The Interval Logarithmic Number System

Mark G. Arnold and Jesus Garcia
Lehigh University
CSE Department
Bethlehem, PA 18015, USA
{marnold,jegj}@eecs.lehigh.edu

Michael J. Schulte
University of Wisconsin-Madison
Department of ECE
Madsion, WI 53706, USA
schulte@engr.wisc.edu

Abstract

This paper introduces the Interval Logarithmic Number System (ILNS), in which the Logarithmic Number System (LNS) is used as the underlying number system for interval arithmetic. The basic operations in ILNS are introduced and an efficient method for performing ILNS addition and subtraction is presented. The paper compares ILNS to Interval Floating Point (IFP) for a few sample applications. For applications like the N-body problem, which have a large percentage of multiplies, divides and square roots, ILNS provides much narrower intervals than IFP. In other applications, like the Fast Fourier Transform, where addition and subtraction dominate, ILNS and IFP produce intervals having similar widths. Based on our analysis, ILNS is an attractive alternative to IFP for application that can tolerate low to moderate precisons.

1. Introduction

Floating-point arithmetic is the dominant choice for scientific and graphics applications; however, other alternatives exist. These alternatives are desirable for niche applications that have special requirements. An alternative with very similar properties to floating point is the Logarithmic Number System (LNS), which can be more cost effective for applications that require low-to-moderate precision. Advantages of LNS include: low power consumption [23], efficient matrix operations [8], and low-cost multiplies, divides, powers, square roots, etc., all of which are useful in massively parallel scientific simulation [19].

Interval arithmetic is an alternative used instead of conventional floating point to give reliable bounds on roundoff, approximation and inexact-input errors. Interval arithmetic is often used in scientific code to explore the effect of measurement error on the final result of a computation. For example, astronomical applications of interval arithmetic have included calculating the effects of experimental error to obtain Newton's gravitational constant to a higher degree of accuracy [12], and analyzing the stability of solar systems [15].

This paper proposes combining LNS and interval arithmetic. To the authors' knowledge, this combination of interval arithmetic and LNS has not been studied previously. This may primarily be due to the perception that interval arithmetic, which is often used for high-precision applications, is incompatible with the low or moderate precision applications where LNS offers significant advantages. We have identified at least two applications (the N-body problem [19] and computer graphics [29]), where this combination of interval arithmetic and LNS arithmetic may offer significant advantages. ILNS is especially attractive in application-specific systems that allow for specialized hardware designs. The simple algorithms chosen in this paper to illustrate the benefits of ILNS frequently are implemented using application-specific systems, rather than general-purpose processors. Future research will investigate algorithms that use interval arithmetic to achieve convergence by interval methods, and the advantages that ILNS yields for them.

The remainder of this paper is organized as follows. Section 2 gives an overview of LNS. Section 3 discusses interval arithmetic and shows how LNS arithmetic can be used to implement interval LNS. Section 4 describes routines that are used to simulate algorithms with conventional and logarithmic interval arithmetic. Section 5 describes how various algorithms perform using these two arithmetics. Section 6 discusses an implementation of the proposed system. Section 7 summarizes the effects observed in the simulations and gives conclusions.

2. The Logarithmic Number System

LNS represents a real number, x, using a sign bit, $\text{sign}(x)$, and the fixed point, base-b logarithm of the absolute value of that real number, such that $\bar{x} = \text{round}(\log_b |x|, \text{mode})$, rounded using some mode (e.g., towards nearest,

towards $\pm\infty$, or by special LNS modes [4]). The representation of a quotient, q, is simply the difference of the logarithms ($\bar{q} = \bar{x} - \bar{y}$) and the exclusive OR of the sign bits ($\text{sign}(q) = \text{sign}(x) \oplus \text{sign}(y)$). A similar approach holds for products by adding logarithms and for square roots by right shifting the logarithm. When the operands are exact and the results do not overflow, the LNS product or quotient is exact, because only fixed point operations are involved on the logarithms. This is an advantage compared to floating point, where only a small subset of products and quotients of exact operands produce exact results. For example, if $b = 2$ with the inputs $x = -8$ and $y = -1/\sqrt{2}$, we have $\bar{x} = 3$, $\text{sign}(x) = 1$, $\bar{y} = -0.5$ and $\text{sign}(y) = 1$, which are exact representations. The quotient, $q = 8\sqrt{2}$, is represented as $\bar{q} = 3.5$ and $\text{sign}(q) = 0$, which is an exact representation in this base-2 LNS.

LNS addition and subtraction [14] are more complicated and their results are seldom exact. Addition can be rewritten as:

$$x + y = y \cdot (1 + x/y). \tag{1}$$

The advantage of (1) is that a function of two variables ($x + y$) has been reduced to a function of one variable (increment of the quotient). In order to increment with LNS, the hardware needs an implementation of the *addition logarithm*,

$$s_b(z) = \log_b(1 + b^z) \tag{2}$$

and (for applications involving adding numbers of opposite signs) the *subtraction logarithm*,

$$d_b(z) = \log_b |1 - b^z|. \tag{3}$$

From (1), when the signs of the operands are the *same*,

$$\begin{aligned} \log(|x| + |y|) &= \log_b(|y| \cdot (1 + |x|/|y|)) \\ &= \bar{y} + \log_b(1 + b^{(\bar{x} - \bar{y})}) \\ &= \bar{y} + s_b(\bar{x} - \bar{y}). \end{aligned} \tag{4}$$

When the signs of the operands *differ*, a similar approach holds:

$$\log ||x| - |y|| = \bar{y} + d_b(\bar{x} - \bar{y}). \tag{5}$$

The result of s_b or d_b must be rounded in some way [4] to make it representable in the finite $b = 2$ LNS.

For example, $x = -8$ and $y = -1/\sqrt{2}$, gives $\bar{x} - \bar{y} = 3.5$ and $\text{sign}(x) = \text{sign}(y)$, Thus, $s_2(3.5) = 3.6221934162022...$, which must be rounded. The computed result of (4), $\log_2(|x| + |y|) = -0.5 + s_2(3.5) \approx 3.122... \approx \log_2(8.707...)$, is only an approximation of the exact value $\log_2(8 + 1/\sqrt{2})$.

3. Interval Arithmetic

Interval arithmetic specifies a precise method for performing arithmetic operations on intervals [21]. An interval-arithmetic operation takes intervals as input operands and produces an interval as the output result. The output interval is defined by lower and upper endpoints, such that the true result is guaranteed to lie in this interval. The width of the interval (i.e., the distance between the two endpoints) indicates the accuracy of the result or the certainty with which the result is known.

Interval arithmetic is implemented by using some underlying real number system and its associated arithmetic operations. For example, the underlying real number system could be exact rationals (as available in LISP) [27] that require an unbounded amount of memory per number. In this case, the resulting intervals are as narrow (precise) as the algorithm (involving the four basic operations on intervals) allows for the given inputs.

More commonly, the underlying real number system is chosen to be one that is easier to implement in hardware. Often, such systems have a constant, rather than unbounded, datum size that is determined when the hardware is fabricated (i.e., the word size). For example, the fixed-point number system has the advantage that overflow-free addition and subtraction are exact using simple integer ALUs, but suffers from the fact that roundoff error occurs for multiplication and division. Another disadvantage of fixed point interval arithmetic is its limited dynamic range [24].

Although specialized hardware for variable-precision floating-point interval arithmetic [26] has been suggested, the most common number system for implementing interval arithmetic has been conventional floating point, as exemplified by the IEEE-754 standard [13]. This paper refers to this implementation as Interval Floating Point (IFP). A description of IFP and its implementation using IEEE-754 arithmetic is provided in [11]. IFP offers greater dynamic range than interval fixed point for the same constant word size, but suffers from roundoff errors, in many cases, for all four basic arithmetic operations (+, -, *, /).

This paper introduces the Interval Logarithmic Number System (ILNS), which uses LNS to perform interval arithmetic. LNS has approximately the same dynamic range as floating point, but LNS has the advantage that roundoff never occurs in the operations of multiplication and division. This suggests ILNS may offer interval widths closer to the theoretical minimum for the same constant word size as IFP. ILNS is most useful for applications that need low-to-moderate precision and that have a predominance of multiplications, divisions, square roots and other powering operations. Although only a subset of applications meet these criteria, for those that do, ILNS may offer significant advantages.

Interval arithmetic was originally proposed as a tool for bounding roundoff errors in numerical computations [21]. It can also used to determine the effects of approximation errors and errors that occur due to non-exact inputs [1]. In-

terval arithmetic is especially useful for scientific computations, in which data is uncertain or can take a range of values. For example, Kreinovich and Bernat have used interval arithmetic to investigate the stability of the solar system [15].

In the discussion to follow, intervals are denoted by capital letters and real numbers are denoted by small letters. The lower and upper endpoints of an interval X are denoted as x_l and x_u, respectively. As defined in [21] and [1], a closed interval $X = [x_l, x_u]$ consists of the set of real numbers between and including the two endpoints x_l and x_u (i.e. $X = \{x : x_l \leq x \leq x_u\}$). An real number x is equivalent to the degenerate interval $[x, x]$.

The ILNS representation, $\bar{X}$, of an interval $X > 0$ is itself an interval involving the fixed-point logarithmic representations of the endpoints: $\bar{X} = [\bar{x}_l, \bar{x}_u]$. When an algorithm allows $x_l < 0$, the ILNS representation must also include sign(x_l) and sign(x_u).

The width and midpoint of an interval X are defined as:

$$\text{width}(X) = x_u - x_l \quad (6)$$

$$\text{midpoint}(X) = (x_l + x_u)/2 \quad (7)$$

The endpoints of a result interval may not be representable in the underlying number system. In this case, the endpoints have to be rounded outwards: the lower towards $-\infty$, which is denoted as $\bigtriangledown()$, and the upper towards $+\infty$, which is denoted as $\bigtriangleup()$. This guarantees that the *true* result is still contained in the new interval, but introduces an undesired uncertainty, by making the interval wider, as defined by (6).

Conventional interval addition and subtraction are defined as [21]:

$$X + Y \approx R = [\bigtriangledown(x_l + y_l), \bigtriangleup(x_u + y_u)] \quad (8)$$

$$X - Y \approx R = [\bigtriangledown(x_l - y_u), \bigtriangleup(x_u - y_l)] \quad (9)$$

The ILNS implementation of addition is:

$$\bar{R} = [\bar{y}_l + \bigtriangledown(f_l(\bar{x}_l - \bar{y}_l)), \bar{y}_u + \bigtriangleup(f_u(\bar{x}_u - \bar{y}_u))], \quad (10)$$

where f_l is either s_b or d_b, depending on whether sign(x_l) = sign(y_l). Likewise, f_u is either s_b or d_b, depending on whether sign(x_u) = sign(y_u). Certain algorithms, like Euclidean-distance calculations, guarantee positive signs, and therefore $f_l = f_u = s_b$.

Conventional interval multiplication [21] is defined as:

$$X \cdot Y \approx R = [\bigtriangledown(\min(x_l y_l, x_l y_u, x_u y_l, x_u y_u)), \bigtriangleup(\max(x_l y_l, x_l y_u, x_u y_l, x_u y_u))]. \quad (11)$$

Alternatively, the interval endpoints of X and Y that give the correct interval product can be determined by examining their sign bits. Since when $x_l > 0$, x_u must also be positive, there are three possible cases for sign(x_l) and sign(x_u) This results in a total of nine cases [22]. With this technique, only two multiplications are required, unless both X and Y cross zero. Note that unlike IFP, ILNS does not require rounding since it is implemented using addition.

Interval division is defined as:

$$X/Y \approx R = [\bigtriangledown(\min(x_l/y_l, x_l/y_u, x_u/y_l, x_u/y_u)), \bigtriangleup(\max(x_l/y_l, x_l/y_u, x_u/y_l, x_u/y_u))] \quad (12)$$

if Y does not contain zero. Again, this can be implemented with no rounding in ILNS, since division is implemented by subtracting logarithms. If Y contains zero, the resulting interval is infinite. To allow for division by an interval that contains zero, extended interval arithmetic is required [10]. Extended interval arithmetic specifies results for division by an interval that contains zero and for operations on plus and minus infinity, as provided by IEEE-754 [13]. ILNS can be extended to deal with this in an analogous way [2]. Like the product, each endpoint of the quotient can be determined by a single division after examining the sign bits [21].

Interval arithmetic is also defined for the elementary functions. If an elementary function $f(x)$ is monotonically increasing on $X = [x_l, x_u]$, the resulting interval is $f(X) = [f(x_l), f(x_u)] \approx [\bigtriangledown(f(x_l)), \bigtriangleup(f(x_u))]$. For example, when $x_l \geq 0$, the square-root function can be computed as

$$\sqrt{X} \approx [\bigtriangledown(\sqrt{x_l}), \bigtriangleup(\sqrt{x_u})]. \quad (13)$$

4. Interval Arithmetic Emulation

To investigate the numerical differences between ILNS and IFP, a library of basic interval operations was written for ILNS and IFP addition, subtraction, multiplication, division, and square root. Algorithms using this library perform each arithmetic operation on interval data. The underlying number representation chosen greatly affects interval results, since every time an endpoint of a result cannot be exactly represented, it must be rounded. The frequency of this rounding and the size of the roundoff error determine the width of the interval results.

The IEEE-754 standard [13] for floating point represents values in a 32-bit single-precision format, where the MSB is the sign bit (sign(x)), the following eight bits represent the exponent (e), and the remaining 23 bits represent the mantissa (m). The mantissa is a fixed-point value, with a binary point to the left of the mantissa and an implicit 1 to the left of the binary point. The value, x, represented by the IEEE-754 single-precision format is:

$$x = (1 - 2\,\text{sign}(x))2^{e-127}(1 + m), \quad (14)$$

except in the subnormal region. The maximum roundoff error in FP is given by

$$\epsilon_{FP} = 2^{e-127} \cdot 2^{-23} = 2^{e-150}, \quad (15)$$

which is relative to the exponent, but not to the mantissa. Thus, the maximum relative error increases as the mantissa decreases.

The selected LNS implementation also uses a 32-bit format. The base is 2, the MSB is the sign bit, and a 31-bit fixed-point exponent l follows it. The first 8 bits in the exponent are the integer part, and the last 23 bits come after the binary point. This allows a similar dynamic range and also similar accuracy [4] compared to single-precision floating point. The value, u, represented by the LNS format is

$$u = (1 - 2\,\mathrm{sign}(u))2^{l-128}, \tag{16}$$

where in the notation of the previous sections, the LNS representation is $\bar{u} = 128 - l$ and sign(u). (16) provides constant relative precision. The maximum roundoff error is given by

$$\epsilon_{LNS} = 2^{l-128}(2^{2^{-23}} - 1) \approx 2^{l-151.5}, \tag{17}$$

which is proportional to the value of u. The value *zero* is considered a special case, and is represented with a reserved value. When zero is used in an operation, it is detected and the result is exact. Denormal and infinite values, although possible in LNS [2], were not implemented for this research.

Emulating IFP operations is straightforward. The IEEE-754 compliant FPU in any modern general-purpose processor allows setting the rounding mode toward $+\infty$ or $-\infty$, as required by the interval operations.

To emulate an LNS ALU, LNS values are stored using the 32-bit sign and exponent representation. Multiplication and division are exact (unless overflow occurs) and can be implemented with the simple fixed-point addition or subtraction of the logarithmic representations. Operations that are not exact require outward rounding. Assuming an exact representation of the logarithm of the absolute value of the result (which, in general, would require infinite precision), the result is truncated to 23 bits. Positive values rounded toward $+\infty$ and negative values rounded toward $-\infty$ need to have 2^{-23} added to the truncated logarithm. This is accomplished by adding 1 to the LSB, to produce the correctly rounded result. In the remaining cases, truncation is enough. The assumption of an exact result is justified for each operation in the next two paragraphs.

In LNS, the square root is equivalent to dividing the exponent by 2, and is therefore implemented using a 1-bit right shift. The exact result can be represented with just one extra precision bit. There is a 50% probability that the endpoints for (13) are exact and do not require rounding. Rounding is straight-forward, since only the the single bit that was shifted out and the rounding direction need to be examined.

Addition and subtraction in LNS are more complicated. Of the steps in (4) and (5), only the computations of s_b and d_b are subject to roundoff errors. To compute the result, (2) and (3) are computed as shown. In the ILNS library routines, the separate steps involving exponentials and logarithms in (2) and (3) are computed in double-precision floating point, and the final result is rounded as required (depending on which interval endpoint is being calculated). Double precision is sufficiently more accurate than the precision used for the simulation that such double-precision results rounded towards $+\infty$ or $-\infty$ should form the correct interval: $[\bigtriangledown(s_b(z_l)), \bigtriangleup(s_b(z_u))]$[1]. In practice, some hardware implementations may reduce the precision with which (2) and (3) are calculated, and force over-approximation or under-approximation, as required to guarantee a correct interval result. The straightforward implementation using double precision indicates the maximum improvement when switching from IFP to ILNS.

To complete the interval support, it is necessary to define how a value is assigned to an interval variable so that it can be used with the interval functions. The assignment converts the value to the number representation being used (FP or LNS). If it has an exact representation, then the interval has zero-width, as in $B = [1, 1]$, which has an exact representation in both FP and LNS. If the value cannot be represented exactly, the nearest upper and lower exactly-representable values are used as the endpoints for the generated interval.

5. Algorithm Comparison

The algorithms chosen to compare the width of result intervals for IFP and ILNS illustrate the viability of using ILNS as an alternative for IFP in certain real-world applications. Due to the nature of the representation, ILNS has the greatest advantage for algorithms that make extensive use of multiplications, divisions, squares and square roots. These include the proposed target applications, such as gravity-force computation in astronomical simulations or normalized-distance calculation for graphics applications. It is also shown that ILNS produces interval widths that are comparable to IFP for other important algorithms, such as the Fast Fourier Transform, where the number of multiplications and divisions is significantly less than the number of additions and subtractions.

5.1. Fast Fourier Transform

The Fast Fourier Transform (FFT), which has been implemented successfully using LNS [28], is a very important algorithm in digital signal processing (DSP), due both its usefulness in changing from the time-domain to

[1] Based on data presented in [25], this seems likely to be true, but we have not yet had a chance to verify it.

the frequency-domain and to its optimized implementation. The FFT requires significantly fewer multiplications than additions, which would seem to favor IFP over ILNS.

The selected implementation is a 32-point, real-input, decimation-in-frequency FFT. The FFT algorithm has five radix-2 stages, after which bit-reversing is applied to the outputs. The complex values involved in the computation of the FFT are represented with a real and an imaginary component. Each of these components is an interval defined by its two endpoints.

Two experiments involving different inputs are presented. The first experiment uses a sinusoidal wave of amplitude 10, plus white noise of amplitude 0.5. The second uses a square wave, also with an amplitude of 10, and the same noise. Both are real-valued inputs, in which the imaginary part of each complex input interval is set to zero. The frequency of both waves is 1.0. Although there is a random component involved, the randomly generated values are stored, and identical inputs are supplied to the IFP and ILNS FFT algorithms. Both the input values and the twiddle factors are generated as double-precision FP numbers. These values are then transformed into intervals with 32-bit endpoints. The width of these intervals depends on the corresponding number system, IFP or ILNS, as is explained in Section 4.

The simulation is executed 10,000 times on noisy input waves. Thus, for each input sequence, there are 10,000 FFT computations, producing 320,000 points, each consisting of two intervals (Re and Im) that are defined by two endpoint values. The outputs vary between simulation runs due to the spectral components introduced by the noise signal.

To establish a fair comparison, we take into account that roundoff errors introduced in the interval calculations are relative to the number being represented, as shown in Section 4. The *relative width* of each interval is defined as

$$\text{width}_{rel}(A) = \frac{\text{width}(A)}{\text{midpoint}(A)}, \qquad (18)$$

where $\text{width}(A)$ is computed using (6) and $\text{midpoint}(A)$ is computed using (7). This is adequate for algorithms in which catastrophic cancellation (i.e., subtraction of similar values, yielding an inaccurate result close to zero) does not occur. In the FFT, however, this problem exists, since intervals whose endpoints surround zero may produce extremely large relative errors, and mask the behavior of the other outputs.

Therefore, the selected measure is the ratio of the IFP and ILNS output interval widths for the same input. The ratio obtained is a measure of how much better ILNS performed for a particular output (i.e., how much wider the IFP output is than the ILNS output). Although the ratios for every interval are comparable, they can be slightly misleading, since a ratio of 0.125 has the same relevance but in the opposite direction as a ratio of 8. Therefore, the $\log_2()$ of each

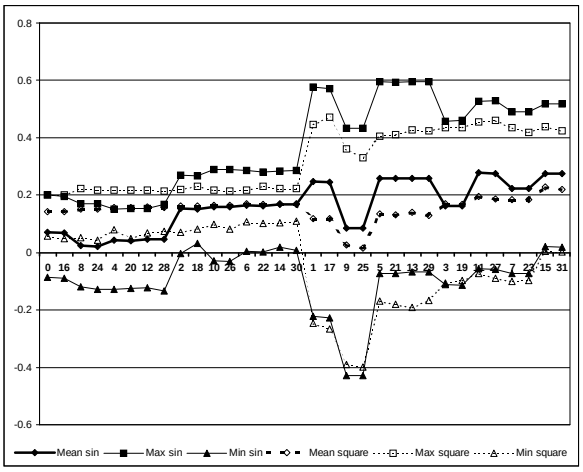

Figure 1. Logarithmic ratios (IFP/ILNS) of real FFT components for sine and square waves.

ratio is calculated. These measures are called the *logarithmic ratios*. This produces an intuitive result, where positive values represent narrower intervals for ILNS, and negative values indicate narrower intervals for IFP.

The results for the real part of the outputs are presented in Figure 1. This graph combines the outputs of both the sine-wave and square-wave experiments. The data presented corresponds to the maximum, minimum and mean of the logarithmic ratios, at each output index for the 32-point FFT. The reason for analyzing the outputs in this way is that the 32-point FFT algorithm represents 32 different sub-algorithms; one for each output. These outputs go through a different number of additions and multiplications, in which different twiddle factors are used. Outputs with the same index, but corresponding to different periods of the input waves, show a very similar behavior, because they are generated by the same sub-algorithm.

The outputs are ordered according to their index before being bit-reversed, since this is helpful in analyzing the results. It is obvious that outputs with an even index (the leftmost sixteen outputs in the plot) present a lower, less variable logarithmic-ratio mean. This is because these outputs are not multiplied in the first FFT stage. The first stage is especially important, because intervals generated in it expand in every consecutive stage. Since even-indexed FFT outputs do not have multiplication in the first stage, ILNS and IFP tend to produce similar intervals. Outputs in the right half of the plot (odd indices) experience a multiplication in the first stage and show, in general, more advantage for ILNS. Multiplication in early stages increase this advantage. Differences in the midpoint values of the intervals being added cause a variability in these results that cannot be predicted just by taking multiplications into account.

The significant observation from Figure 1 is that for the sine- and square-wave inputs given, the mean of the logarithmic ratios are positive for all FFT outputs, indicating similar or slightly better ILNS results. There is variation in the logarithmic ratios between the different points produced by the FFT. For example, some outputs (like 1, 3 and 31) are noticeably better in ILNS, while others (like 8, 24, 9 and 25) are not. We attribute much of this variation to the presence or absence of multiplication (which favors ILNS) in the first stage of the FFT.

5.2. Gravitational-force computation

N-body simulation is a technique that consists of simulating the evolution of a system with N bodies, where the force exerted on each body depends on its interaction with every other body in the system. Typical applications for this technique are studies of astrophysical and molecular systems, where the size of the bodies (stars and molecules, respectively) is negligible compared to the distances between them, which allows bodies to be represented by simple points in a 3-D space.

In astrophysical N-body simulation, the trajectories of stars are calculated by integrating the force (acceleration) due to gravitational interaction. One straightforward method is to calculate, for each star, the component due to each other star present in the simulation. This is accomplished by evaluating the expression

$$\vec{a}_i = \sum_{j=1}^{N} \frac{\vec{x}_j - \vec{x}_i}{(r_{ij}^2 + \epsilon^2)^{3/2}}, \tag{19}$$

where $\vec{a}_i$ is the gravitational acceleration at the position of particle i, $\vec{x}_i$ is the position of particle i, r_{ij} is the distance between particles i and j, and ϵ is the artificial-potential softening used to suppress the divergence of the force as $r_{ij} \to 0$ [19]. The x-coordinate terms in $\vec{a}_i$ are calculated as

$$\frac{x}{(x^2 + y^2 + z^2 + \epsilon^2)^{3/2}}, \tag{20}$$

where $\vec{x}_j - \vec{x}_i = (x, y, z)$ is computed in fixed point and then x, y and z are converted to LNS. This approach is used in the GRAPE-3 processor designed by Makino [19]. Equation (20) is used in this paper to compare the interval widths obtained when using IFP or ILNS. In a complete system, the final addition of every gravity vector exerted on a given body should be taken into account. The whole N-body algorithm has not been implemented for this paper due to its complexity; however, note that the result of operations on intervals of non-minimum width is much more dependent on the widths of the inputs than on the associated rounding. Therefore evaluating equation (20) is considered significant.

The abundance of squares in (20) suggests ILNS will show a clear improvement over IFP. To test this, (20) is executed 100,000 times using both IFP and ILNS. The set of inputs, x, y, z and ϵ, are generated randomly for each computation. The variables x, y, and z take values from $[0, 1)$ and the error variable ϵ takes values from $[0, 0.001)$ (to reflect the smaller magnitude it usually has). This produces two sets of 100,0000 output intervals, one for each number representation. These outputs are similar to the calculations in one step of an N-body astrophysical simulation, where 100,000 stars are uniformly distributed in a normalized tridimensional cube.

To compare results when the inputs are not as narrow as minimum-width intervals, a similar experiment was conducted which increases the width of the input intervals. A new value x_W was generated for input x as

$$x_W = x + x \cdot 2^{-23+W} = x(1 + 2^{-23+W}), \tag{21}$$

and the endpoints of the interval were obtained by rounding x and x_W to the selected 32-bit number representation. This created input intervals with an approximately constant relative width of 2^{-23+W} (it is not perfectly constant because of rounding).

Input Intervals	Output Logarithmic Ratio			
	mean	σ^2	Max	Min
W = 0	0.676	0.043	1.508	-0.273
W = 1	0.395	0.032	2.036	-0.322
W = 2	0.234	0.010	0.909	-0.410
W = 3	0.130	0.003	0.469	-0.118
W = 4	0.069	0.001	0.246	-0.048
W = 5	0.035	0.000	0.126	-0.036

Table 1. Logarithmic-ratio statistics for the gravity equation with 100,000 sets of random inputs.

The comparison between IFP and ILNS output data sets was performed using the logarithmic ratio ($\log_2$(IFP/ILNS)). Statistical analysis was performed on the logarithmic ratios, and the results are shown in Table 1. As explained above, the experiment was repeated for minimum-width input intervals, with $W = 0$, and wider input intervals, with W ranging from 1 to 5. Since this algorithm only adds positive numbers, catastrophic cancellation never occurs. Statistics obtained using relative widths as defined in equation 18 agree with these results.

Table 1 shows that ILNS produces narrower intervals. Using the logarithmic ratios, the mean for minimum-width inputs is 1.598 ($\approx 2^{0.676}$) times wider for IFP output intervals than it is for ILNS output intervals. Calculating the ratio with the mean of the relative widths, the value obtained is 1.606. This is nearly a 60% wider FP interval on average, showing a very important advantage to using ILNS. Also

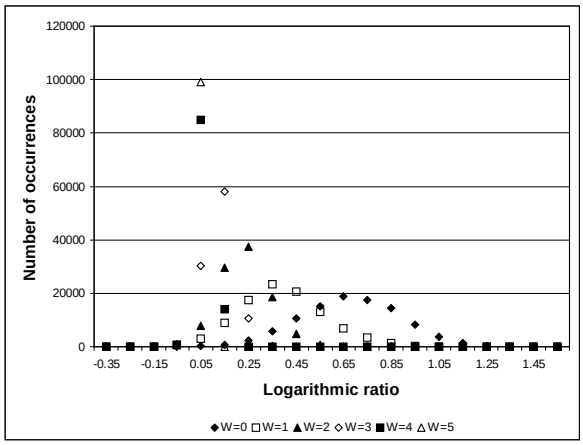

Figure 2. Absolute frequency of the logarithmic ratios of the gravity equation for various input interval widths.

relevant is that the difference between ILNS and IFP decreases with an increase in the width of the input intervals. This is expected, as the width of the input interval, which is equal for both number representations, becomes the most important factor for the width of the output interval. The interval widening due to roundoff errors, where ILNS gets its advantage, become less and less relevant. This effect is summarized in Figure 2, which shows the graph for each input interval width. For minimum-width inputs ($W = 0$), the logarithmic ratios present a Gaussian-bell-like distribution around 0.676. As the widths of the input intervals increase, the mean moves toward zero and measures are more concentrated around zero, as IFP and ILNS become almost equivalent.

5.3. Normalized distance computation

Calculating normalized distances is a common operation in graphic applications. It consists of normalizing each component of a position vector, $\vec{r}$, by dividing it by $|\vec{r}|$. This computation is similar to the one considered in Section 5.2, and again the normalization of only one component is simulated, since components are random and it does not matter which component is calculated. The equation that is evaluated is

$$x_{norm} = \frac{x}{(x^2 + y^2 + z^2)^{1/2}}, \qquad (22)$$

where $\vec{r} = (x, y, z)$. Values for x, y and z are randomly selected from the interval $[0, 1)$.

Equation (22) is similar to (20), yet there are important differences in the results. We performed the same analysis as in Section 5.2, and the results are presented in Table 2, with the simulation repeated for several input interval widths. The data show a behavior similar to Table 1, except the mean of the logarithmic ratios increases from 0.676 to 0.903 for minimum-width intervals. (22) favors ILNS over IFP more than (20), since (20) has three additions, rather than two.

Input Intervals	Output Logarithmic Ratio			
	Mean	σ^2	Max	Min
W = 0	0.903	0.083	1.736	-0.269
W = 1	0.499	0.041	1.494	-0.281
W = 2	0.297	0.016	0.878	-0.212
W = 3	0.164	0.006	0.459	-0.130
W = 4	0.086	0.001	0.222	-0.095
W = 5	0.044	0.000	0.119	-0.043

Table 2. Logarithmic-ratio statistics for the normalized distance equation, for 100,000 sets of random inputs.

Relative widths for each set of input interval widths, both for ILNS and IFP, again agree with the logarithmic ratios. These data show a similar behavior as (20) in Section 5.2, only with a larger difference in favor of ILNS. The output interval width increases by a factor of approximately 2 when W is incremented, showing how the input width directly impacts the output width.

6. Implementation

As described earlier, ILNS multiplication, division, squaring, and square root are trivial to implement using simple fixed-point circuits. For very limited-precision ILNS systems, direct lookup of $\nabla(s_b(z_l))$ and $\triangle(s_b(z_u))$ from precomputed ROMs is possible.

Daumas and Matula [9] have studied rounding of elementary functions in floating point. Their method involves the introduction of extra flags in the floating-point ALU for correct directed rounding and faithful rounding. Although such an approach could be used to obtain $\nabla(s_b(z_l))$ and $\triangle(s_b(z_u))$, the cost may be excessive.

Many approximations have been proposed in the conventional LNS literature, some of which support high precision

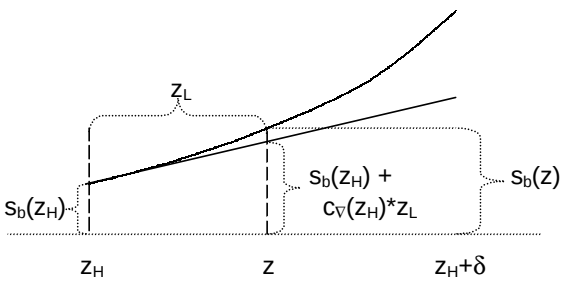

Figure 3. Linear-Taylor Interpolation.

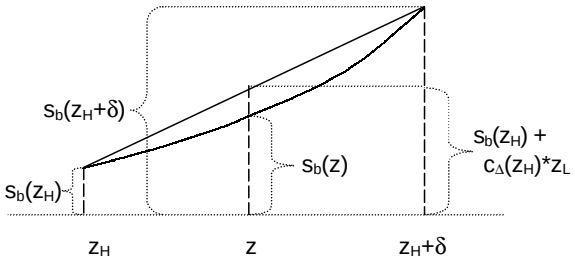

Figure 4. Linear-Lagrange Interpolation.

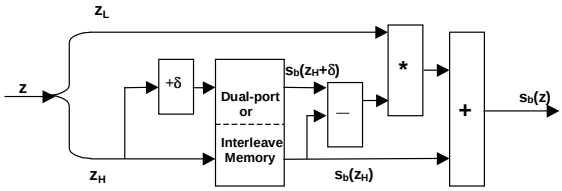

Figure 5. Lagrange Interpolation with Dual-port memory.

[7]. One way to approximate $s_b(z)$ that supports moderate precision is unpartitioned linear interpolation [6]:

$$s_b(z) \approx s_b(z_H) + c(z_H) \cdot z_L \tag{23}$$

where z_L is the low portion of z such that $0 \leq z_L < \delta$, z_H is the high portion of z such that $z_H \leq z < z_H + \delta$, and $c(z_H)$ is the slope for the line used in that linear-interpolation region. The pre-computed values of $c(z_H)$ and $s_b(z_H)$ can be stored in ROMs that require $N + K$ address bits each, where K is the number of bits for the integer part of the z_H bus, and N is the number of bits for the fractional part of the z_H bus. The term *unpartitioned* here means $\delta = 2^{-N}$ is the constant distance between tabulated points, in contrast to partitioned methods [16] in which δ varies in an effort to minimize memory.

As is apparent in Figure 3, linear-Taylor interpolation forms a lower bound:

$$\bigtriangledown(s_b(z_H) + c_\bigtriangledown(z_H) \cdot z_L) \leq s_b(z) \tag{24}$$

where $c_\bigtriangledown(z_H) = s'(z) = 1/(b^{-z} + 1)$. Similarly, as shown in Figure 4, linear-Lagrange interpolation forms a upper bound on s_b:

$$s_b(z) \leq \bigtriangleup(s_b(z_H) + c_\bigtriangleup(z_H) \cdot z_L) \tag{25}$$

where $c_\bigtriangleup(z_H) = (s_b(z_H) - s_b(z_H + \delta))/\delta$. The linear-Lagrange method has the further advantage that the $c_\bigtriangleup$ memory can be eliminated when a dual-port [6] or interleaved [17] memory is used to hold $s_b(z_H)$, as shown in Figure 5. Of course, this technique only works with Lagrange interpolation.

Although two separate prior art techniques like these represented in Figures 3 and 4 could implement ILNS, this paper proposes a novel technique that achieves the same effect using less memory. Rather than requiring separate memories to compute $\bigtriangledown(s_b(z_l))$ and $\bigtriangleup(s_b(z_u))$, the proposed technique (illustrated in Figure 5) is able to generate both correct upper and lower bounds from a memory that only contains $s_b(z_H)$. These tabulated $s_b(z_H)$ points are stored with an additional G guard bits, making the memory size $2^{K+N} \times (N + G)$ bits. Computing $\bigtriangledown(s_b(z_l))$ and $\bigtriangleup(s_b(z_u))$ from the same dual-ported memory eliminates about $2^{K+N} \times (2N + 2)$ additional bits that would otherwise be required to store $c_\bigtriangledown(z_H)$ and $c_\bigtriangleup(z_H)$ with the prior approach.

In the proposed method, the values of $c_\bigtriangledown(z_H)$ and $c_\bigtriangleup(z_H)$ are reasonably close to each other, and only need to be computed with about $N + 1$ bits of accuracy [4] in order for (23) to produce a reasonable approximation. Therefore, the proposed design computes $c_\bigtriangledown(z_H)$ from $c_\bigtriangleup(z_H)$, which, in turn, is computed using the two values obtained from the dual-ported memory. The key to this novel interpolation approach is the following relationship between the slopes of the upper and lower bounds:

$$c_\bigtriangleup(z_H) - c_\bigtriangledown(z_H) \approx s_b''(z_H) \cdot \frac{\delta}{2} = b^{\log_b(s_b''(z_H)) + \log_b(\frac{\delta}{2})}. \tag{26}$$

In order to capitalize on this relationship, we use a special property of the s_b function [5]:

$$\log_b(s_b''(z_H)) = z_H - 2 \cdot s_b(z_H) + \log_b(\ln b). \tag{27}$$

Substituting (27) into (26) we have

$$c_\bigtriangleup(z_H) - c_\bigtriangledown(z_H) \approx b^{z_H - 2 \cdot s_b(z_H) + \log_b(\ln b) + \log_b(\frac{\delta}{2})}. \tag{28}$$

For $b = 2$, $c_\bigtriangleup(z_H) - c_\bigtriangledown(z_H) \approx 2^x$, where $x \approx z_H - 2 \cdot s_b(z_H) - (N + 1.5)$ which can be implemented to sufficient accuracy using two $(K + G)$-bit adders. Furthermore, Mitchell's method [20] can approximate the base-two antilogarithm to sufficient accuracy (4 bits) as

$$c_\bigtriangleup(z_H) - c_\bigtriangledown(z_H) \approx 2^{\text{int}(x)}(1 + \text{frac}(x)) \tag{29}$$

using only a $(G + 1)$-bit shifter. Thus, a small amount of extra logic allows the same dual-ported memory to be used for computing the upper and lower bounds.

7. Conclusions

Sections 5.1 to 5.3 show the comparison between the interval widths obtained with IFP and ILNS for three algorithms. For gravity-force calculations in N-body simulation, ILNS improves the interval widths over IFP. IFP produces intervals an average of 60% wider, and the worst-case

relative interval width is roughly 2.5 times wider in IFP in our experiments. These facts support the decision to implement ILNS for applications where the proportion of multiplications and squares is high. The advantage of ILNS is even more pronounced in the normalized distance calculation, primarily due to fewer additions in the algorithm.

The FFT algorithm is not favorable toward ILNS, because of the low number of multiplications compared to additions, and the poorer representation that ILNS provides for twiddle factors. Furthermore, it appears the truncation required by ILNS is inherently inferior to the truncation required by IFP [18]. Despite these limitations, it has been shown that, on average for typical inputs, the FFT interval results are narrower with ILNS, although a particular interval output may be wider.

Implementation of multiplication, division and square root are significantly cheaper and faster in ILNS than in IFP. Addition and subtraction are more dificult. All but low-precision LNS implementations require table lookups and interpolation to implement addition. This is also the case for ILNS, but a naïve implementation would require separate memories for the upper and lower bounds of the s_b function. This paper has proposed a novel method to implement these upper and lower bounds using no more memory than required for a non-interval LNS.

References

[1] G. Alefeld and J. Herzberger, *Introduction to Interval Computations*, New York Academic Press, New York, 1983.

[2] M. G. Arnold, T. A. Bailey, J. R. Cowles and M. D. Winkel, "Applying Features of IEEE 754 to Sign/Logarithm Arithmetic," *IEEE Trans. Comput.*, vol. 41, pp. 1040–1050, Aug. 1992.

[3] M. G. Arnold, T. A. Bailey, J. R. Cowles and M. D. Winkel, "Arithmetic Cotransformations in the Real and Complex Logarithmic Number Systems," *IEEE Trans. Comput.*, vol. 47, no. 7, pp. 777-786, July 1998.

[4] M. G. Arnold and C. D. Walter, "Unrestricted Faithful Rounding is Good Enough for Some LNS Applications," *15th International Symposium on Computer Arithmetic*, Vail, Colorado, pp. 237-245, 11-13 June 2001.

[5] M. G. Arnold and M. D. Winkel, "A Single-Multiplier Quadratic Interpolator for LNS Arithmetic," *Proceedings of the 2001 International Conference on Computer Design: ICCD 2001*, Austin, TX, pp. 178-183, 23-26 September, 2001.

[6] M. Arnold, "Slide Rules for the 21st Century: Logarithmic Arithmetic as a High-speed, Low-cost, Low-power Alternative to Fixed Point Arithmetic," *Second Online Symposium for Electronics Engineers*, `www.techonline.com/community/20140`.

[7] C. Chen and C. H. Yang, "Pipelined Computation of Very Large Word-Length LNS Addition/Subtraction with Polynomial Hardware Cost," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 716-726, July 2000.

[8] E. I. Chester and J. N. Coleman, "Matrix Engine for Signal Processing Applications Using the Logarithmic Number System," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, San Jose, CA, pp. 315-324, 17-19 July 2002.

[9] M. Daumas and D. W. Matula, *Rounding of Foating Point Intervals*, Research Report no. 93-06, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Superiéure de Lyon, Lyon, France, March 1993.

[10] E. Hansen, *Global Optimization Using Interval Analysis*, Marcel Dekker, New York, 1992.

[11] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval Arithmetic: From Principles to Implementation," *Journal of the ACM*, vol. 48, no. 5, pp 1038–1068, September, 2001.

[12] O. Holzmann, B. Lang, and H. Schütt, "Newton's Constant of Gravitation and Verified Numeric Quadrature," *Reliable Computing*, no. 3, 1996.

[13] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, IEEE, 1985.

[14] N. G. Kingsbury and P. J. W. Rayner, "Digital Filtering Using Logarithmic Arithmetic," *Electron. Lett.*, vol. 7, no. 2, pp. 56-58, Jan., 1971.

[15] V. Kreinovich and A. Bernat, "Is Solar System Stable? A Remark,' *Reliable Computing*, vol. 3, no. 2, pp. 149-154, 1997.

[16] D. M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System," *IEEE Trans. Comput.*, vol. 39., pp. 1325-1336, Nov. 1990.

[17] D. M. Lewis, "Interleaved Memory Function Interpolators With Application to an Accurate LNS Arithmetic Unit," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974-982, Aug. 1994.

[18] J. D. Marasa and D. W. Matula, "A Simulative Study of Correlated Error in Various Finite-Precision Arithmetics," *IEEE Transactions on Computers*, vol. C-22, pp. 587-597, June 1973.

[19] J. Makino and M. Taiji, *Scientific Simulations with Special-Purpose Computers—the GRAPE Systems*, John Wiley and Sons, Chichester, 1998.

[20] J. N. Mitchell, "Computer Multiplication and Division using Binary Logarithms," *IEEE Transactions on Electronic Computers*, vol. EC-11, pp. 512-517, August 1962.

[21] R. E. Moore, *Interval Analysis*, Prentice Hall, Englewood Cliffs, N.J., 1966.

[22] R. E. Moore, "Computing to Arbitrary Accuracy," *Computational and Applied Mathematics I: Algorithms and Theory*, pp. 327-336, North-Holland, Amsterdam, 1992.

[23] V. Paliouras and T. Stouraitis, "Low Power Properties of the Logarithmic Number System," *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, pp. 229-236, 11–13 June 2001.

[24] N. Pollard and D. May, "Using Interval Arithmetic to Calculate Data Sizes for Compilation to Multimedia Instruction Sets," *Proceedings of the Sixth ACM International Conference on Multimedia*, pp. 279-284, 1998.

[25] M. Schulte and E. Swartzlander, "Exact Rounding of Certain Elementary Functions," 11th IEEE Symposium on Computer Arithmetic, Windsor, Ontario, pp. 138-145, June, 1993.

[26] M. Schulte and E. Swartzlander, "A Family of Variable-Precision Interval Arithmetic Processors," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 387-397, May 2000.

[27] G. L. Steele, Jr., *Common LISP, 2nd ed.*, Digital Press, 1990.

[28] E. E. Swartzlander, et al., "Sign/logarithm Arithmetic for FFT Implementation," *IEEE Trans. Comput.*, vol. C-32, pp. 526-534, 1983.

[29] A. Wrigley, *Real-Time Ray Tracing on a Novel HDTV Framestore*, Ph. D. University of Cambridge, England, 1993.