Affine Arithmetic
(draft)

Marcus Vinicius Alvim Andrade[1]
João Luiz Dihl Comba[2]
Jorge Stolfi[1]

[1]Computer Science Department (DCC–IMECC)
Universidade Estadual de Campinas
Caixa Postal 6065 – 13081 Campinas, SP, Brasil
`stolfi,marcus@dcc.unicamp.br`

[2]Computer Graphics Laboratory (LCG-COPPE)
Universidade Federal do Rio de Janeiro
Caixa Postal 68511 - Rio de Janeiro, RJ, Brasil
`comba@lcg.ufrj.br`

1 Introduction

Interval arithmetic (IA), also known as *interval analysis*, is a technique for numerical computation where each quantity is represented by an interval of floating-point numbers. 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 [3, 4].

Since its introduction in the 60's by R. E. Moore, IA became widely 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.

The main weakness of IA is that it tends to be too conservative: the intervals it produces are often much wider than the true range of the corresponding quantities, 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 the computer graphics applications mentioned above.

To address this problem, we propose here a new model for numerical computation, which we call *affine arithmetic* (AA). Like standard IA, AA keeps track automatically of the round-off and truncation errors affecting 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 intervals than IA, especially in long computation chains.

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, including computer graphics.

Section 2 of the paper is a brief review of standard IA and its "error explosion" problem. Section 3 defines *affine forms*, the representation of quantities in the AA model. Section 4 gives the basic principle for computing with affine forms, and applies it to a couple of basic operations (addition, multiplication, and square root). Section 5 describes some technical details of our implementation of AA. Finally, section 6 discusses some applications of AA in computer graphics.

2 Standard interval arithmetic

In standard IA, each quantity x arising in a computation is represented by an interval $\bar{x} = [x.lo \mathinner{..} x.hi]$ of real numbers, meaning that the "true" value of x is known to satisfy $x.lo \leq x \leq x.hi$.

For any operation $f(x, y, \ldots)$ from reals to reals (such as sum, product, square root, etc.), one defines in IA a corresponding operation on intervals $\bar{f}(\bar{x}, \bar{y}, \ldots)$. 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 $\bar{x}, \bar{y}, \ldots$

Thus, for example, one defines the sum and difference of two intervals $\bar{x}, \bar{y}$ as

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

Multiplication and division can be handled by slightly more complex formulas; and the same holds for square root and the other basic mathematical functions.

2.1 The error explosion problem

Note that the standard IA operations always assume that the (unknown) values of the arguments may vary *independently* over the given intervals. 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 returned by the IA operation may be much wider than the true range of the result quantity.

As an extreme example, when we evaluate the expression $x-x$ with standard IA, we get the interval $\bar{x}-\bar{x} = [x.lo-x.hi, x.hi-x.lo]$, which is twice as wide as the original interval $\bar{x}$ — instead of $[0 .. 0]$, which is the true range of the expression. Note that 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 $\bar{x} = [4 .. 6]$. Applying the IA formulas blindly, we get

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

On the other hand, a trivial analysis shows that the true range of $x(10-x)$ is $[24 .. 25]$. 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.

Let $\bar{z}$ be the result of evaluating some expression $\bar{f}(\bar{x}, \bar{y}, \ldots)$ according to the rules of IA. Let also $\bar{z}^*$ be the "true range" of that expression — that is, the smallest interval that contains all values of $f(x, y, \ldots)$, when $x, y, \ldots$ range over the given intervals $\bar{x}, \bar{y}, \ldots$ We define the *relative accuracy* ρ of the IA computation as the width of $\bar{z}^*$ divided by that of $\bar{z}$. Thus, in the $x(10-x)$ example above, we had $\rho = (25-24)/(36-16) = 0.05$, meaning the resulting interval was 20 times wider than what it should be.

The over-conservatism of IA is particularly bad in a long computation chain, because the overall ρ of the chain tends to be the product of the ρs 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 1 below 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 $[-2 .. +2]$. Figure 2 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.

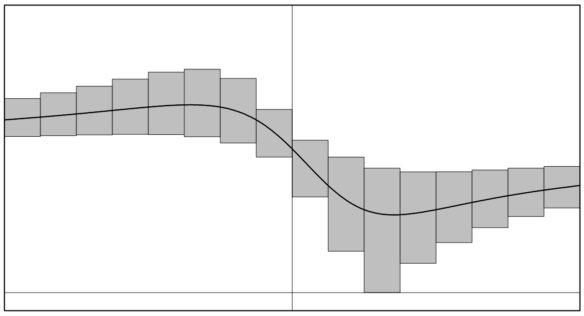

Figure 1: $g(x) = \sqrt{x^2-x+1/2}/\sqrt{x^2+1/2}$.

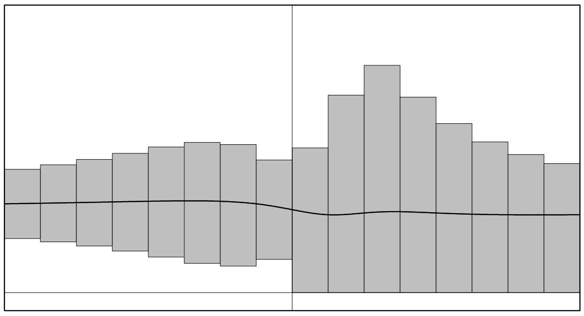

Figure 2: $h(x) = g(g(x))$.

(Standard interval arithmetic.)

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. So, if the relative accuracy of a computation is too small to be useful by a factor of 1000, we will probably have to split the domain into 1000 sub-intervals to obtain a useful result.

3 Affine arithmetic

Affine arithmetic (AA) is a computation model that attempts to retain the advantages of IA while ameliorating this "error explosion" problem. The key feature of AA is an extended encoding of quantities from which one can determine, in addition to their ranges, also certain relationships to other quantities

— such as the ones existing between x and $10-x$ in our earlier example.

Specifically, in AA, a partially unknown quantity x is represented by an *affine form* $\hat{x}$, which is a first-degree polynomial

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n \qquad (1)$$

Here the x_i are known real coefficients (stored as floating-point numbers), and the ε_i are symbolic variables whose values are unknown but assumed to lie in the interval $\mathbf{U} = [-1 .. +1]$.

Each ε_i stands for an independent source of error or uncertainty that contributes to the total uncertainty of the quantity x. The source may be external (due to original uncertainty in some input quantity) or internal (due to round-off and truncation errors committed in the computation of $\hat{x}$). The corresponding coefficient x_i gives the magnitude of that contribution.

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*.

Obviously, affine arithmetic is more complex (and expensive) than ordinary interval arithmetic. 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. See section 6 for specific examples.

3.1 Conversions between IA and AA

If $\hat{x} = x_0 + x_1\varepsilon_1 + \cdots x_n\varepsilon_n$ is an affine form for a quantity x, then the value of the latter is guaranteed to be in the *range of* $\hat{x}$, the interval

$$[\hat{x}] = [x_0 - \xi .. x_0 + \xi], \quad \xi = \sum_{i=1}^{n} |x_i|$$

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

Conversely, suppose we are given an ordinary interval $\bar{x} = [x.lo .. x.hi]$ representing some quantity x. We can generate from it a valid affine form $\hat{x} = x_0 + x_k\varepsilon_k$ for the same quantity, with

$$x_0 = \frac{x.hi + x.lo}{2} \qquad x_k = \frac{x.hi - x.lo}{2} \qquad (2)$$

The noise symbol ε_k symbolizes the uncertainty in the value of x. Since the interval $\bar{x}$ 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 previously used in the same computation.

The key feature of this model is that the same noise symbol ε_i 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 $\hat{x}, \hat{y}$ indicates some partial dependency between the underlying quantities x, y. The magnitude and sign of the dependency is determined by the corresponding coefficients x_i, y_i. (Note that the signs of x_i and y_i are not significant in themselves; only their relative signs are important.)

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

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

¿From this data we can tell that x lies in the interval $[6 .. 14]$, and y lies in $[12 .. 28]$; 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 region of $\mathbf{R}^2$ depicted in figure 3 (dark grey), which is substantially smaller than the rectangle $[6 .. 14] \times [12 .. 28]$ (light grey).

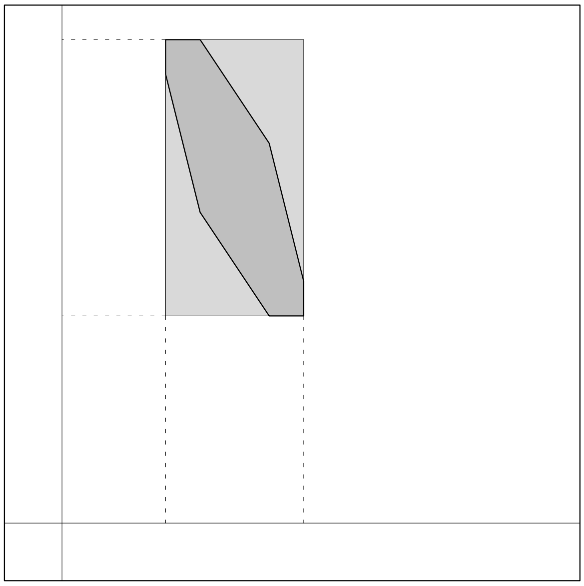

Figure 3: Joint range of (x, y)

Obviously, this dependency information would be lost if we were to replace $\hat{x}$ and $\hat{y}$ by the ordinary intervals $[\hat{x}]$ and $[\hat{y}]$, even though the latter encode precisely the same ranges of values as the former.

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

4 Computing with Affine Forms

To evaluate a formula in AA, we must replace each of its elementary operations $z \leftarrow f(x, y)$ on real numbers by an equivalent step $\hat{z} \leftarrow \hat{f}(\hat{x}, \hat{y})$ on affine forms, where $\hat{f}$ is a procedure that computes an affine form for $z = f(x, y)$ that is consistent with $\hat{x}, \hat{y}$.

By definition,

$$\begin{aligned} x &= x_0 + x_1\varepsilon_1 + \cdots x_n\varepsilon_n & (3)\\ y &= y_0 + y_1\varepsilon_1 + \cdots y_n\varepsilon_n & (4) \end{aligned}$$

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

$$\begin{aligned} z &= f(x, y)\\ &= f(x_0 + x_1\varepsilon_1 + \cdots x_n\varepsilon_n,\ y_0 + y_1\varepsilon_1 + \cdots y_n\varepsilon_n)\\ &= f^*(\varepsilon_1, .. \varepsilon_n) \qquad (5) \end{aligned}$$

The challenge now is to replace $f^*(\varepsilon_1, .. \varepsilon_n)$ by an affine form

$$z = z_0 + z_1\varepsilon_1 + \cdots z_n\varepsilon_n$$

that preserves as much information as possible about the constraints between x, y, and z that are implied by (3–5), but without implying any other constraints that cannot be deduced from the given data. The remainder of this section is devoted to this issue.

4.1 Affine operations

If f itself is an affine function of its arguments x, y, then the function $f^*(\varepsilon_1, .. \varepsilon_n)$ in (5) can be expanded into an affine combination of the noise symbols ε_i. In that case, by performing the expansion and collecting similar terms we get a valid affine form for $z = f(x, y)$. In particular,

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

for any affine forms x, y, and any $\alpha \in \mathbf{R}$. Note that these formulas are exact, in the sense that the resulting affine form contains all the information about the quantity z that can be deduced from the given affine forms x and y.

Note also that, according to those formulas, the difference $\hat{x} - \hat{x}$ between an affine form and itself is identically zero. In this case, the fact that the two operands share the same noise symbols with the same coefficients reveals that they are actually the same quantity, and not just two quantities that happen to have the same range of possible values. Thanks to this feature, in AA we also have $(\hat{x} + \hat{y}) - \hat{x} = \hat{y}$, $(3\hat{x}) - \hat{x} = 2\hat{x}$, and so on.

4.2 Non-affine operations

In general, when f is not an affine operation, the function $f^*(\varepsilon_1, .. \varepsilon_n) = f(\hat{x}, \hat{y})$ of (5) cannot be expressed as an affine combination of the ε_i. In that case, we must pick some affine function

$$f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n) = z_0 + z_1\varepsilon_1 + \cdots + z_n\varepsilon_n \qquad (6)$$

that approximates $f^*(\varepsilon_1, .. \varepsilon_n)$ 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,

$$\begin{aligned} \hat{z} &= f^{\mathrm{a}}(\varepsilon 1, .. \varepsilon_n) + z_k\varepsilon_k\\ &= z_0 + z_1\varepsilon_1 + \cdots + z_n\varepsilon_n + z_k\varepsilon_k \end{aligned}$$

Here ε_k must be a brand new noise symbol (distinct from all other noise symbols in the same computation) and z_k must be an upper bound on the absolute difference between f^{a} and f^*, for all possible values of $\varepsilon_1, .. \varepsilon_n$; that is,

$$\max\{\, |f^*(\varepsilon_1, .. \varepsilon_n) - f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n)| \,:\, \varepsilon_1, .. \varepsilon_n \in \mathbf{U} \,\}$$

Note that the substitution of $f^{\mathrm{a}} + z_k\varepsilon_k$ for f^* defeats in part the goal of AA: 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.

In order to minimize the loss of information, we should choose the coefficients $z_0, z_1, .. z_n$ so as to make the new error term $z_k\varepsilon_k$ as small as possible. In other words, f^{a} should be the first-degree polynomial that best approximates f^* over $\mathbf{U}^n$, in the Chebyshev sense of minimizing the maximum error.

Below we develop this approach in detail for two representative operations, namely multiplication and square root. The techniques illustrated by these examples can be easily extended to handle most of the other elementary operations and functions.

4.3 Multiplication

Let's now consider the multiplication of affine forms, that is, $z = f(x, y) = x \cdot y$. The quantity z is a quadratic polynomial $f^*(\varepsilon_1, .. \varepsilon_n)$ on the noise symbols:

$$z = f^*(\varepsilon_1, .. \varepsilon_n)$$

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

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

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

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 $\mathbf{U}^n$, 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

$$\hat{z} = A(\varepsilon_1, .. \varepsilon_n) + \frac{a+b}{2} + \frac{b-a}{2} \varepsilon_k \tag{7}$$

where ε_k is a "new" noise symbol.

A quick conservative estimate for the range of Q is the symmetric interval $[-uv \,..\, +uv]$, where

$$u = \sum_{i=1}^{n} |x_i| \qquad v = \sum_{i=1}^{n} |y_i|$$

This choice gives $(a+b)/2 = 0$, $(b-a)/2 = uv$; therefore,

$$\begin{aligned}
\hat{z} &= A(\varepsilon_1, .. \varepsilon_n) + uv\, \varepsilon_k \\
&= x_0 y_0 \\
&\quad + (x_0 y_1 + y_0 x_1)\, \varepsilon_1 \\
&\quad + \cdots \\
&\quad + (x_0 y_n + y_0 x_n)\, \varepsilon_n \\
&\quad + uv\, \varepsilon_k
\end{aligned} \tag{8}$$

This interval may be up to twice as wide as the exact range of Q; but, in any case, its size is still quadratic in the interval half-widths u and v. More precise estimates for the range of Q can be obtained by somewhat more complex formulas. In fact, the *exact* range of Q can be computed in $O(m \log m)$ time, where m is the number of nonzero partial deviations in $\hat{x}$ and $\hat{y}$. Lack of space prevents us from discussing these alternatives; and, anyway, it is not clear that their modest advantage in accuracy is worth their cost and complexity.

To illustrate these formulas, let's evaluate the expression $z = (10+x+r) \cdot (10-x+s)$ for $x \in [-2 \,..\, +2]$, $r \in [-1 \,..\, +1]$, $s \in [-1 \,..\, +1]$. Converting the ordinary intervals to affine forms, we get

$$\begin{aligned}
x &= 0 + 2\varepsilon_1 \\
r &= 0 + 1\varepsilon_2 \\
s &= 0 + 1\varepsilon_3 \\
10 + x + r &= 10 + 2\varepsilon_1 + 1\varepsilon_2 \\
10 - x + s &= 10 - 2\varepsilon_1 + 1\varepsilon_3
\end{aligned}$$

therefore

$$z \;=\; 100 + 10\varepsilon_2 + 10\varepsilon_3 + (2\varepsilon_1 + 1\varepsilon_2)(-2\varepsilon_1 + 1\varepsilon_3)$$

The quick estimate for the range of the quadratic part is

$$\bar{q} \;=\; [-(3 \cdot 3) \,..\, +(3 \cdot 3)] \;=\; [-9 \,..\, +9]$$

(The exact range being $[-9 \,..\, +1]$.) Hence, the result of the computation is

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

Observe that, in this example, influence of the noise symbol ε_1 in the factors happened to cancel out (to first order).

The true range of z is $[71 \,..\, 121]$. The range of z implied by this affine form above is

$$[100 - 29 \,..\, 100 + 29] \;=\; [71 \,..\, 129] \quad (\rho = 0.86)$$

By comparison, standard IA would return

$$[7 \,..\, 13] \cdot [7 \,..\, 13] \;=\; [49 \,..\, 169] \quad (\rho = 0.42)$$

4.4 Round-off errors

The discussion so far assumed that the coefficients z_i of the affine approximation $f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n)$ can be computed exactly. In a practical implementation, however, they will be computed with floating-point arithmetic, and thus affected by round-off errors. As a matter of fact, such errors will occur even when f is an affine operation like sum or difference.

In standard IA, these errors can be handled by simply making sure that the lower endpoint of the result interval is always rounded down, and the higher endpoint is always rounded up. In affine arithmetic, however, any kind of rounding is wrong. When computing an affine form $\hat{z}$, if we round off a partial deviation z_i, in either direction, then $\hat{z}$ will no longer correctly describe the quantity z, because it will imply slightly different (hence incorrect) relationships with other quantities. To preserve the validity of the affine form, we must determine an upper bound δ_i to the round-off error committed in the computation of each z_i, and then we must add all these δ_i to the linearization error z_k (rounding upwards).

4.5 Square root

Let's now consider the AA evaluation of $z = \sqrt{x}$. As before, we need to approximate the function

$$f^*(\varepsilon_1, .. \varepsilon_n) = \sqrt{x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n} \tag{9}$$

over the hypercube $\mathbf{U}^n$ by some affine function

$$f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n) = z_0 + z_1\varepsilon_1 + \cdot\cdot\, z_n\varepsilon_n$$

and then add to the latter an extra term $z_k\varepsilon_k$, to account for the difference $|f^* - f^{\mathrm{a}}|$, where ε_k is a brand-new noise symbol.

Note that the function $f^*(\varepsilon_1, .. \varepsilon_n)$ is constant over any hyperplane of $\mathbf{U}^n$ that is orthogonal to the vector $(x_1, .. x_n)$. It is not hard to show that the best (Chebyshev) affine approximation to f^* must also have this property. That is, we need only consider approximations $f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n)$ of the form

$$\begin{aligned} f^{\mathrm{a}}(\varepsilon_1, .. \varepsilon_n) &= \alpha\hat{x} + \beta \\ &= \alpha(x_0 + x_1\varepsilon_1 + \cdot\cdot\, x_n\varepsilon_n) + \beta \end{aligned} \tag{10}$$

It is also easy to show that the values of α and β that minimize the maximum error of $f^{\mathrm{a}} - f^*$ are precisely the coefficients of the optimum (Chebyshev) first-degree approximation $\alpha x + \beta$ to $\sqrt{x}$ in the interval $[\hat{x}]$. Thus, the problem of approximating the original n-variable function reduces to that of approximating a single-variable function.

If $[a\ ..\ b]$ is the interval $[\hat{x}]$, then the optimum Chebyshev coefficients α and β are

$$\begin{aligned} \alpha &= \frac{1}{\sqrt{b} + \sqrt{a}} \\ \beta &= \frac{\sqrt{a} + \sqrt{b}}{8} + \frac{1}{2}\frac{\sqrt{a}\sqrt{b}}{\sqrt{a} + \sqrt{b}} \end{aligned}$$

and the maximum approximation error is

$$\delta = \frac{1}{8}\frac{(\sqrt{b} - \sqrt{a})^2}{\sqrt{a} + \sqrt{b}}$$

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

Once α, β and δ are known, we can return

$$\begin{aligned} z_0 &= \alpha x_0 + \beta && (11) \\ z_i &= \alpha x_i \quad (i = 1, .. n) && (12) \\ z_k &= \delta && (13) \end{aligned}$$

This analysis assumes that we can compute α, β, and δ exactly. In practice, the computation of α must carried out in floating point, and so we will get only an approximation $\tilde{\alpha}$ to the optimum slope α. We must then choose the intercept β so as to minimize the maximum of $|\tilde{\alpha}x + \beta - \sqrt{x}|$, instead of $|\alpha x + \beta - \sqrt{x}|$. Again, we will only be able to compute an approximation $\tilde{\beta}$ to this optimum β. The approximation error δ is then the maximum of $\left|\tilde{\alpha}x + \tilde{\beta} - \sqrt{x}\right|$; again, we will only be able to compute an upper bound $\tilde{\delta}$ to it.

Formulas (11–13) must then be changed to use $\tilde{\alpha}$, $\tilde{\beta}$, and $\tilde{\delta}$ instead of α, β, and δ. Naturally, the computation of $z_0, z_1, .. z_n$ by these formulas will be affected by round-off error; we must therefore determine upper bounds $\tilde{\delta}_0, \tilde{\delta}_1, .. \tilde{\delta}_n$ for these errors, and add them to the linearization error $\tilde{\delta}$, always rounding up, to obtain the error term z_k.

On machines that support program-directed rounding, such as mandated by the IEEE floating-point standard [10], all these computations can be performed at reasonable cost: two square roots, plus a few floating-point operations for each x_i.

4.6 Accuracy of AA

Numerical experiments seems to confirm our claim that AA is in general substantially more precise than standard IA, and less prone to error explosion. For instance, consider figures 4 and 5 below. They show the same functions of figures 4 and 5, evaluated with AA instead of IA, over the same intervals.

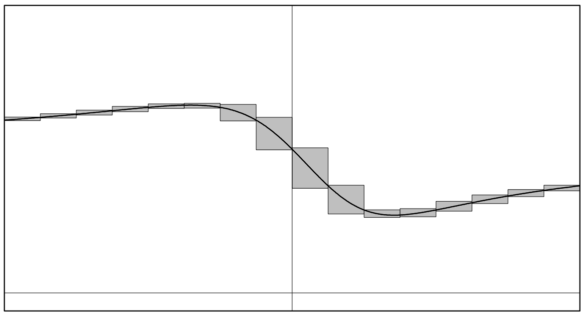

Figure 4: $g(x) = \sqrt{x^2 - x + 1/2}/\sqrt{x^2 + 1/2}$.

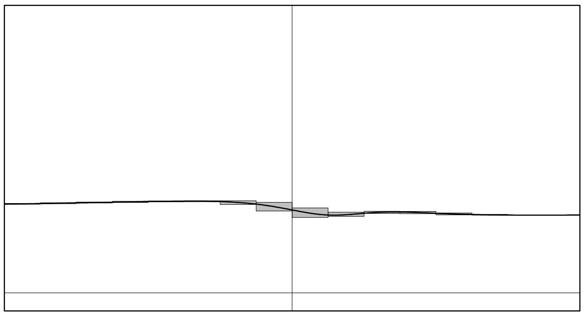

Figure 5: $h(x) = g(g(x))$.

(Affine arithmetic.)

The main reason why AA is usually more accurate than IA is the cancellation phenomenon described in section 4.3, which tends to make the range of computed quantities smaller than the corresponding intervals computed by standard IA. Indeed, except for round-off errors, any computation chain that involves only affine operations will be evaluated by AA with relative accuracy $\rho = 1$ — that is, the range of the computed affine form will be the true range of the corresponding quantity.

Even in the case of a non-linear operation, the loss of information (measured by the magnitude of the error term $z_k \varepsilon_k$) is often smaller for AA than for IA. Moreover, in a multi-step computation, the affine form of each computed quantity keeps track of how much of its uncertainty is attributable to the linearization error committed at each of the previous operations. Thus, these linearization errors themselves may cancel out in later operations, instead of always adding up (as they usually do in ordinary interval arithmetic). This same observation applies to the floating-point round-off errors committed in each step.

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 round-off 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 round-off 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.

Yet another reason for AA to be more accurate than IA is that the magnitude of the linearization error z_k in each operation will in general depend quadratically on the input deviations x_i, y_i. 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 (see section 2.1) 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.

5 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 will now describe some implementation choices that we made, but which are not part of the AA model proper.

5.1 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 the 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 i.

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.

5.2 Memory and index management

Storage for the affine forms themselves is normally allocated from a separate 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 explained above, new noise symbols are constantly being invented 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 round-off 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, it is advisable to manage the "index space" 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.

5.3 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 round-off 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.

Many applications of AA, such as those described in section 6, can be coded as procedures that take ordinary intervals as parameters, convert them to affine forms, and evaluate a 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. Moreover, the noise symbol indices would then be superfluous, and the AA arithmetic operations could be partly expanded in-line.

5.4 Shared sub-expressions

In actual programs, it is common for the same subformula 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 $(x^2 + 1)/(x^2 - 1)$ for AA evaluation, it is doubly important to identify common sub-expressions like x^2, and compute each of them only once.

6 Applications

We will now describe two applications of AA to computer graphics, which in fact motivated us to develop the whole model. However, it is our belief that affine arithmetic is a general-purpose technique which, like standard IA, will turn out to be useful in all sorts of numerical applications.

Computer graphics researchers have recently "discovered" standard interval arithmetic, and have successfully used it to solve various numerical problems that arise in this field, such as determining line-surface intersections for ray tracing, constructing octrees for implicitly defined surfaces, and detecting collisions between moving objects [1, 5, 6, 7, 8].

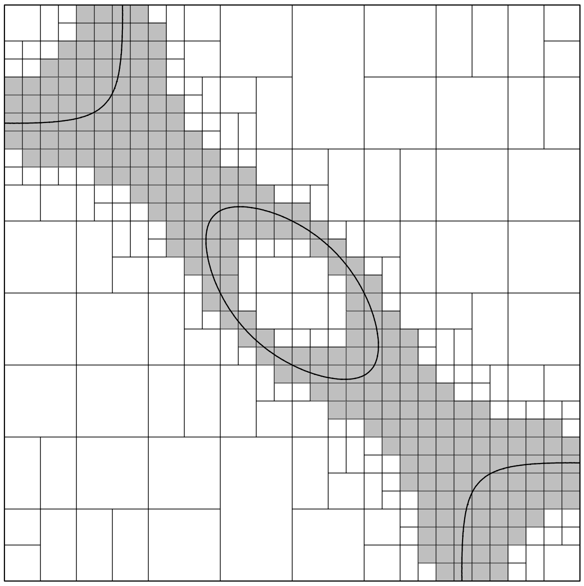

Figure 6: Interval arithmetic:
847 evaluations, 180 empty cells retained.

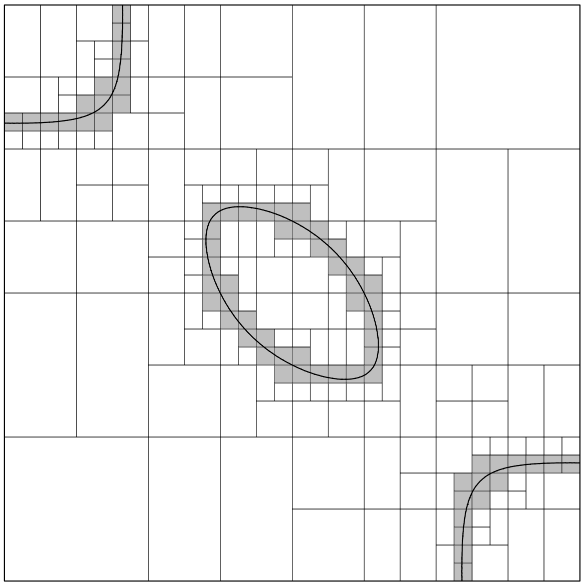

Figure 7: Affine arithmetic:
451 evaluations, 4 empty cells retained.

6.1 Computing octrees and quad-trees

Interval arithmetic has been used as a tool for the construction of octrees for implicit surfaces [1, 6]. The basic principle is that we can test if the surface $F(x, y, z) = 0$ enters a given axis-aligned box

$$B = [x.lo \,..\, x.hi] \times [y.lo \,..\, y.hi] \times [z.lo \,..\, z.hi]$$

of $\mathbf{R}^3$ by evaluating the expression $\bar{u} \leftarrow \bar{F}(\bar{x}, \bar{y}, \bar{z})$, with standard IA, on the intervals $\bar{x} = [x.lo \,..\, x.hi]$, $\bar{y} = [y.lo \,..\, y.hi]$, $\bar{z} = [z.lo \,..\, z.hi]$. If the resulting interval $\bar{u}$ does not include 0 — that is, if $u.lo > 0$ or $u.hi < 0$ — then we know that the surface does not extend into the box.

Of course, if the interval $\bar{u}$ does contain 0, we cannot tell whether the surface does extend into the box B; the computation may just have been too inaccurate. In that case, we must bisect B and repeat the test on each half-box. Typically we truncate the recursion when B gets too small, at which point we just add it to the octree.

We can improve the performance of this algorithm by using affine arithmetic to evaluate $\bar{u} = \bar{F}(\bar{x}, \bar{y}, \bar{z})$, instead of standard IA. This change generally reduces the width of the computed intervals $\bar{u}$. As a consequence, the algorithm will be able to discard empty regions of space earlier in the recursion; in particular, it will be able to discard more cells before giving up on the recursion, thus reducing the size of the final octree.

Figures 6 and 7 illustrate a two-dimensional analog of this process [8], the construction of a quadtree for the curve $F(x, y) = 0$, where $F(x, y) = x^2 + y^2 + xy - (xy)^2/2 - 1/4$, in the square $[-2 \,..\, +2] \times [-2 \,..\, +2]$. In both cases the minimum cell size (at the bottom of the octree) was $(1/8) \times (1/8)$; the curve actually enters only 66 of these cells.

Octrees built as described above are often used to speed up the ray-tracing of complex surfaces and solids [2, 9]. In this application, each empty node of the octree that is wrongfully retained (because of $\bar{u}$ beeing too wide) may lead to thousands of spurious ray-surface intersection tests. So, even though evaluating $F(\bar{x}, \bar{y}, \bar{z})$ with AA is more expensive than with IA, the benefits are usually worth the cost.

6.2 Ray tracing

Interval analysis has also been used for reliable ray-tracing of surfaces; specifically, to determine all intersections between an implicit surface

$$F(x, y, z) = 0$$

and a line segment pq (the "ray") [6].

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

$$f(t) = F(\bar{t}p_x + tq_x,\ \bar{t}p_y + tq_y,\ \bar{t}p_z + tq_z)$$

where $t \in [0 \,..\, 1]$ and $\bar{t} = 1 - t$. A robust and reasonably efficient algorithm for the latter combines interval analysis with Newton's root-finding method. We evaluate $\bar{u} = f(\bar{t})$ using IA, for the whole interval $\bar{t} = [0 \,..\, 1]$. If the resulting interval $\bar{u}$ is strictly positive or strictly negative, we know that the ray does not intersect the surface. Otherwise, we evaluate the derivative $\bar{v} = f'(\bar{t})$, 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 contained in $\bar{t}$. If $\bar{t}^*$ is less than half as wide as $\bar{t}$, 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 $f(\bar{t})$ in the IA model is equivalent to evaluating $F(\bar{x}, \bar{y}, \bar{z})$ on the intervals $\bar{x} = [p_x \mathinner{..} q_x]$, $\bar{y} = [p_y \mathinner{..} q_y]$, $\bar{z} = [p_z \mathinner{..} q_z]$ — that is, evaluating F 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 $F(x, y, z)$ 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 $F(\bar{x}, \bar{y}, \bar{z})$ 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.

Here again, replacing standard IA by affine arithmetic 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 $f(\bar{t})$.

Moreover, as the width of the interval $\bar{t}$ 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, 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.

Acknowledgements

All three authors are curently supported in part by grants from Brazilian research funding agencies (CNPq and FAPESP). Some of this research was performed at the DEC Systems Research Center (SRC) in Palo Alto, whose director Dr. Robert W. Taylor we wish to thank for his friendly encouragement and generosity. Lyle Ramshaw of DEC SRC and Luiz Henrique Figueiredo of IMPA contributed many helpful comments and suggestions.

References

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

[2] A. Glassner, *Space subdivision for fast ray tracing.* IEEE Computer Graphics & Applications, Oct. 1984.

[3] R. E. Moore, *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ (1966).

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

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

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

[7] J. M. Snyder, A. R. Woodbury, K. Fleischer, B. Currin, and A. Barr, *Interval analysis for multi-point collisions between time-dependent curved surfaces.* Proceedings of SIGGRAPH'93, in ACM Computer Graphics Annual Conference Series (1993), 321–334.

[8] K. G. Suffern and E. D. Fackerell, *Interval methods in computer graphics.* Computers & Graphics **15**, 3 (1991), 331–340.

[9] G. Wyvill, T. .L .Kunii, and Y. Shirai, *Space division for ray tracing in CSG.* IEEE Computer Graphics & Applications, April 1986.

[10] *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Standard 754-1985, IEEE, New York (1985).