Testing Floating-Point Applications

Alan A. Jorgensen, Connie R. Masters

AAJ@TrueNorthFloatingPoint.com
CM@TrueNorthFloatingPoint.com

Abstract

Testing the results of floating-point in large, complex scientific and engineering applications is problematic. Though floating point is standardized by IEEE, there are serious deficiencies. IEEE standard floating point has no indication of whether representation of a real value is exact, much less anywhere near correct. The correctness of the real value represented is tainted by accuracy and completeness of source data, algorithmic error, and accumulated floating-point error. Available techniques for testing complex calculations are expensive and unreliable. This paper describes a new testing technology for complex floating-point calculations employing bounded floating point, which is a device and methodology for calculating and propagating the bounds on floating-point calculations and alarming when a calculation does not meet a specified precision. This paper presents a demonstration of a software model of bounded floating point detecting a failure in Muller's recursion, an algorithm known to fail undetectably using IEEE standard floating point.

Biography

Alan Jorgensen is a computer engineer and scientist with decades of hardware and software design experience, particularly test engineering. He has degrees in Electrical Engineering and Computer Science. He has spoken internationally on software testing as well as previously at PNSQC. In addition to teaching as Adjunct Faculty at various universities, his latest work has been obtaining patents for an "Apparatus for Calculating and Retaining a Bound on Error during Floating-Point Operations and Methods Thereof," a technology called "Bounded Floating Point" first presented at PNSQC 2018 as a poster.

1 Introduction

I have spent my career improving computer systems' quality. Hardware was usually easy, but software quality often proved problematic. One problem in particular has plagued me - enough that I have lost sleep over it. This is the longstanding problem of unknown inaccuracies when using floating point in computers to represent values and results.

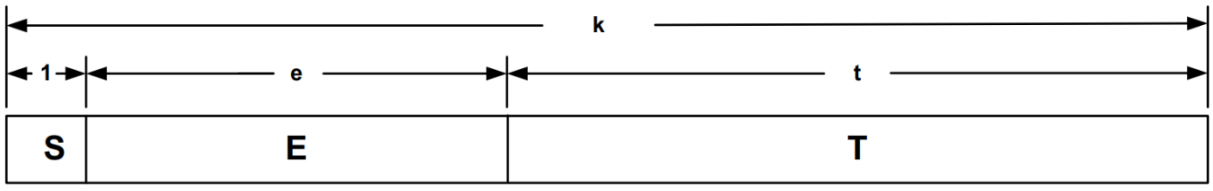
The first shock to my psyche occurred after I developed a test to evaluate compatibility between software floating point and a hardware floating-point implementation. My test showed that they were not compatible. Management decided that was okay. That was okay? It was okay to get two different answers? This was vexing to me, so over time I began to study floating point and to deliberate on possible approaches to resolve the problem of inaccuracies in results due to floating point.

As an engineer I have learned good problem-solving techniques that I have applied, for instance, to system problems in live nuclear power plant computers. I endeavored to apply these problem-solving techniques to the unresolved floating-point problem, but it was not an easy problem to solve.

2 IEEE Standard Floating Point

Floating point was standardized in 2011 at the instigation of the "Father of Floating Point," William Kahn [IEEE 2011]. Standard floating point is a data structure emulating scientific notation with a sign, exponent, and fraction (known as the significand). Fig. 1 shows the generalized floating-point format.

Figure 1 IEEE Standard Floating-Point Format



As I studied standard floating point, to my horror I discovered that floating-point error has killed dozens of soldiers due to a Patriot missile failure [GAO 1992], has destroyed a $370M rocket [Lions 1996], and has halved the value of a public stock trading business [Lilley 1983], [Kahan 2011]. I came to the realization that standard floating point is a significant computer system quality problem.

Sierra, the author of the first patent for floating point that I could find, said in his patent:

> "… under some conditions, the major portion of the significant data digits may lie beyond the capacity of the registers. Therefore, the result obtained may have little meaning if not totally erroneous." [Sierra 1962, p. 2]

Totally erroneous? How can we proceed with a calculation when we know the result obtained may be totally erroneous?

What to do? We know we can't "test-in" quality, but I realized it would help if we could at least KNOW if we HAD a problem.

3 Testing Floating-Point Applications

I investigated to see if others had proposed means to determine if a result was erroneous. Professor Emeritus William Kahan is very helpful in this matter. He identifies 3 testing methods as follows [Kahan 2011, pp. 39-48]:

- Recomputation with Redirected Rounding;
- Recomputation with Higher Precision; and
- Temporary Exception Handling.

The first two methods mentioned by Kahan are fault injection techniques, and the third is error trapping.

3.1 Recomputation with Redirected Rounding

The default rounding mode in standard floating point is “round to nearest,” but other modes are available, such as round to zero, round to +infinity, and round to -infinity [IEEE 2011]. Recomputation with redirected rounding uses a comparison of the results when the calculation is preformed using multiple modes of rounding.

Some language instantiations provide a means of setting the rounding mode. For instance, GNU’s gcc library [GNU Rounding] provides:

int **fesetround** *(int round)*

where round = FE_TONEAREST, FE_UPWARD. FE_DOWNWARD, and FE_TOWARDZERO

The application can be run in more than one mode, and the results of running the application with each mode will differ, but the results should not differ radically. If they do, a branch and bound technique will be required to determine the source of significant error. This is an expensive, time-consuming, repetitious, and manual method.

In contrast, using bounded floating point with required intermediate precision will isolate the location where precision is lost. Bounded floating point presents a range where the lower bound is the standard floating-point result, rounded to zero, and the upper bound is computed based on D, the number of lost bits, which is the number of bits that are not significant.

3.2 Recomputation with Higher Precision

The second method to test the accuracy of a result is to compute the result using an original precision, and then test it by using a higher precision. Standard floating-point precisions are 32-bit (single precision) and 64-bit (double precision). Some environments allow 128-bit (quad precision). Though using higher precision is a means of testing the accuracy of a result, higher precision requires more time and space, which may make implementing and testing in this way problematic.

GNU gcc, which does not have a direct “quad” type declaration, attempted to provide an indirect means of defining a lengthened precision, which is only partially functional. In this indirect method, normal C language functions, like casting and automatic type casting are not available [GNU 2018]. The method compares the results of an application test with normal precision, and then the result of the same application test with lengthened precision, which should indicate only a small difference. If there is a greater difference, there is a hidden floating-point defect undetected by IEEE standard floating point. In contrast, this defect would be detected during the execution of the calculation by the use of bounded floating point.

3.3 Temporary Exception Handling

The third conventional testing method, temporary exception handling, uses error trapping. Standard floating point provides several different exceptions, such as overflow, underflow, divide by zero, and Not a Number, (NaN). An exception is a hardware generated interrupt (or trap) that provides program notification of an error that requires special handling by the software. Examples include, dividing by zero or taking the square root of a negative number. Exceptions are “handled” by servicing the interrupt. The “Try-Catch” pair of operators is a high-level language construct to provide a scope in which an exception interrupt may be recognized and serviced. The software “Throw” command creates a hardware exception

interrupt and is used whenever the program discovers a failure from which it cannot gracefully recover. If any hardware exception is not serviced by the application generating the exception, then the operating system services the exception and fails the application catastrophically. A similar exception handling policy caused by floating point error lead to the failure of the Ariane 5 rocket launched by the European Space Agency [Lions 1996].

The Try-Throw-Catch exception generating/handling mechanism may catch something you were not fishing for [Kahan 2011, p. 55]. But it is best to keep exception handling in the test/diagnostic environment, and using it sparingly in production applications, particularly in mission critical systems where an improperly handled exception may cause a catastrophic system failure.

In gcc, Try-Throw-Catch, requires the use of the <setjmp.h> functions. I have included the exception code that I use for checking significance of floating-point values in Appendix A.

In contrast, bounded floating point provides the capability (under program control) of generating an exception when a result does not have sufficient precision.

4 Bounded Floating Point

I think my serious consideration of floating-point error started in the late 90's when I had some free time at work, and, being a life-long tester, began playing with the Microsoft Calculator available on my Windows desktop. It did not take long to discover anomalies, which I documented later in my Ph.D. dissertation [Jorgensen 1999, pp. 138-146]. Many of these anomalies were caused by incorrect floating point.

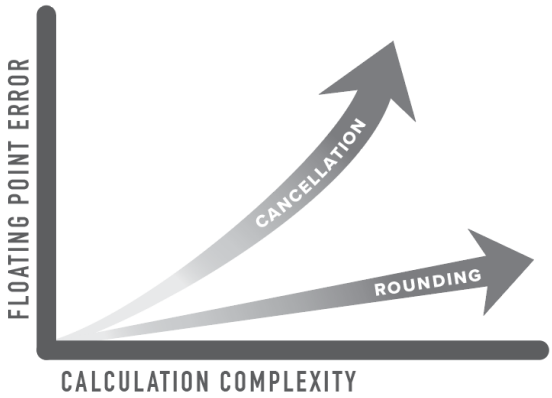
Some of this work led to my publication of "Why Software Fails." In summary, that work presents that the frequent cause of software failure is the failure of the programmer to constrain program operations, such as constraining input, calculations, storage, and output to reasonable values [Whittaker et al. 1999].

Floating-point error continued to be a burl in my brain. I would go to sleep thinking of a problem caused by floating point, but instead of waking up with an answer, I would wake up with a question to ask. I knew this problem had been around for a long time and bright minds had tackled it. So, why hadn't anyone solved it?

I began pouring information from leaders in the field into my brain [Kahan 2011], [Goldberg 1991], [Muller et al. 2010]. I examined the various attempts that others had made to solve floating-point problems, such as interval arithmetic [Hickey et al. 2001], error analysis [Higham 1996, pp. 74-78], and more.

While absorbing these various approaches and the current state of the art, I was seeking an insight that might lead me down a path to a solution. I went to sleep mulling over the problem and woke up with the realization that the current format did not provide sufficient information – more bits were needed for storing additional information (whether the standard format was shortened or more bits were added to the standard format). Based on this realization, I added information to the IEEE standard format by adding another field. This new field would be one that described the error in the value, limited that error, or provided a bound for that error. Thus, "Bounded Floating Point" (BFP) was born, creating a method of constraining floating-point operations.

After a few false starts, I came to realize that the problem was that there are two kinds of error: rounding and cancellation. To produce a solution, I must find a way to simultaneously retain information about both kinds of error.



Rounding error occurs because it is necessary to truncate the representation of some numbers. An easy example in the decimal numbering system is Pi or 1/3; both of these require an infinite number of digits to express exactly. Correspondingly, standard binary floating point cannot accurately represent any number that is not the sum of a limited range of powers of two. For example, one tenth (0.1) is a repeating fraction in binary and must be truncated, with rounding. Rounding error accumulates in a linear fashion.

Cancellation error, which occurs when subtracting "similar" numbers, multiplies existing error – exponentially.

Adding error from both cancelation and rounding is like adding apples and oranges *i.e.* things that cannot be practically compared. Yet, I needed to do this. And, as apples and oranges can be added by using a common characteristic, such as fruit or weight, I found that the common characteristic of cancellation error and rounding error was logarithms.

Consequently, the error field of bounded floating point is composed of the summation of cancellation error (which is logarithmic) and the logarithm of the summation of rounding error.

Figure 2 Bounded Floating-Point Format

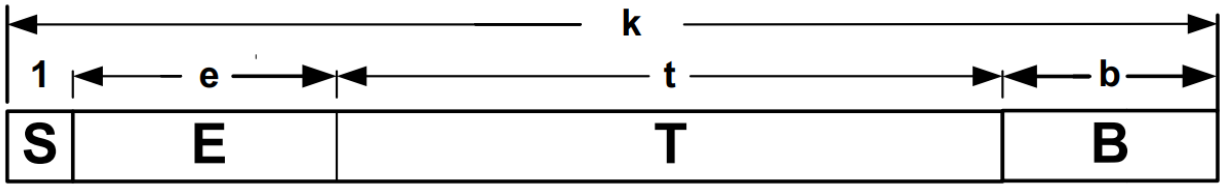
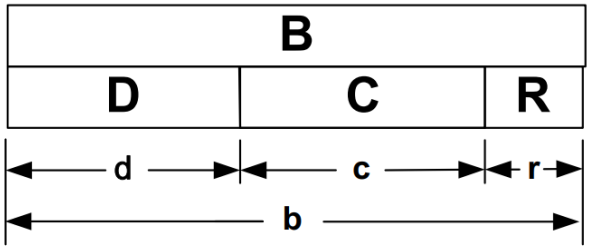


Figure 3 Format of Bound Field, B



The final revelation came from a visit with the Father of Floating Point, himself. This discovery was that operations on exact floating-point numbers yield exact results. "Exact" means that the numbers are accurate to + or – 1/2 unit in the last place (ulp), because this is as accurately as a number can be represented in the standard floating-point system [Jorgensen et al., 2020]. However, standard floating point has no way to inform you whether or not a result is exact. In contrast, bounded floating point can tell you if a number is exact because if the number is exact, the number of lost bits, D, would be zero.

5 Examples of Testing with Bounded Floating Point

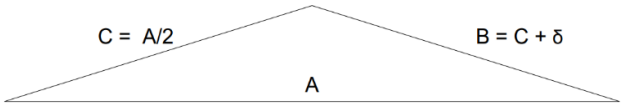
I have developed a C language bounded floating-point model and have provided input, output, and debug capabilities. The model provides conditional display of specified system logic. It is this model, which uses 80-bit bounded floating-point format, that I have used in these testing examples.

Bounded floating point is modeled with k = 80 (FIG. 2); with the bound, B, having b = 16 bits; and with S, E, and T remaining identical to 64-bit IEEE standard floating point where e = 11 and t = 52. The bound field B is defined with D (FIG. 3, with width d=6), c = 6, and r = 4. The C and R fields contain the current sum of the rounding error.

Having an operational model of bounded floating point meant that I could perform testing with it. So, I applied the model to some problems that were known to be difficult. The first problem I attacked was finding the area of a thin triangle [Kahan 2014].

5.1 Kahan's Thin Triangle

Figure 3 A Thin Triangle.



As the triangle becomes smaller and smaller (δ => 0), the area has fewer and fewer significant digits IEEE floating point will not tell you this, but bounded floating point will. Using the model, we are able to use a stress test (shown below in Table 1) in which we require greater and greater precision until we have a failure. This technique allows us to stress test significant scientific or engineering applications.

This thin triangle problem comes from Kahan's work [Kahan 2014], which extends the work of Pat H. Sterbenz [Sterbenz 1976 , pp. 152-153] who states:

> *"However, we can produce a good solution for the problem if we assume that A, B, and C are given **exactly** as numbers in [floating point]." [Emphasis added]*

But what happens when A or B or C are NOT exact? Further, standard floating point does not even provide any indication that A or B or C is exact. But bounded floating point establishes exactness, where a representation is exact if and only if the dominate bound lost bits field, D, is zero; this indicates that there are no insignificant bits in the representation.

The Kahan formula [Kahan 2014] for determining the area of the thin triangle is shown in (1), where A ≥ B ≥ C. The Area =

$$\text{SQRT}(A+(B+C))(C-(A-B))(C+(A-B))(A+(B-C)))/4 \quad (1)$$

Table 1 presents the data from stress testing the Kahan formula for 3 values of δ (delta), where the difference between the length of C and the length of B is δ. The results from standard double precision floating point (64-bit), standard quad precision floating point (128-bit) and bounded floating point (80-bit) compared. A one ulp error is injected into the "A" values by adding 1 to the significand field (T) of the 64-bit and 128-bit standard floating-point values and by adding 1 to the lost bits field (D) of the bounded floating-point value. Injecting a 1 ulp error into any one of the values, A, B, or C serves to inject an error into the Kahan equation subjecting it to cancellation error.

In the results from Triangle One, using B=0.8001 (where δ=0.0001), the first column of the first row shows that 10 significant digits are required. In the second column, double precision standard floating point indicates that there are seventeen significant digits. In the third column, using the recomputation with higher precision testing technique described above, quad precision identifies (by manually comparing the

two results) that only eleven of those seventeen are actually significant. In the fourth column bounded floating point displays only the digits that are significant, which in this case is eleven. Thus, bounded floating point verifies the results of the recomputation with higher precision test and identifies the actual number of significant digits in the double precision result. In the second row, 11 significant digits are required, and we calculate with the same values and get the same result – there are 11 significant digits. In the third row of Triangle One, 12 significant digits are required, and we calculate with the same values. The double and quad precision results are the same, but bounded floating point displays a quiet Not-a-Number (qNaN.sig) indicating Equation 1 cannot be solved for Triangle One using double precision to achieve a result accurate to 12 significant digits.

Table 1 Stress tests of Kahan's thin triangle equation for various increasing required significant digits

Stress Tests			
δ is embedded within the B value			
Required Significant Digits	Area Using Double Precision	Area Using Quad Precision	Area Using Bounded Floating Point
For Triangle One, A= 1.6, B=0.8001, and C=0.8			
10	0.00715552931654077180000	0.00715552931654910837456	0.0071555293165
11	0.00715552931654077180000	0.00715552931654910837456	0.0071555293165
12	0.00715552931654077180000	0.00715552931654910837456	qNaN.sig
For Triangle Two, A=1.6, B=0.80000001, and C=0.8			
6	0.0000715541743799515320 0	0.0000715541753917966676 8	0.00007155417
7	0.0000715541743799515320 0	0.0000715541753917966676 8	0.00007155417
8	0.0000715541743799515320 0	0.0000715541753917966676 8	qNaN.sig
For Triangle Three, A=1.6, B=0.800000000001, and C=0.8			
2	0.00000071545439186697031	0.00000071554175280004451	0.000000715
3	0.00000071545439186697031	0.00000071554175280004451	0.000000715
4	0.00000071545439186697031	0.00000071554175280004451	qNaN.sig

The stress test determines, for a specific triangle, where the algorithm (Equation 1) fails for a specific number of required significant digits. For each of the triangles in the test reported in Table 1 (using a given value of δ for each triangle), the required number of significant digits is increased until bounded floating point indicates by a quiet Not a Number (qNaN.sig) that Equation 1 cannot be solved for the specific triangle and achieve the specified number of required significant digits.

5.2 Muller's Recursion

Another test I performed was on Muller's recursion. This is a nasty counter example that converges – but to a completely incorrect value [Kahan 2011, pp. 37-38].

Muller's (Counter) Example:

$$X_{N+1} := 111 - (1130 - 3000/X_{N-1})/X_N \text{ for } N = 1, 2, 3, ... \quad (2)$$
$$\text{Where } X_0 := 2 \text{ and } X_1 := -4$$

Using bounded floating point's ability to generate an exception when insufficient digits are available for an intermediate result, I detected the precise point in the recursion where it fails.

Table 2 Muller's (Counter) Example - Bounded Floating Point vs. IEEE 64-Bit Floating Point

X[0] =	2.000000000000000	2.00000000000000000
X[1] =	-4.000000000000000	-4.00000000000000000
X[2] =	18.50000000000007	18.50000000000000000
X[3] =	9.37837837837878	9.37837837837837900
X[4] =	7.801152737756	7.80115273775216790
X[5] =	7.15441448103	7.15441448097533340
X[6] =	6.8067847377	6.80678473692481220
X[7] =	6.592632780	6.59263276872180270
X[8] =	6.44946611	6.44946593405408120
X[9] =	6.348454	6.34845206074892940
X[10] =	6.27448	6.27443866276447610
X[11] =	6.2193	6.21869676916201720
X[12] =	6.187	6.17585386514404090
X[13] =	sNaN.sig	6.14262732158489030
X[14] =		6.12025116507937560
X[15] =		6.16612674271767690
X[16] =		7.23566541701194320
X[17] =		22.06955915453103100
X[18] =		78.58489258126825000
X[19] =		98.35041655134628500
X[20] =		99.89862634218410200
X[21] =		99.99387444125312600
X[22] =		99.99963059549460800
X[23] =		99.99997774322417900
X[24] =		99.99999865997196500
X[25] =		99.99999991936725500

As seen in Table 2, Muller's recursion converges to 100.0 when the correct answer is 6.0, but bounded floating point signals a loss of precision at a midpoint in the calculation of the recursion. In recursions, loss of significance in an intermediate result propagates to the final result.

Interestingly, when using floating point, error in intermediate results may not propagate to the final result, because successive alignment of the binary point may eliminate previous error bits by shifting them out of range.

Nevertheless, bounded floating point will track the error in the calculation in real time.

6 Summary

Bounded floating point, implemented in hardware or software, is very useful for testing complex scientific and engineering calculations, as shown by the application to Kahan's thin triangle problem and to Muller's Recursion.

Instead of a designer of an application hoping that a result has the required number of significant digits, bounded floating point will assure that it does or notify if it does not.

Bounded floating point can also identify whether a number is exact.

No longer do we need to proceed with a calculation knowing that the result obtained may be totally erroneous.

7 Appendix A

```
// TRY-THROW-CATCH Defined
#include <setjmp.h>
#define Place jmp_buf
#define Mark ex_buf__
#define MarkPlace(place) setjmp(place)
#define ReturnToMark(ReturnValue) longjmp(Mark,ReturnValue)

#define TRY do{ Place Mark; if( !MarkPlace(Mark) ){
#define CATCH } else {
#define TRY_End } }while(0)
#define THROW(Exception) ReturnToMark(Exception)
```

Application to bounded floating point sNaN.sig:

```
char Exception[25];
int BFPChk(bfp op1, int MaxLostBits)
{
     int Exception = (int) "sNaN.sig";

  TRY
  {
        if (D(op1.B) > MaxLostBits)
             THROW(Exception);
        return 0;  // No Exception
  }
  CATCH
  {
        return (int) Exception;  // Returns integer pointer to "sNaN.sig"
  }
  TRY_End;

  return 0;
}
```

Where D(op1.B) is a macro to retrieve the lost bits of the bounded floating point operand, “op1”

8 References

Investigations and Oversight, Committee on Science, Space, and Technology. House of Representatives. Accessed 5 June 2020, https://www.gao.gov/assets/220/215614.pdf.

GNU; 2018. “The GCC Quad-Precision Math Library.” Accessed 3 June 2020, https://gcc.gnu.org/onlinedocs/gcc-8.4.0/libquadmath.pdf.

GNU; “C Library 20.6 Rounding Modes.” Accessed 6/4/2020, https://www.gnu.org/software/libc/manual/html_node/Rounding.html.

Goldberg, D. “What Every Computer Scientist Should Know About Floating-Point Arithmetic.” ACM Computing Surveys, vol. 23, no. 1, pp. 5-47, 1991.

Hickey, T., Q. Ju, M.H. van Emden. 1999. “Interval Arithmetic: from Principles to Implementation.” Journal of the ACM (JACM). Accessed 6 June 2020, http://fab.cba.mit.edu/classes/S62.12/docs/Hickey_interval.pdf.

Higham, N. J. “Accuracy and Stability of Numerical Algorithms.” Philadelphia, PA: SIAM., 1996, pp. vii-xxviii,1-688.

ISO/IEC/IEEE 60559. “Information technology - Microprocessor Systems – Floating-Point Arithmetic.” Piscataway, NJ. Institute of Electrical and Electronics Engineers, 2011.

Jorgensen, Alan A. “Software Design Based on Operational Modes”, A dissertation submitted to the Graduate School of Florida Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, Melbourne, Florida. 1999. Accessed 11 June 2020, https://repository.lib.fit.edu/handle/11141/103.

Jorgensen, A. A., A. Masters. 2020. “Exact Floating Point.” To be published. Springer Nature - Book Series: Transactions on Computational Science & Computational Intelligence.

Jorgensen, A. A., A. Masters and R. Guha. 2019. “Assurance of Accuracy in Floating-Point Calculations - A Software Model Study,” in 2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA.

Kahan, W. M. “Desperately needed remedies for the undebuggability of large floating-point computations in science and engineering.” *IFIP/SIAM/NIST Working Conference on Uncertainty Quantification in Scientific Computing*. Boulder, Colorado. 2011. Accessed 6/5/2020, https://people.eecs.berkeley.edu/~wkahan/Boulder.pdf.

–. “Miscalculating Area and Angles of a Needle-like Triangle.” 4 September 2014.. Accessed 13 January 2020, https://people.eecs.berkeley.edu/~wkahan/Triangle.pdf.

Lilley, Wayne. "Vancouver stock index has right number at last." The Toronto Star. November 29, 1983.

Lions, J. L. “Flight 501 Failure, Report by the Inquiry Board.” Chairman of the Board. Accessed 5 June 2020, http://sunnyday.mit.edu/nasa-class/Ariane5-report.html.

Muller, Jean-Michael, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehle, and Serge Torres. 2010. “Handbook of Floating-Point Arithmetic.” Boston: Birkhauser.

Sierra, H. M. “Floating Decimal Point Arithmetic Control Means for Calculator.” 5 June 1962. US Patent No. Patent 3,037,701.

Sterbenz, P. H. “Floating-Point Computation.” 1976. Prentice-Hall, Inc. Englewood Cliffs, N.J.

United States General Accounting Office. February 1992. “PATRIOT MISSILE DEFENSE Software Problem Led to System Failure at Dhahran, Saudi Arabia.” Report to the Chairman, Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, House of Representatives. Accessed 5 June 2020, https://www.gao.gov/assets/220/215614.pdf.

J. Whittaker and A. A. Jorgensen. “Why software fails.” ACM SIGSOFT Software Engineering Notes, vol. 24, no. 4, p. 81–83, 1999.