Precise Numerical Computation

Mark Sofroniou [†] *& Giulia Spaletta* [‡]

Dedicated to Jerry B. Keiper (1953—1995) and Nicholas C. Metropolis (1915—1999)

Arithmetic systems such as those based on IEEE standards currently make no attempt to track the propagation of errors. However a formal error analysis can be complicated and is often confined to the realm of experts in numerical analysis. In recent years there has been a resurgence in interest in automated methods for accurately tracking the propagation of errors. In this article we will describe a model known as significance arithmetic. We will give details of the implementation in *Mathematica* along with several examples that illustrate the design goals and differences over conventional fixed precision floating point systems.

[†] *Wolfram Research Inc, Champaign, Illinois, U. S. A.*
[‡] *Department of Mathematics, University of Bologna, Italy*

Introduction

Knuth has expressed a desiderata of floating point arithmetic as follows [Knuth 1998]:

> It would be nice if we could give our input data for each problem in an unnormalized form which expresses how much precision is assumed, and if the output would indicate just how much precision is known in the answer.

This enhancement would assist those who do not wish to undertake a rigorous analysis of computational error.

Significance arithmetic is one approach for providing such a facility. It provides local error monitoring. The computational paradigm is not a panacea however. Knuth goes on to say (see also [Ashenhurst 1965]):

> The results of local error monitoring should be combined with a global analysis to take into account the interdependence which generally exists between the operands involved in an elaborate computation.

A premise of *Mathematica*'s arbitrary precision significance arithmetic is that it is preferable that the (dynamic control of) digits in low−orderoperations should be based upon the precision of the operands rather than by considerations such as fixed format storage limitations.

Significance arithmetic is not new and has been around since the late 1950s (see [Gray 1959], [Metropolis 1958] and [Metropolis 1959]). Despite an initial period of enthusiasm, one of the reasons the model fell out of favor was its inefficiency compared with alternative hardware implementations of fixed precision arithmetic models such as that which evolved into the IEEE 754 standard. Other reasons for the demise were that hardware implementations impose a severe restrictions on the range of numbers that can be represented and error estimates that were obtained using an integral value for the associated error gave results that were frequently too pessimistic to be of practical value [Sterbenz 1974].

Since significance arithmetic dynamically adjusts the number of digits used in computations, it is ideally suited to a software implementation where the mantissa can readily change in size over a very wide range. Many modern computational environments are equipped with facilities for carrying out arithmetic in software. Furthermore, there seems to be an increasing demand from the user community for tools which assist in a numerical study of problems, but at the same time give some guarantee of reliability of the solution.

The goal of this article is to describe the current state of significance arithmetic as it is implemented in *Mathematica.* We will attempt to give a comprehensive description of how errors are represented and combined in the implementation. We will also describe some enhancements since the early days of development. Examples are then given to compare the arithmetic with more conventional models.

```
Off[General::spell, General::spell1];
SetOptions[{Precision, Accuracy}, Round->False];
```

Definitions

In this section we introduce some standard definitions and explain the terminology that is used in *Mathematica*.

■ Standard terminology

The terms accuracy and precision are often confused or used interchangeably. In most numerical analysis contexts accuracy refers to the absolute or relative error of an approximate quantity. Precision usually refers to the number of digits with which the basic arithmetic operations +, −,*, / are performed. See [Higham 1996] for more discussion of this topic.

■ Representation of numbers

Here we recall several definitions that serve to illustrate how real numbers are represented in a computer (see [Koren 1993]). A base β number X of length n = k + m is an n−tuplewith radix point between the k most significant digits and the m least significant digits:

$$\underbrace{x_{k-1}\, x_{k-2} \cdots x_0}_{\text{integer part}} . \underbrace{x_{-1}\, x_{-2} \cdots x_{-m}}_{\text{fractional part}}$$

The *fixed point representation* of X is given by:

$$\begin{aligned} &= x_{k-1}\,\beta^{k-1} + \ldots + x_1\,\beta + x_0 + x_{-1}\,\beta^{-1} + \ldots + x_{-m}\,\beta^{-m} \\ &= \sum_{i=-m}^{k-1} x_i\,\beta^i, \quad 0 \le x_i < \beta \quad \forall\, i \end{aligned}$$

Commonly used positions for the radix point are at the rightmost side of the number (pure integers, m=0) and at the leftmost side of the number (pure fractions, k=0).

The *signed-magnitude* representation of a floating-point number X in base β consists of three parts, the *sign* S, the (unsigned) *mantissa* or *significand* M and the *exponent* E:

$$X = (S,\ M,\ E)_\beta = (-1)^S\, M\, \beta^E$$

The mantissa $M \neq 0$ is said to be *normalized* if $x_{k-1} \neq 0$.

A normalized mantissa M can be represented as an integer by taking $\beta^{n-1} \le M < \beta^n$ or as a fraction by taking $1/\beta \le M < 1$ and adjusting the exponent accordingly.

A term that we will use occasionally is the *weight* β^{-m} of the least significant digit, which is commonly referred to as an *ulp* and signifies a *Unit in the Last Place* [Knuth 1998] or *Unit in the Last Position* [Koren 1993].

In IEEE 754 arithmetic the base is taken as $\beta = 2$ and numbers are represented as native types in hardware (the IEEE 854 standard allows choices other than $\beta = 2$). In contrast, arbitrary precision numbers are represented in software where M is usually an array of unsigned integers. Commonly used bases are $\beta = 2^{16}$, $\beta = 2^{32}$ or $\beta = 2^{64}$ depending on the architecture and whether hardware integer arithmetic or assembler or software routines are used to multiply and add single digits.

■ Condition numbers

Let us examine the result of applying a unary function f to an approximate value $x + \Delta_x$. Assume for simplicity that f is twice continuously differentiable, then we can expand using a Taylor series to obtain:

$$f(x + \Delta_x) - f(x) = f'(x)\,\Delta_x + O({\Delta_x}^2)\ .$$

Neglecting higher order terms we have the absolute error measure

$$\Delta_{f(x)} = f(x + \Delta_x) - f(x) \approx f'(x)\,\Delta_x\ . \qquad (1)$$

Taking absolute values and logarithms we can define the approximate scale of the error (**Accuracy**) of a function f in terms of the scale of the error of the input.

The quantity

$$c_a(\mathrm{f},\ \mathrm{x}) = \left| f'(x) \right|$$

is known as *absolute condition number* and is the amount by which an error in x gets magnified to become an error in $f(x)$.

The following function can be used to compute the absolute condition number.

```
AbsoluteConditionNumber[f_, x_] := Abs[D[f, x]];
```

Normalizing by dividing both sides in (1) by $f(x) \neq 0$ gives a relative error measure

$$\frac{\Delta_{f(x)}}{f(x)} \approx \frac{f'(x)\,\Delta_x}{f(x)} .$$

Rewriting the term on the right hand side for x≠0 gives

$$\frac{\Delta_{f(x)}}{f(x)} \approx \left(\frac{x\,f'(x)}{f(x)}\right)\left(\frac{\Delta_x}{x}\right) .$$

By taking absolute values and logarithms, we can therefore relate the **Precision** of the input to the **Precision** of f.

The quantity

$$c_r(\mathrm{f},\,\mathrm{x}) = \left|\frac{x\,f'(x)}{f(x)}\right|$$

is known as the *relative condition number* of f (see for example [Bevilacqua 1992], [Skeel 1993], [Higham 1996]). The relative condition number measures the relative change in the output for a given relative change in the input.

The following function can be used to compute the relative condition number.

```
RelativeConditionNumber[f_, x_] := Abs[x D[f, x] / f];
```

Example 1: The relative condition number of exp(x) is |x| and so the relative error in the exponential function is sensitive for x ≫ 1 and insensitive for x ≪ 1.

Example 2: The relative condition number of log(x), $x \neq 0$ is |1/log(x)| and so the relative error in the logarithm function is sensitive for x ≈ 1.

■ Terminology used in *Mathematica*

The decimal scale of a number x is defined as:

$$\log_{10}(|x|). \tag{2}$$

We can define a function to compute the scale of a number in decimal as follows, where a special case handles arguments near zero.

```
Scale[x_] := If[x == 0, Accuracy[x], Log[10, Abs[x]]];
```

Denote the *absolute error* of a number x as Δ_x and denote the *relative error* of x≠0 as Δ_x / x.

In *Mathematica*'s significance arithmetic, **Accuracy** is the negative of the scale of the absolute error of x:

$$-\log_{10}(|\Delta_x|). \tag{3}$$

Precision is the negative of the scale of the relative error:

$$-\log_{10}\left(\left|\frac{\Delta_x}{x}\right|\right). \tag{4}$$

Comparing (2), (3) and (4), **Scale**, **Accuracy** and **Precision** are related in the following way:

$$\texttt{Precision[x] == Scale[x] + Accuracy[x]} \tag{5}$$

The value x=0 is an exception, which reflects the fact that the relative error is undefined.

Here are pictorial representations of the quantities, where the radix point is assumed to be fixed in absolute position at zero.

When **Accuracy**, **Precision** and **Scale** are all positive we have:

$$\overbrace{\underbrace{x_1\, x_2 \cdots x_s}_{\textbf{Scale}}.\underbrace{x_{s+1}\, x_{s+2} \cdots x_{s+a}}_{\textbf{Accuracy}}}^{\textbf{Precision}}$$

When **Accuracy**, **Precision** are positive and **Scale** is negative we have:

$$\overbrace{\underbrace{.0\,0 \cdots 0}_{\textbf{Scale}}\underbrace{x_1\, x_2 \cdots x_p}_{\textbf{Precision}}}^{\textbf{Accuracy}}$$

When **Precision**, **Scale** are positive and **Accuracy** is negative we have:

$$\overbrace{\underbrace{x_1\, x_2 \cdots x_p}_{\textbf{Precision}}\underbrace{0\,0 \cdots 0.}_{\textbf{Accuracy}}}^{\textbf{Scale}}$$

Any of the values for **Accuracy**, **Precision** and **Scale** can be negative and are not necessarily integers, which means that the depicted quantities do not align exactly at the digit boundaries.

`Accuracy` can be thought of as the approximate number of digits to the right of the decimal point, **`Scale`** as the approximate number of digits to the left of the decimal point and **`Precision`** as the approximate number of significant digits. While these notions can be reasonable aids to understanding the terminology, they should not obscure the definitions in terms of absolute and relative errors.

We will endeavor to use the highlighted words **`Accuracy`** and **`Precision`** when referring to the terms in *Mathematica* in order to distinguish from the standard definitions of accuracy and precision.

■ Error Propagation

In order to see why **`Accuracy`** and **`Precision`** have been defined the way they have, it is useful to consider some examples that combine errors in an absolute and relative fashion.

▪ Addition and multiplication

We can define two intervals:

$$\hat{x} = x + \delta_x, \quad \hat{y} = y + \delta_y \tag{6}$$

in terms of the *absolute half−lengths* δ_x and δ_y:

$$\delta_x = \pm \frac{1}{2}\, 10^{-\mathbf{Accuracy}[x]} = \pm \frac{1}{2}\, |\Delta_x|\,,$$
$$\delta_y = \pm \frac{1}{2}\, 10^{-\mathbf{Accuracy}[y]} = \pm \frac{1}{2}\, |\Delta_y|\,. \quad (7)$$

The sum of the two intervals defined by (6) and (7) is given by:

$$\hat{x} + \hat{y} = \mathrm{x} + \mathrm{y} + \delta_x + \delta_y.$$

Most errors do not combine according to such a simple rule. Significance arithmetic in *Mathematica* combines errors using a linear, or leading term, approximation.

We can also define two intervals as:

$$\hat{x} = x * (1 + \varepsilon_x), \quad \hat{y} = y * (1 + \varepsilon_y) \quad (8)$$

where the *relative half-lengths* ε_x and ε_y for x, y≠0 are defined as:

$$\varepsilon_x = \pm \frac{1}{2}\, 10^{-\mathbf{Precision}[x]} = \pm \frac{1}{2}\, \left| \frac{\Delta_x}{x} \right|,$$
$$\varepsilon_y = \pm \frac{1}{2}\, 10^{-\mathbf{Precision}[y]} = \pm \frac{1}{2}\, \left| \frac{\Delta_y}{y} \right|. \quad (9)$$

The product of the two intervals defined by (8) and (9) is given by:

$$\hat{x} * \hat{y} = \mathrm{x}^*\mathrm{y}^*(1 + \varepsilon_x + \varepsilon_y + \varepsilon_x * \varepsilon_y)\,. \quad (10)$$

Mathematica's implementation of significance arithmetic neglects terms that are higher than first order, so that the following approximation is used in place of (10):

$$\hat{x} * \hat{y} \approx \mathrm{x}^*\mathrm{y}^*(1 + \varepsilon_x + \varepsilon_y)\,.$$

For real numbers x and y, **Accuracy** combines in the following way under addition:

$$\mathbf{Accuracy}[x + y] == -\mathrm{Log}[10, 10^{-\mathbf{Accuracy}[x]} + 10^{-\mathbf{Accuracy}[y]}]. \quad (11)$$

Likewise **Precision** combines in the following way under multiplication:

$$\mathbf{Precision}[x * y] == -\mathrm{Log}[10, 10^{-\mathbf{Precision}[x]} + 10^{-\mathbf{Precision}[y]}]. \quad (12)$$

■ Linearized error propagation for functions

Numerical algorithms for computing an elementary function can be written in terms of addition and multiplication at some level. However, relying on the error propagation rules for these operations would often give very pessimistic error bounds.

Much better bounds can be obtained by directly imposing error estimates based on properties of functions during their numerical computation.

Example: Exp

The error for the exponential function at x propagates as

$$\Delta_{\exp(x)} \approx \exp(\mathrm{x})\, \Delta_x \,,$$

or

$$\frac{\Delta_{\exp(x)}}{\exp(x)} \approx \Delta_x \,.$$

Thus the **Precision** of **Exp** is related to the **Accuracy** of the input.

Example: Log

The error for the logarithm function at $x \neq 0$ propagates as

$$\Delta_{\log(x)} \approx \frac{\Delta_x}{x} \,.$$

Thus the **Accuracy** of **Log** is related to the **Precision** of the input.

Example: Sin and Cos

The error propagation for certain functions are related to each other. For sin and cos at x errors are related in the following way:

$$\Delta_{\sin(x)} \approx \cos(\mathrm{x})\, \Delta_x$$

$$\Delta_{\cos(x)} \approx \sin(\mathrm{x})\, \Delta_x$$

Thus the **Accuracy** of **Sin** and **Cos** are related to the **Accuracy** of the input.

Example: Power

For powers x^y we have:

$$\Delta_{x^y} \approx y\, x^{y-1}\, \Delta_x \,,$$

which for x≠0 can be rewritten as

$$\frac{\Delta_{x^y}}{x^y} \approx y\, \frac{\Delta_x}{x} \,.$$

Here we se that at x=0 the **`Accuracy`** of the input is related to the **`Accuracy`** of the output, otherwise the **`Precision`** of the input is related to the **`Precision`** of the output.

For exponents |y| > 1 the **`Precision`** of the output is lower than the **`Precision`** of the input, whereas for exponents |y| < 1 the **`Precision`** of the output is greater than the **`Precision`** of the input.

Syntax for significance arithmetic

In this section we describe in some detail how *Mathematica*'s significance arithmetic works, giving examples of how errors are represented and propagated through computations.

■ Input syntax

SetAccuracy[x, n] (**SetPrecision[x, n]**) yields a number with **Accuracy** (**Precision**) of n.

To create a number with **Precision** 20 we can use any of the following forms:

```
SetPrecision[53124, 20]
```

```
53124.000000000000000
```

```
53124`20
```

```
53124.000000000000000
```

```
5.3124`20 10^4
```

```
53124.000000000000000
```

```
5.3124`20*^4
```

```
53124.000000000000000
```

Analogously, to create a number with **Accuracy** 20 we can use any of the following forms:

```
SetAccuracy[53124, 20]
```

```
53124.00000000000000000000
```

```
53124``20
```

```
53124.00000000000000000000
```

```
5.3124``24 10^4
```

```
53124.000000000000000000000
```

```
5.3124``20*^4
```

```
53124.000000000000000000000
```

Note that since zero is an exceptional value in the significance arithmetic model, it is not possible to create a zero that only has **Precision** n.

```
SetPrecision[0, 5]
```

```
0
```

In contrast, it is possible to create a zero with **Accuracy** n.

```
SetAccuracy[0, 5]
```

```
0. × 10^-5
```

This can be thought of as an interval containing zero in which none of the digits of the number at the center of the interval are known to be correct and is sometimes referred to as an *order of magnitude zero* [Knuth 1998].

A user may know that, due to the algorithm they are using, rounding errors may cancel in some way that is not best represented by an independent error model. One role of **SetAccuracy** and **SetPrecision** is to allow a shift of the boundary between the 'good' digits and the 'bad' digits, so that the functions allow a user to override the default error accumulation model and impose their own alternative.

Here we create a **Precision** 16 software number from the machine floating point number 0.1.

```
SetPrecision[0.1, 16]
```

```
0.1000000000000000
```

We now increase the **Precision** of the previous software number. If the boundary is moved past the end of the stored digits, more digits are appended. The extra digits that are added are zero in the internal base representation, but may appear as nonzero values in the decimal form that is used for output.

```
SetPrecision[%, 50]
```

```
0.10000000000000000555111512312578270211815834045410
```

An example where it is useful to explicitly control the number of digits in a computation is in Newton's method in arbitrary precision computations. At each step of an iteration we know that errors are strongly correlated and combine in such a way that the number of correct digits approximately doubles at each iteration.

■ Basics

For exact numbers, **Precision** and **Accuracy** are infinite.

```
Precision[2]
```

```
∞
```

```
Accuracy[2]
```

```
∞
```

For a general expression, **Accuracy** and **Precision** return the minimum of the values of the constituent subexpressions.

```
Precision[ 1.`50 + f[3 / 2 + g[2.7`20]] ]
```

```
20.
```

This confirms that the **Precision** of the output is the same as the **Precision** of the input.

```
Precision[3.14159`23.2]
```

```
23.2
```

There is no monitoring of **Precision** or **Accuracy** when machine numbers are involved: the assumption is that if machine numbers are being used then the primary concern is run–time efficiency. Furthermore, machine arithmetic is contagious: if a machine number is involved the result should be a machine number.

This adds a machine number to a software number with **Precision** 20.

```
1.2 + 33.2`20
```

```
34.4
```

The result is a machine number.

```
MachineNumberQ[%]
```

```
True
```

The example above assumes that the result can be represented as a machine number, but machine underflow or overflow can cause numbers to be forced into the realm of software numbers.

Here is the maximum representable number for this machine:

```
maxmachnum = $MaxMachineNumber
```

$$1.79769 \times 10^{308}$$

This adds two machine numbers; the result is a machine overflow, which is detected and trapped internally by *Mathematica* and a software number or *bigfloat* result is returned.

```
maxmachnum + maxmachnum
```

$$3.595386269724631 \times 10^{308}$$

```
MachineNumberQ[%]
```

```
False
```

■ Implementation details

In order to implement a significance arithmetic scheme, it is necessary to add an additional field to the floating point number This number is used to keep track of the scale of the error. Early implementations of significance arithmetic added an integer number, or *index of significance*, that was an indicator of which bits in the mantissa were to be considered correct [Metropolis 1958].

The first version of *Mathematica* used a method referred to as the unnormalized method of significance arithmetic, which essentially counted the number of significant digits remaining after leading zeros [Gray 1959], [Metropolis 1959].

In versions prior to release 2.0 of *Mathematica* the values of **Accuracy** and **Precision** were integers that reflected the index of significance [Metropolis 1958]. In version 2.0 the values were still integers, but the case of combining two numbers with the same **Accuracy** or **Precision** was corrected (downwardly) by one unit.

As we have seen, **Accuracy** and **Precision** can be defined in a more precise manner than by counting significant digits using an index of significance. In versions after 2.0 of *Mathematica* **Accuracy** and **Precision** were changed to reflect a *continuous* error model (see [Metropolis 1977a] and [Jones 1984]). Values were computed using a single precision floating point value which was then stored in the structure of bigfloat numbers. A minimax approximation was used to combine **Accuracy** and **Precision** (see [Keiper 1995]) according to the rules described above.

Accuracy and **Precision** used to return an integer number, which was a reflection of the early implementation. In version 3.0 and onward facilities were added for returning unrounded values, though these were never formally documented.

■ Current state

Accuracy and **Precision** are now computed as double precision numbers. Internally the values reflect the binary number system and are converted to the decimal system on output. The function for computing the logarithm in base two now relies on the machine library logarithm function which is faster and more portable across different architectures. Additional binary argument reduction is performed as a preprocessing step so that the computed error is minimized (see for example [Plauger 1992]). Various optimizations are used, such as detecting identical and disparate values of **Accuracy** and **Precision** of two operands, in order to circumvent potentially expensive machine precision exp–logcomputations which implement the rules (11) and (12).

Since the values of **Accuracy** and **Precision** are floating point numbers a recent enhancement that has improved robustness (and in some cases efficiency) uses a tolerance to compare values. For a discussion of floating

point comparison see section 4.2.2 of [Knuth 1998}. Note that the user−level relational operators in *Mathematica* such as **Equal**, **GreaterEqual** etc. use a tolerance in floating point comparison.

■ Guard digits

Bigfloat numbers in *Mathematica* can be visualized pictorially as

$$\overbrace{x_1\, x_2\, \cdots\, x_s}^{\textbf{Scale}}.\overbrace{x_{s+1}\, x_{s+2}\, \cdots\, x_{s+a}}^{\textbf{Accuracy}}\overbrace{z_1\, z_2\, \cdots\, z_n}^{\textbf{Guard digits}}$$

where the guard digits represent trailing digits that are not certain to be correct. In early implementations an integral quantity, sometimes referred to as the *residence of the least significant digit* [Metropolis 1977a], was used to represent the boundary between significant and insignificant digits.

Guard digits have the standard use of safeguarding against incorrect rounding (see for example [Santoro 1989]), but other roles arise due to the properties of significance arithmetic. Storing a few extra words allows efficient shifting of the boundary between significant and insignificant digits, by preventing against excessive data copying that would otherwise be needed to create a software number each time **SetAccuracy** or **SetPrecision** is invoked. There is a trade−off here, since storing too many guard digits can slow down arithmetic operations. Storing too many guard digits can also have the drawback of leading to a coarser representation error in the stored floating point number that is propagated when combining **Accuracy** and **Precision** in arithmetic operations.

Examples

■ Cancellation of rounding errors using machine numbers

Consider the following example that turns up in several practical situations [Higham 1996].

$$f(x) = \frac{(\exp(x) - 1)}{x}\,. \tag{13}$$

Here is the limiting value for small arguments, which can be seen by taking L'Hospital's rule (see for example [Hairer 1995]).

```
Limit[(E^x - 1)/x, x → 0]
```

```
1
```

This defines a routine for computing the function given in (13). A special case handles arguments near zero.

```
f1[x_Real] := If[Abs[x] <= $MinMachineNumber, 1., (E^x - 1)/x];
```

Using machine precision, severe cancellation for small arguments $x \ll 1$ causes the result to be inaccurate.

```
Table[f1[1. 10^-i], {i, 10, 16}]
```

```
{1., 1., 1.00009, 0.999201, 0.999201, 1.11022, 0.}
```

■ Remedies

A remedy for machine precision is to rewrite the function defined in (13) by computing *both* exp and log as follows [Higham 1996].

$$g(x) = \frac{(y-1)}{\log(y)}, \quad y = \exp(x). \tag{14}$$

For small x, let $\hat{y}$ denote the value of exp(x)(1+δ) with $|\delta| \leq$ ulp. Neither $\hat{y} - 1$ nor $\log(\hat{y})$ represent accurate approximations. However, the errors made in computing $\hat{y} - 1$ and $\log(\hat{y})$ almost completely cancel, so that

$$\frac{(\hat{y}-1)}{\log(\hat{y})},$$

is a very good approximation to

$$\frac{(y-1)}{\log(y)}.$$

This defines a routine that implements the remedy given in (14).

```
f2[x_] := With[{y = e^x}, If[y == 1, y, (y - 1)/Log[y]]];
```

Now we obtain an accurate result for small machine numbers. The default printing of machine numbers yields only 6 digits, so we use **InputForm** to display all the digits.

```
InputForm[Table[f2[1. 10^-i], {i, 10, 16}]]
```

```
{1.0000000000499998, 1.000000000005, 1.0000000000005,
 1.00000000000005, 1., 1., 1.}
```

Another remedy worth mentioning is the function expm1 that the C language provides for the accurate computation of exp(x)−1 in situations when the argument x is near one in magnitude. The onus is clearly on the implementor to write code to take advantage of such facilities when available.

■ Adaptive precision

Mathematica's significance arithmetic raises **Precision** based on the conditioning of the exponential. If we take an input value with **Precision** 16 for example, then we see that the **Precision** in the output increases according to the **Scale** of the input.

```
Exp[1.`16 10^-10]
```

```
1.00000000010000000000050000
```

```
Precision[%]
```

```
26
```

After subtraction we are left with a value which has **Precision** 16.

```
Exp[1.`16 10^-10] - 1
```

```
1.000000000050000×10^-10
```

In the following example we obtain a result which has the same **Precision** as the input *without* needing to reformulate the function defined in (13).

```
f3[x_] := If[x == 0, SetAccuracy[1, Accuracy[x]], (e^x - 1)/x];
```

```
Table[f3[1.`16 10^-i], {i, 10, 16}]
```

```
{1.000000000050000, 1.000000000005000,
 1.000000000000500, 1.000000000000050,
 1.000000000000005, 1.000000000000000, 1.000000000000000}
```

■ A common misconception

A common misconception in numerical computation is that it is possible to compute a quantity using, say, single and double precision and if the two results agree to some number of digits, then those digits must be correct. The following example of Rump demonstrates that this argument is not valid [Rump 1988].

▪ A polynomial example of Rump

Here we define a function of two arguments that can be used to evaluate a polynomial of Rump numerically.

```
RumpPolynomial[x_, y_] :=
  1335 y^6/4 + x^2 (11 x^2 y^2 - y^6 - 121 y^4 - 2) + 11 y^8/2 + x/(2 y);
```

The polynomial was evaluated by Rump at the values x = 77617 and y = 33096 using single, double and extended precision computations on an S/370 computer. The input data are exactly representable and so the only source of error is the rounding that occurs during evaluation. The results are summarized in the following table [Hansen 1992].

Single precision :	1.172603
Double precision :	1.1726039400531 ...
Extended precision :	1.172603940053178 ...

Unfortunately none of these results are correct. In fact, even the sign is wrong!

Using machine arithmetic in *Mathematica* we get a similarly wrong, but somewhat different, result. The result obtained depends on the scheme used for integer exponentiation and on the order of evaluation of arithmetic

operations, since floating point addition and multiplication are not associative; see [Knuth 1998] for a discussion of these topics.

```
RumpPolynomial[77617., 33096.]
```

```
-1.18059×10^21
```

■ Fixed precision arithmetic

Mathematica allows the use of fixed precision arithmetic as an alternative to significance arithmetic. In contrast to significance arithmetic, fixed precision computations make no attempt to track errors.

In order to illustrate how fixed precision can be used, we define a function **FixedPrecisionEvaluate** which locally sets the same value for the system parameters **$MinPrecision** and **$MaxPrecision** that govern the minimum and maximum number of digits that should be used in computations. The function uses the **HoldFirst** attribute which ensures that numerical evaluation is not carried out until the parameters that control the number of digits to be used are locally reset in the scoping construct **Block**.

```
SetAttributes[FixedPrecisionEvaluate, HoldFirst];

FixedPrecisionEvaluate[input_, digits_] :=
  Block[{$MaxPrecision = digits, $MinPrecision = digits},
    input
  ];
```

Once again we return to the example of Rump for illustration. Here we construct a list of inputs that numerically evaluate the polynomial for a fixed but increasing numbers of digits.

```
Table[
 FixedPrecisionEvaluate[
  RumpPolynomial[
   SetPrecision[77617, p], SetPrecision[33096, p]],
  p],
 {p, 30}]
```

```
{6.×10^27, 6.3×10^27, 6.34×10^27, 6.339×10^27, 4.2937×10^23,
 4.29366×10^23, 4.293664×10^23, 4.2939327×10^23,
 4.29393273×10^23, 7.672444915×10^18, 7.6724449152×10^18,
 7.67244491518×10^18, 7.672699138588×10^18,
 7.6726991385883×10^18, 8.98936655052812×10^13,
 8.989366550528117×10^13, 8.9893665505281173×10^13,
 8.98961657692171726×10^13, 8.989616576921717260×10^13,
 8.95877131726039400 53×10^7, 8.958771317260394005 32×10^7,
 8.95877131726039400531 8×10^7, 8.958771317260394005317 9×10^7,
 8.958771317260394005317 86×10^7,
 -0.82739605994682136814116 51,
 -0.82739605994682136814116 510,
 -0.82739605994682136814116 5095,
 -0.82739605994682136814116 50955,
 -0.82739605994682136814116 509548,
 -0.82739605994682136814116 5095480}
```

What we see is that, even for a small number of digits, results appear to be converging but as the number of digits used is increased then the numerical contribution from smaller terms in the polynomial start to contribute to the result. Eventually using enough digits we begin to converge to the correct value.

The auxiliary package **NumericalMath`ComputerArithmetic`** can be used to carry out investigations in fixed precision using alternative rounding modes when the speed of operations is not a primary concern.

■ Using interval arithmetic

An alternative computational model that rigorously encapsulates errors is *interval arithmetic*. A *real interval* is a non−empty,closed, and bounded subset of the real numbers ℝ:

$$[x] = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$$

where $\underline{x}$ and $\overline{x}$ represent the *lower* and *upper bounds* of the interval [x]. At each step of a computation the upper and lower bounds are rounded outwardly so that the interval is guaranteed to contain the result.

The following code creates and displays two intervals with machine double precision bounds. Each interval has one ulp discrepancy in the upper and lower bounds from the input to **Interval**.

```
xapprox = Interval[77617.];
```

```
InputForm[xapprox]
```

```
Interval[{77616.99999999999, 77617.00000000001}]
```

```
yapprox = Interval[33096.];
```

```
InputForm[yapprox]
```

```
Interval[{33095.99999999999, 33096.00000000001}]
```

This evaluates the polynomial of Rump using interval arithmetic.

```
RumpPolynomial[xapprox, yapprox]
```

```
Interval[{-4.36819×10^22, 4.13207×10^22}]
```

The resulting interval is very wide, which sends us a warning that we might need to rewrite the problem or use more than double precision in order to get a reasonable answer.

■ Using significance arithmetic

If we use *Mathematica*'s significance arithmetic with a **Precision** of 16, about the same number of digits as double precision, then we see that the result has no significance which is reflected by the printed zero for the mantissa. This gives us an indication that the problem is ill−conditioned.

```
RumpPolynomial[77617.`16, 33096.`16]
```

```
0.×10^22
```

Here we specify that each input number should have a **Precision** of 50.

```
rpapprox = RumpPolynomial[77617.`50, 33096.`50]
```

```
-0.82739605995
```

The result only has a **Precision** of about 12, so we see that approximately 38 digits have been lost due to propagation of errors in the problem –a result which reflects the conditioning of the problem.

```
Precision[rpapprox]
```

```
11.815
```

We can use rational arithmetic to get the exact result to compare against.

```
rpexact = RumpPolynomial[77617, 33096]
```

$$-\frac{54767}{66192}$$

By subtracting the exact value, we see that the digits that were displayed are correct.

```
rpexact - rpapprox
```

$$0.\times 10^{-12}$$

In the next section it will be shown why a loss of approximately 38 digits can be observed in this example.

■ Error propagation for the polynomial of Rump

If the input x or the function f is a vector then we can define an absolute or relative condition matrix. In practice, norms are used to represent the maximum change by a single number. Let us return to the example of Rump as illustration. For a multivariate polynomial p acting on an n–tuple x = $(x_1, \ldots, x_n)^T$, the relative condition number can be defined for $p(x) \neq 0$ as:

$$c_r(\mathrm{p}, \mathrm{x}) = \sum_{i=1}^{n} \left| \frac{x_i \, \frac{\partial p(x)}{\partial x_i}}{p(x)} \right| .$$

We can define a function to evaluate $c_r(p, x)$ for the polynomial of Rump as follows.

```
rpoly = RumpPolynomial[x, y];

rpolycond = RelativeConditionNumber[rpoly, x] +
   RelativeConditionNumber[rpoly, y];
```

We can evaluate the relative condition number at exact values and then approximate the result using the function **N**. Note that using approximate numbers as inputs would give rise to the same form of difficulty that was experienced in numerically evaluating the original polynomial.

```
N[ rpolycond /. {x → 77617, y → 33096} ]
```

$$3.82748 \times 10^{37}$$

Taking the logarithm to base 10, the resulting decrease in relative error is very close to the loss of 38 digits of **Precision** that was observed using significance arithmetic.

```
-Log[10, %]
```

$$-37.5829$$

■ A polynomial of Kulisch

The following polynomial example of Kulisch is related to the convergents in the continued fraction expansion of the quadratic irrational number $\sqrt{3}$ (see exercise 31 in section 4.2.2 of [Knuth 1998]).

This defines a function for evaluating the polynomial for real number inputs. The reason for the restriction to real number inputs will become clear later on.

```
KulischPolynomial[x_Real, y_Real] := 2 y^2 + 9 x^4 - y^4;
```

This evaluates the polynomial using double precision arithmetic.

```
KulischPolynomial[408855776., 708158977.]
```

$$-3.68935 \times 10^{19}$$

This defines a function for evaluating the polynomial of Kulisch where the second term is represented in factored form.

```
FactoredKulischPolynomial[x_, y_] :=
  2 y^2 + (3 x^2 - y^2) (3 x^2 + y^2);
```

This evaluates the factored form of the polynomial using double precision arithmetic.

```
FactoredKulischPolynomial[408855776., 708158977.]
```

$$1.00298 \times 10^{18}$$

The result is completely different from the unfactored form and does not even have the same sign.

■ Significance arithmetic

Evaluating the unfactored and factored forms of the polynomial using software numbers with **Precision** 16 we see that none of the digits are indicated as being correct.

```
KulischPolynomial[408855776.`16, 708158977.`16]
```

$0. \times 10^{20}$

```
FactoredKulischPolynomial[408855776.`16, 708158977.`16]
```

$0. \times 10^{20}$

The polynomial example of Rump illustrated how we can estimate the relative condition number of a multivariate polynomial. For the polynomial of Kulisch, the logarithm of the relative condition number shows that we should expect a loss of approximately 36.3 decimal digits.

It would be useful to have a way of automatically obtaining a result to within a prescribed error tolerance. This would be advantageous in the absence of a concrete measure for estimating the conditioning of a problem and it would also simplify the process of finding how many digits should be used in computations when the conditioning of a problem is known. The next section describes such a facility.

■ An automatic strategy to overcome cancellation

Evaluating the polynomial of Kulisch in factored form using exact arithmetic yields the following results. Intermediary computations here involve arbitrary precision integers.

```
FactoredKulischPolynomial[408855776, 708158977]
```

```
1
```

Unfortunately, using exact arithmetic can be prohibitively expensive in general. It would be useful to have a way of automatically returning a reliable result using floating point arithmetic. As we have seen *Mathematica*'s significance arithmetic has numbers that are equipped with intrinsic error estimates. It is therefore possible to use the size of errors in inputs to computations and in output from computations in an attempt to get an output within a specified error tolerance. The function **N** can be used to accomplish this.

Setting the following attribute allows **N** to inherit the knowledge that the function **KulischPolynomial** takes inexact numbers as inputs and returns an inexact number for the result.

```
SetAttributes[KulischPolynomial, NumericFunction];
```

In this example **N** converts exact numbers to have an initial **Precision** and then examines the **Precision** of the result. If the **Precision** of the result is deficient, linear extrapolation is used to augment the initial **Precision** and the computation is then repeated. The restriction of the definition of the function **KulischPolynomial** to work with real numbers ensures that the input does not evaluate using exact arithmetic.

```
N[KulischPolynomial[408855776, 708158977], 20]
```

```
1.0000000000000000000
```

■ Informal verification

In high level systems like *Mathematica* there is more than one way of obtaining a result to a given computation, so that we can use the system introspectively to validate results. Using the analysis of linearized error propagation for functions we can define some routines to compute error propagation quantities from their mathematical definitions. Then we can compare results with the built in functions and examine how closely the implementation matches the theory.

Here we define a routine which finds the **Accuracy** of the result of a unary function f at a given point x.

```
PropagateAccuracy[f_, x_] := Accuracy[x] - Scale[f'[x]];
```

This defines a real valued number with **Precision** 20.

```
val = SetPrecision[2 10^-10, 20]
```

$$2.0000000000000000000 \times 10^{-10}$$

This invokes the function **PropagateAccuracy** for the function **Exp** at the previous value.

```
PropagateAccuracy[Exp, val]
```

```
29.699
```

The result is the same as that of the built−infunction.

```
Accuracy[ Exp[val]]
```

```
29.699
```

We now define a routine which finds the **Precision** of the result of a unary function at a given point.

```
PropagatePrecision[f_, x_] :=
    PropagateAccuracy[f, x] + Scale[f[x]];
```

Here we see that the **Precision** of the result of applying **Sqrt** to the number assigned to **val** is augmented from 20 by one binary bit, or $\log_{10}(2) \approx 0.301$.

```
PropagatePrecision[Sqrt, val]
```

```
20.301
```

The reason for the increase in **Precision** can be seen by considering the (logarithm of) the relative condition number for $f(x) = \sqrt{x}$. The result is independent of the actual value of x, so that we can use a symbol **x** as input.

```
-Log[10, RelativeConditionNumber[√x, x]]
```

$$\frac{\text{Log}[2]}{\text{Log}[10]}$$

The result of the function **PropagatePrecision** is the same as the **Precision** of the built−in function for computing **Sqrt**.

```
Precision[ √val ]
```

```
20.301
```

Conclusions

There are alternative arithmetic models to significance arithmetic whose purpose is to accurately track the propagation of errors. One approach is interval arithmetic which we have briefly discussed and is described in more detail in [Hammer 1995] and [Rump 1999]. Although *Mathematica* currently only implements real−valued intervals, interval bounds can be specified to arbitrary precision. For an interesting application of interval arithmetic to verified solutions for ordinary differential equations see [Corliss 1998]. Another interesting project is MPFR (Multiple Precision Floating point Reliable [MPFR 2000]). We have shown how *Mathematica*'s implementation of significance arithmetic varies the number of digits used in computations according to the conditioning of a function. In contrast MPFR endeavours to give a fixed precision result that is completely accurate, which may involve internally raising the number of digits used and redoing a computation. Another interesting feature of MPFR is that it provides several modes for directional rounding. Other recent work in the study of automatic error analysis is given in [Rowan 1990] and a recent description of some precise numerical methods can be found in [Aberth 1998].

There is no substitute for traditional methods of numerical analysis, such as forward and backward analysis which provide tangible error estimates. These tools provide the necessary insight that is required in a thorough investigation of a problem (see for example [Chaitin 1996] and [Higham 1996]). However, we believe that significance arithmetic is a useful addition to the repertoire of available tools for the numerical investigation of problems. We also believe that the model provides a useful aid for users who are not experts in the analysis and construction of numerical methods but are interested in investigating and solving problems, often against the industrial backdrop of pressing deadlines. It is our hope that the examples shown here provide some justification for these claims. Some additional details and examples of *Mathematica*'s implementation of significance arithmetic and numerical functions are given in [Keiper 1992] and [Skeel 1993]. Additional information relevant to significance arithmetic can be found in [Hunter 1975], [Metropolis 1973], [Metropolis 1974], [Metropolis 1977b], [Tanny 1973], [Yap 1997].

As we have seen, the use of increasing precision in a fixed precision arithmetic system should be used with caution, a point which is summed up in the following cautionary extract [Higham 1996]:

> When the only source of errors is rounding, a common technique for estimating the accuracy of an answer is to recompute it at a higher precision and to see how many digits of the original and the (presumably) more accurate answer agree. We would intuitively expect any desired accuracy to be achievable by computing at a high enough precision. This is certainly the case for algorithms possessing an error bound proportional to the precision, which includes all the algorithms described in subsequent chapters of this book. However, since an error bound is not necessarily attained, there is no guarantee that a result computed in t digit precision will be more accurate than one computed in s digit precision, for a given t > s; in particular, for a very ill conditioned problem both results could have no correct digits.

Significance arithmetic is applicable to a wider range of problems than interval arithmetic and works reasonably well when the assumptions of the model are justified. Numerous simplifications and operations involving exact quantities in *Mathematica* are actually resolved numerically, such as comparisons like $\pi > 22/7$ (see for example [Sofroniou 1996]). Such applications would not be possible if some form of error certification were not used. Another application of significance arithmetic in *Mathematica* is the package **NumericalMath`Microscope`** which is useful for verifying the correctness of machine arithmetic libraries in modern compilers.

The choice of significance arithmetic in *Mathematica* has not been entirely favorable (see for example [Fateman 1992]) although some of these criticisms relate to early deficiencies in the implementation. The tools for verifying results that have been presented here are now used by the quality assurance department at Wolfram Research as a routine control in the development of *Mathematica*. It is our hope that this will improve the reliability of the program and placate the criticism and concerns relating to the implementation. Further criticism, somewhat justifiably, has centered on the lack of formal documentation explaining how significance has been implemented and we hope that the description provided here goes some way towards addressing this aspect.

It is an open question as to how generically useful the assumption of independence of errors in significance arithmetic is and whether corrective steps to better reflect the correlation of errors can be carried out. Indeed *Mathematica* uses fixed precision arithmetic in its large scale numerical routines such as in linear algebra and the numerical solution of differential equations; the error bounds provided by these numerical methods are well studied and provide much tighter bounds than those based on random assumptions of error accumulation. The issue of error correlation also arises in interval arithmetic where it gives rise to what is sometimes referred to as the *dependency problem* [Hansen 1992].

There is some inevitable overhead involved in the tracking of errors that is associated with significance arithmetic. However, by embedding run time error bounds in the arithmetic itself, the user is freed from maintaining manual error bounds and it can become possible to carry out reliable computations much more efficiently than was previously considered possible. Unlike interval arithmetic, the error bounds in significance arithmetic are approximate. As long as error intervals are kept sufficiently small, the model can still be used to accurately track errors. At least some computational processes fit the assumptions of significance arithmetic very well and as evidence of this the authors have recently developed the most efficient known implementation for computing continued fractions of real numbers [Sofroniou 2000].

Acknowledgements

Part of the description given here, in particular the historical implementation, has been expanded from unpublished notes of Jerry B. Keiper. Thanks to Nicholas J. Higham for many useful suggestions in improving the presentation of the work presented here, Paul Zimmermann for bringing the MPFR project to our attention and also to Jean Buck for help in tracking down some of the references to significance arithmetic.

References

[Aberth 1998] O. Aberth, *Precise Numerical Methods Using C++*, Academic Press, San Diego, 1998.

[Ashenhurst 1965] R. L. Ashenhurst, Techniques for automatic error monitoring and control, in Error in Digital Computation, Vol 1, L. B. Rall (ed), Wiley, New York, pp. 43—59, 1965.

[Bevilacqua 92] R. Bevilacqua, D. Bini, M. Capovani and O. Menchi, *Metodi Numerici*, Zanichelli, Bologna, 1992.

[Chaitin 96] F. Chaitin–Chatelinand V. Fraysse', *Lectures on Finite Precision Computations*, Series on Software, Environment and Tools, SIAM, Philadelphia, 1996.

[Corliss 1998] G. F. Corliss, *Guaranteed error bounds for ODEs*, in Theory and Numerics of Ordinary and Partial Differential Equations, Advances in Numerical Analysis, Vol. IV, Ainsworth, Levesley, Light and Marletta eds., Oxford Science Publications, Clarendon Press, Oxford, pp. 1—75, 1995.

[Gray 1959] H. L. Gray and C. Harrison, Normalised floating point arithmetic with an index of significance, In Proceedings of the Eastern Joint Computer Conference 16 (Boston, Mass. Dec 1—3, 1959), pp. 244—248.

[Fateman 1992] R.J. Fateman, A review of Mathematica. J. Symb. Comp. 13, pp. 545—579, 1992.

[Hairer 1995] E. Hairer and G. Wanner, *Analysis by Its History*, Springer, New York, 1995.

[Hammer 1995] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *C++ Toolbox for Verified Computing. Basic Numerical Problems*, Springer, Berlin, 1995.

[Hansen 1992] E. Hansen, Global optimization using interval analysis, Marcel Dekker Inc., New York, 1992.

[Higham 1996] N. J. Higham, Accuracy and stability of numerical algorithms. SIAM, Philadelphia, 1996.

[Hunter 1975] G. Hunter, A quantitative measure of precision, Comput. J. Vol. 18 No. 3, pp. 231—233, August 1975.

[Jones 1984] C. B. Jones, A significance rule for multiple precision arithmetic, A. C. M. Transactions on Mathematical Software, Vol. 10 No. 1, pp. 97—107, March 1984.

[Keiper 1992] J. B. Keiper and D. Withoff, Numerical computation in *Mathematica*, in Course notes of the *Mathematica* conference, 1992.

[Keiper 1995] J. B. Keiper, Interval arithmetic in *Mathematica*. The Mathematica Journal, Vol. 5 No. 2: pp. 66—71, 1995. Reprinted from the proceedings of the International Conference on Numerical Analysis with Automatic Result Verification, Interval Computations 3, 1993.

[Knuth 1998] D.E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical algorithms*, 3rd edition, Addison–Wesley, London, 1998.

[Metropolis 1958] N. Metropolis and R. L. Ashenhurst, Significant digit computer arithmetic, IRE Trans. Electron. Comput., EC–7, pp. 265—267, 1958.

[Metropolis 1959] N. Metropolis and R. L. Ashenhurst, Unnormalised floating point arithmetic, J. ACM, Vol. 6 No. 3, pp. 415—428, July 1959.

[Metropolis 1973] N. Metropolis, G. C. Rota and S. M. Tanny, Significance arithmetic: the carrying algorithm, Journal of Combinatorial Theory, (A) Vol. 14 No. 3, pp. 386—421, 1973.

[Metropolis 1974] N. Metropolis and G. C. Rota, Significance arithmetic – on the algebra of binary strings, in Collection: Studies in numerical analysis (papers in honour of Cornelius Lanczos on the occasion of his 80th birthday), Academic Press, London, pp. 241—251, 1974.

[Metropolis 1977a] N. Metropolis, Methods of significance arithmetic, in The state of the art in numerical analysis (Proc. Conf., Univ. York, Heslington, 1976), Academic Press, London, pp. 179—192, 1977.

[Metropolis 1997b] N. Metropolis and S. M. Tanny, Significance arithmetic: the probability of carrying, Computers and Mathematics with Applications, Vol. 3 No. 1, pp. 77—81, 1977.

[MPFR 2000] Details of the Multiple Precision Floating point Reliable project can be found at: http://www.loria.fr/projets/mpfr

[Plauger 1992] P. J. Plauger, *The Standard C Library*, Prentice–Hall,Englewood Cliffs, New Jersey, 1992.

[Rowan 1990] T. H. Rowan, *Functional Stability Analysis of Numerical Algorithms*, Ph. D. Thesis, University of Texas at Austin, Austin, TX, USA, 1990.

[Rump 1988] S. M. Rump, Algorithm for verified inclusions – theory and practice, in *Reliability in Computing*, R. E. Moore ed., Academic press, San Diego, pp. 109—126, 1988.

[Rump 1999] S. M. Rump, INTerval LABoratory, 1999. The INTLAB home page is located at http://www.ti3.tu–harburg.de/~rump/intlab/index.html

[Santoro 1989] M. R. Santoro, G. Bewick and M. A. Horowitz, Rounding algorithms for IEEE multipliers. In Milos D. Ercegovac and Earl Swartzlander, eds., Proceedings 9th Symposium on Computer Arithmetic, pp. 176—183. IEEE Computer Society Press, 1989.

[Skeel 1993] R. D. Skeel and J. B. Keiper, Elementary numerical computing with *Mathematica,* McGraw–Hill,New York, 1993.

[Sofroniou 1996] M. Sofroniou, Numerics in *Mathematica* 3.0, The *Mathematica* Journal, Vol. 6 No. 4. pp. 64—73. 1996.

[Sofroniou 2000] M. Sofroniou and G. Spaletta, Efficient computation of continued fractions (to appear).

[Sterbenz 1974] P. H. Sterbenz, *Floating point computation*, Prentice Hall, Englewood Cliffs, NJ, USA, 1974.

[Tanny 1973] S. M. Tanny, Studies in significance arithmetic, Doctoral dissertation, M. I. T., Jan. 1973.

[Yap 1997] C. Yap and K. Ouchi, The Real/Expr package homepage is located at http://simulation.nyu.edu/projects/exact/