The Mathematics of Floating-Point Arithmetic

NICHOLAS J. HIGHAM

Floating-point arithmetic is ubiquitous in computing and its implementation in evolving computer hardware remains an active area of research. Its mathematical properties differ from those of arithmetic over the real numbers in important and sometimes surprising ways. We explain what a mathematician should know about floating-point arithmetic, and in particular we describe some of its not so well known algebraic properties.

Floating-point arithmetic has been in use for over seventy years, having been provided on some of the earliest digital computers. For the first half of that period there was tremendous variation in floating-point formats and in the ways the arithmetics were implemented. Some floating-point arithmetics could produce anomalous results and it was difficult or impossible to write programs that were portable, i.e., produce similar results on different computer systems with little or no change.

The 1985 ANSI/IEEE Standard for Binary Floating-Point Arithmetic [9] provided binary floating-point formats and precise rules for how to carry out arithmetic on them. Carefully designed over several years by a committee of experts, it brought much-needed order to computer arithmetic and within a few years virtually all computer manufacturers had adopted it.

From a mathematical perspective we can ask several questions about a floating-point arithmetic.

- What mathematical properties does it have compared with exact arithmetic?
- What sort of mathematical structure is it?
- How can we understand the accuracy of computations carried out in it?

First, we need to define the set of numbers under consideration. A floating-point number system is a finite subset $F = F(\beta, t, e_{\min}, e_{\max})$ of the real numbers $\mathbb{R}$ whose elements have the form

$$x = \pm m \times \beta^{e-t+1}. \tag{1}$$

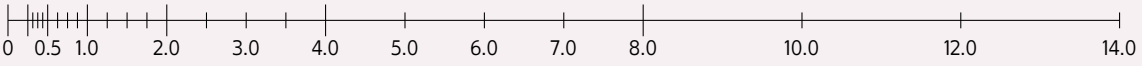
Here, β is the base, which is 2 on virtually all current computers. The integer t is the precision and the integer e is the exponent, which lies in the range $e_{\min} \le e \le e_{\max}$. The significand m is an integer satisfying $0 \le m \le \beta^t - 1$. To ensure a unique representation for each nonzero $x \in F$ it is assumed that $m \ge \beta^{t-1}$ if $x \ne 0$, so that the system is *normalised*.

The reason for the "+1" in the exponent of (1), which could be avoided by redefining $e_{\min}$ and $e_{\max}$, is for consistency with the IEEE standard. The standard also requires that $e_{\min} = 1 - e_{\max}$.

The largest and smallest positive numbers in the system are $x_{\max} = \beta^{e_{\max}}(\beta - \beta^{1-t})$ and $x_{\min} = \beta^{e_{\min}}$, respectively. Two other important quantities are $u = \frac{1}{2}\beta^{1-t}$, the *unit roundoff*, and $\epsilon = \beta^{1-t}$, the *machine epsilon*, which is the distance from 1 to the next larger floating-point number. See the box for a simple example of a floating-point number system.

A toy floating-point number system

This diagram shows the nonnegative (normalised) numbers in a binary floating-point number system with $t = 3$, $e_{\min} = -2$, and $e_{\max} = 3$. Note that the floating-point numbers are equally spaced between powers of 2 and the spacing increases by a factor of 2 at each power of 2. Here, the unit roundoff is $u = 0.125$ and the machine epsilon $\epsilon = 0.25$; these are the distances from 1 to the next smaller and next larger floating-point number, respectively.



The system F can be extended by including *subnormal numbers*, which have the minimum exponent and $m < \beta^{t-1}$, so that they are not normalised; they fill the gap between 0 and $x_{\min}$ with numbers having a constant spacing $\beta^{e_{\min}+1-t}$. The IEEE standard includes subnormal numbers.

We assume throughout the rest of this article that F is a binary system ($\beta = 2$), and that it follows the IEEE standard by including the special numbers $\pm\infty$ and NaN (Not a Number). The numbers $\pm\infty$ obey the usual mathematical conventions regarding infinity, such as $\infty + \infty = \infty$, $(-1) \times \infty = -\infty$, and $(\text{finite})/\infty = 0$. A NaN is generated by operations such as $0/0$, $0 \times \infty$, ∞/∞, $(+\infty) + (-\infty)$, and $\sqrt{-1}$.

We also assume, again following the IEEE standard, that the results of the elementary operations of addition, subtraction, multiplication, division, and square root are the same as if they were carried out to infinite precision and then rounded back to F, and that rounding of $x \in \mathbb{R}$ to F is done by mapping to the nearest floating-point number, with ties broken by rounding to the floating-point number with a zero last bit. We denote the operation of rounding by fl. With a standard abuse of notation, $\mathrm{fl}(\mathrm{expr})$, where expr is an arithmetic expression, is also used to denote the result of evaluating expr in floating-point arithmetic in some specified order.

With the inclusion of ∞ and NaN, F is a closed number system: every floating-point operation on numbers in F produces a result in F.

Algebraic properties

The real numbers form a field under addition and multiplication. It is natural to ask what sort of mathematical structure floating-point numbers form under the elementary (floating-point) arithmetic operations. To investigate this question we will explore some basic algebraic properties of floating-point arithmetic.

Let $a, b \in F$. By definition, $\mathrm{fl}(a + b)$ and $\mathrm{fl}(b + a)$ are equal, as are $\mathrm{fl}(a * b)$ and $\mathrm{fl}(b * a)$. However, with three numbers the usual rules of arithmetic break down: $\mathrm{fl}((a + b) + c))$ is not necessarily equal to $\mathrm{fl}(a + (b + c))$ and $\mathrm{fl}((a * b) * c)$ is not necessarily equal to $\mathrm{fl}(a * (b * c))$. In other words, floating-point addition and multiplication are not associative. For example, in our toy system $\mathrm{fl}(0.25 + (8.0 - 7.0)) = 1.25$ but $\mathrm{fl}((0.25 + 8.0) - 7.0) = \mathrm{fl}(8.0 - 7.0) = 1.0$. Similarly, $\mathrm{fl}(a * (b + c))$ is not necessarily equal to $\mathrm{fl}(a * b + a * c)$, so the distributive law does not hold.

If $a > b > 0$ then $\mathrm{fl}(a + b) > a$ need not hold. The reason is that b may be so small that a is unchanged after adding b and rounding. Indeed $\mathrm{fl}(1 + x) = 1$ for any positive floating-point number $x < u$.

Does the equation $x * (1/x) = 1$ hold in floating-point arithmetic? The following result of Edelman says that it may just fail to do so [6, Prob. 2.12].[1]

Theorem 2. *For $1 < x < 2$, $\mathrm{fl}(x * (1/x))$ is either 1 or $1 - \epsilon/2$*

A closely related question is which floating-point numbers are possible reciprocals of $x \in F$. Muller [10] showed that when $1/x \notin F$ there are two possibilities.

Theorem 3. *The only $z \in F$ that can satisfy $\mathrm{fl}(x * z) = 1$ are $\min\{ y : y \geq 1/x, y \in F \}$ and $\max\{ y : y \leq 1/x, y \in F \}$.*

Perhaps surprisingly, these two possible z can simultaneously give equality, so a floating-point number can have two floating-point reciprocals. In fact, of the 24 positive numbers in the toy system, eight have two floating-point reciprocals; for example, $y = 0.625$ and $y = 0.75$ both satisfy $\mathrm{fl}(1.5 * y) = 1$, and these are the two nearest floating-point numbers to $1/1.5 = 2/3$.

Now consider the computation $n * (m/n)$, where m and n are integers. If m/n is a floating-point number then $\mathrm{fl}(n * \mathrm{fl}(m/n)) = \mathrm{fl}(n * (m/n)) = \mathrm{fl}(m) = m$, as no rounding is needed. Kahan proved that the same identity holds for many other choices of m and n [4, Thm. 7].

Theorem 4. *Let m and n be integers such that $|m| < 2^{t-1}$ and $n = 2^i + 2^j$ for some i and j. Then $\mathrm{fl}(n * \mathrm{fl}(m/n)) = m$.*

The sequence of allowable n begins $2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 17, 18, 20$ (and is A048645 in the On-Line Encyclopedia of Integer Sequences), so Theorem 3 covers many common cases. Nevertheless, the equality does not hold in general.

It can be shown that $\mathrm{fl}\left(\sqrt{x^2}\right) = |x|$ for $x \in F$, as long as x^2 does not underflow (round to zero) or overflow

[1] We give a minimal set of references in this article. Original sources can be found in the references cited.

(exceed the largest element of F), but $\mathrm{fl}((\sqrt{x})^2) = |x|$ is not always true (by the pigeonhole principle) [6, Prob. 2.20].

Rounding (to nearest) is monotonic in that for $x \in \mathbb{R}$ and $y \in \mathbb{R}$, the inequality $x \le y$ implies $\mathrm{fl}(x) \le \mathrm{fl}(y)$. As a result, it is easy to show that

$$x \le \mathrm{fl}\left(\frac{x+y}{2}\right) \le y.$$

While this result holds for base 2, the computed midpoint can be outside the interval for base 10.

The inequality $\mathrm{fl}\left(x/\sqrt{x^2+y^2}\right) \le 1$ always holds, barring overflow and underflow [6, Prob. 2.21]. Although this fact may seem unremarkable, in some pre-IEEE standard arithmetics this inequality could be violated, causing failure of an attempt to compute one of the angles in a right-angled triangle with shortest sides of lengths x and y as $\mathrm{acos}\left(x/\sqrt{x^2+y^2}\right)$.

The following result of Sterbenz guarantees that subtraction is exact for two numbers that are at most a factor 2 apart.

Theorem 5. *If x and y are floating-point numbers with $y/2 \le x \le 2y$ then $\mathrm{fl}(x-y) = x-y$ (assuming $x-y$ does not underflow).*

This result is notable because inaccurate results are often blamed on subtractive cancellation. It is not the subtraction itself that is dangerous but the way it brings into prominence errors already present in the numbers being subtracted, making these errors much larger relative to the result than they were to the arguments.

Finally, we note that a NaN is unique among elements of F in that it compares as unordered (including unequal to) everything, including itself. In particular, a statement "if x = x" returns false when x is a NaN. This is why some programming languages provide a function to test for a NaN (e.g., isnan in MATLAB).

We conclude that floating-point arithmetic is a rather strange mathematical object that does not correspond to any standard algebraic structure. These examples could make one pessimistic about our ability to carry out reliable numerical computations. Fortunately, these peculiar features of floating-point arithmetic are not a barrier to its successful use or to deriving satisfactory error bounds, as we now illustrate.

Experimenting with different floating-point arithmetics

It is instructive to run experiments in floating-point arithmetics based on different parameters t, $e_{\min}$, and $e_{\max}$.

We used the MATLAB function `chop`[a] [8] for this purpose. This function rounds single or double precision numbers to a specified target format (limited to $e_{\min} = 1 - e_{\max}$) and supports several rounding modes and other options. A library CPFloat offers similar functionality for C [3].

[a] `https://github.com/higham/chop`

Error analysis

If we want to understand the effects of rounding errors on a floating-point computation then we need to analyse how the individual rounding errors interact and propagate. A natural way to try to do this is to define "circle operators" $\oplus$, $\ominus$, $\otimes$, and $\oslash$ by

$$x \oplus y = \mathrm{fl}(x+y), \quad x \ominus y = \mathrm{fl}(x-y),$$
$$x \otimes y = \mathrm{fl}(x*y), \quad x \oslash y = \mathrm{fl}(x/y),$$

and then rewrite the expressions being evaluated in terms of these operators. For example, consider the evaluation of the cubic polynomial $p = ax^3 + bx^2 + cx + d$ by Horner's rule as $p = ((ax+b)x+c)x+d$ (using 6 operations instead of the 8 required if we explicitly form x^3 and x^2). We would then write the computed $\widehat{p}$ as

$$\widehat{p} = \big((a \otimes x \oplus b) \otimes x \oplus c\big) \otimes x \oplus d.$$

However, we cannot easily simplify this expression because the circle operators do not satisfy the associative or distributive laws.

The right way to do error analysis is to obtain equations in terms of the original operators and individual rounding errors. We need the result that [6, Thm. 2.2]

$$x \in \mathbb{R} \implies \mathrm{fl}(x) = x(1+\delta), \quad |\delta| \le u, \qquad (2)$$

where u is the unit roundoff. Since $\mathrm{fl}(x \mathrm{\ op\ } y)$ is defined to be the rounded exact value, it follows that for $\mathrm{op} = +, -, *, /$ we have

$$\mathrm{fl}(x \mathrm{\ op\ } y) = (x \mathrm{\ op\ } y)(1+\delta), \quad |\delta| \le u. \qquad (3)$$

This is the standard model of floating-point arithmetic used for rounding error analysis. Note that it does not fully characterise floating-point arithmetic because (2) does not fully characterise rounding: for some x, two different floating-point numbers y satisfy $y = x(1+\delta)$ with $|\delta| \le u$. It is possible to use more refined models of floating-point arithmetic that more fully reflect the definition of rounding, which tends to give results with slightly smaller constants at the cost of a much more complicated analysis. The main purpose of a rounding error analysis, though, is to gain insight into accuracy and stability rather than to optimise constants.

For our cubic example, we can use the model to write

$$\begin{aligned}\widehat{p} &= \Big(\Big((ax(1+\delta_1)+b)(1+\delta_2)x(1+\delta_3)+c\Big)\\ &\qquad \times (1+\delta_4)x(1+\delta_5)+d\Big)(1+\delta_6)\\ &= ax^3(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4)(1+\delta_5)(1+\delta_6)\\ &\quad + bx^2(1+\delta_2)(1+\delta_3)(1+\delta_4)(1+\delta_5)(1+\delta_6)\\ &\quad + cx(1+\delta_4)(1+\delta_5)(1+\delta_6) + d(1+\delta_6),\end{aligned}$$

where $|\delta_i| \le u$ for all i. This expression is rather messy, but we can rewrite it as

$$\begin{aligned}\widehat{p} &= ax^3(1+\theta_6) + bx^2(1+\theta_5)\\ &\quad + cx(1+\theta_3) + d(1+\theta_1),\end{aligned} \tag{4}$$

where the θ_i are bounded by the following lemma [6, Lem. 3.1].

Lemma 1. *If $|\delta_i| \le u$ and $\rho_i = \pm 1$ for $i = 1\colon n$, and $nu < 1$, then*

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

where

$$|\theta_n| \le \frac{nu}{1-nu} =: \gamma_n.$$

Applying the lemma to (4), we obtain

$$|p - \widehat{p}| \le \gamma_6(|a||x|^3 + |b||x|^2 + |c||x| + |d|), \tag{5}$$

which is a concise and easily interpretable error bound, with constant $\gamma_6 = 6u + O(u^2)$.

With careful use of the lemma, the profusion of $1+\delta_i$ terms that arise in a rounding error analysis can be kept under control and manipulated, using the usual rules of arithmetic, into a useful bound.

Fused multiply-add operation

Since the 1990s some processors have provided a *fused multiply-add* (FMA) operation that computes $x+y*z$ with just one rounding error instead of two, so that

$$\mathrm{fl}(x + y * z) = (x + y * z)(1+\delta), \quad |\delta| \le u.$$

The motivation for an FMA is speed, as it is implemented in such a way as to take the same time as a single multiplication or addition.

When an FMA is used the number of rounding errors in a typical computation is halved. Our cubic polynomial can be evaluated with three FMAs, giving

$$\begin{aligned}\widehat{p} &= \Big(\Big((ax+b)(1+\delta_1)x + c\Big)\\ &\qquad \times (1+\delta_2)x + d\Big)(1+\delta_3)\\ &= (ax^3 + bx^2)(1+\theta_3) + cx(1+\theta_2)\\ &\quad + d(1+\theta_1),\end{aligned}$$

which is more favourable than (4).

Although it generally brings improved accuracy, an FMA can also lead to some unexpected results.

If we compute the modulus squared of a complex number from the formula

$$(x+\mathrm{i}y)^*(x+\mathrm{i}y) = x^2 + y^2 + \mathrm{i}(xy - yx)$$

then the result is real, because $\mathrm{fl}(xy) = \mathrm{fl}(yx)$. But if an FMA is used in evaluating $xy-yx$ then the imaginary part may evaluate as nonzero.

Similarly, if the discriminant $b^2 - 4ac$ of a quadratic is nonnegative then the computed result is guaranteed to be nonnegative by the monotonicity of floating-point arithmetic, but with an FMA the result can be negative.

Error analysis strategy

Even with the use of Lemma 1, rounding error analysis can be tedious, and it is natural to ask whether it can be automated. Can we harness a computer to carry

out the necessary manipulations? For focused classes of algorithms some progress has been made [1], but in general the task is difficult or impossible to automate. The reason is that the hardest part of an error analysis is deciding what one wants to prove. For the evaluation of the cubic we obtained a bound (5) on the error in the computed p, known as the forward error. Along the way we obtained (4), which is a backward error result: it shows that the computed $\widehat{p}$ is the exact result for a polynomial with perturbed coefficients $a(1+\theta_6)$, $b(1+\theta_5)$, $c(1+\theta_3)$, and $d(1+\theta_1)$, and it bounds the size of the relative perturbations by γ_6.

In general, for a computation $y = f(x)$, where x and y are vectors (say), we have three measures of error for the computed $\widehat{y}$:

- forward error: $\|y - \widehat{y}\|/\|y\|$,
- backward error:

$$\min\left\{ \frac{\|\Delta x\|}{\|x\|} : \widehat{y} = f(x + \Delta x) \right\},$$

- mixed backward–forward error: the smallest ϵ for which there exist Δx and Δy such that

$$\widehat{y} + \Delta y = f(x + \Delta x), \quad \frac{\|\Delta x\|}{\|x\|} \le \epsilon, \quad \frac{\|\Delta y\|}{\|\widehat{y}\|} \le \epsilon.$$

Depending on the problem, any one of these errors may be the best one to bound in a rounding error analysis, or perhaps the only one that it is feasible to bound. Determining the right approach and working out how to achieve a result that is readable, understandable, and insightful can be difficult.

Backward error analysis was developed by J. H. Wilkinson in the 1950s and 1960s [5]. It has the attractive feature of decoupling the numerical stability properties of an algorithm from the conditioning of the underlying problem (its sensitivity to perturbations in the data).

What is quite remarkable is that despite the strange behaviour of floating-point arithmetic illustrated above, it is possible to carry out rounding error analysis of a wide variety of algorithms and obtain useful results.

Recovering the error

The sum $s = a + b$ of $a, b \in F$ is not in general in F, so the computed sum $\widehat{s} = \mathrm{fl}(a + b)$ may be inexact.

However, the error $e = s - \widehat{s}$ is in F, and for $|a| \ge |b|$ it can be computed (exactly) as [11, Sec. 4.3.1]

$$e = b - (\widehat{s} - a),$$

so that

$$a + b = \widehat{s} + e.$$

This computation is known as Fast2Sum. Let us denote it by $[s, e] = \mathrm{Fast2Sum}(a, b)$. (There are other, more complicated, ways of computing e that do not require $|a| \ge |b|$.)

Of course, if we try to form $\mathrm{fl}(\widehat{s} + e)$ then we will just obtain $\widehat{s}$, because $\widehat{s}$ is the best floating-point representation of $a+b$. However, in a sequence of operations we can add the error from an earlier operation into a later operation, where it can potentially have an effect.

An important usage of Fast2Sum is in *compensated summation*, proposed by Kahan in 1965, which computes $\sum_{i=1}^{n} x_i$ by

```
1  s = x_1
2  e = 0
3  for i = 2: n
4      t = x_i + e
5      [s, e] = Fast2Sum(s, t)
6  end
```

For standard recursive summation the computed $\widehat{s}$ satisfies

$$|s - \widehat{s}| \le c_n u \sum_{i=1}^{n} |x_i| + O(u^2)$$

with $c_n = n - 1$, whereas the computed $\widehat{s}$ from compensated summation satisfies the same bound with $c_n = 2$ (even though compensated summation does not sort the arguments of Fast2Sum). For large n, this reduction in the constant makes a significant difference.

The 2019 revision of the IEEE standard includes so-called augmented arithmetic operations for addition, subtraction, and multiplication, which (like Fast2Sum) return both the computed result and the error in it.

Probabilistic analysis and stochastic rounding

A numerical computation with $n \times n$ matrices usually has a rounding error bound proportional to $c_n u$ with c_n growing at least linearly. Traditionally, numerical computations have been done in single precision arithmetic or double precision arithmetic, with unit roundoffs u of order 10^{-8} or 10^{-16}, respectively. Hence $nu \ll 1$ for practical problems.

However, half precision arithmetic is now increasingly available in hardware, with u of order 10^{-3} for the bfloat16 format and 10^{-4} for the IEEE half precision format. In these arithmetics, $nu = 1$ for quite modestly-sized problems, and in these cases an error bound proportional to nu provides no information.

Mixed-precision algorithms

It is becoming common for computer systems to offer half precision, single precision, and double precision floating-point arithmetics in hardware, possibly with quadruple precision arithmetic in software. In designing algorithms we wish to exploit the speed of execution of lower precision arithmetic while ensuring that enough higher precision is used to deliver a result of the desired accuracy. Rounding error analysis, parametrised by the unit roundoffs for the different precisions, helps to identify suitable algorithms.

Traditional rounding error bounds, such as those above for Horner's rule, are worst-case bounds. As Stewart observes [12], "To be realistic, we must prune away the unlikely. What is left is necessarily a probabilistic statement." The idea of obtaining probabilistic rounding error bounds by modelling rounding errors as random variables is not new, but a rigorous treatment producing bounds valid for any dimension has only recently been developed, by Connolly, Higham, and Mary [2], [7]. This analysis proves that under the assumption that the rounding errors are mean independent random variables of mean zero, error bounds with constants $\sqrt{f(n)}u$ hold with high probability in place of worst-case bounds $f(n)u$.

A form of rounding called *stochastic rounding* has recently been finding use in deep learning and other areas. It rounds a number lying between two adjacent floating-point numbers $a < b$ to a with a probability proportional to the distance to b, and conversely for b. Stochastic rounding is somewhat worse behaved than round to nearest vis-à-vis its algebraic properties for individual operations. However, the random nature of the rounding is beneficial. It can be shown [2] that the rounding errors from stochastic rounding are random variables satisfying both the mean independence and the mean zero assumptions, so that the $\sqrt{f(n)}u$ bounds hold unconditionally. This means that stochastic rounding can provide more accurate results than round to nearest for large problems.

As a simple example, we computed $\sum_{i=1}^{10^4} x_i$ in IEEE half precision arithmetic, where x_i is $1/i$ rounded to half precision with round to nearest. The sum computed with round to nearest had relative error 2.7×10^{-1}, whereas the minimum, mean, and maximum errors over ten sums computed with stochastic rounding were 2.2×10^{-3}, 1.2×10^{-2}, 3.0×10^{-2}, respectively. In this example, round to nearest suffers from stagnation, whereby the smallest terms cannot change the computed partial sum. By contrast, stochastic rounding gives all terms a nonzero probability of increasing the sum, and in fact it does so in just the right way to ensure that the expected value of the computed sum is the exact sum [2].

Outlook

The provision of half precision floating-point arithmetic in hardware is motivated by machine learning, where its greater speed is proving beneficial despite its lower accuracy. Half precision can also be exploited in general scientific computing, but rounding error analysis is needed to determine whether sufficiently accurate results are being computed.

An example of how half precision arithmetic can be harnessed to great effect is the HPL-AI Mixed Precision Benchmark[2], which is one of the benchmarks that the TOP500 project uses to rank the world's most powerful supercomputers. This benchmark solves a double precision nonsingular linear system $Ax = b$ of order n using an LU factorisation computed in half precision and it refines the solution using iterative refinement in double precision. As of November 2020, the world record

[2] icl.bitbucket.io/hpl-ai

execution rate for the benchmark is 2.0 ExaFlop/s (2×10^{18} floating-point operations per second, where most of the operations are half precision ones) for a matrix of size $16,957,440$, which was achieved by the Fugaku supercomputer in Japan. For a successful benchmark run, the relative residual $\|A\widehat{x} - b\|/(\|A\|\|\widehat{x}\| + \|b\|)$ of the computed $\widehat{x}$ must be no larger than a threshold that is about 10^{-8} in this case. So after approximately $2n^3/3 \approx 3 \times 10^{21}$ floating-point operations, Fugaku's computed solution $\widehat{x}$ had a small residual, which is a testament to effectiveness of floating-point arithmetic given that each half precision operation has a relative error of order 10^{-4}.

Despite floating-point arithmetic having some strange mathematical properties, seventy years of experience show that it usually works well in practice, and it is supported by rigorous mathematical analysis—both worst-case and probabilistic. With hardware implementations of floating-point arithmetic evolving constantly and new algorithms regularly being developed, interesting mathematical questions will continue to arise over the coming years.

Acknowledgements

I thank Michael Connolly, Massimilano Fasi, Sven Hammarling, Theo Mary, Mantas Mikaitis, and Srikara Pranesh for suggesting improvements to a draft of this article. This work was supported by the Royal Society and Engineering and Physical Sciences Research Council grant EP/P020720/1.

FURTHER READING

[1] P. Bientinesi and R.A. van de Geijn. Goal-oriented and modular stability analysis. *SIAM J. Matrix Anal. Appl.* 32 (2011) 286–308.

[2] M.P. Connolly, N.J. Higham, and T. Mary. Stochastic rounding and its probabilistic backward error analysis. *SIAM J. Sci. Comput.*, 2021. To appear.

[3] M. Fasi and M. Mikaitis. CPFloat: a C library for emulating low-precision arithmetic. MIMS EPrint 2020.22, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Oct. 2020.

[4] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23 (1991) 5–48.

[5] S. Hammarling and N.J. Higham. Wilkinson and backward error analysis. `https://nla-group.org/2019/02/18/wilkinson-and-backward-error-analysis/`, Feb. 2019.

[6] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd ed., 2002.

[7] N.J. Higham and T. Mary. A new approach to probabilistic rounding error analysis. *SIAM J. Sci. Comput.* 41 (2019) A2815–A2835.

[8] N.J. Higham and S. Pranesh. Simulating low precision floating-point arithmetic. *SIAM J. Sci. Comput.* 41 (2019) C585–C602.

[9] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985.

[10] J.-M. Muller. Some algebraic properties of floating-point arithmetic. In: P. Kornerup (ed.) *Proceedings of the Fourth Conference on Real Numbers and Computers* (2000), pp. 31–38.

[11] J.M. Muller, N. Brunie, F. de Dinechin, C.P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, MA, USA, 2nd ed., 2018.

[12] G.W. Stewart. Stochastic perturbation theory. *SIAM Rev.* 32 (1990) 579–610.

Photo credit: Rob Whitrow

Nicholas J. Higham

Nick is Royal Society Research Professor and Richardson Professor of Applied Mathematics in the Department of Mathematics at the University of Manchester. His current research interests include mixed precision numerical linear algebra algorithms. He blogs about applied mathematics at `https://nhigham.com/`. Nick shudders to recall that in some pre-IEEE standard computer arithmetics one could have $\mathrm{fl}(1.0 * x) \neq x$ for a floating-point number x.