilon_n + z_k\varepsilon_k.
\end{aligned}
$$

The term $z_k\varepsilon_k$ will represent the *residual* or *approximation error*:

$$e^*(\varepsilon_1, .. \varepsilon_n) = f^*(\varepsilon_1, .. \varepsilon_n) - f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n).$$

The noise symbol ε_k must be distinct from all other noise symbols that already appeared in the same computation, and the coefficient z_k must be an upper bound on the absolute magnitude of e^*; that is,

$$|z_k| \geq \max\{\,|e^*(\varepsilon_1, .. \varepsilon_n)| \;:\; \varepsilon_1, .. \varepsilon_n \in \mathbf{U}\,\}.$$

Note that the substitution of $z_k\varepsilon_k$ for $e^*(\varepsilon_1, .. \varepsilon_n)$ represents a loss of information: from this point on, the noise symbol ε_k will be implicitly assumed to be independent from $\varepsilon_1, .. \varepsilon_n$, when in fact it is a function of them. Any subsequent operation that takes $\hat{z}$ as input will not be aware of this constraint between ε_k and $\varepsilon_1, .. \varepsilon_n$, and therefore may return an affine form that is less precise than necessary.

However, as we shall see, if the approximation f^{a} is properly chosen, then the error term z_k will depend quadratically on the widths of the ranges of the input variables $\hat{x}$ and $\hat{y}$, so that its magnitude will decrease (even in the relative sense) as those ranges become smaller.

3.7.1 Selecting the affine approximation

There are $n+1$ degrees of freedom in the choice of the affine approximation f^{a}. In order to keep the algorithms reasonably simple and efficient, we will consider only approximations f^{a} that are themselves affine combinations of the input forms $\hat{x}$ and $\hat{y}$, that is,

$$f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n) = \alpha\hat{x} + \beta\hat{y} + \zeta. \tag{3.7}$$

Thus, we have only three parameters to determine, instead of $n+1$.

For some operations f, the most accurate affine approximation to $f^*(\varepsilon_1, .. e_n)$ may not be of the form (3.7). However, the restriction to (3.7) has relatively minor consequences. The reason is that, for smooth functions f, the difference between the two optimal approximations, restricted and unrestricted, depends quadratically on the size of the input ranges.

Moreover, for univariate functions $f(x)$ the restriction is perfectly harmless, because it can be shown that the best affine approximation to f^* is indeed of the form $\alpha\hat{x} + \zeta$.

3.7.2 The general algorithm

Once we have selected the approximation f^{a} of the form (3.7), we can use the general-purpose routine `AA.affine` of Section 3.6.3 to compute the affine form $f^{\mathrm{a}}(\hat{x}, \hat{y}) = \alpha\hat{x} + \beta\hat{y} + \zeta$. To this form, we then add the extra term $z_k\varepsilon_k$, which can be combined with the roundoff error incurred by `AA.affine`.

In summary, the general binary operation $z \leftarrow f(x, y)$ can be implemented as follows:

```
AA.Form AA.f(x̂, ŷ: AA.Form) ≡
| Computes f(x̂, ŷ).
  ⟨ Choose α, β, ζ ⟩
  ⟨ Find δ ≥ max { |f(x̂, ŷ) − (αx̂ + βŷ + ζ)| : ε1, .. εn ∈ U } ⟩
  return AA.affine(x̂, ŷ, α, β, ζ, δ)
```

Of course, this same approach can be used for operations with one argument, or more than two arguments.

3.8 Optimal affine approximations

There are many goals we can aim for when choosing the affine approximation (3.7). Accuracy is usually an important goal, but hardly the only one. We will often have to settle for a less accurate solution in exchange of efficiency, code simplicity, or other practical criteria.

3.8.1 Accuracy measures

The accuracy of the result $\hat{z}$ can be quantified in many ways. For instance, we can measure its error by the magnitude of the extra coefficient z_k. This number measures the uncertainty in the true value of quantity z that the affine form $\hat{z}$ allows but fails to relate to the argument uncertainties $\varepsilon_1, .. \varepsilon_n$.

Alternatively, we can use the volume of the polytope P_{xyz} jointly determined by the affine forms of $\hat{x}$, $\hat{y}$, and $\hat{z} = \hat{f}(\hat{x}, \hat{y})$. This volume measures the uncertainty in the location of the point (x, y, z).

Fortunately, it turns out that, for approximations of the form (3.7), the two error measures are equivalent. It is easy to see that, in this case, the polytope P_{xyz} is a prism with vertical axis and parallel oblique bases, whose projection on the x–y plane is the joint polytope P_{xy} defined by $\hat{x}$ and $\hat{y}$, and whose height in the z direction is $2|z_k|$. Therefore, the volume of P_{xyz} is $2|z_k|$ times the volume of P_{xy}. Since the latter does not depend on the approximation f^{a}, minimizing the volume of P_{xyz} is equivalent to minimizing $|z_k|$.

3.8.2 Chebyshev (minimax) approximations

Approximations that minimize the maximum absolute error are the subject of *Chebyshev approximation theory.*

Specifically, let $\mathcal{F}$ be some space of functions, (polynomials, affine forms, etc.). An element of $\mathcal{F}$ that minimizes the maximum absolute difference from a given function f over some specified domain Ω is known as a *Chebyshev* (or *minimax*) $\mathcal{F}$-approximation to f over Ω.[1]

Chebyshev approximation theory is a well-developed field with many non-trivial results and a vast literature. Fortunately for us, the subtheory of affine approximations is relatively simple and easy to understand in geometric terms.

Univariate Chebyshev affine approximations

In particular, for univariate functions, the minimax affine approximation is characterized by the following property [**47**]:

Theorem 1 *Let f be a bounded and continuous function from some closed and bounded interval $I = [a \,\_\_\, b]$ to $\mathbf{R}$. Let h be the affine function that best approximates f in I under the minimax error criterion. Then, there exist three distinct points u,v, and w in I where the error $f(x)-h(x)$ has maximum magnitude; and the sign of the error alternates when the three points are considered in increasing order.*

This theorem provides an algorithm for finding the optimum approximation in many cases, via the following corollary:

[1] Minimum-error Chebyshev approximations are not to be confused with the truncated expansions of f in the *Chebyshev orthogonal polynomial basis* [**1**]. The latter do not minimize the maximum error, although they usually come quite close to.

Theorem 2 *Let f be a bounded and twice differentiable function defined on some interval $I = [a \mathinner{\ldotp\ldotp} b]$, whose second derivative f'' does not change sign inside I. Let $f^a(x) = \alpha x + \zeta$ be its minimax affine approximation in I. Then:*

- *The coefficient α is simply $(f(b) - f(a))/(b - a)$, the slope of the line $r(x)$ that interpolates the points $(a, f(a))$ and $(b, f(b))$.*
- *The maximum absolute error will occur twice (with the same sign) at the endpoints a and b of the range, and once (with the opposite sign) at every interior point u of I where $f'(u) = \alpha$.*
- *The independent term ζ is such that $\alpha u + \zeta = (f(u) + r(u))/2$, and the maximum absolute error is $\delta = |f(u) - r(u)|/2$.*

Note that this result gives us an algorithm for finding the optimum coefficients α and ζ, as long as we can solve the equation $f'(u) = \alpha$.

Geometry of Chebyshev approximations

Recall that the goal of AA is to keep track of the relationships between the quantities occurring in a computation. When we use a Chebyshev minimum-error approximation in the computation of $\hat{z} \leftarrow f(\hat{x})$, we are in a sense trying to preserve as much information as we can about the relationship of $\hat{z}$ and $\hat{x}$. More precisely, consider the set P of all possible pairs of values (x, z) that are consistent with the affine forms $\hat{x}$ and $\hat{z}$; that is,

$$P = \{\ (x,z) :\quad x = x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n,\ z = \alpha x + \zeta + z_k\varepsilon_k,\ \varepsilon_1, \mathinner{\ldotp\ldotp} \varepsilon_n, \varepsilon_k \in \mathbf{U}\ \}.$$

The set P is a parallelogram with altitude $\bar{x}.hi - \bar{x}.lo$ and base $2z_k$, rotated 90° (see Figure 3.2). Clearly, by minimizing the approximation error δ, we are minimizing the area of this parallelogram, which we can view as a measure of how much information was lost about the relationship between x and z.

3.9 Square root

To illustrate the use of Theorem 2, let's examine in detail how the square root operation $z = \sqrt{x}$ is implemented in AA.

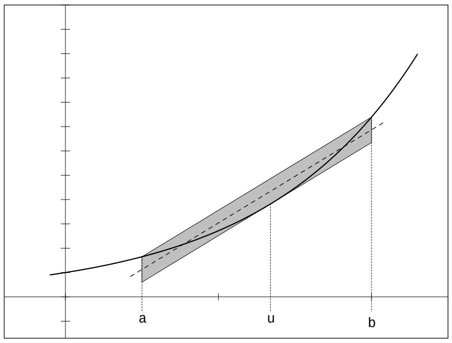

Figure 3.2: Geometry of Chebyshev approximations.

3.9.1 The "exact" solution

As explained in Section 3.7, the first step is to select a good affine approximation for the non-affine function

$$\sqrt{\hat{x}} = \sqrt{x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n},$$

when the noise variables $\varepsilon_1, .. \varepsilon_n$ range independently over $\mathbf{U}$. Since square root is an univariate function, it can be shown that the best affine approximation (in the sense of minimizing the maximum absolute error) has the form

$$\alpha\hat{x} + \zeta = \alpha(x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n) + \zeta.$$

In fact, the problem reduces to finding the best affine approximation $\alpha x + \zeta$ to the univariate function $\sqrt{x}$, when x ranges over the interval $[\hat{x}] = [a \,\_\_\, b]$. See Figure 3.3.

For the time being, let's assume that $a \geq 0$. Since $\sqrt{x}$ has negative second derivative for all positive x, Theorem 2 applies, and tells us that α is the slope of the line $r(x)$ that goes through $(a, \sqrt{a})$ and $(b, \sqrt{b})$, namely

$$\alpha = \frac{\sqrt{b} - \sqrt{a}}{b - a} = \frac{1}{\sqrt{b} + \sqrt{a}}\ . \tag{3.8}$$

The point u where the graph of $\sqrt{x}$ has slope α is the solution of $1/(2\sqrt{u}) = \alpha$, namely

$$u = \frac{1}{4\alpha^2} = \frac{a + b + 2\sqrt{a}\sqrt{b}}{4}\ . \tag{3.9}$$

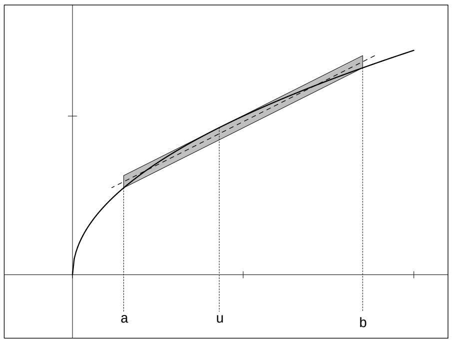

Figure 3.3: Chebyshev approximation for the square root function.

According to Theorem 2, the optimum independent term is

$$\zeta = \frac{f(u) + r(u)}{2} - \alpha u = \frac{\sqrt{a} + \sqrt{b}}{8} + \frac{1}{2}\frac{\sqrt{a}\sqrt{b}}{\sqrt{a} + \sqrt{b}} \tag{3.10}$$

and the maximum error is

$$\delta = \frac{f(u) - r(u)}{2} = \frac{1}{8}\frac{(\sqrt{b} - \sqrt{a})^2}{\sqrt{a} + \sqrt{b}} \tag{3.11}$$

The maximum absolute error δ occurs at the endpoints of the interval, where the curve lies below the line $\alpha x + \zeta$, and at the point $c = (\sqrt{a} + \sqrt{b})^2/4$, where the curve lies above the line.

Therefore, the optimal affine form for $z = \sqrt{x}$ is

$$z_0 + z_1\varepsilon_1 + \cdots z_n\varepsilon_n + z_k\varepsilon_k,$$

where ε_k is a new noise variable, and

$$z_0 = \alpha x_0 + \zeta \tag{3.12}$$

$$z_i = \alpha x_i \qquad (i = 1, .. n) \tag{3.13}$$

$$z_k = \delta \tag{3.14}$$

3.9.2 Geometric interpretation

Geometrically, these computations determine the parallelogram P with two vertical sides that encloses the graph of $\sqrt{x}$ in the interval $\bar{x}$ and has the smallest possible vertical extent (see Figure 3.3). Since the width of $\bar{x}$ is fixed, that is also the enclosing parallelogram with minimum area. The affine function $\alpha x + \zeta$ is the oblique axis of P, and the maximum error δ is half of P's vertical extent.

The parallelogram P is merely the joint range of the pair (z, x), as implied by the affine forms $\hat{x}$ and $\hat{z}$. Thus, by minimizing the maximum error, we are preserving as much information as we can about the relationship of $z = \sqrt{x}$ and x.

In contrast, consider evaluating $\bar{z} = \sqrt{\bar{x}}$ with standard IA, assuming that $\bar{x}$ and $\hat{x}$ have the same range. The pairs of values (x, y) consistent with these intervals cover the entire rectangle $R = \bar{x} \times \bar{z}$, whose area is greater area than that of P.

3.9.3 Coping with roundoff errors

Formulas (3.8–3.11) assume that we can compute α, ζ, and δ exactly. In practice, the computation of α must be carried out in floating point, and so we will get only an approximation $\tilde{\alpha}$ to the optimum slope α (see Figure 3.4).

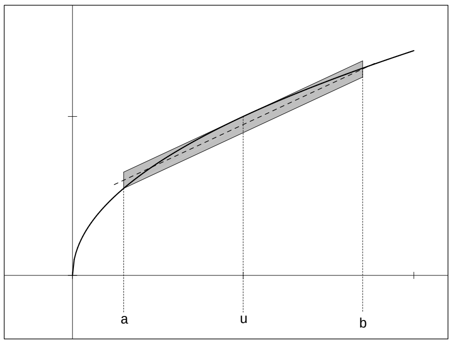

Figure 3.4: Approximation to optimum slope.

Now, to compute ζ, we cannot simply substitute $\tilde{\alpha}$ for α in formula (3.10), because the derivation of that formula used the fact that α was the slope of the chord $r(x)$. Instead, we must compute conservative estimates for the difference $\sqrt{x} - \tilde{\alpha}x$ at the endpoints of I, and at the point v of I where the slope of $\sqrt{x}$ equals $\tilde{\alpha}$:

$$\begin{aligned} d_a &= \downarrow\sqrt{a} - \tilde{\alpha}a\downarrow & (3.15)\\ d_v &= \uparrow\sqrt{v} - \tilde{\alpha}v\uparrow & (3.16)\\ d_b &= \Big\downarrow\sqrt{b} - \tilde{\alpha}b\Big\downarrow . & (3.17) \end{aligned}$$

Assuming $\tilde{\alpha}$ is close enough to α that v lies inside I, d_v will be greater than the other two values; and the difference $\sqrt{x} - \tilde{\alpha}x$, for any $x \in I$, will lie in the interval spanned by d_a, d_v, and d_b. We may then take the approximate midpoint of this range as the independent term $\tilde{\zeta}$, and its approximate radius as the maximum error estimate $\tilde{\delta}$ (see Figure 3.4).

The point v is $1/(4\tilde{\alpha}^2)$, but we do not need to compute it explicitly. Substituting symbolically $1/(4\tilde{\alpha}^2)$ into equation (3.16), we get

$$d_v = \sqrt{\frac{1}{4\tilde{\alpha}^2}} - \tilde{\alpha}\frac{1}{4\tilde{\alpha}^2} = \frac{1}{4\tilde{\alpha}} .$$

Formulas (3.12–3.14) must then be changed to use $\tilde{\alpha}$, $\tilde{\zeta}$, and $\tilde{\delta}$ instead of α, ζ, and δ. Naturally, the computation of $z_0, z_1, .. z_n$ by these formulas will be affected by roundoff errors, which must be estimated and combined with $\tilde{\delta}$, to obtain the error term z_k.

Actually, the computations become a bit simpler if we work with $\gamma = 1/\alpha$ instead of α itself. Here is the detailed code:

```
AA.sqrt(x̂: AA.Form): AA.Form ≡
| Computes √x̂.
  if x̂ = [] or x̂ = R then
    return x̂
  x̄ ← AA.to.IA(x̂) ∧ [0 __ +∞]
  if x̄ = [] then return []
  if x̄.hi = +∞ then return R
  (γ, ζ, δ) ← Cheb.sqrt(x̄)
  return AA.invaffine(x̂, γ̃, ζ, δ)
```

The routine `AA.invaffine` computes $\hat{x}/\gamma + \zeta \pm \delta$, in a manner entirely similar to `AA.affine` (Section 3.6.3).

The code for `Cheb.sqrt` is

$\texttt{Cheb.sqrt(}\bar{x}\texttt{: Interval): (}\gamma\texttt{, }\zeta\texttt{, }\delta\texttt{: Float)} \equiv$
| *Computes a Chebyshev approximation* $x/\gamma + \zeta \pm \delta$
| *to* $\sqrt{x}$ *for* $x \in \bar{x}$.
| *Assumes* $\bar{x}$ *is non-empty and bounded, and* $\bar{x}.lo \geq 0$.
  $r_a \leftarrow \big\downarrow \sqrt{\bar{x}.lo} \big\downarrow$
  $r_b \leftarrow \big\uparrow \sqrt{\bar{x}.hi} \big\uparrow$
  $\gamma \leftarrow \uparrow r_a + r_b \uparrow$
  $d_a \leftarrow \downarrow r_a(1 - r_a/\gamma) \downarrow$
  $d_b \leftarrow \downarrow r_b(1 - r_b/\gamma) \downarrow$
  $d_{\min} \leftarrow \min\{d_a, d_b\}$
  $d_{\max} \leftarrow \uparrow \gamma/4 \uparrow$
  $\zeta \leftarrow \texttt{IA.mid(}[d_{\min} \;\_\_\; d_{\max}]\texttt{)}$
  $\delta \leftarrow \texttt{IA.rad(}[d_{\min} \;\_\_\; d_{\max}]\texttt{)}$
  return (γ, ζ, δ)

There are several subtle points in this code. First, the interval $[a \;\_\_\; b]$, over which the affine approximation is computed, is not the range $[\bar{x}.lo \;\_\_\; \bar{x}.hi]$ of $\hat{x}$, but the slightly wider interval $[r_a^2 \;\_\_\; r_b^2]$. That is, a and b are defined retroactively as the (exact) squares of the (approximate) square roots r_a and r_b. This convention allows us to avoid some square roots. For instance in the computation of $d_b = \big\downarrow \sqrt{b} - b/\gamma \big\downarrow$, where we need $\sqrt{b}$ rounded down, we can use r_b in its place, because $\big\downarrow \sqrt{b} \big\downarrow = \big\uparrow \sqrt{b} \big\uparrow = r_b$. On the other hand, we must use $\uparrow r_b \uparrow^2$ instead of $\bar{x}.hi$ for the second b in that formula.

The procedure also assumes that the value γ lies between $2r_a$ and $2r_b$, which is true in the IEEE floating-point standard. This condition ensures that the maximum of $\sqrt{x} - x/\gamma$ lies between r_a and r_b, and therefore that the minimum is either at r_a or r_b. Actually, for maximum accuracy, γ should be rounded to the nearest `Float`. Rounding up (or down) is more efficient, however, and the precision loss is minimal.

Note that the implied range of $\hat{x}$ is clipped to the interval $[0 \;\_\_\; +\infty]$. As we observed in Section 2.3.3, it is convenient to assume that any intrusion of $\bar{x}$ into the negative numbers may be due to sloppiness of the range computation, and doesn't imply that the real quantity x can assume negative values.

The handling of overflows could be improved. As it is, the procedure returns $\mathbf{R}$ if the computation of $[\hat{x}]$ overflows, which may happen if some of the coefficients x_i are near the end of the finite `Float` range. However, the square root of $|x_0| + |x_1| + \cdots |x_n|$ is always a finite value, so the overflow could be avoided with some care. Spurious overflows may also occur in the computation of $\uparrow r - d^2 \uparrow$.

Note that the cost of this algorithm is essentially two square roots, plus a few floating-point operations for each x_i.

3.9.4 Overshoot

The use of a Chebyshev approximation in the computation of $\sqrt{\hat{x}}$ has one significant drawback. Note that the range of values for z that is implied by the affine form $\hat{z}$ (that is, the vertical extent of the parallelogram P in Figure 3.2) is actually *wider* than the range that would be computed using ordinary interval arithmetic! The explanation is that the new noise variable ε_k actually has a hidden (non-linear) dependency on the other noise variables, such that the value of $z_k\varepsilon_k$ is negative when the other terms approach the maximum value. If we were to take this dependency into account, we would conclude that the maximum value of $\hat{z}$ is indeed $\sqrt{b}$; but since we assume that the ε_i are independent, we must count $z_k\varepsilon_k$ as positive at the upper end of the interval.

This problem is particularly vexing when the range of x, as implied by its affine form, is partially negative. If we compute $\hat{z}$ as described, the implied range for z will contain some negative values—even though the square root function is never negative.

So, the affine form based on Chebyshev approximation trades some knowledge about the range of z for knowledge about the relationship between z and x. If there is any merit to the AA approach, then in complex computations the trade should generally be worthwhile: that is, in subsequent operation we hope to gain enough by cancellation of noise terms to compensate for the extra-wide interval.

In any case, recall that the coefficient z_k depends quadratically on the width of the input interval, which is still a qualitative improvement over the first-order errors of ordinary interval arithmetic.

3.10 The min-range approximation

As a matter of fact, we *can* produce a affine form $\hat{z}$ for $\sqrt{\hat{x}}$ that implies a tight range for z, if we settle for less than the optimal Chebyshev approximation. We only need to choose the coefficients α and ζ in such a way that the joint range P of the forms $\hat{x}$ and $\hat{z} = \alpha\hat{x} + \zeta \pm \delta$ has the same vertical extent as the piece of the graph of $\sqrt{x}$ subtended by the interval $[a \mathrel{..} b]$ (see Figure 3.5).

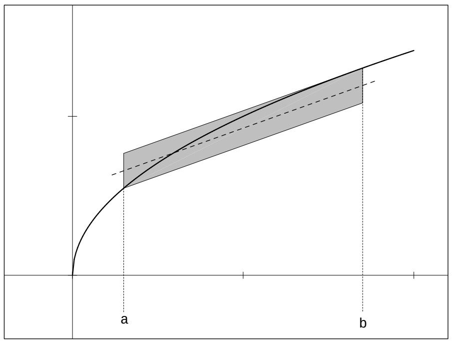

Figure 3.5: Min-range approximation for the square root.

It is easy to see that the smallest such parallelogram has the top side tangent to the graph, at the higher endpoint of the interval $[a \mathrel{..} b]$. The parameters of this approximation are

$$\alpha = \frac{1}{2\sqrt{b}}, \qquad \zeta = \frac{\sqrt{b}}{2}, \qquad \delta = \frac{(\sqrt{b} - \sqrt{a})^2}{2\sqrt{b}}.$$

We say that $\alpha x + \zeta \pm \delta$ is a *min-range affine approximation* to $\sqrt{x}$ in the interval $[a \mathrel{..} b]$.

Note that the set P is still a proper subset of the rectangle $R = [a \mathrel{..} b] \times [\sqrt{a} \mathrel{..} \sqrt{b}]$; so the resulting affine form $\hat{z}$ is strictly more informative than the result of ordinary interval arithmetic. Moreover, the ratio of the areas is

$$\frac{|P|}{|R|} = \frac{2\delta}{\sqrt{b} - \sqrt{a}} = 1 - \sqrt{\frac{a}{b}} = 1 - \sqrt{1 - \frac{b-a}{b}}\,,$$

which goes to zero linearly as the relative width $(b - a)/b$ goes to zero. In other words, the modified AA approximation above still has higher order of convergence than IA.

Incidentally, the IA square root algorithm can be viewed as this same idea taken to the extreme, where the parallelogram P is the whole rectangle $[a \mathrel{..} b] \times [\sqrt{a} \mathrel{..} \sqrt{b}]$ (see Figure 3.6). This corresponds to approximating $\sqrt{x}$ by a constant function in the interval $[a \mathrel{..} b]$, that is, choosing

$$\alpha = 0, \qquad \zeta = \frac{\sqrt{a}+\sqrt{b}}{2}, \qquad \delta = \frac{\sqrt{b}-\sqrt{a}}{2}.$$

With these choices, the returned affine form $\hat{z} = \alpha\hat{x} + \zeta + \delta\varepsilon_k$ contains only the independent term ζ and the single noise term $\delta\varepsilon_k$, which is not related to any other quantity; and this is essentially the affine form interpretation of the ordinary interval $[\sqrt{a} \mathrel{..} \sqrt{b}]$.

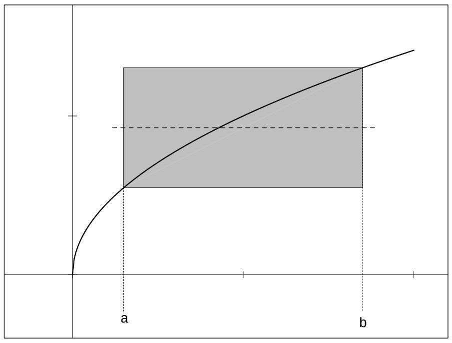

Figure 3.6: The IA approximation for square root.

There are other reasons that may justify the choice of a sub-optimal affine approximation, such as avoiding overflows, simplifying the algebra or the handling of rounding errors. We will see some examples in the following sections.

3.10.1 Handling roundoff errors

The routine `MinRange.sqrt` below implements the min-range approximation formulas, with due care for roundoff errors. It is meant to be a replacement for the routine `Cheb.sqrt`, called in `AA.sqrt`.

As in `Cheb.sqrt`, the approximation is actually computed for the interval $I = [r_a^2 \mathrel{..} r_b^2]$, which contains $\bar{x}$. Since $1/\gamma \leq (\sqrt{x})'$ for all $x \in I$, the difference $\sqrt{x} - x/\gamma$ is minimum at $x = r_a^2$ and maximum at $x = r_b^2$. The correctness of the result then follows.

`MinRange.sqrt(`$\bar{x}$`: Interval): (`γ`, `ζ`, `δ`: Float)` $\equiv$
| *Computes an affine approximation* $x/\gamma + \zeta \pm \delta$
| *to* $\sqrt{x}$ *for* $x \in \bar{x}$*, minimizing the output range.*
| *Assumes* $\bar{x}$ *is non-empty and bounded, and* $\bar{x}.lo \geq 0$.
  $r_a \leftarrow \left\downarrow \sqrt{\bar{x}.lo} \right\downarrow$
  $r_b \leftarrow \left\uparrow \sqrt{\bar{x}.hi} \right\uparrow$
  $\tilde{\gamma} \leftarrow \uparrow 2 r_b \uparrow$
  $d_{\min} \leftarrow \downarrow r_a(1 - r_a/\tilde{\gamma}) \downarrow$
  $d_{\max} \leftarrow \uparrow r_b(1 - r_b/\tilde{\gamma}) \uparrow$
  $\tilde{\zeta} \leftarrow$ `IA.mid(`$[d_{\min} \mathrel{\_\_} d_{\max}]$`)`
  $\tilde{\delta} \leftarrow$ `IA.rad(`$[d_{\min} \mathrel{\_\_} d_{\max}]$`)`
  return (γ, ζ, δ)

3.11 Exponential

The exponential function $f(x) = \exp(x) = e^x$ in AA is quite similar to square root. The key step is computing an affine approximation $f^{\mathrm{a}}(x) = \alpha x + \zeta \pm \delta$ to $\exp(x)$ in the interval $\bar{x} = [\hat{x}]$.

3.11.1 The Chebyshev approximation

Since the second derivative of exp is everywhere positive, the Chebyshev approximation is parallel to the chord, i.e.,

$$\alpha = \frac{e^b - e^a}{b - a}.$$

We cannot compute the exact value of α, but only some approximation $\tilde{\alpha}$. In any case, as long as $\tilde{\alpha}$ lies between e^a and e^b, the difference $e^x - \tilde{\alpha}x$ will be maximum at either $x = a$ or $x = b$, and minimum at $x = u = \log \tilde{\alpha}$, the abscissa where e^x has slope $\tilde{\alpha}$. If $\tilde{\alpha} \geq e^b$, then the difference $e^x - \tilde{\alpha}x$ will be maximum at either $x = a$ and minimum at $x = b$. See Figure 3.7a. Either way, from these extremal differences we can compute ζ and δ, as in the square root formulas.

The complete procedure is:

```
AA.exp(x̂: AA.Form): AA.Form ≡
| Computes exp(x̂).
  if x̂ = [] or x̂ = R then
    return x̂
  x̄ ← AA.to.IA(x̂)
  if x̄ = [] then return []
  if x̄.hi = +∞ then return R
  (α, ζ, δ) ← Cheb.exp(x̄)
  return AA.affine(x̂, α, ζ, δ)
```

where Cheb.exp is

```
Cheb.exp(x̄: Interval): (α, ζ, δ: Float) ≡
| Computes a Chebyshev approximation αx + ζ ± δ
| to exp(x) for x ∈ x̄.
| Assumes x̄ is non-empty and bounded from above.
  e_b ← ↑exp(x̄.hi)↑
  w ← ↓x̄.hi − x̄.lo↓
  if w = +∞ then
    e_a ← 0
    α ← 0
  else
    e_a ← ↓exp(x̄.lo)↓
    α ← ↑(e_b − e_a)/w↑
  if α = 0 then
    d_min ← e_a
    d_max ← e_b
  else if α ≥ e_b then
    d_min ← ↓exp(x̄.hi) − αx̄.hi↓
    d_max ← ↑exp(x̄.lo) − αx̄.lo↑
  else
    d_a ← ↑exp(x̄.lo) − αx̄.lo↑
    d_b ← ↑e_b − αx̄.hi↑
    d_min ← ↓α(1 − log α)↓
    d_max ← max {d_a, d_b}
  ζ ← IA.mid([d_min __ d_max])
  δ ← IA.rad([d_min __ d_max])
  return (α, ζ, δ)
```

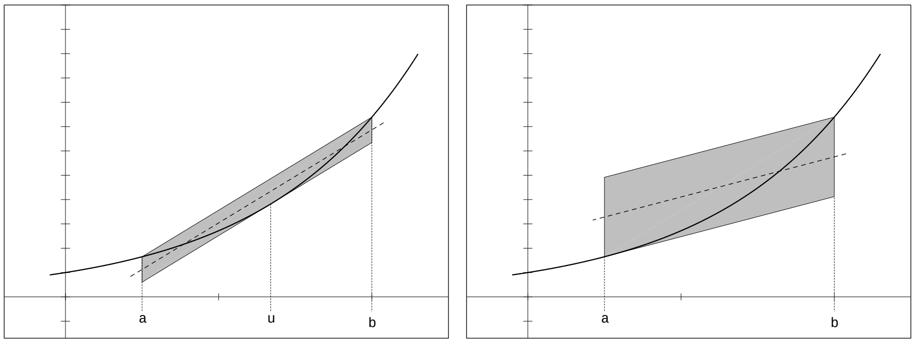

Figure 3.7: Chebyshev (left) and min-range (right) approximation for exp.

3.11.2 The min-range approximation

The Chebyshev approximation is somewhat tricky and expensive to compute, and is plagued by large undershoots when the interval is moderately wide. In particular, it extends into the negative range for $\mathrm{rad}(\hat{x}) > 1$. The reason is obvious if one looks at a plot of $y = e^x$ over such a wide range.

In practice, therefore, one may prefer to use the min-range approximation (see Section 3.10), which is easier to compute and has no over- or undershoot. This approximation preserves less information on the dependency between x and e^x; but this loss is significant only for wide intervals, where the dependency is mostly non-linear anyway. For small intervals, the min-range approximation still has quadratic error.

In the min-range approximation, the slope α is chosen as the derivative of e^x at the lowest end of the argument interval $[a \ \_\_ \ b]$, that is, e^a. See Figure 3.7b. The coefficients ζ and δ are then computed as in `Cheb.exp`.

```
MinRange.exp(x̄: Interval): (α, ζ, δ: Float) ≡
| Computes a min-range affine approximation αx + ζ ± δ
| to exp(x) for x ∈ x̄. Assumes x̄ is non-empty and bounded.
  e_a ← ↓exp(x̄.lo)↓
  e_b ← ↑exp(x̄.hi)↑
  α ← e_a
  if α = 0 then
    d_max ← e_b
    d_min ← 0
  else
    d_max ← ↑e_b − αx̄.hi↑
    d_min ← ↓e_a − αx̄.lo↓
  ζ ← IA.mid([d_min ‥ d_max])
  δ ← IA.rad([d_min ‥ d_max])
  return (α, ζ, δ)
```

3.12 Reciprocal

For the reciprocal $f(x) = 1/x$ in AA, we proceed pretty much as for the square root; except that overflow and undershoot are a major concern. For these reasons, and for the sake of code simplicity, it seems best to use the min-range approximation, instead of the Chebyshev one (Figure 3.8).

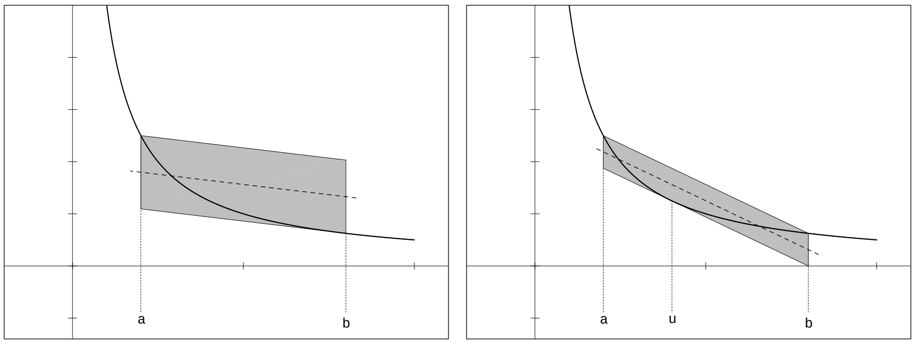

Figure 3.8: Min-range (left) and Chebyshev (right) approximation for $1/x$.

If the interval $[\hat{x}] = [a \mathrel{{.}\,{.}} b]$ includes zero, then the reciprocal may have arbitrarily large and/or arbitrarily small values, so the only valid result is $\mathbf{R}$. If the interval is entirely positive ($a > 0$), then the slope of the min-range approximation is the derivative of $1/x$ at $x = b$, namely $\alpha = -1/b^2$. We must round $1/b^2$ downwards, in order to prevent overflow. The case of negative argument range ($b < 0$) is similar.

```
AA.inv(x̂: AA.Form): AA.Form ≡
| Computes 1/x̂.
  if x̂ = [] or x̂ = R then
    return x̂
  x̄ ← AA.to.IA(x̂)
  if x̄ = [] then return []
  if x̄.lo ≤ 0 and x̄.hi ≥ 0 then return R
  (α, ζ, δ) ← MinRange.inv(x̄)
  return AA.affine(x̂, α, ζ, δ)
```

where `MinRange.inv` is

```
MinRange.inv(x̄: Interval): (α, ζ, δ: Float) ≡
| Computes a min-range approximation αx + ζ ± δ
| to 1/x for x ∈ x̄.
| Assumes x̄ is non-empty and zero-free.
  a ← min{|x̄.lo|, |x̄.hi|}
  b ← max{|x̄.lo|, |x̄.hi|}
  α ← −↓1/b²↓
  d_max ← ↑1/a − αa↑
  d_min ← ↓1/b − αb↓
  ζ ← IA.mid([d_min __ d_max])
  if x̄.lo < 0 then ζ ← −ζ
  δ ← IA.rad([d_min __ d_max])
  return (α, ζ, δ)
```

3.13 Multiplication

Let's now consider the multiplication of affine forms; that is, the evaluation of $z = f(x, y) = xy$, given the affine forms $\hat{x}$ and $\hat{y}$ for the operands x and y.

The product of the affine forms is, of course, a quadratic polynomial $f^*(\varepsilon_1, .. \varepsilon_n)$ on the noise symbols:

$$\begin{aligned} f^*(\varepsilon_1, .. \varepsilon_n) &= \hat{x} \cdot \hat{y} \\ &= (x_0 + \sum_{i=1}^{n} x_i \varepsilon_i) \cdot (y_0 + \sum_{i=1}^{n} y_i \varepsilon_i) \\ &= x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\, \varepsilon_i + (\sum_{i=1}^{n} x_i \varepsilon_i) \cdot (\sum_{i=1}^{n} y_i \varepsilon_i). \end{aligned}$$

It is not hard to see that the best affine approximation to $f^*(\varepsilon_1, .. \varepsilon_n)$ consists of the affine terms from the expansion above

$$A(\varepsilon_1, .. \varepsilon_n) = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) \varepsilon_i,$$

plus the best affine approximation to the last term

$$Q(\varepsilon_1, .. \varepsilon_n) = (\sum_{i=1}^{n} x_i \varepsilon_i) \cdot (\sum_{i=1}^{n} y_i \varepsilon_i) = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i y_j \, \varepsilon_i \varepsilon_j .$$

Observe that Q is a center-symmetric function, in the sense that $Q(-\varepsilon_1, .. -\varepsilon_n) = Q(\varepsilon_1, .. \varepsilon_n)$. Moreover, its domain $\mathbf{U}^n$ is also center-symmetric, that is, $(\varepsilon_1, .. \varepsilon_n) \in \mathbf{U}^n$ iff $(-\varepsilon_1, .. -\varepsilon_n) \in \mathbf{U}^n$. From these properties, it follows easily that the best (Chebyshev) affine approximation to Q over $\mathbf{U}^n$ is itself a center-symmetric affine function — that is to say, a constant function.

More precisely, if a and b are the minimum and maximum values of $Q(\varepsilon_1, .. \varepsilon_n)$ over $\mathbf{U}^n$, then the best affine approximation to the latter is the constant function $(a + b)/2$, and its maximum error is $(b - a)/2$. Thus, we should return

$$\hat{z} = A(\varepsilon_1, .. \varepsilon_n) + \frac{a + b}{2} + \frac{b - a}{2} \varepsilon_k,$$

where ε_k is a "new" noise symbol.

Computing the extremal values a and b of Q in U is not trivial. The best algorithm known to the authors runs in $O(m \log m)$ time, where m is the number of nonzero noise terms in $\hat{x}$ and $\hat{y}$.

Fortunately, the exact bounds are not necessary. We can use instead the trivial range estimate $\bar{Q} = \pm\,\mathrm{rad}(\hat{x})\,\mathrm{rad}(\hat{y})$. That is, we can return

$$\hat{z} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\varepsilon_i + \mathrm{rad}(\hat{x})\,\mathrm{rad}(\hat{y})\varepsilon_k\,. \tag{3.18}$$

It can be shown that the error of this approximation is at most four times the error of the best affine approximation. The worst case happens when the joint range of $\hat{x}$ and $\hat{y}$ is a square rotated 45° with respect to the axes; that is, when $\hat{x} = x_0 + \hat{u} + \hat{v}$ and $\hat{y} = y_0 + \hat{u} - \hat{v}$, where $\hat{u}$ and $\hat{v}$ are affine forms with the same range $[-r \;\_\_\; +r]$, but disjoint noise symbols. In that case, the residual Q is $\hat{u}^2 - \hat{v}^2$, and its true range is $[-r^2 \;\_\_\; +r^2]$. On the other hand, since $\mathrm{rad}(\hat{x}) = \mathrm{rad}(\hat{y}) = 2r$, the trivial estimate will be $\bar{Q} = [-4r^2 \;\_\_\; +4r^2]$.

The reader may have noticed that the trivial estimate $\bar{Q}$ is precisely the range of Q as it would be computed by standard IA, without taking into account any correlations between the two factors. But then, how could the result of formula (3.18) be more accurate that the ordinary IA product of $[\hat{x}]$ and $[\hat{y}]$? The answer is that formula (3.18) uses standard IA only to estimate the quadratic residual. The formula still allows negatively correlated terms to cancel out in the linear part; whereas in standard IA even the linear part may be overestimated.

Indeed, even though it may be four times wider than the optimum, the trivial estimate $\bar{Q} = \pm\,\mathrm{rad}(\hat{x})\,\mathrm{rad}(\hat{y})$ is still quadratic in the total width of the input intervals $[\hat{x}]$ and $[\hat{y}]$; which is enough to make the AA multiplication asymptotically more accurate than standard IA, as the input ranges get smaller.

3.13.1 Multiplication example

To illustrate these formulas, let's evaluate the expression

$$z = (10 + x + r) \cdot (10 - x + s)$$

for $x \in [-2 \;\_\_\; +2]$, $r \in [-1 \;\_\_\; +1]$, and $s \in [-1 \;\_\_\; +1]$. Converting the ordinary intervals to affine forms, we get

$$x = \;0 + 2\varepsilon_1, \qquad r = \;0 + 1\varepsilon_2 \qquad s = \;0 + 1\varepsilon_3.$$

Therefore

$$\begin{aligned} 10 + x + r &= 10 + 2\varepsilon_1 + 1\varepsilon_2 \\ 10 - x + s &= 10 - 2\varepsilon_1 + 1\varepsilon_3 \\ z &= (10 + x + r) \cdot (10 - x + s) \\ &= 100 + 10\varepsilon_2 + 10\varepsilon_3 + (2\varepsilon_1 + 1\varepsilon_2)(-2\varepsilon_1 + 1\varepsilon_3). \end{aligned}$$

In the quadratic term, each factor (considered independently) has the range $[-3 \mathbin{\_\_} +3]$. Therefore, a quick estimate for the range of that term is

$$\bar{Q} = [-(3 \cdot 3) \mathbin{\_\_} +(3 \cdot 3)] = [-9 \mathbin{\_\_} +9].$$

Using this estimate, we obtain for z the affine form

$$\hat{z} = 100 \ + \ 10\varepsilon_2 + 10\varepsilon_3 \ + \ 9\varepsilon_4.$$

The range of z implied by this affine form above is

$$[100 - 29 \mathbin{\_\_} 100 + 29] = [71 \mathbin{\_\_} 129].$$

A more precise analysis reveals that the true range of the quadratic term $\bar{Q}$ above, assuming the input noise symbols $\varepsilon_1, .. e_3$ are independent, is actually $[-9 \mathbin{\_\_} 1]$; and that of the product z is $[71 \mathbin{\_\_} 121]$. The relative accuracy of this AA computation is therefore $(121 - 71)/(129 - 71) = 0.86$. For comparison, standard IA would return

$$[7 \mathbin{\_\_} 13] \cdot [7 \mathbin{\_\_} 13] = [49 \mathbin{\_\_} 169],$$

whose relative accuracy is $(169 - 49)/(129 - 71) = 0.42$. In words, the IA interval is more than twice as wide as it should be, whereas the AA result is only 15% wider.

Observe that, in this example, the uncertainty associated to the noise symbol ε_1, which is shared by both operands, happened to cancel out to first order in the final result. This cancellation does not occur in the IA computation, which is the main reason for the larger uncertainty in the IA result.

3.13.2 The multiplication routine

As usual, the implementation must also estimate the roundoff errors and add them to the term z_k. Here is the complete code:

```
AA.mul(x̂, ŷ: AA.Form): AA.Form ≡
| Computes x̂ · ŷ.
  if x̂ = [] or ŷ = [] then
    return []
  else if x̂ = R or ŷ = R then
    return R
  else
    var p̄: Interval
    r_x ← AA.rad(x̂)
    r_y ← AA.rad(ŷ)
    δ ← ↑r_x r_y↑
    p̄ ← IA.mul([x0 __ x0], [y0 __ y0])
    α ← y0
    β ← x0
    ζ ← −IA.mid(p̄)
    δ ← ↑δ + IA.rad(p̄)↑
    return AA.affine(x̂, ŷ, α, β, ζ, δ)
```

Note that the form $\alpha\hat{x} + \beta\hat{y}$ includes two instances of the term $x_0 y_0$, one of which is canceled by the term ζ. Implementors should consider expanding the call to `AA.affine` in-line and optimizing the computation of z_0 to be just ζ.

3.14 Division

Division of affine forms is harder than multiplication. To begin with, division by quantities that may wander close to zero (that is, whose uncertainty is comparable to their average magnitude) is inherently inaccurate and unstable: a modest slop in the computation of the divisor may cause its range to overlap zero, in which case the division cannot be carried out.

That being said, there are many ways to compute an acceptable affine form $\hat{z}$ for the quotient $\hat{x}/\hat{y}$. The simplest is to rewrite $\hat{x}/\hat{y}$ as a product

$\hat{x} \times (1/\hat{y})$. This two-step approach does have quadratic convergence, because any first order correlations between $\hat{x}$ and $\hat{y}$ are preserved by the AA reciprocal routine.

```
AA.div(x̂, ŷ: AA.Form): AA.Form ≡
| Computes x̂/ŷ.
  return AA.mul(x̂, AA.inv(ŷ))
```

3.15 The mixed AA/IA model

The overshoot (and undershoot) problem that we observed in the analysis of $\sqrt{x}$ and $\exp(x)$ show that AA's goal of recording the correlation between quantities leads sometimes to range estimates for individual variables that are worse than those produced by standard IA. This problem is more likely to happen in simple computations, where uncertainty cancellation does not have a chance to occur.

We have already seen one way of coping with this problem, namely using min-range approximations instead of Chebyshev ones. This solution does cure the overshoot problem, but loses some of the correlation information. For example, the min-range approximation to $\sin x$ over $[-\pi/2 \,\_\_\, \pi/2]$ is 0 ± 1, which contains no hint of the fact that $\sin x$ is monotonically increasing in that interval.

Another solution to the overshoot problem, which actually improves the overall accuracy of computations, is to combine the AA and IA representations in a single model. That is, the representation $\hat{\bar{x}}$ of a quantity x consists of both an ordinary interval $\bar{x}$ and an affine form $\hat{x}$. The purpose of the former is to provide tight ranges for individual variables in simple operations, while the latter is optimized to record correlations between quantities. The joint range implied by representations $\hat{\bar{x}}$, $\hat{\bar{y}}$, ... is then the intersection of the joint range of $\hat{x}$, $\hat{y}$, ... (a center-symmetric convex polytope) and the box $\bar{x} \times \bar{y} \times \cdots$.

There is more to this *mixed AA/IA model* (AAIA) than just performing the same computation in AA and IA and intersecting the resulting ranges. The two models can and should interact synergistically at each step, with each model using the other's information to improve its own accuracy.

Specifically, the AAIA procedure that implements $\hat{\bar{z}} \leftarrow \hat{\bar{f}}(\hat{\bar{x}}, \hat{\bar{y}})$ will use the IA ranges of the arguments $\bar{x}$ and $\bar{y}$ when selecting the affine approximation $\alpha\hat{x} + \beta\hat{y} + \zeta \pm \delta$ for f, and use it to compute the AA component $\hat{z} \leftarrow \hat{f}(\hat{x}, \hat{y})$ of the result. The procedure will then compute the IA component as $\bar{z} \leftarrow \bar{f}(\bar{x}, \bar{y}) \wedge [\hat{z}]$.

Since the IA ranges $\bar{x}$ and $\bar{y}$ generally tighter than the AA-implied ranges $[\hat{x}]$ and $[\hat{y}]$, the affine approximation chosen by the AAIA procedure is likely to be tighter (in the sense of having a smaller error term δ) than its pure-AA counterpart. Conversely, whenever the correlation information results in an accurate AA form, the interval $\bar{z}$ will be tighter than its pure-IA counterpart.

Thus, the mixed AA/IA model can often produce better results than running the IA and AA computations in parallel, and may produce usable results even in cases where both pure models fail due to error explosion.

3.16 Comparing AA and IA

Numerical experiments seems to confirm our claim that AA is in general substantially more precise than standard IA, and less prone to error explosion.

Obviously, AA is more complex (and expensive) than IA. However, we believe that its higher accuracy will be worth the extra cost in many fields where IA's "error explosion" may be a problem, such as computer graphics. We shall see some examples in Chapter 4.

3.16.1 Example: iterated functions

For instance, consider the function

$$g(x) = \sqrt{x^2 - x + 1/2}/\sqrt{x^2 + 1/2}\ ,$$

which we used in Section 2.6.1 to illustrate the error explosion problem. Figure 3.9 shows the result of evaluating g and its iterate $h(x) = g(g(x))$ with AA, over 16 equal intervals in $[-2 \,\_\_\, +2]$. This picture should be compared to Figure 2.2, which shows the results of standard IA over the same intervals.

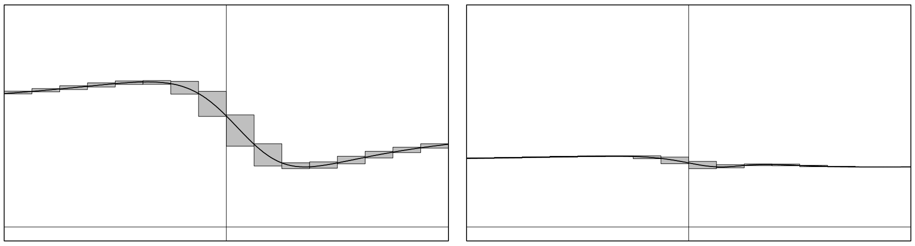

Figure 3.9: Avoiding error explosion in iterated functions with AA.

3.16.2 Cancellation of uncertainty

The main reason why AA is usually more accurate than IA is the cancellation phenomenon described in Section 3.13, which tends to make the range of computed quantities smaller than the corresponding intervals computed by standard IA. Indeed, except for roundoff errors, any computation chain that involves only affine operations will be evaluated by AA with relative accuracy 1 — that is, the range of the computed affine form will be the true range of the corresponding quantity.

3.16.3 Cancellation of internal errors

Another feature of AA is that the affine form of each computed quantity keeps track of how much of its uncertainty is attributable to the linearization and roundoff errors committed at each previous step, separately. Thus, these linearization errors themselves may cancel out in later operations, instead of always adding up (as they usually do in IA).

For example, let $\hat{x} = x_0 + x_1\varepsilon_1$ and $\hat{y} = y_0 + y_2\varepsilon_2$, and consider the following AA computation:

$$\hat{u} \leftarrow \hat{x}/\hat{y}; \quad \hat{v} \leftarrow \sqrt{\hat{u}}; \quad \hat{z} \leftarrow \hat{u} - \hat{v}.$$

The first step will compute an affine form $\hat{u} = u_0 + u_1\varepsilon_1 + u_2\varepsilon_2 + u_3\varepsilon_3$, where the term $u_3\varepsilon_3$ represents the linearization and roundoff errors of the division. Similarly, the second step will compute $\hat{v} = v_0 + v_1\varepsilon_1 + v_2\varepsilon_2 + v_3\varepsilon_3 + v_4\varepsilon_4$, where $v_4\varepsilon_4$ represents the linearization and roundoff errors of the square root. Note the term $v_3\varepsilon_3$, which records the uncertainty in v that was inherited from the previous division step. In the last step, this term will be *subtracted* from $u_3\varepsilon_3$, meaning that the error committed in the division does not affect $\hat{z}$ as much as it affects $\hat{u}$.

Needless to say, in standard IA the errors corresponding to v_3 and u_3 would be added, instead of subtracted.

3.16.4 Quadratic convergence

Since the AA model keeps track of the first-order dependency between variables, the loss of information in an AA computation—as measured by the "new" error coefficients z_k—will in general depend quadratically on the size of the input intervals. Therefore, as the ranges of the operands get smaller, the error term $z_k \varepsilon_k$ will become less important—not only in absolute terms, but also relative to the other terms.

That is, in AA the *relative* accuracy of each operation (Section 1.2.4) will be inversely proportional to the width of the input intervals. Thus, in a long computation chain, halving the input intervals will not just halve the output ones, but will also make all steps of the chain more accurate, and therefore improve the accuracy of the result by a factor that is roughly exponential in the length of the chain.

One should keep in mind, however, that quadratic convergence applies only to the affine approximation errors, and not to arithmetic roundoff errors, which are proportional to the magnitude of the *quantities*, irrespective of their uncertainties. The relative roundoff errors are small (about 10^{-15} for double precision coefficients), but they do define a lower limit to the size of input intervals. Once roundoff errors begin to dominate the uncertainty of the result, reducing the width of the inputs (e.g., by domain subdivision) will not be of much help.

3.16.5 When to use AA

Since AA errors are quadratic, whereas IA errors are linear, there is usually a critical width for the input intervals beyond which AA is not accurate enough to be worth its added expense. Therefore, in applications such as global optimization and zero finding, which depend on recursive domain exploration, one should ideally use the faster IA model at first, and switch to AA once the subregions have become small enough.

The problem with this idea is that the critical width cannot be effectively determined beforehand. Therefore, in practice one will simply try computing the range with IA, and redo the test with AA if IA was inconclusive.

3.17 Implementation issues

To test the practicality and usefulness of AA, we have implemented the basic operations ($+$, $-$, $\times$, $\div$, $\sqrt{\ }$) in C for the Sun SPARCstation. We describe below some choices that we made in our prototype implementation [**56**], but which are not part of the AA model proper.

3.17.1 Representation of affine forms

In our prototype implementation of AA, we represent an affine form $\hat{x}$ depending on m noise symbols by an array of $2m+2$ consecutive 32-bit words. The first two words contain the central value x_0 and the number m; then come the m terms, each consisting of a partial deviation x_i, and the corresponding *index* i — an integer value that uniquely identifies the noise symbol ε_i. All real quantities are encoded as IEEE 32-bit floating-point numbers.

The noise symbol indices need to be stored because affine forms are quite sparse: although a long-running program may create billions of independent noise symbols, each affine form will typically depend only on a small subset of them. Therefore, it is imperative that we store for each affine form $\hat{x}$ only the terms $x_i\varepsilon_i$ that are not zero.

Thus, in general, each affine form that occurs in a computation will have a different number of terms, with a different set of noise symbol indices. Two affine forms are dependent only when they include terms with the same index.

Algorithms that operate on two or more affine forms, such as the addition and multiplication routines described above, typically need to match corresponding terms from the given operands, while computing the terms of the result. In order to speed up this matching, we make sure that the terms of every affine form are always sorted in increasing order of their noise symbol indices.

3.17.2 Memory and index management

Affine forms are typically stored in a special storage pool `SA`, which is managed like a stack. In general, a routine that performs AA computations should reset the `SA` top-of-stack pointer, right before exiting, to the value it had on entry. This action implicitly discards all affine forms computed during the routine's execution, and recycles their storage. Of course, if the routine is supposed to return any of these affine forms, then it must copy them to the new top-of-stack position, and adjust the pointer accordingly.

As mentioned above, new noise symbols are constantly being created while the program runs. Practically every time we compute a new affine form, we need to introduce a brand new noise symbol, to represent the linearization and roundoff errors committed in that operation. The noise symbols do not consume any storage by themselves, but each requires a distinct index. For this purpose, we use a global counter that keeps track of the highest index in use at any moment.

To avoid running out of indices after 2^{32} AA operations, an "industrial strength" implementation of AA should to manage the noise symbol namespace too as a stack: when exiting from a procedure, one should reset the noise symbol counter to the value it had upon entry. This action implicitly "discards" all the noise symbols created during the procedure, and allows their indices to be "recycled". If the procedure returns an affine form as its result, then any new noise symbols that occur in the latter must be renumbered while the result is copied to its proper location.

3.17.3 Space and time cost

Consider the AA evaluation of an expression (or a sequence of chained expressions) with m operations, where the input values are affine forms that depend on a certain set of n noise symbols $\varepsilon_1, .. \varepsilon_n$. Each operation will contribute one more noise symbol to this set, representing the linearization and roundoff errors of that step. Therefore, each computed value will depend at most on $n + m$ noise symbols. Since the cost of any basic AA operation is proportional to the size of the operands, the whole expression can be evaluated in $O(m(n + m))$ time and space.

3.18 Optimization techniques

High computational cost is the main obstacle to the use of AA in practical applications, especially those with modest precision requirements. Implementors of AA libraries should therefore be sensitive to efficiency issues. We will now describe some helpful optimization techniques for AA programs.

3.18.1 Condensing noise variables

On long computations, with $m \gg n$, most terms in each affine form will be recording errors due to previous operations. In general, it is not worth keeping track of all those errors separately. For the sake of efficiency, it is desirable to insert at selected points extra code to "condense" the affine forms.

The idea is to replace two or more terms $z_i\varepsilon_i$, $z_j\varepsilon_j$, ... by a single term $z_k\varepsilon_k$, where $z_k = |z_i| + |z_j| + \cdots$, and ε_k is a brand-new noise symbol. This operation reduces the size of the affine form $\hat{z}$, possibly at the cost of losing correlation information.

No information will be lost if the noise variables ε_i, ε_j, ... are exclusive to $\hat{z}$, that is, they do not appear in any other affine form that is still alive. (A value is *alive* at some point if it may be used further on.) Also, the loss of information is likely to be minimal if the condensed coefficients $|z_i|$, $|z_j|$, ... are small compared to the other noise coefficients of $\hat{z}$. The condensation might make a difference only if the larger coefficients happened to cancel out later on. Hence, we may condense all terms of $\hat{z}$ form which are smaller than some fraction of $[\hat{z}]$.

For example, suppose the computation is a loop where only two variables $\hat{x}$ and $\hat{y}$ are carried from one iteration to the next. Suppose $\hat{x}$ and $\hat{y}$ begin as simple affine forms, each depending on a single noise variable. At the end of the first iteration, the joint range of those two variables will be a center-symmetric polygon D whose complexity is at worst $2(m+2)$, where m is the number of operations performed in the body of the loop. If we just went on, each iteration would add another $2m$ terms to those forms.

We can solve this problem by condensing all noise variables at the end of each iteration. That is, we replace $\hat{x}$ and $\hat{y}$ by the new forms

$$\begin{aligned}
\hat{x} &= x_0 + x_i\varepsilon_i + x_j\varepsilon_j \\
\hat{y} &= y_0 + y_i\varepsilon_i + y_j\varepsilon_j,
\end{aligned}$$

where ε_i, ε_j are two brand new noise symbols. The joint range of these new forms is a parallelogram P with sides parallel to (x_i, y_i) and (x_j, y_j).

The new coefficients x_i, x_j, y_i, y_j should be chosen so that this parallelogram contains the original domain D, preserving that AA invariant.

The parallelogram of minimum area enclosing the convex polygon D can be computed in $O(m \log m)$ steps, and its area is at most $4/3$ times the area D. Thus, at a modest cost in time and accuracy, we can keep the size of the affine forms bounded by $O(m)$, indefinitely.

Depending on the context, it may be important to know how the final result correlates with the input quantities. In that case, we should condense only "internal" noise variables (i.e., those that were created during the computation itself), but preserve the "external" ones (i.e., those that were present in the input forms).

This technique begs the question, how many terms do we *really* need to keep in the affine forms? Suppose that, at some point of the computation, there are k affine forms that are still "alive" (i.e., that may be used later). In principle, at that point we could replace all those forms by a new set of k affine forms, depending exclusively on k new noise variables, in such a way that the volume of the joint range increases only by a constant factor (that depends on k).

Unfortunately, this result is not very useful, since computing the smallest k-dimensional paralelotope that contains the old joint range is a difficult problem when $k \geq 3$.

3.18.2 Static storage allocation

Another promising optimization is the "compilation" of AA algorithms into an ordinary programming language, like C.

Many applications of AA, such as those described in Chapter 4, can be coded as procedures that take ordinary intervals as parameters, convert them to affine forms, and evaluate a linear (non-looping) chain of expressions on those values. In such cases, the compiler could predict statically the set of noise symbols affecting each computed affine form. The compiler could then allocate the affine forms statically, on the ordinary procedure-call stack, using a separate simple variable for each coefficient. The noise symbol indices would then be superfluous. In this context, the AA arithmetic operations that loop over the terms (such as `AA.affine` and its variants) would be expanded in-line, avoiding the overhead of merging the term lists.

3.18.3 Shared sub-expressions

In actual programs, it is common for the same sub-formula to appear as an operand of two or more operations. With ordinary floating-point, or with standard IA, evaluating such shared sub-expressions more than once is merely a waste of time. With AA, however, multiple evaluations may also make the results less accurate. The reason is that each evaluation of a shared sub-formula represents the linearization errors of the latter by a different set of noise symbols, preventing those errors from canceling out in later steps. Therefore, when coding expressions like $(x^2 - y^3)/(x^2 + y^3)$ for AA evaluation, it is doubly important to identify common sub-expressions like x^2 and y^3, and compute each of them only once. Symbolic manipulations programs can identify common sub-expressions and make coding much easier. For example, in Maple we have:

```
C((x^2-y^3)/(x^2+y^3),optimized);
      t1 = x*x;
      t2 = y*y;
      t3 = t2*y;
      t7 = (t1-t3)/(t1+t3);
```

3.19 Hansen's Generalized Interval Arithmetic

Affine arithmetic can be viewed as a simplification of *generalized interval arithmetic* (GIA), a computation model proposed in 1975 by E. R. Hansen [**20**]. (For a fuller discussion about GIA, including applications, see [**60**].)

In its original formulation, GIA addresses specifically the problem of computing one or more functions from a fixed set of n quantities $x_1, .. x_n$, which are given as ordinary intervals $\bar{x}_1, .. \bar{x}_n$. Every quantity z that is derived from these variables, including the function result, is represented as a list $\tilde{z}$ of $n + 1$ intervals $\bar{\zeta}_0, \bar{\zeta}_1, .. \bar{\zeta}_n$, with the understanding that

$$z \in \bar{\zeta}_0 + \bar{\zeta}_1 x_1 + \bar{\zeta}_2 x_2 + \cdots \bar{\zeta}_n x_n.$$

Note that the x_i in this formula are unknown quantities, so the formula is kept unevaluated—just as affine forms of AA. From this representation we can obtain a range for z, namely

$$\bar{z} = \bar{\zeta}_0 + \bar{\zeta}_1 \bar{x}_1 + \bar{\zeta}_2 \bar{x}_2 + \cdots \bar{\zeta}_n \bar{x}_n,$$

where the formula is evaluated as in standard IA.

This model is obviously similar to AA, not only in its representation, but also on its main virtue—namely, that in principle it can model affine dependencies between quantities, with error that shrinks quadratically with the size of the input intervals.

3.19.1 Conceptual differences

However, GIA and AA are not mathematically equivalent, and neither is a special case of the other. In general, conversion from one representation to the other entails loss of information. The difference is evident when one considers the joint range of two quantities u and v when they are described in either model.

As we have seen, in AA the joint range is always a center-symmetric convex polygon in the u–v plane. In a computation with n input variables and m steps, the joint range may have up to $2(n+m)$ sides. In GIA, the joint range may be a non-convex polygon, whose complexity is proportional to n alone. Consider, for example, the GIA forms

$$\begin{aligned} \tilde{u} &= [0 \mathinner{\ldotp\ldotp} 0] + [1 \mathinner{\ldotp\ldotp} 3]\bar{x} \\ \tilde{v} &= [0 \mathinner{\ldotp\ldotp} 0] + [1 \mathinner{\ldotp\ldotp} 4]\bar{x}, \end{aligned}$$

where x is the only input variable. If x ranges over $[-1 \mathinner{\ldotp\ldotp} +1]$, then the value pairs (u, v) that are allowed by these forms is the bowtie-shaped region shown in Figure 3.10.

Moreover, the fact that the GIA coefficients are intervals, rather than numbers, implies that uncertainty cancellation will not be as complete as it can be in the AA model. For example, if $\tilde{u}$ is the form given above, then the GIA evaluation of $\tilde{u} - \tilde{u}$ will produce

$$[0 \mathinner{\ldotp\ldotp} 0] + [-2 \mathinner{\ldotp\ldotp} +2]x + [0 \mathinner{\ldotp\ldotp} 0]y,$$

whereas the analogous AA computation returns exactly zero. For the same token, we can expect that error explosion will occur in GIA more often than in AA.

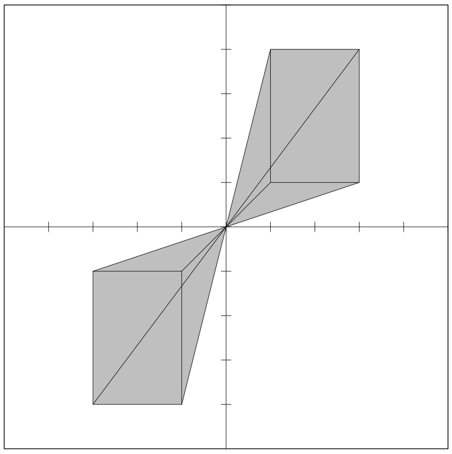

Figure 3.10: Joint range in GIA.

Finally, by lumping all internal errors into the independent interval $\bar{\zeta}_0$ (or, in the coefficients $\bar{\zeta}_1, .. \bar{\zeta}_n$), the GIA model prevents those errors from canceling out.

3.19.2 Practical differences

GIA and AA differ also in a number of practical details that affect their efficiency and flexibility. First, in the GIA representation, each coefficient takes twice as much space as in AA; and the extra space does not seem to translate into increased accuracy.

Another difference is that the number of terms in a GIA form is fixed, and their order is tied to the order of the function arguments. This choice forces one to handle and operate on all n terms, even when the quantity at hand depends on a small subset of the input variables.

Moreover, the GIA representation is implicitly tied to a specific set of variables, which limits the scope of GIA forms to the body of a single function (or to a set of functions with the same arguments); a feature that hinders the composition of complex functions from simpler ones.

For example, suppose that the function $f(x, y)$ is defined in terms of some previously defined function $g(u, v, w)$. The values of u, v, and w computed by f with GIA will be expressed as GIA forms with three coefficients over the variables x and y; whereas the values handled internally by g are four-term GIA forms over u, v, and w. Therefore, the call to g from f will require some non-trivial conversion between the two types of GIA forms, usually with loss of information.

Affine arithmetic is more flexible in this regard. In the implementation originally proposed by Comba and Stolfi **[10]**, and detailed in Section 3.17, the "names" of the noise variables are recorded in the affine form, and their scope is the entire program. As a consequence, affine forms can be passed across function boundaries without conversion or loss of information. An example of application where this issue is quite relevant is the ray-tracing of implicit surfaces, described in Section 4.3.

Chapter 4

Some applications

In this chapter, we describe some applications of interval arithmetic to problems in numerical analysis, computer graphics, geometric modeling, and global optimization. We also show that affine arithmetic can produce better results than IA [**10–13**].

4.1 Zeros of functions

Solving equations is a fundamental problem in mathematics.[1] The simplest kind of equation is $f(x) = 0$, defined by a real function $f \colon \Omega \to \mathbf{R}$ of a single variable x. A *solution* (also called a *root*) is of course any number $\tilde{x} \in \Omega$ such that $f(\tilde{x}) = 0$. We also say that $\tilde{x}$ is a *zero* of f. Sometimes, finding any one zero of f is sufficient. Sometimes, all zeros are required.

There exist explicit formulas for computing all zeros of polynomial functions of degree at most 4. The formula for degree 1 is trivial. The formula for degree 2 is the well known quadratic formula. The formulas for degrees 3 and 4, based on the famous Cardano solution of the cubic, are less well known, and not frequently used, because they may require complex arithmetic, even when all roots are real.

No formulas exist for solving polynomial equations of higher degree or transcendental equations. Thus, in general, we must resort to numerical approximation methods. Moreover, we should keep in mind that

[1] It can be argued that mathematics advances by continuously redefining what "equation" and "solution" mean.

it is very probable that the zeros of f will have *no* exact floating-point representation, even if we know in theory how to compute them exactly. For example, the trivial equation $3x - 1 = 0$ has no exact floating-point solution in base 2 (or 10). So, approximate solutions are all we can obtain with a computer.

What does it mean to solve an equation $f(x) = 0$ approximately? Two natural interpretations are:

1. *Find a number close to a root.* More precisely: given $\delta > 0$, find $\hat{x} \in \Omega$ such that $|\hat{x} - \tilde{x}| < \delta$, for some exact root $\tilde{x}$ of f. Clearly, the whole point is to find such $\hat{x}$ without knowing $\tilde{x}$.

2. *Find a root of an equation that is close to the original equation.* More precisely: given $\epsilon > 0$, find $\hat{x} \in \Omega$ such that $|f(\hat{x})| < \epsilon$. In this case, $\hat{x}$ is a solution of the equation $f(x) - f(\hat{x}) = 0$, which may be thought as a "perturbation" of the original equation, because $f(\hat{x})$ is small.

Both definitions of approximate solution are useful, sometimes even in combination; the adequate definition depends on the application. In any case, both definitions make sense in the floating-point world, provided that the tolerances δ and ϵ are not too small.

There are many classical numerical methods for finding zeros of functions. However, to a certain extent, their success depends on having previously *isolated* a root. Once a root has been isolated, these methods usually converge quickly. Thus, the hard part in solving equations is isolating the roots.

One way to isolate a root is to find a small interval $[a, b] \subseteq \Omega$ such that $f(a)$ and $f(b)$ have different signs. In this case, we say that $[a, b]$ is a *bracketing interval* for f. By the Intermediate Value Theorem, if f is continuous in a bracketing interval $[a, b]$, then f must have at least one zero in $[a, b]$.

4.1.1 Bisection

A simple algorithm that is *guaranteed* to find a root of f in a bracketing interval $[a, b]$ is *bisection.* This algorithm successively divides a bracketing interval at its midpoint into two parts, choosing the half that is still a bracketing interval:

```
bisect(a, b, f_a, f_b : R): R ≡
| Finds a zero of f in a bracketing interval [a, b].
  c ← (a + b)/2
  if (b − a)/2 ≤ δ then return c
  f_c ← f(c)
  if f_a f_c < 0 then
    return bisect(a, c, f_a, f_c)
  else
    return bisect(c, b, f_c, f_b)
```

Starting with a call to `bisect`$(a, b, f(a), f(b))$, this algorithm computes a sequence of nested bracketing intervals, each interval half as wide as its predecessor. Thus, bisection always converges to a root of f. (This observation can be used as the basis of a constructive proof of the Intermediate Value Theorem.)

There are many things that may go wrong when we try to implement the bisection algorithm in floating point. For one thing, if a and b are finite but large, then the formula $(a + b)/2$ may return $+\infty$. Also, if $(b - a)/2$ is not rounded properly (i.e., upwards), then the result may not be within distance δ of a root. We can avoid these problems by using `IA.mid` and `IA.rad` to compute these quantities (see Section 2.5).

Another possible problem is that the product $f_a f_c$ may underflow to zero even when $f_a < 0$ and $f_c > 0$. If this happens, then the procedure will take the wrong branch, and the result may be arbitrarily far from any root. Thus, instead of multiplying the values, we must work with the signs only.

A subtler problem is that `bisect` may go into infinite recursion if c gets rounded to a or b while $(b - a)/2$ is still greater than δ. If `IA.mid` has been properly coded, then this can happen only when a and b are consecutive `Float` values. In that case, the user has chosen too small a tolerance, and there is no way to get any closer to the bracketed root, or to tell which of a and b is closer to it. We should then give up and return either a or b.

Incorporating these changes into the basic algorithm, we get

```
FP.bisect(a,b: Finite; σ: Finite): Finite ≡
| Given a finite non-empty interval [a .. b] and σ ∈ {−1,+1},
| such that σf(a) ≤ 0 and σf(b) ≥ 0,
| returns a value c that is either within distance δ of a root of f,
| or is one of the two representable Floats closest to such root.
  c ← IA.mid([a .. b])
  if c = a or c = b or IA.rad([a .. b]) ≤ δ then return c
  f_c ← σf(c)
  if f_c > 0 then
    return FP.bisect(a,c,σ)
  else if f_c < 0 then
    return FP.bisect(c,b,σ)
  else
    return c
```

The search begins with a call to `FP.bisect(`$a, b, \mathrm{sign}(f(b))$`)`.

Bisection is slow: only one bit of precision is obtained at each iteration. Thus, it needs $\lceil \log_2(|a-b|/\delta) \rceil$ steps to reach tolerance δ.

The well-known iterative method of Newton, if properly used, will have faster convergence (doubling the number of bits at each step). However, Newton's method is not guaranteed to converge in all cases. Even when the mathematical function f satisfies all the conditions for convergence, the rounding errors in its floating-point implementation $\tilde{f}$ may cause Newton's method to loop or diverge. In Section 4.1.3, we will see how to achieve convergence rates similar to Newton's while retaining the robustness of bisection.

4.1.2 Interval bisection

Bisection is one example of a guaranteed method, not a common situation in numerical methods. However, bisection has limitations: it needs to start with a bracketing interval; and it only finds *one* root in the interval, even if there are many roots.

A more serious limitation is that `bisect` does not find a root of the intended function f, but rather a root of its floating-point implementation $\tilde{f}$. Even if the values of f and $\tilde{f}$ are very close, their roots may be arbitrarily different.

As long as we are confined to the realm limited-precision arithmetic, there is no satisfactory solution to this problem. The inevitable roundoff errors can always turn a near-root into a true root, or vice-versa. However, if we have an interval implementation $\bar{f}$ of f, then we can use it to discard parts of the domain $\bar{x}$ that are guaranteed *not* to contain any roots of f.

The idea is to evaluate $\bar{z} \leftarrow \bar{f}(\bar{x}')$ for various sub-intervals $\bar{x}'$ of Ω, starting with Ω itself. If $\bar{z}$ does not contain the value zero, then $\bar{x}'$ cannot contain a root of f, and can be discarded. Otherwise the sub-interval is split into two halves, which are recursively tested. This process continues until the intervals are smaller than a specified tolerance δ, or cannot be subdivided any further.

In the end, we are left with a subset x^* of Ω, consisting of zero or more intervals, that is guaranteed to contain *all* the roots of f in Ω, even if Ω is not bracketing for f. Moreover, the intervals that make up x^* are small or indivisible, and, when tested with $\bar{f}$, their signs turn out ambiguous. We will say that x^* is an *approximate root set* of the original function f.

```
IA.roots(x̄: Interval): stream of Interval ≡
| Given a finite interval x̄, outputs a sequence of sub-intervals
| that are either indivisible or have radius at most δ,
| and constitute an approximate root set of f.
  if 0 ∈ f̄(x̄) then
    c ← IA.mid(x̄)
    if c = x̄.lo or c = x̄.hi or IA.rad(x̄) ≤ δ then
      output x̄
    else
      output IA.roots([x̄.lo __ c])
      output IA.roots([c __ x̄.hi])
```

For many applications, the approximate root set x^* computed by `IA.roots` is an acceptable surrogate for the true set of roots of f. In a sense, this improved bisection algorithm is able to find *all* roots of f inside any given interval, Obviously, the accuracy (and usefulness) of x^* is limited by the accuracy of $\bar{f}$ and by our computation budget.

Since `IA.roots` explores the left half before the right half, it finds all roots *in order*, from left to right, as they occur in $\bar{x}$. If only the smallest root is required, then the method can be modified to stop as soon as a root is found. This variant is useful in applications such as ray tracing, when one is interested in finding the first intersection of a ray with a surface (see Section 4.3), or spectral analysis of matrices, when one is usually interested only in the first few eigenvalues.

Note that `IA.roots` may report two or more intervals for each root (or near-root) of f. The reason is that the range estimator $\bar{f}$ is liable to return an ambiguous sign for intervals that are close to a root but do not straddle it. Therefore, we may want to pipe the output of `IA.roots` through a filter that merges any consecutive abutting intervals. This step does not guarantee that each root of f will be represented by exactly one interval, but it may reduce the caller's overhead.

4.1.3 Using affine arithmetic

If the ranges computed by $\bar{f}$ are much wider than the true range of f in $\bar{x}$, then `IA.roots` may become quite slow, since it will be forced to split the intervals more finely in order to resolve sign ambiguities. It may also output a large number of intervals for each root, and also many intervals that contain no root.

As we observed in Section 2.6, this problem is particularly likely when f is a complicated function with correlated sub-expressions. When this "interval explosion" may be a problem, we may consider adapting `IA.roots` to use affine arithmetic instead of standard IA.

The trivial approach is to use AA only inside the range estimator $\bar{f}$: that is, we compute $\bar{z} \leftarrow \bar{f}(\bar{x})$ by converting $\bar{x}$ to an affine form $\hat{x}$, then evaluating $\hat{z} \leftarrow \hat{f}(\hat{x})$ in the AA model (or the mixed AA/IA model), and returning $\bar{z} = [\hat{z}]$.

However, if we are going to use AA, then we might as well take advantage of the extra information it provides. Specifically, we should replace `IA.roots` by a similar routine `AA.roots` that evaluates $\hat{z} \leftarrow \hat{f}(\hat{x})$ itself, and then uses the information contained in the affine form $\hat{z}$ to choose the split point c.

The routine `AA.roots` begins by converting the interval $\bar{x}$ to the affine form $x_0 + x_1\varepsilon_1$. The result of $\hat{z} \leftarrow \hat{f}(\hat{x})$ will be

$$\hat{z} = z_0 + z_1\varepsilon_1 + z_2\varepsilon_2 + \cdots + z_n\varepsilon_n.$$

We may then conclude that the graph of f in $\bar{x}$ is contained in the parallelogram P with center (x_0, z_0) and horizontal projection $\bar{x}$, whose top and bottom sides have slope z_1/x_1, and whose left and right sides are vertical and have length 2λ, where $\lambda = |z_2| + |z_3| + \cdots + |z_n|$. It follows, then, that any roots of f in $\bar{x}$ must be confined to the sub-interval $\bar{x}'$ where the parallelogram P intersects the x-axis. Thus, we can replace $\bar{x}$ by $\bar{x}'$ before splitting the interval in half. Here is the detailed code:

```
AA.roots(x̄: Interval): stream of Interval ≡
| Given a finite interval x̄, outputs a sequence of sub-intervals
| that are either indivisible or have radius at most δ,
| and constitute an approximate root set of f.
  var k ← newsym()
  var x̂: AA.Form ← IA.mid(x̄) + IA.rad(x̄)ε_k
  var ẑ: AA.Form ← f̂(x̂)
  x̄ ← x̄ ∧ AA.rootAux(x̂, ẑ, k)
  if x̄ ≠ [] then
    c ← IA.mid(x̄)
    if c = x̄.lo or c = x̄.hi or IA.rad(x̄) ≤ δ then
      output x̄
    else
      output AA.roots([x̄.lo __ c])
      output AA.roots([c __ x̄.hi])
```

The routine `AA.rootAux` computes the intersection of P and the x-axis:

```
AA.rootAux(x̂, ẑ: AA.Form; k: Index): Interval ≡
| Given affine forms x̂ = x_0 + x_k ε_k and ẑ,
| returns an interval x̄' containing the intersection
| of the x-axis with the joint range of x̂ and ẑ.
  var λ ← ↑Σ{|z_i| : i ∈ ℰ(ẑ) \ {k}}↑
  var z̄ ← [↓z_0 − λ↓ __ ↑z_0 + λ↑]
  var ū ← IA.div([−x_k __ −x_k], [z_k __ z_k])
  return IA.shift(ū, x_0)
```

For smooth functions, `AA.roots` will exhibit quadratic convergence (doubling the number of bits at each step) until the interval $\bar{x}$ becomes so small that roundoff errors start to dominate. In any case, convergence will be at least linear (one bit gained per iteration), as in `IA.roots`.

4.2 Level sets

Many applications deal with scalar fields $h\colon \Omega \to \mathbf{R}$, where Ω is a subset of $\mathbf{R}^d$. For example, h may be temperature, pressure or height. Of special interest are the subsets of Ω where h is constant, called the *level sets* of h, i.e., the sets $h^{-1}(c) = \{x \in \Omega : h(x) = c\}$, for $c \in \mathbf{R}$. A picture showing how the level sets of h are distributed and how their geometry changes as c varies contains a great deal of qualitative information about the behavior of h (Figure 4.1). Such contour maps are widely used in scientific applications. Familiar examples are temperature charts in weather maps and level curves in altitude maps.

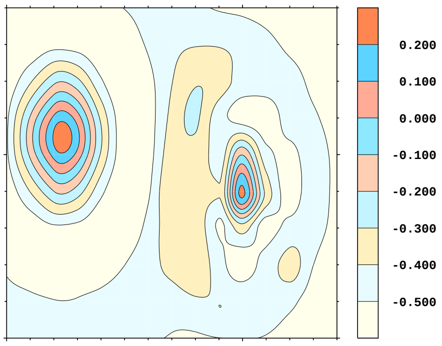

Figure 4.1: Contour map of a scalar field.

4.2.1 Enumeration

Consider the computation of an approximation of a single level set C. Without loss of generality, we may take $C = h^{-1}(0)$, the level set at level zero, because the level set at level c of h is the level set at level zero of $h - c$. For concreteness, we shall consider only the two-dimensional case, i.e., $d = 2$. However, the discussion below can be modified for arbitrary dimension.

The zero set C is the set of solutions of the equation $h(x) = 0$. Since we are now dealing with an equation in several unknowns, the set C does not need to be a single point, or even finite. In general, C is a curve, but it can be almost anything.[2] We also say that C is an *implicit*

[2]Every closed set in $\mathbf{R}^d$ is the zero set of a C^∞ function.

curve, defined by the equation $h(x) = 0$. Implicit curves and surfaces are important in geometric modeling.

A simple and general technique for computing an approximation of a level curve C in Ω is *enumeration*:

1. decompose Ω into small cells;
2. identify which cells intersect C;
3. approximate C within each intersecting cell.

The simplest cellular decompositions used in step 1 are regular grids of squares or triangles (Figure 4.2). Such decompositions are often used in practice because their topology and geometry are well understood. If the cells are sufficiently small, then C can be approximated by linear segments in step 3.

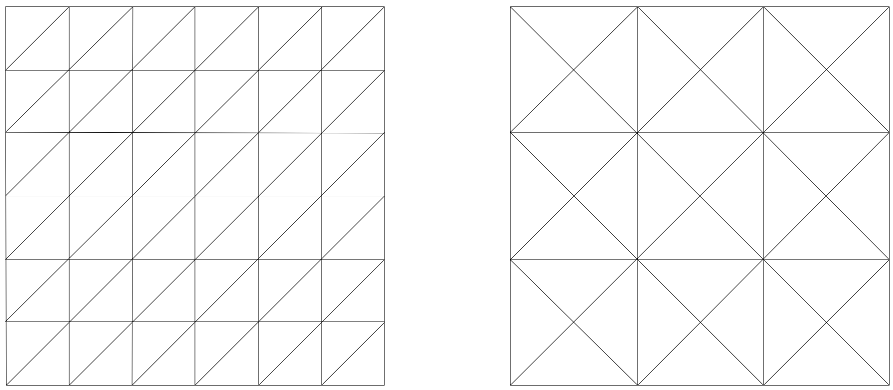

Figure 4.2: Simple cellular decompositions.

Step 2, the enumeration of the intersecting cells, is usually the most expensive step in this method. In the simplest schema, the cells that intersect C are identified by sampling. The function h is evaluated at the vertices of each cell; if the signs of those values are not identical, then the cell necessarily intersects C (again, this follows from the Intermediate Value Theorem, if h is continuous). The points where C intersects the boundary of the cell can be found by bisection, or by simple linear interpolation, if the cell is small.

Obviously, the converse does not hold: when all the values of h at the vertices have the same sign, we cannot conclude that the cell does not intersect C. This is a form of aliasing in the sampling related to the size of the cell. Therefore, cell sizes must be carefully chosen to avoid missing features due to undersampling (Figure 4.3). On the other hand, choosing very small cells can be expensive: a uniform cellular decomposition of Ω having n cubes along each main direction has n^d cubes, but only $O(n^{d-1})$ cubes will intersect C. Thus, choosing a smaller cell size to avoid aliasing in the sampling will greatly increase the number of cells to be scanned, and also increase the fraction of "useless" tests. Therefore, the approximation of level curves with uniform enumeration is simple but not efficient.

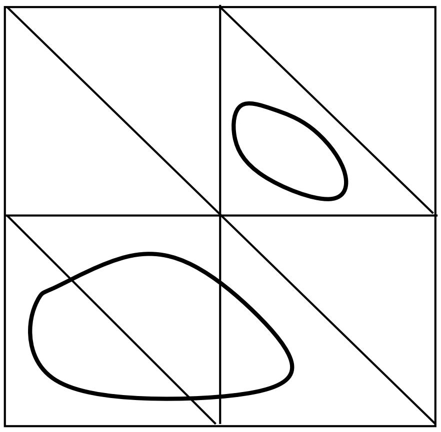

Figure 4.3: Missing features due to undersampling.

4.2.2 Adaptive enumeration

To find the cells intersecting a level curve C without visiting all cells in a cellular decomposition of Ω, we need a way to discard, quickly and reliably, large portions of Ω that cannot contain pieces of C. Point sampling can only prove the *presence* of C in some region of Ω; to reduce the number of cells scanned, we need a test procedure that can also prove the *absence* of C in a region.

Range analysis can provide such a test. If we have an interval version $\bar{h}$ of h, then we can enumerate the cells intersecting C by an adaptive procedure, which is essentially a two-dimensional version of `IA.roots` (see Section 4.1.2). We explore Ω recursively, starting with Ω itself as the initial cell. If a cell is proved to be empty, then it is ignored; otherwise, it is subdivided into smaller cells, which are then explored recursively, until the cells are small enough to approximate C [**14, 54, 57–59**]

For rectangular decompositions, a simple way to divide a cell into subcells is to bisect it orthogonally to its widest direction, or cyclically bisect along one of the coordinate directions at each step, resulting in a 2-d tree decomposition of Ω. Another popular variant divides the cells into four equal parts, resulting in a quadtree decomposition of Ω.

The meaning of "small enough" depends on the application. For rendering, it might mean "smaller than a pixel". For other applications, such as modeling, it may depend on some other numerical criterion. For instance, testing how closely h can be approximated by a linear function inside the cell allows polygonal approximations to adapt to the curvature of C.

Note that the test procedure is not required to be complete, in the sense that it may fail to prove either the presence or the absence of C in a given cell. In particular, a cell that is declared "small enough" may still have unknown status. Each application must decide what to do with those "indeterminate" cells: discard them, treat them just like the cells that do intersect C, or handle them in some special way. Point sampling may be useful at this stage to help identify some intersecting cells.

To test whether a cell $K \subseteq \Omega$ intersects C, we evaluate $\bar{h}(\bar{x}, \bar{y})$, where $\bar{x}$ and $\bar{y}$ are the projections of K onto the coordinate axes. The interval thus computed is guaranteed to contain all values of h for points inside K. If this interval does not contain zero, then K cannot contain zeros of h. Of course, the converse does not hold—if the interval contains zero, we cannot conclude that h vanishes somewhere in K.

An adaptive enumeration method based on IA is:

```
IA.roots_2(K: Cell): stream of Cell ≡
| Enumerates a set of sub-cells of cell K
| that are either small or indivisible, and
| which constitute an approximate solution of h(x, y) = 0 in K.
  if 0 ∈ h̄(K) then
    if is_small(K) then
      output K
    else
      (K_1, .. K_n) ← IA.divide_2(K)
      if n = 1 then
        output K
      else
        for i in {1..n} do
          output IA.roots_2(K_i)
```

4.2.3 Examples

Figure 4.4a shows a full enumeration of the cubic curve defined implicitly by $y^2 - x^3 + x = 0$, in the square $\Omega = [-2 \mathbin{\_\_} 2] \times [-2 \mathbin{\_\_} 2]$, using a 16×16 grid. Intersecting cells were identified by sampling and appear in grey (note how few they are: only 44 out of 256). The points where the curve crosses cell edges (marked with white dots) were computed by simple linear interpolation, and have been joined into a polygonal approximation for the curve. Note that a level curve can have several connected components. Figure 4.4b shows an adaptive enumeration based on IA of the same curve, but now using a 32×32 grid. Note that large portions of Ω were discarded at early stages.

Figure 4.5 shows adaptive enumerations, using 2-d trees, of the quartic curve defined by $h(x, y) = x^2 + y^2 + xy - (xy)^2/2 - 1/4$ in the square $\Omega = [-2 \mathbin{\_\_} 2] \times [-2 \mathbin{\_\_} 2]$. Note how AA was able to be compute a much better approximation than IA, because of the many correlations in h.

4.3 Ray tracing

Interval analysis has also been used for reliable ray-tracing of surfaces [**7, 40**]; specifically, to determine all intersections between an implicit surface $h(x, y, z) = 0$ and a line segment pq (the "ray").

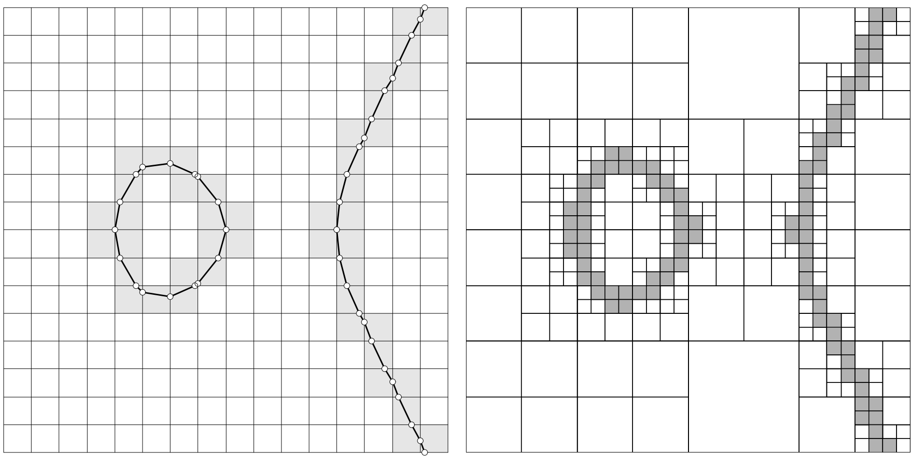

Figure 4.4: Full enumeration (left) and hierarchical enumeration using IA (right).

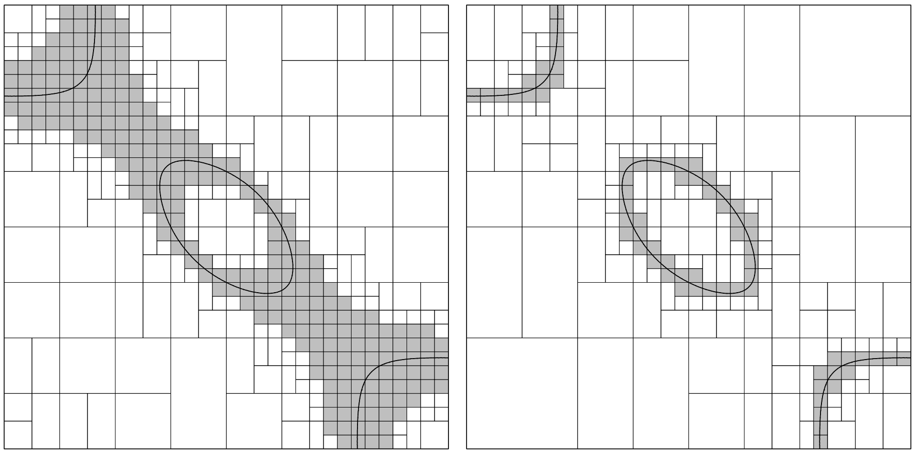

Figure 4.5: Adaptive enumeration of quartic with IA (left) and AA (right).

This problem is equivalent to that of finding the roots of the univariate function

$$f(t) = h((1-t)x_p + tx_q,\ (1-t)y_p + ty_q,\ (1-t)z_p + tz_q)$$

in the interval $[0 \,{\scriptstyle --}\, 1]$. A robust and reasonably efficient algorithm for the latter combines interval analysis with Newton's root-finding method. We evaluate $\bar{u} = \bar{f}(\bar{t})$ using IA, for the whole interval $\bar{t} = [0 \,{\scriptstyle --}\, 1]$. If the resulting interval $\bar{u}$ is strictly positive or strictly negative, then we know that the ray pq does not intersect the surface. Otherwise, we evaluate the derivative $\bar{v} = \overline{f'}(\bar{t})$, in that interval, also using IA. From the intervals $\bar{u}$ and $\bar{v}$, we can compute a sub-interval $\bar{t}^*$ of $\bar{t}$ that must contain all the roots of f in $\bar{t}$. If $\bar{t}^*$ is less than half as wide as $\bar{t}$, then we repeat the search in $\bar{t}^*$, recursively. Otherwise, we split $\bar{t}^*$ in two equal parts, and repeat the search recursively in each half. The recursion stops when the interval $\bar{t}$ is small enough for the application.

The order of convergence of this algorithm is somewhere between linear and quadratic, depending on the accuracy of the computed intervals $\bar{u}$ and $\bar{v}$. However, evaluating $\bar{f}(\bar{t})$ in the IA model is equivalent to evaluating $\bar{h}(\bar{x}, \bar{y}, \bar{z})$ on the intervals $\bar{x} = [x_p \,{\scriptstyle --}\, x_q]$, $\bar{y} = [y_p \,{\scriptstyle --}\, y_q]$, $\bar{z} = [z_p \,{\scriptstyle --}\, z_q]$ — that is, evaluating h on the axis-aligned bounding box of the segment pq, instead of only along the segment itself. Once again, the problem arises because the IA routines have no way of knowing that the arguments x, y, and z of $h(x, y, z)$ are highly correlated.

Obviously, the bounding box of the segment pq may intersect the surface even when the segment itself does not. Even assuming that $\bar{h}(\bar{x}, \bar{y}, \bar{z})$ will be computed accurately (which, as we saw, is unlikely to happen with standard IA), this fact alone will surely lead to slow convergence, and to many evaluations of $\bar{f}$ on ray segments that eventually turn out not to contain any roots.

Replacing standard IA by AA will generally improve the performance of this algorithm. Even without any algebraic manipulation, AA will automatically notice that the affine forms $\hat{x}$, $\hat{y}$, and $\hat{z}$ are strongly correlated, and will use this fact to produce tighter bounds for $f(t)$.

Moreover, as the interval $[\hat{t}]$ decreases, the deviation of the computed affine form $\hat{u}$ should be increasingly dominated by the single error term $u_j\varepsilon_j$ whose noise symbol ε_j is that of the input interval $\hat{t}$. (Recall that if f is moderately well behaved, then the other partial deviations of $\hat{u}$

should decrease quadratically with the size of $[\hat{t}]$.) But in that case the coefficient u_j is a good estimate of the derivative of f in the interval, and we can use it to guess the position of the root for the next iteration. In other words, AA allows us to carry out Newton's root-finding algorithm without explicitly computing the derivative of f.

4.4 Global optimization

Another important and difficult problem is *global optimization*, that is, the computation of the global maximum or minimum of a function over its domain. There are two main variants for this problem: *unconstrained optimization*, which is over the entire domain, and *constrained optimization*, which is only in a subregion of the domain, usually defined implicitly by non-linear equations and inequations.

We shall consider the *box-constrained global minimization problem*: given a *d-dimensional box* $\Omega \subseteq \mathbf{R}^d$ (that is, the Cartesian product of d real intervals), and a continuous *objective function* $f\colon \Omega \to \mathbf{R}$, find its *global minimum* $f^* = \min\{\, f(x) : x \in \Omega \,\}$, and the set of all *global minimizers* $\Omega^*(f) = \{\, x^* \in \Omega : f(x^*) = f^* \,\}$. Actually, we shall consider the approximate numerical version of this problem: instead of finding all minimizers $\Omega^*(f)$, we seek only to identify some subset $\widehat{\Omega}$ of Ω that is guaranteed to contain Ω^*. The goal then becomes to make the measure of $\widehat{\Omega}$ as small as possible, for a given computation budget.

There are many methods for findind local minima, but it would seem that finding a global minimum with a computer is a hopeless task. Indeed, this is probably correct, if we are restricted to computing the values of f at a finite set of sample points in Ω, because f may oscillate arbitrarily between these sample points. Nevertheless, combining general branch-and-bound techniques with range analysis can provide robust algorithms for global optimization, because range estimates can be used to discard large subregions of Ω that cannot contain a global minimum.

4.4.1 Branch-and-bound methods for global optimization

Branch-and-bound is a general numerical technique for solving global minimization problems. A branch-and-bound algorithm generally alternates between two main steps: *branching*, which is a recursive subdivision of the domain Ω; and *bounding*, which is the computation of lower and upper bounds for the global minimum of f in a subregion of Ω. By keeping track of the current best upper bound for the global minimum of f, one can discard subregions that cannot contain a global minimizer, i.e., subregions where the lower bound for f is greater than the current upper bound for the global minimum f^*. Subregions that cannot be discarded in this way are kept in a list $\mathcal{L}$ to be further processed. Thus, at any time, the set $\widehat{\Omega} = \cup\mathcal{L}$ is a valid solution to the global minimization problem, such as defined above. The algorithm stops when the current solution $\widehat{\Omega}$ is adequate for the application (based on the sizes of the boxes in $\mathcal{L}$, on the estimated range for f^*, or on some other criterion).

This algorithm converges provided that: the function f is continuous; the branching step is such that the width of the widest box in $\mathcal{L}$ tends to zero; and the range estimates for $f(\Delta)$ shrink to a single value as the diameter of Δ goes to zero.

The basic branch-and-bound algorithm, outlined above, admits endless variations, depending on how the branching and bounding steps are implemented **[26, 29, 30, 51, 52]**. The simplest branching method is to bisect the current box orthogonally to its widest direction **[41]**. Alternatively, one can cyclically bisect along one of the coordinate directions at each step **[43]**. (This similar to enumeration techniques for level sets.)

The correctness of general branch-and-bound methods requires range estimates that are guaranteed to contain the values of f in a subregion Δ of Ω. On the other hand, the efficiency of such methods depends on the quality of those estimates. One usually trades quality for speed when computing estimates; however, tight estimates, even if more expensive to compute, sometimes provide overall faster algorithms.

4.4.2 Example

Consider the Goldstein-Price function:

$$\begin{aligned} f(x,y) &= [1 + (x+y+1)^2(19 - 14x + 3x^2 - 14y - 6xy + 3y^2)] \cdot \\ &\quad [30 + (2x-3y)^2(18 - 32x + 12x^2 + 48y - 36xy + 27y^2)]. \end{aligned}$$

The global minimum of f in the box $\Omega = [-2 \mathrel{..} +2] \times [-2 \mathrel{..} +2]$ is $f^* = 3 = f(0, -1)$.

This is an easy function for local optimization, but which is very difficult for a branch-and-bound IA algorithm. The dependency problem generates a large number of boxes and causes it to partition the region much more finely than is required for AA, as shown in Figure 4.6. In this figure, boxes shown in white have been eliminated, and boxes shown in grey remain at termination, and are thus guaranteed to contain all global minimizer for f in Ω. Intuitively, a "good" algorithm should generate a picture with few, large white boxes, and few, small grey boxes. This is interpreted as its ability to both quickly discard large subregions of Ω and locate all global minimizers very precisely.

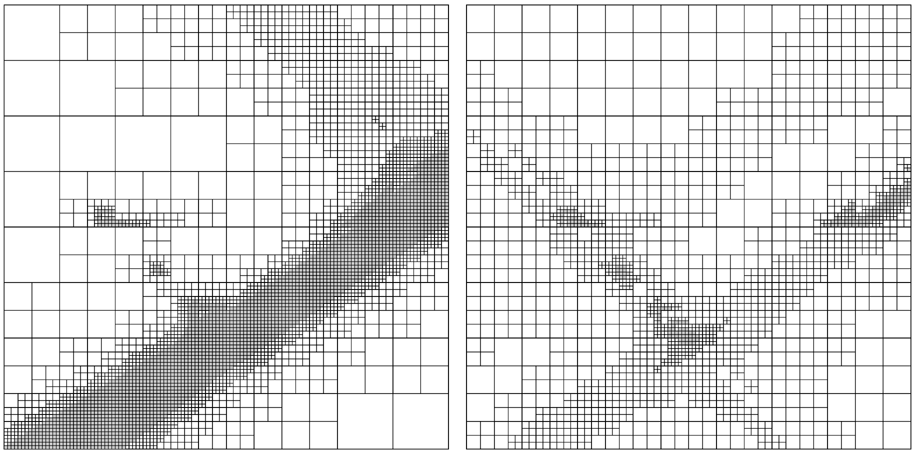

Figure 4.6: Domain decompositions for minimizing the Goldstein-Price function with IA (left) and AA (right).

4.5 Surface intersection

Parametric surfaces are the most popular primitives used in computer aided geometric design (CAGD). They are easy to approximate and render, and there is a huge literature on special classes of surfaces suitable for shape design, such as Bézier and splines surfaces, for which special algorithms exist [**5**]. However, using parametric surfaces for modeling solids in CSG systems requires efficient and robust methods for computing surface intersection, mainly for trimming surfaces into patches that can be sewn together to bound complex shapes.

4.5.1 Methods for computing surface intersections

Several methods have been proposed for solving the important problem of computing the intersection of two parametric surfaces. These methods can be classified into two major classes: *continuation methods* and *decomposition methods*.

Continuation methods, also called *marching methods*, use a local approach to the surface intersection problem. Starting from a point known to be on both surfaces, these methods build an approximation for the intersection curve by marching along the curve, successively computing a new point based on the previous point (or points) [**6**]. Continuation methods must use numerical approximations not only for marching along the curve, but also for finding starting points. Since the intersection curve may have several connected components, a starting point is needed on each component. Moreover, care must be taken for handling closed components correctly. In some applications, such as trimming, intersection curves computed with continuation methods must be mapped back to the parameter domains to define trimming curves. This may be a difficult inverse problem.

Decomposition methods, on the other hand, use a more global approach to the problem. A simple decomposition method is to build polygonal approximations for both surfaces and then intersect the corresponding polyhedral surfaces. Although it is easy to build polygonal approximations for parametric surfaces, such approximations need to be very fine to provide a good approximation for the intersection. A naive polygonal approximation is obtained by simply subdividing the parameter domain uniformly into many small rectangles. However, intersecting such fine polygonal approximation is itself a difficult task. Even if we do not care about geometric degeneracies [**25, 55**], this is a high complexity task: If there are n rectangles along each main direction in parameter space, then there are n^2 faces in each polyhedron. A naive algorithm that computes the intersection of the two polyhedra by testing every possible pair of faces has to consider n^4 cases, most of which do not contribute to the intersection. This algorithm is not practical because it is very expensive to refine an approximation.

Since decomposition methods work directly on parameter domains, no inverse problem needs to be solved to find trimming curves. On the other hand, decomposition methods compute trimming curves in a piecewise, unstructured way; the pieces must be somehow glued together into complete curves.

Adaptive decomposition methods avoid the cost of uniform decompositions by subdividing the domain until the surface is approximately planar. In that way, the associated polygonal approximation is adapted to the local curvature of the surface, being finer in regions of high curvature and coarser in regions of low curvature, where the surface is almost flat. Such methods are generally restricted to specific types of surfaces, whose nature can be exploited to derive efficient tests for local flatness [**16**].

4.5.2 A decomposition method based on interval analysis

The decomposition method proposed by Gleicher and Kass [**18**] takes a global approach for subdividing the domains, using range analysis. Given a rectangle in each domain, use IA to compute an estimate for the range of values taken by the corresponding parametric function on each rectangle. This estimate is a bounding box for a surface patch, i.e., a rectangular box in 3d space, aligned with the coordinate axes, and guaranteed to contain the piece of the surface corresponding to the given rectangle in parameter space. If two bounding boxes do not intersect, then the corresponding surfaces patches cannot intersect. If the bounding boxes do intersect, then the surfaces patches *may* intersect. In this case, the corresponding rectangles in the domains are subdivided into four equal pieces, and the process is repeated until either the surfaces patches are proved disjoint or a user defined tolerance is reached; the patches are then assumed to intersect. In this way, a quadtree decomposition is built for each domain.

For efficiency, Gleicher and Kass keep track of all pairs of patches that might intersect: each leaf node in one quadtree contains a list of leaf nodes in the other quadtree that it overlaps. This list is refined and distributed to its children when a node is subdivided.

4.5.3 Examples

We show two examples of how the Gleicher-Kass algorithm for surface intersection can be improved by using AA instead of IA, specially for surfaces that are common in CAGD.

Lofted parabolas

Consider a cubic patch obtained by lofting a parabola to another parabola. More precisely, take three points a_0, a_1, a_2 in $\mathbf{R}^3$, and consider the quadratic Bézier curve defined by these points:

$$\alpha(u) = a_0(1-u)^2 + 2a_1 u(1-u) + a_2 u^2, \qquad u \in [0,1].$$

Take three other points b_0, b_1, b_2 in $\mathbf{R}^3$, and the Bézier parabola defined by them:

$$\beta(u) = b_0(1-u)^2 + 2b_1 u(1-u) + b_2 u^2, \qquad u \in [0,1].$$

Now, sweep α to β linearly to obtain a surface:

$$f(u,v) = (1-v)\alpha(u) + v\beta(u), \qquad u,v \in [0,1].$$

Lofting is a common operation in CAGD.

Because the parametrization f contains several occurrences of u and $1-u$, and of v and $1-v$, the terms are strongly correlated, and we expect AA to provide tighter bounds for f than IA. This expectation is met: Figure 4.8 shows the domain decompositions built with IA and AA for computing the intersection of the two lofted parabolas shown in Figure 4.7, using six levels of recursive subdivision. Note how AA exploits correlations to give much tighter approximations for the intersection, quickly discarding large parts of both domains.

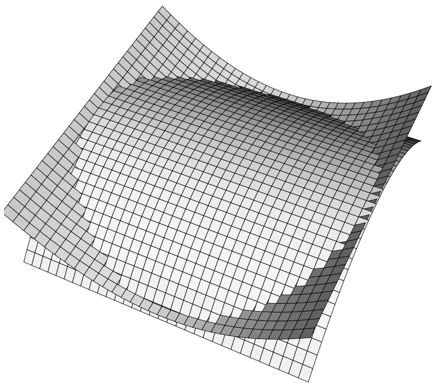

Figure 4.7: Two intersecting lofted parabolas.

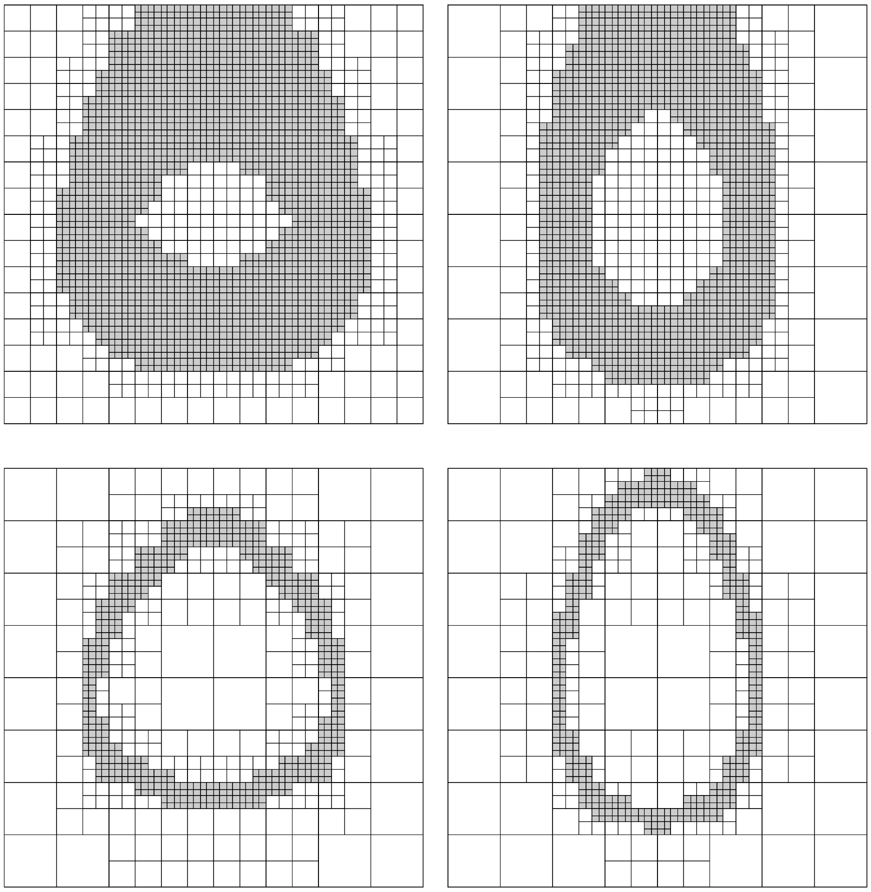

Figure 4.8: Domain decompositions computed with IA (top) and AA (bottom) for intersecting the two skew parabolic cylinders shown in Figure 4.7.

Bicubic patches

Consider now bicubic patches, the most common surface patches in CAGD. A bicubic patch is a tensor product Bézier surface, defined by a mesh of sixteen control points $a_{ij} \in \mathbf{R}^3$ $(i, j = 0..3)$:

$$f(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} B_i^3(u) B_j^3(v),$$

where $u, v \in [0,1]$ and B_i^n is the i-th Bernstein polynomial of degree n:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

(Lofted parabolas are also tensor product Bézier surfaces.)

Figure 4.10 shows the domain decompositions built with IA and AA for computing the intersection of the two bicubic patches shown in Figure 4.9. Because tensor product parametrizations contain many occurrences of strongly correlated terms, we expect AA to provide tighter bounds than IA. Again, this expectation is met. An extra subdivision step with AA is sufficient to show that the intersection curve is not a loop (Figure 4.11). Figure 4.12 shows the trimming curves corresponding to the intersection curve.

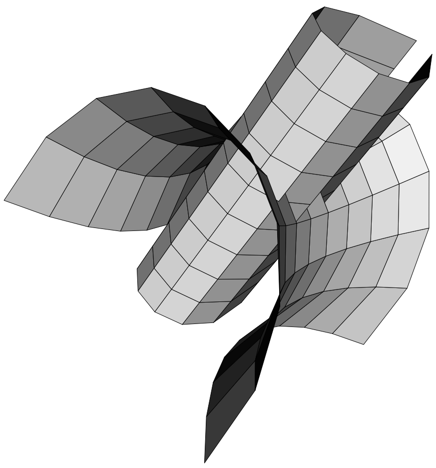

Figure 4.9: Two intersecting bicubic patches.

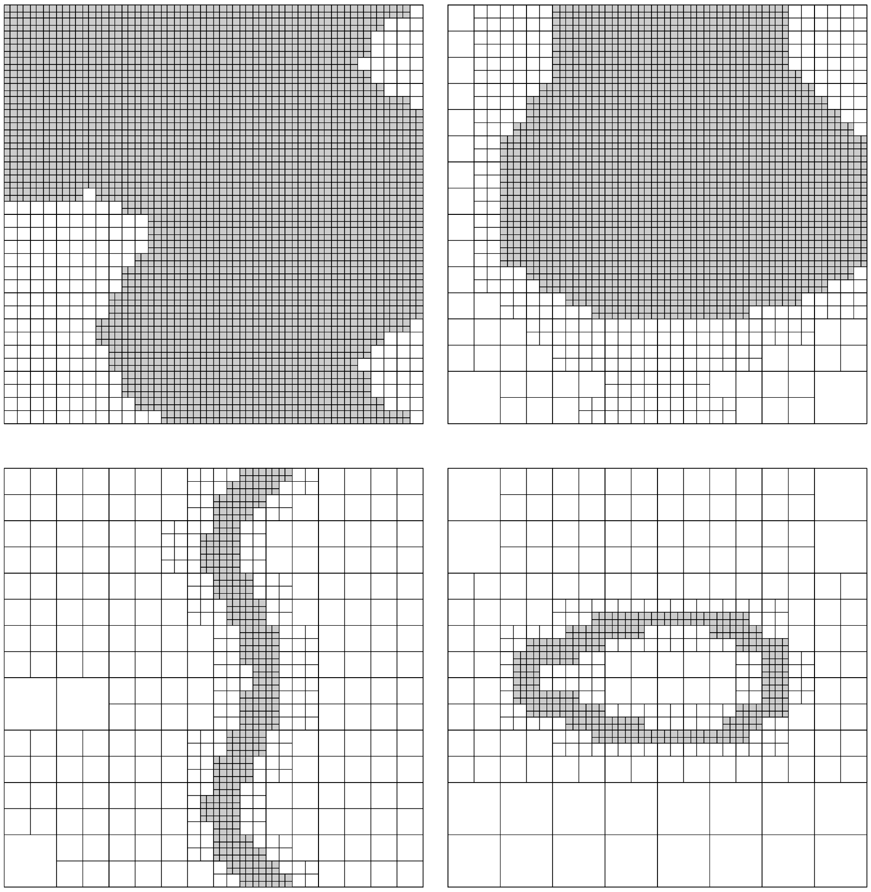

Figure 4.10: Domain decompositions computed with IA (top) and AA (bottom) for intersecting the two bicubic patches shown in Figure 4.9.

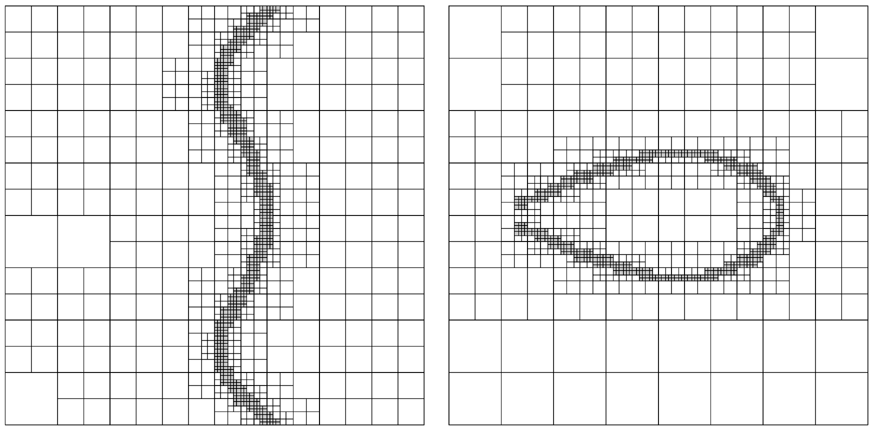

Figure 4.11: Extra subdivision step with AA shows that intersection curve is not a loop.

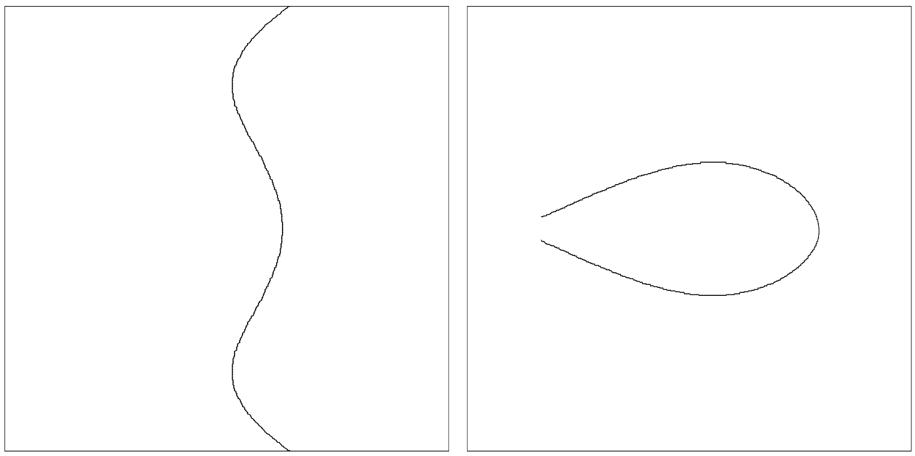

Figure 4.12: Trimming curves for the intersecting bicubic patches.

Bibliography

[1] M. Abramowitz and I. A. Stegun, editors. *Handbook of mathematical functions with formulas, graphs, and mathematical tables.* Dover Publications Inc., New York, 1992. Reprint of the 1972 edition.

[2] G. Alefeld, A. Frommer, and B. Lang (eds.). *Scientific Computing and Validated Numerics*, volume 90 of *Mathematical Research.* Akademie-Verlag, 1996. Proceedings of the International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN-95), Wuppertal, Germany.

[3] ANSI/IEEE, ANSI/IEEE Std 754-1985, IEEE, New York. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[4] R. Baker Kearfott. *Rigorous Global Search: Continuous Problems.* Kluwer, October 1996.

[5] R. E. Barnhill. Surfaces in computer-aided geometric design: A survey with new results. *Computer Aided Geometric Design*, 2(1–3):1–17, 1985.

[6] R. E. Barnhill, G. Farin, M. Jordan, and B. R. Piper. Surface/surface intersection. *Computer Aided Geometric Design*, 4(1-2):3–16, 1987.

[7] W. Barth, R. Lieger, and M. Schindler. Ray tracing general parametric surfaces using interval arithmetic. *The Visual Computer*, 10(7):363–371, 1994.

[8] F. L. Chernousko and A. I. Ovseevich. Method of ellipsoids: Guaranteed estimation in dynamical systems un der uncertainties and

control. In *Abstracts of the International Conference on Interval and Compute r-Algebraic Methods in Science and Engineering (INTERVAL/94)*, page 66, St. Petersburg, Russia, April 1994.

[9] D. M. Cláudio and S. M. Rump. Inclusion methods for real and complex functions in one variable. *Revista de Informática Teórica e Aplicada*, 2(1):125–136, January 1995.

[10] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *Proceedings of VI SIBGRAPI (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 9–18, 1993. Available at `http://dcc.unicamp.br/~stolfi/`.

[11] L. H. de Figueiredo. Surface intersection using affine arithmetic. In *Proceedings of Graphics Interface '96*, pages 168–175, May 1996. Available at `ftp://csg.uwaterloo.ca/pub/lhf/gi96.ps.gz`.

[12] L. H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296, 1996.

[13] L. H. de Figueiredo R. Van Iwaarden and J. Stolfi. Fast interval branch-and-bound methods for unconstrained global optimization with affine arithmetic. March 1997. Submitted to *SIAM Journal of Optimization*. Available at `ftp://ftp.icad.puc-rio.br/pub/lhf/doc/go.ps.gz`.

[14] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):131–138, July 1992.

[15] R. Farouki and V. Rajan. Algorithms for polynomials in Bernstein form. *Computer Aided Geometric Design*, 5(1):1–26, 1988.

[16] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1986.

[17] Geomview. Software written at the Geometry Center, University of Minnesota. Available at `http://www.geom.umn.edu/software/`.

[18] M. Gleicher and M. Kass. An interval refinement technique for surface intersection. In *Proceedings of Graphics Interface '92*, pages 242–249, May 1992.

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

[20] E. Hansen. A generalized interval arithmetic. In K. Nickel, editor, *Interval Mathematics*, number 29 in Lecture Notes in Computer Science, pages 7–18. Springer Verlag, 1975.

[21] E. Hansen. Global optimization using interval analysis — the one-dimensional case. *Journal of Optimization Theory and Applications*, 29(3):331–344, 1979.

[22] E. Hansen. Global optimization using interval analysis — the multi-dimensional case. *Numerische Mathematik*, 34(3):247–270, 1980.

[23] E. Hansen. *Global Optimization using Interval Analysis.* Number 165 in Monographs and Textbooks in Pure and Applied Mathematics. M. Dekker, New York, 1988.

[24] J. Hass, M. Hutchings, and R. Schlafly. The double bubble conjecture. *Electronic Research Announcements of the American Mathematical Society*, 1(3):98–102 (electronic), 1995.

[25] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann, 1989.

[26] R. Horst and H. Tuy. *Global Optimization.* Springer-Verlag, Berlin, 1990.

[27] K. Ichida and Y. Fujii. An interval arithmetic method for global optimization. *Computing*, 23:85–97, 1979.

[28] Interval computations. `http://cs.utep.edu/interval-comp/main.html`.

[29] R. B. Kearfott. Interval Newton/generalized bisection when there are singularities near roots. *Annals of Operations Research*, 25:181–196, 1990.

[30] R. B. Kearfott. An interval branch and bound algorithm for bound constrained optimization problems. *Journal of Global Optimization*, 2:259–280, 1992.

[31] R. B. Kearfott. Algorithm 763: INTERVAL_ARITHMETIC — a Fortran 90 module for an interval data type. *ACM Transactions on Mathematical Software*, 22(4):385–392, 1996.

[32] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[33] O. Knüppel. BIAS — basic interval arithmetic subroutines. Technical Report 93.3, Department of Computer Science III, Technical University of Hamburg-Harburg, July 1993. Avaliable at `http://www.ti3.tu-harburg.de/`.

[34] O. Knüppel. PROFIL — programmer's runtime optimized fast interval library. Technical Report 93.4, Department of Computer Science III, Technical University of Hamburg-Harburg, July 1993. Avaliable at `http://www.ti3.tu-harburg.de/`.

[35] O. Knüppel and T. Simenec. PROFIL/BIAS extensions. Technical Report 93.5, Department of Computer Science III, Technical University of Hamburg-Harburg, November 1993. Avaliable at `http://www.ti3.tu-harburg.de/`.

[36] D. E. Knuth. Ancient Babylonian algorithms. *Communications of the ACM*, 15(7):671–677, 1972. Errata, ibid. 19(2):108, 1976.

[37] U. Kulisch and H. J. Stetter, editors. *Scientific computation with automatic result verification*, volume 6 of *Computing Supplementum*. Springer-Verlag, Vienna, 1988. Papers from the conference held in Karlsruhe, September 30–October 2, 1987.

[38] A. B. Kurzhanski. Ellipsoidal calculus for uncertain dynamics. In *Abstracts of the International Conference on Interval and Computer-Algebraic Methods in Science and Engineering (INTERVAL/94)*, page 162, St. Petersburg, Russia, April 1994.

[39] D. Michelucci and J-M. Moreau. Lazy arithmetic. Technical report, Ecole des Mines de St-Etienne, 1995. Available at `ftp://ftp.emse.fr/pub/papers/LAZY/lazy.ps.gz`.

[40] D. P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface '90*, pages 68–74, May 1990.

[41] R. Moore, E. Hansen, and A. Leclerc. Rigorous methods for global optimization. In C. A. Floudas and P. M. Pardalos, editors, *Recent Advances in Global Optimization*, pages 321–342. Princeton University Press, Princeton, NJ, 1992.

[42] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

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

[44] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics & Applications*, 4(2):7–17, 1984.

[45] O. Neugebauer. *A History of Ancient Mathematical Astronomy*. Springer-Verlag, New York, 1975. Studies in the History of Mathematics and Physical Sciences, No. 1.

[46] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, New York, 1990.

[47] C. E. Pearson, editor. *Handbook of Applied Mathematics: Selected Results and Methdos*. Van Nostrand Reinhold, second edition, 1983.

[48] A. Preusser. xfarbe: Visualization of 2D-arrays. Available at `http://www.fhi-berlin.mpg.de/grz/pub/xfarbe_doc.html`.

[49] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions*. Ellis Horwood Ltd., 1984.

[50] H. Ratschek and J. Rokne. *New Computer Methods for Global Optimization*. Ellis Horwood Ltd., 1988.

[51] H. Ratschek and R.L. Voller. What can interval analysis do for global optimization? *Journal of Global Optimization*, 1(2):111–130, 1991.

[52] D. Ratz. Box-splitting strategies for the interval Gauss-Seidel step in a global optimization method. *Computing*, 53:1–16, 1994.

[53] S. M. Rump. Algorithms for verified inclusions: theory and practice. In R. E. Moore, editor, *Reliability in computing: The role of interval methods in scientific computing*, volume 19 of *Perspectives in Computing*, pages 109–126. Academic Press Inc., Boston, MA, 1988.

[54] J. M. Snyder. Interval analysis for computer graphics. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):121–130, July 1992.

[55] A. J. Stewart. Local robustness and its applications to polyhedral intersection. *International Journal on Computational Geometry and Applications*, 4(1):87–118, 1994.

[56] J. Stolfi. `libaa`: An affine arithmetic library in C, 1993. Avaliable at `http://www.dcc.unicamp.br/~stolfi/`.

[57] K. G. Suffern. Quadtree algorithms for contouring functions of two variables. *The Computer Journal*, 33:402–407, 1990.

[58] K. G. Suffern and E. D. Fackerell. Interval methods in computer graphics. *Computers & Graphics*, 15:331–340, 1991.

[59] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer Graphics and Applications*, 14:14–23, 1994.

[60] J. A. Tupper. Graphing equations with generalized interval arithmetic. Master's thesis, Graduate Department of Computer Science, University of Toronto, 1996. Available ay `http://www.dgp.utoronto.ca/people/mooncake/msc.html`.

[61] R. van Iwaarden. *An Improved Unconstrained Global Optimization Algorithm.* PhD thesis, Department of Mathematics, University of Colorado at Denver, 1996. Available at `http://www.cs.hope.edu/~rvaniwaa/phd.ps.gz`.

[62] J. H. Wilkinson. *Rounding errors in algebraic processes.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1963.