Implementing Algorithms in Fixed-Point Math on the Intrinsity™ FastMATH™ Processor

George P. Yost, Ph. D.

Floating point [sic] computation is by nature inexact, and programmers can easily misuse it so that the computed answers consist almost entirely of "noise." One of the principal problems of numerical analysis is to determine how accurate the results of certain numerical methods will be. There's a credibility gap: We don't know how much of the computer's answers to believe.

——Donald E. Knuth, *The Art of Computer Programming*, [7].

The Intrinsity™ FastMATH™ processor is an extremely fast computing engine optimized for parallel processing applications. A fixed-point machine, it can be used to process algorithms traditionally implemented in floating-point math. We discuss the issues that are important in implementing an algorithm in fixed-point math.

There are robust procedures for understanding how to do this. We describe useful principles and practices. These practices help to convert an existing algorithm from a floating-point implementation or design an implementation from scratch. Computational accuracy, which affects both fixed-point and floating-point math, is also addressed.

The FastMATH processor offers a 2-GHz cycle speed (1-GHz in the power-saving FastMATH-LP processor) and an efficient architecture for signal-processing applications. Featuring the reduced power consumption and improved instructions per clock cycle possible with a fixed-point machine, the processor rewards the developer willing to convert a floating-point application.

1. Introduction

There is a general need for a thorough discussion of the issues surrounding the implementation of algorithms in fixed-point math on the Intrinsity FastMATH processor. This should be of interest to people with a very broad spectrum of interest and experience. This paper includes the following:

- Some basic principles of floating- and fixed-point math
- Issues of overflow, choice of radix point (binary equivalent to decimal point), and scaling to avoid overflow or preserve accuracy
- A suggested methodology (see Section 4)
- Calculation accuracy, including statistical and mathematical issues

  (As you become more familiar with fixed-point math and as a fixed-point implementation progresses, the above sections become more valuable.)
- Examples that illustrate the concepts discussed
- A technique known as block floating point (BFP) (see Section 7)
- Two examples demonstrating analysis of accuracy issues (see Section 8)

The FastMATH machine offers 16-fold parallelism in a matrix unit coupled with a synchronous MIPS32™-compliant scalar unit. The matrix unit features both native vector and full matrix operation in a 4×4 configuration of processing elements that operate in single instruction, multiple data (SIMD) mode. The 2-GHz clock rate (1-GHz for the FastMATH-LP processor) affords up to 64 giga operations per second (GOPS) peak execution rate. Two bi-directional RapidIO™ ports, each capable of up to 1-Gbyte per second peak simultaneous data transfer rates in both directions, service the high-speed processing. There is a 1-Mbyte L2 cache that directly feeds data into 16 matrix registers. This cache is large enough that most signal processing applications are able to load one data set with the direct memory

access (DMA) unit while at the same time processing a second data set with the matrix engine. A pair of 40-bit multiply-accumulators (MACs), associated with each of the 16 SIMD processing elements, support single-issue multiply-add instructions. A sequence of such operations can be done with no intermediate round-off errors–an advantage over floating-point processors.

We describe the most important points in developing an algorithm for running on a fixed-point machine. From the programmer's point of view the principal difference between floating-point and fixed-point computing is that in floating-point computation the computer keeps track of the decimal points and in fixed-point computation the developer has that responsibility. For example, if we add 5.00 + 500.0, we must make sure we get 505.00, not 1,000. The decimal points have to line up for correct addition. In binary representation the "decimal point" is often called the "radix point." Floating-point math keeps track of all that for the developer, but there is a price.

Implementation of fixed-point arithmetic is much easier in hardware. The simplicity often means a faster cycle speed for a fixed-point machine. Often, there are more operations per clock tick because fixed-point operations may require fewer cycles. Finally, certain applications require "bit exact" results, which is very difficult except in fixed-point implementations.

For many embedded-processor applications a data stream with a limited number of bits emerges from a device such as an analog-to-digital converter (ADC) at a high rate. In such cases, the number of bits limits both the dynamic data range and the precision of the data. This kind of application is ideal for a fixed-point machine, where fast processing capabilities are important and the range of data does not require special handling. However, with proper attention to the issues discussed here, successful implementation of almost any algorithm is possible in a fixed-point mode. If the rounding errors discussed in Section 5.3, "Sources of Error in a Fixed-Point Architecture," are small in comparison with the granularity of the data, then precision may not be an issue. This is likely the case as long as the word size is sufficiently greater than the number of bits in the data. However, if there is right-shifting of the data or the intermediate results to avoid overflows (Section 3.1), then caution is necessary.

Floating-point math has its greatest advantage in working with numbers of very different magnitudes, i.e., a large dynamic range. The programmer's job is simplified and the accuracy may be improved because large numbers and small numbers are represented by the same number of bits. Fixed-point algorithms can also deal with large dynamic ranges. However, it's important to understand how. Fixed-point math forces the developer to pay more attention to the numerical properties of an algorithm. This investment in the development phase may pay off in other ways, such as in more efficient or more accurate algorithms.

We discuss a few of the issues involved in fixed-point versus floating-point arithmetic. Usually most of the complications are restricted to a few points where a small number of key procedures or library functions can be inserted, perhaps as macros. As an example, in the Cholesky decomposition algorithm (see Section 8.2), a 32-bit complex-number multiplication procedure repeats in every one of three stages, and a fixed-point division procedure repeats in two of the three stages. We can write either of these as a macro or as a separate function. Other than the flow logic, this is the bulk of the code.

Recall the principals of positional number systems such as the decimal system. For example, the number $LMN.XYZ_{10}$ is equal to $L \times 10^2 + M \times 10^1 + N \times 10^0 + X \times 10^{-1} + Y \times 10^{-2} + Z \times 10^{-3}$. The power of 10 is based on the position relative to the decimal. The digits L, M, N, X, Y, Z are selected from {0, 1, …, 9}. Similarly, in the common computer representation, the hexadecimal system, $LMN.XYZ_{16}$ would represent $L \times 16^2 + \ldots$, and so on; the digits belong to the set {0, 1, …, 9, A, B, C, D, E, F}. The binary system, where the digits come from the set {0, 1}, functions similarly.

In any positional system some numbers extend for a large number of places or never terminate. Forced to be approximately represented by a finite number of places, accuracy is lost. Only a finite set of numbers, separated by 1 unit of least precision (ULP), is capable of accurate representation. Numbers in between are forced to one of their neighbors according to some prescription.

Numbers that terminate in one positional system may not terminate in another. For example, in the decimal system the number 0.1, an exact terminating series, converted to binary representation becomes 0.0001 1001 1001..., a non-terminating series. Because both floating-point and fixed-point representations must have a finite number of bits, we must pay attention to the application's accuracy requirements. For suitable choice of word size and radix point posi-

tion (Section 3, “Fixed-Point Arithmetic”) the fixed-point form may make more bits available. We discuss accuracy issues in Section 5.

For algorithms that cannot conveniently be coded without a small amount of floating-point math, emulation software is available for the FastMATH processor. We also have the block floating point method (Section 7) available, which accommodates a large dynamic range within the fixed-point format.

As Proakis and Manolakis [9] point out, designing a high-performance algorithm requires consideration of computational complexity, memory requirements, and finite word-length effects. Computational complexity refers to more than the obvious; namely, the number of arithmetic operations the algorithm requires and the order in which they are performed. It also includes data transfers (e.g., to and from the L2 cache and SDRAM), which can dominate certain problems. Fortunately, the FastMATH processor reduces data movement time by using a 1-Mbyte L2 cache (optionally partly configurable as SRAM), a descriptor-based DMA controller, and two parallel RapidIO ports. It accomplishes the loading of a 64 byte FastMATH matrix register from L2 cache with a zero-cycle load-to-use penalty. Computational complexity also includes the parallelization issues fundamental to effective use of the FastMATH processor. The memory issues overlap with the data size factors and include consideration of the size of the code. The *Intrinsity Software Application Writer's Manual* [5] addresses memory and complexity issues for optimum algorithm choice and coding.

Choice of fixed-point representation can also affect algorithm choice. We start with a discussion of the alternative.

2. Floating-Point Arithmetic

The IEEE standard (IEEE 754, [4]) for the floating-point representation of numbers specifies the generally accepted format. Single-precision, floating-point representation in a 32-bit word calls for a sign bit, an 8-bit exponent, and a 23-bit significand. The significand is also called the “mantissa.” The IEEE standard also calls for the number to be normalized; that is, the exponent is chosen such that the highest-order bit in the significand is a 1. Since this is always the case[1], the standard stipulates omission of that fixed-value bit, effectively providing a 24-bit significand. This number of bits sets the limit on the precision of single-precision floating-point arithmetic. One of the advantages of floating point numbers is that the full 24 bits of accuracy is always available for storing a number. Another strength is that, thanks to the automatic exponent, this supports a large dynamic range of values.

Accuracy remains an issue in mathematical operations. If we add two numbers with different exponents, the one with the lower exponent shifts to the right by enough bits to line up the radix points before addition. Bits shifted off the right side are lost.[2] The sum retains the most significant 24 bits of the result. The order in which operations are performed is important. A summation involving a large range of values is most accurate if the smallest values are handled together, before they lose low-order bits when added to larger numbers. Then the carry bits from their sums are accurate. Otherwise, losses of bits carried up from low-order positions can accumulate and become significant. We cannot necessarily assume that the results of a sequence of operations are accurate to 24 bits.

Floating-point multiplication requires that the significands be multiplied and the exponents added. The numbers are then re-normalized.

The IEEE standard provides for reproducibility. The result of any given IEEE-compliant floating-point calculation will be identical on any machine, regardless of the machine's architecture.

To avoid negative exponents the standard calls for the exponent to be biased. That is, for a single precision number the value 127 is added to all exponents. Then an exponent of 1 represents 2^{-126} and an exponent of 255 represents 2^{+128}. An exponent of 0 and a significand of 0 represents a value of 0. Note that a number that is an exact power of 2, 2^n, will have $n + 127$ in the exponent and all zeros in the significand, since the normalization bit, 1, is implied.

IEEE floating-point numbers are represented in sign-magnitude format, where the representation of a given number and its negative differs only in the sign bit: a sign of 1 signifies a negative number.

1 With the exception of very small numbers represented in a denormalized form to be addressed.
2 The least significant bit may be rounded after shifting, if desired, saving some accuracy.

Many architectures do not implement the full IEEE 754 standard. These architectures work the same but without some additional features needed for some applications. For example, to represent numbers smaller than possible with normalized numbers,[3] the full standard permits a "denormalized" representation. In such numbers, the exponent is set to 0 and the significand is not normalized. Because some significance is retained even when the results of a computation are extremely small, this practice is sometimes called "gradual underflow." This can be of value in some applications, but may not be needed with sufficient care in code design. It also causes great complications in hardware design. Therefore, simpler designs omit this feature. Underflow results are set to 0.0 and underflow data on input is treated as 0.0. However, Donald Knuth[7] recommends against this practice. Since there is no generally adopted practice for guarding against exponent overflow, this generates one of five possible "exceptions" in the full IEEE standard.[4] However, even with full compliance, most high-level programming languages do not provide a satisfactory way to test for this exception. On many architectures the developer who suspects that either overflow or underflow may occur can use double-precision floating-point arithmetic. This option, if available, offers both increased range of values through a greater number of bits in the exponent (11) and increased accuracy through a greater number of bits in the significand (53, counting the implied normalization bit). In many processors, double-precision mode reduces execution speed.

3. Fixed-Point Arithmetic

With a machine like the FastMATH processor, we can think of the word as an integer, but we also have the freedom to picture the radix point as being at any place in the word. Thus, we can write a 32-bit, fixed-point number as (s15.16) or as (s7.24). In this notation the "s" refers to the sign bit, the number to the left of the point is the number of bits in the integer part (not an exponent, as in floating point), and the number to the right of the point is the number of bits in the fractional part of the number. For example, in a 16-bit word the number 0000 0001 1010 1001 can represent $+425.0_{10}$ in (s15.0) format or $+13.28125_{10}$ in (s10.5) format. We shifted the radix point five places to the left, which is equivalent to dividing by 32. The responsibility for keeping track of the radix point is transferred from the hardware to the programmer. However, for applications intended for frequent use, this effort is amortized over the useful life of the code and may be more than rewarded with performance.

Negative numbers are represented in two's complement arithmetic on the FastMATH processor. A number that is the negative of a positive number x is obtained by inverting every bit in x and then adding 1. This is equivalent to subtracting the positive number from 2. Thus, in the 16-bit example in the previous paragraph, the negative of 0000 0001 1010 1001 becomes 1111 1110 0101 0111, regardless of the position of the radix point. To convert a negative number x into its positive equivalent, carry out the same steps: invert all bits and add 1. Carry bits off the left are discarded. In the most significant bit, all negative numbers have a 1 and all positive numbers have a 0, which explains the designation of that bit as the sign bit. The FastMATH processor also provides for unsigned numbers in which the highest-order bit is created as part of the number rather than as a sign. We use fixed-point notation of the form; e.g., (16.16) or (8.24) for unsigned formats.

In two's complement arithmetic addition is carried out bit by bit, including the sign bit, just as if the numbers were unsigned. Carries that overflow off to the left are always discarded. For example, the sum of the positive number 0 1111 = $+15_{10}$ in (s4.0) format (a sign bit and a 4 bit integer, no fractional part) and the negative number 1 0010 = -14_{10} is 10 0001; with the overflow carry bit in the left-most position dropped, the result becomes 0 0001, as desired.

Multiplication is also carried out bit-by-bit, except that the sign bit is special when the operation involves signed numbers. See Section 6, "Example: 32-bit Multiplication in (s15.16) Format" for examples.

Two's complement arithmetic ensures that $x + (-x)$ does equal 0 and that there is only one representation for 0.[5] We can verify that the sum of these is exactly 0 (dropping the carry bit shifted off to the left). The most negative number is 100…000; it is the only number for which there is no corresponding positive number. Inverting every bit in this number and adding 1 returns the same number. In 32-bit arithmetic in (s31.0) format (with the radix point at the far

3 Smaller than 1×2^{-126} for single-precision numbers.
4 Does not necessarily apply to non-standard implementations.
5 Opposed to a different positive and negative zero, which occur in other schemes.

right) this is -2,147,483,648. In this representation the number -1 becomes 111…111. Adding 1 results in 0, with the carry bit discarded to the left.

A negative number right-shifted with sign extension (arithmetic right shift) fills the left-side vacated bits with 1s. For example, an arithmetic right shift of 0000 0001 1010 1001 by 2 bits yields 0000 0000 0110 1010, and an arithmetic right shift of its negative 1111 1110 0101 0111 yields 1111 1111 1001 0101. Any sign-extended bits become part of the number and are used in all arithmetic operations. Further examples appear in Section 6.

While floating-point numbers always feature a sign bit, fixed-point numbers may be unsigned, freeing up the sign bit. In our notation, an unsigned 16-bit integer is represented as (16.0), e.g., (10.6) represents a number with six fractional bits. The FastMATH architecture includes instructions for unsigned arithmetic, as provided for in the C programming language.

3.1. Fixed-Point Overflow

In computations, if necessary we can lose low-order bits off to the right, but we can not lose bits off to the left. That situation is known as overflow and can be the result of arithmetic or shifting operations. Overflow can also take the form of moving a numerical bit into the sign bit, also usually with catastrophic consequences. There is no exception trap for overflow. The software must be constructed so as to avoid it or, at a minimum, detect it. The FastMATH processor offers means for detecting the condition in some cases. However, best performance is achieved if the code is constructed to avoid it. The most common method is to scale the input data and, perhaps, also the results of intermediate steps, so that overflow cannot occur no matter what the input data. The FastMATH multiply-accumulators (MACs) contain 40 bits to protect intermediate results from multiply-add operations from overflow up to a certain number of accumulations.

An important strength of two's complement arithmetic is that the sum of a sequence of numbers is correct as long as the *final result* lies within the range represented, even though intermediate portions of the sums may overflow. For example, in (s4.0) representation, the sum 0 1111 + 0 1100 + 1 0010 [$+15_{10} + 12_{10} + (-14_{10})$] becomes 10 1101 and, as always, the overflow carry outside the five bits is dropped. The final result is 0 1101 = $+13_{10}$, as desired. The sum of the first two terms, both positive, overflows into the sign bit: 0 1111 + 0 1100 = 1 1011, a negative number (-5_{10}).[6] The carry bit from the second addition is needed to complete the series, (-5_{10}) + (-14_{10}), 1 1011 + 1 0010 = 10 1101 → 0 1101. In effect, overflows beyond the sign bit transport the previous overflow bit off to the left, where it is dropped. Here, we add two negatives and get a positive result. Thus, even though the intermediate sums would not have been correct separately, we obtain the correct final result. If the third term in the series had instead been (-11_{10} = 1 0101), there would have been *another* carry into the sign bit, and the cancellation would not have worked. This is because that final result, $+16_{10}$, is not in the range representable by the (s4.0) format (its negative, the largest possible negative number, 1 0000, is the number for which there is no positive equivalent).

This feature of two's complement arithmetic is a difference with floating-point math. In floating-point calculations, normalization by shifting occurs for each partial sum. Thus, overflow is avoided at every stage, but at the possible expense of bits that might be useful at later stages. In fixed-point math, shifting is necessary only if overflow of the final result is anticipated.

One approach to overflow conditions is saturation arithmetic, which replaces faulty values resulting from overflow with the largest value that the word permits. The FastMATH processor has three instructions for halfword saturation after addition or subtraction. Under most circumstances this provides a reasonable safety valve for overflow. Accuracy may suffer, but the general calculation stays on course and the result of a sequence of operations may still satisfy requirements. Because only the final result of a sequence of additions is of interest, do not employ saturation prematurely.

The FastMATH matrix processor supports 16-bit and 32-bit addition and subtraction, as well as 16-bit multiplication. Division in the matrix unit is by software only.[7] Multiplication of 32-bit numbers is done by breaking the numbers

6 The correct result, 27_{10}, is not representable using (s4.0) format.

7 The FastMATH processors support hardware division in the scalar unit consistent with the MIPS32 standard.

into 16-bit halves, performing 16-bit multiplications of all the combinations of halfwords, and summing the results. For details, see Section 6, “Example: 32-bit Multiplication in (s15.16) Format.”

Table 1 describes the overflows possible from these operations. For example, addition of two four-bit numbers can yield a 5-bit number due to a carry; e.g., 1111 + 1111 = 1 1110. Addition of two unsigned numbers can yield a carry bit. Two additions can yield a maximum of two carry bits, and four additions can yield a maximum of three carries. This is not to say that these carries necessarily occur, but we need to prepare for that possibility. If the data to be added fills the highest order bits, the carries can become overflows. Signed 16-bit words have 15 bits in the significand, and their sum or difference can yield a 16-bit result, plus the sign. That is, any overflow from the sum or difference occupies the bit being used as a sign bit, the true sign bit shifts to the left out of the halfword, and the result is incorrect. To avoid overflow, we may pre-shift the data to the right, transferring the loss of bits to the least significant instead of the most significant bits.

Table 1: Worst-case possibilities for overflow in fixed-point math

Operation	Details	Max number of resultant bits
Addition or subtraction of m-bit unsigned words	2^n terms in sum; $n > 0$	m bits and $n + 1$ carry bits
Addition or subtraction of m-bit signed words	2^n terms in sum; $n > 0$	m bits, including sign, and up to n additional carry bits[a]
Multiplication	Two 16-bit signed words	30 bits and sign bit
Multiplication	Two 32-bit signed words	62 bits and sign bit
Multiplication	Two 16-bit unsigned words	32 bits
Multiplication	Two 32-bit unsigned words	64 bits

a. Sign bit is the first bit overwritten by any overflow.

3.2. Radix Point

The position of the radix point does not affect the number of bits in the result of an arithmetic operation. However, the final result must be stored back into a register or memory location. Then the result might need to be shifted to preserve the location of the radix point. For example, since there are no bits in the integer portion, in (s0.15) format for a 16-bit signed value, the radix point is located such that all numbers are less than 1. This may be accomplished by prescaling all data, if necessary. That is, the data are multiplied by a common factor, usually a factor of 2 so that the multiplication (or division) is accomplished by shifting. Section 6, “Example: 32-bit Multiplication in (s15.16) Format,” on page 22 illustrates this behavior.

Multiplication of two (s0.15) numbers yields 30 significant bits, plus a sign bit. The FastMATH processor provides two 40-bit multiply-accumulators to accomodate a sum of a sequence of such 16-bit multiplications, as in a dot product. Since the inputs are less than 1, the products must also be less than 1. Each individual product is in (s0.30) format, with a total of 31 bits needed.

In order to re-format a single (s0.30) product in (s0.15) format, it is necessary to right-shift the result 15 bits. This does not scale the number. Instead, it moves the radix point to the desired location in the word. This completely avoids overflow in multiplication, but at the expense of the 15 least-significant bits of the product. If summing multiple products, as in the dot product, these may be accumulated in the multiply-accumulator (MAC) without having to be moved back into a register. Carry bits may occur and use the high bits of the accumulator. The extra bits in the accumulator are sometimes called guard bits. The guard bits allow the accumulation of a series of multiplications before overflow becomes a concern. Thus, the accumulator protects against loss of the high-order bits of a dot product (until the 40 bits are used, following the prescriptions of Table 1). These bits must be recovered and right-shifted

in order to subsequently fit into (s0.15) for storage.[8] The scaling factors must be preserved so the scaling may be undone at the end. Section 7, “Block Floating Point” discusses the use of temporary scaling factors for blocks of data.

Section 5 addresses accuracy in more detail. In general, 32-bit, fixed-point arithmetic has the potential to give more accurate results than floating-point arithmetic, which offers only 24 bits of precision, provided that the range of the data is not too large. Counting the sign bits, the potential advantage over floating-point math is 7 bits, or a factor of 128. This advantage is realized if all bits are used. Note that floating-point numbers always use their full 24 bits, even if the lowest order bits may be zeros. If the range of the data or of the intermediate results is larger than a factor of 128, then some fixed-point terms may use less than 24 bits. In that case, it is possible that accuracy may not be as good as for floating point, unless magnitude-dependent shifting is done, as for block floating point math (see Section 7).

Thus, fixed-point accuracy depends on the degree to which the bits in the words are utilized and under what circumstances. For example, summations of unsigned numbers can use all 32 bits if the radix position is chosen so that the final result in the sum uses all 32 bits. Smaller numbers in the sum may lose some of their precision if they must be right-shifted to line up the radix points, but the resultant has 32 bits of precision. Multiplications take on the precision of the least-precise multiplicand.

The code must address these considerations. If additions or subtractions are part of the algorithm the radix point must stay at the same point in all summands. Which fixed-point representation we choose (i.e., where we place the radix point) depends on the number of bits we need for the integer part, the most significant part. It is sometimes convenient to choose the radix point at a byte boundary to facilitate shifting operations.

3.2.1. Scaling

Once we choose where the radix point will lie we are always free to apply a scale factor to make our data fit within that format. Scaling most often takes the form of a shift (equivalent to multiplication or division by a power of 2). All terms in a sum or difference calculation must, again, have the same scaling, enforced by the programmer. In some algorithms a common scale factor can be applied to blocks of data, with perhaps different scale factors applied to different blocks. Section 7, “Block Floating Point,” describes this. The scale factors are stored in separate locations in memory or hard-wired into the code. Although this can be at times inconvenient, it allows the programmer to choose a radix point that efficiently uses the available bits and avoids overflows. The programmer may then adjust the radix point as blocks of data require.

4. Methodology: Fixed-Point Implementation

There is a clean methodology for porting an application from floating-point to fixed-point. We summarize some of the main points here. The individual developer may, of course, prefer to vary these steps according to individual need. However, we recommend that these points be given serious consideration.

- Develop a working version of the algorithm with floating-point math. This is not only convenient for rapid application development, it provides experience with the algorithm. Once verified, it can confirm the operation of the fixed-point version. Performance is not an issue here. Usually, emulated floating point in scalar (non-SIMD) mode on the FastMATH processor is satisfactory at this stage.
- Consider simulating the problem and algorithm at this stage with an appropriate tool. Numerical properties of the algorithm for random input can be studied at length.
- Examine the data and intermediate results to establish the fixed-point convention. The most important choice is between a 16-bit word size and a 32-bit word size. The 16-bit word size will provide faster execution, but may impact accuracy (Section 5, “Accuracy). It is, of course, possible to use 16-bit words in some portions of the algorithm and 32-bit words in other parts. Obviously, that affects complexity and, therefore, development time and code readability.

8 This is scaling because the location of the radix point is shifted to the left of the significant bits.

If divisions are involved, these tends to expand the number of bits needed. For example, division of one 16-bit word by another (non-zero) yields results with values ranging from up to a full 32 bits (31 with sign) to as little as 1 bit in the lowest order position. This does not depend on the location of the radix point in the numerator or denominator (as long as it is the same in both). In that case, 32-bit words in a symmetric (s15.16) or unsigned (16.16) format may be advisable for the quotient, even with 16-bit input, unless the developer can be sure that all divisions will yield a smaller range. The important point is that divisions tend to double the dynamic range. For overflow protection in a division, scale the quotient after division, expand the number of bits available, scale the numerator and denominator differently, or adopt some other technique.

Multiplications also expand the number of bits involved. The difference between handling multiplications and handling divisions is that down-scaling both terms in a product by 2^z scales their product by 2^{2z}, but does not change their quotient at all. Therefore, scaling of input data or intermediate results is effective in a multiplication to avoid potential overflow bits at the cost of low-order bits (or the opposite, up-scaling to increase the number of bits used). Then the operation to fit the product into a fixed word size has the smallest possible impact. There is some unavoidable cost in precision, but with careful handling it is usually tolerable.

It is often most advisable to place the radix point between bytes, to allow byte-shifting of products (or quotients) to recover the radix point location. (See Section 6, “Example: 32-bit Multiplication in (s15.16) Format.”)

- Examine the algorithm in its floating-point implementation to determine points at which overflow might be encountered, given the choice of fixed-point representation. Examine the condition number (Section 5.4) for a number of test cases to evaluate the risk of overflow. Determine if scaling is necessary. This can take the form of one-time prescaling of each set of input data, or stage-by-stage scaling of intermediate results in a multi-stage process such as a fast Fourier transform (FFT). Scaling can either decrease large values for overflow prevention or increase small values for better use of available bits.

  At this point, it is sometimes advisable to reassess the choice of fixed-point implementation. Understanding the data is important. The most general adaptation of an algorithm may require very conservative scaling to guard against the worst case. However, if it is known that the range of admissible data permits, it may be possible to reduce the amount of scaling and preserve more of the least significant bits.

  It is sometimes advisable to construct a complicated algorithm in blocks and tailor any scaling for each block separately.

- Construction of a standard C-language fixed-point scalar version of the algorithm is sometimes advisable; that is, one without the SIMD (matrix unit) implementation. This simplified version allows the developer to run the fixed-point math through its paces and can confirm its numerical properties before investing time in parallelizing the algorithm. Use typical data, if possible. Run as many cases as practical and examine each case for problems in comparison with the floating-point version of the algorithm. Develop and run “corner cases,” designed to stress the accuracy properties of the code. The object is to test the fixed-point properties of the solution, not to optimize execution speed.
- Once the fixed-point performance of the algorithm is established, construct the actual SIMD code in C, using Intrinsity C-language constructs that implement the matrix unit instructions (an example appears in Figure 4 on page 25). Compare output with corresponding output from the floating-point or scalar fixed-point code developed above. As before, run as many cases as practical, especially the “corner cases.”
- Optimize the execution time of the algorithm, using practices described elsewhere[5]. A final check to confirm that the optimized algorithm continues to satisfy the specified accuracy requirements is recommended.

5. Accuracy

Calculation errors for computing algorithms arise from what are collectively known as finite-word-length effects. These include quantization errors, which are shifting errors, rounding errors, or truncation errors. These are sensitive to data or intermediate result precision, algorithm choice, calculation sequence, rounding mode choice, scaling, and, perhaps, other factors that depend on the data itself. These same types of errors affect both fixed-point and floating-

point calculations, but in different ways. Because the results must be stored in a fixed-word-length, there is a fixed quantum of resolution; hence the characterization of this error.

There are many other types of errors that may be more familiar to engineers performing the measurements and designing the algorithms. These errors can be due to such things as accuracy of measurement devices and analog/digital (A/D) converters, noise introduced by the system (such as interference and fading over the air link for transmission systems), thermal noise, premature truncation of the calculation of iterative algorithms, simplifications in the mathematical model, flaws in the mathematical model, and even human errors. These other sources are the same whether we are using fixed-point or floating-point math. However, they may have different affects on a fixed-point versus a floating-point implementation of an algorithm. We cannot leave them out of the picture. In addition, the mathematics describing these types of errors has a lot in common with the mathematics for computational errors. Therefore, though we discuss the management of fixed-point computational errors, we demonstrate statistical techniques relevant to all types of errors.

Floating-point arithmetic has the flexibility that 24 bits of precision are retained until a mathematical operation is performed. Only in the course of an operation may a number have to be shifted so the radix points line up. Fixed-point arithmetic does not have that inherent flexibility. When data or the results of mathematical operations are stored, the radix point location determines how many bits are retained.

We distinguish between relative errors and absolute errors. If variable x has an error Δx, that is its absolute error and $\Delta x/x$ is the relative error. An incorrect bit in the least significant position is an error of 1 ULP. The size of 1 ULP is also known as the resolution. A 1-ULP error in a (normalized) floating-point number is a relative error ranging from $1/2^{24} = 6 \times 10^{-8}$ to $1/2^{23} = 12 \times 10^{-8}$, varying periodically between these limits as the value increases. That is, 1 ULP is 1 bit out of 24 and, for a given exponent, its size relative to the significand varies with the 24 bits of the significand, keeping in mind that the most significant bit omitted from the word is always 1. Phrased another way, 1 ULP is a fixed absolute error for a given exponent, but for a given significand 1 ULP is a fixed relative error. It is approximately correct, and simplest, to view 1 ULP as a fixed relative error, roughly 10^{-7}. The absolute error depends on the exponent. For large numbers, the resolution is coarse; for small numbers it is fine.

In fixed-point math a 1-ULP error is a fixed absolute error whose value depends on the location of the radix point. Moving the radix point exchanges dynamic range for resolution or vice-versa. The relative error depends on the number of bits being used in the number. A 1-ULP error in a fixed-point number with $m > 0$ significant bits, excluding sign (i.e., the highest 1-bit is in the m^{th} position), is a relative error of $1/2^{m-1}$, or 9×10^{-10} if $m = 31$.

For both fixed-point and floating-point math the 1-ULP precision leads to a potential error of as much as ±1 ULP in the representation of a number. The exact error depends on how the least significant bit (LSB) is determined after a mathematical operation: truncation, rounding, or convergent rounding (see Section 5.3, “Sources of Error in a Fixed-Point Architecture.”) Again, because this is an irreducible quantum of precision, it is known as a quantization error. For example, the representation of decimal 0.1 mentioned previously has an *absolute* error of slightly more than 1/2 ULP in (s0.8) if truncation is used (0.0001 1001 1001… is truncated to 0.0001 1001), but slightly more than 1/4 ULP in (s0.6). However, 1 ULP in (s0.8) is 1 part in 2^8, and in (s0.6) it’s 1 part in 2^6. The *relative* errors are much larger than the absolute errors because the number of significant bits does not include the leading 3 zeros. The binary number 0.1111 1111 1… has an absolute (and also relative) error of approximately 1 ULP if truncated anywhere within the stream of 1s, the size of the ULP depending on the truncation location.

For fixed-point calculations, once the maximum data value and the maximum values at intermediate steps in the calculation are known, the radix point can be chosen to maximize the number of bits used. The relative error of a fixed-point algorithm depends on the number of significant bits at the stages of the calculation, not the location of the radix point.

If an algorithm can efficiently use 32 bits in a fixed-point number, signed or unsigned, then the fixed-point calculation potentially has an accuracy advantage over the 24 bits plus sign available in floating-point numbers. A given algorithm may or may not be able to make full use of this potential. For example, if as much as one data point at a key stage of the fixed-point computation has lost, say, all but 10 bits or so, then that sets an accuracy limit for all multiplications or divisions that involve that data point. Loss of significant bits can occur, for example, due to scaling during computation—scaling intended for worst-case overflow protection might be overly penalizing for average-case data.

If this is common, one might look to dynamic scaling (at a performance penalty) or perhaps block floating point, Section 7.

So, the factors determining accuracy in a fixed-point machine are the distribution of the number of significant bits in the data values and intermediate results and how these are used. An algorithm that only involves sums or differences of numbers may well be more accurately done in fixed-point math.

One of the considerations the programmer must bear in mind is any parts of an algorithm where divisions or functions such as square roots must be evaluated. These are usually done by a method of successive approximation. The FastMATH processor allows 16 of these operations to be performed in parallel, thus amortizing the time to a large degree. Nevertheless, there is usually a performance advantage to choosing an algorithm that reduces dependency on such operations.

Mathematically equivalent algorithms may not be numerically equivalent. For example, a sum of numbers with a large dynamic range can be done in any order if infinite precision is available, as in a mathematical expression. Summing them on a finite-word-length machine may cause smaller numbers to be completely lost when shifted to align radix points. This symptom applies to both floating- and fixed-point math. As mentioned previously, it is usually superior to group the smaller values together and sum them with a radix point position appropriate to their values, then applying the final shifting to their sum. That way, any carry bits from the smaller values are accurately calculated before the original values are shifted away.

5.1. Statistical Treatment of Errors

Calculation errors are deterministic in that the same data, manipulated the same way, always yields identical calculation errors. However, the data varies randomly. This means the calculational errors do likewise from one data set to the next. It therefore makes sense to treat these errors as random variables and analyze them statistically.

Understanding of errors is important in understanding the operation of an algorithm. Key elements are the error variance, its bias, correlations among errors, and correlations between errors and signal values. The most straightforward means of testing is to prototype the same algorithm in emulated floating-point code as described in Section 4, “Methodology: Fixed-Point Implementation,” and input randomized data. This also serves the purpose of facilitating debugging of the algorithm for the FastMATH unit. We need to be alert to the fact that floating-point math is also subject to errors as mentioned in Section 5, and, if the problem is ill-posed (Section 5.4), these errors may propagate through the algorithm and yield significant errors in the results. Ill-posedness affects both fixed-point and floating-point versions of the same algorithm, but in different ways.

We note that testing of algorithms with random input may not efficiently uncover all numerical problems. With sufficient running, of course, any random input test process eventually finds all the bad cases, but it is sometimes more efficient for the programmer to insert judgement and identify and directly implement tests of “corner” cases, as mentioned in Section 4. These are cases where the algorithm is expected, based on the developer’s understanding of the algorithm, to show the greatest numerical instability.

5.2. Definitions; a Little Statistics

The rms (root mean square) error of a sequence of N terms, each subject to error, is the square root of the mean square error (MSE):

$$\mathrm{MSE} = \frac{1}{N}\sum_{n=0}^{N-1} (\mathrm{error}_n)^2 \quad , \qquad \textit{Eq.(1)}$$

where estimation of the error is the hard part. In this expression we have assumed that the error in every term is independent of every other term. For a single term we can write the MSE as

$$MSE = E[(x - x_t)^2] = E[x^2] - 2x_t\mu + x_t^2 \quad , \qquad \textit{Eq.(2)}$$

where the $E[\,]$ operator represents the expectation value, x is an estimate (measurement) of a quantity for which x_t is the (usually unknown) true value, and μ is the expected (mean) value of the measurement: $E(x) = \mu$. Note that μ does not necessarily equal x_t. In that case, we say that the measurement has a bias b, $b = \mu - x_t$.

The variance of a random variable x represents the square of the random spread of x about its mean value μ (not about the true value x_t, unless the bias is zero):

$$V(x) = E[(x-\mu)^2] = E[x^2] - \mu^2 \quad , \qquad Eq.(3)$$

where the second equality follows from the definition of μ. The square root of the variance is known as the standard deviation (often denoted by σ). As a reference, if the random variate is distributed as a Gaussian, then the interval $[\mu - \sigma, \mu + \sigma]$ contains approximately 67% of the probability. Note that if the bias is large we could have a situation where the data x cluster with small variance, but around a mean value μ that is far removed from where we would want it, x_t. Even though the variance may be small, if the bias is large the error is large. Simple algebra gives us

$$MSE = V(x) + b^2 \quad . \qquad Eq.(4)$$

Therefore, the MSE incorporates the important bias term as well as the random errors. You can often use its square root, the rms error, to avoid dealing with squared quantities. The rms gives a linear representation of the scale of the problem. If we run a large number of simulations we can get a good estimate of the means, the biases, and the variances. Although many workers prefer the lowest possible bias solution in a given problem, occasionally you can achieve a better MSE by accepting a procedure that gives a small bias. The biased procedure may gain more by reduction of variance than it loses by a non-zero bias. In a computing environment, a small bias may be an outgrowth of the management of finite-word-size effects (Section 5.3), and many people accept that in order to avoid seriously impacting performance. It is worth noting the existence of problems for which a bias is unacceptable, even if it results in better MSE. Since the bias is the expected (mean) value of the error, in some problems it can be calculated. That opens the possibility that the final results can be adjusted for the bias for reduced MSE.

In an ideal situation, the variance and the bias do not depend on the data. We may have to deal with exceptions to this situation, in which case we cannot predict the error without knowing the data. We also must take into account the fact that computational errors do not arise from the measurements themselves but from the operations performed on them.

5.2.1. Fluctuations in Data and Output Results

There are at least three types of random fluctuations in the problem. The largest is usually the fluctuations due to the randomness of the data from one data sample to another. This has nothing to do with measurement or computation error because it constitutes the new information in the new signal. It may, of course, include random errors due to factors to be discussed next.

The second type is the errors listed at the start of Section 5 due, for example, to inaccuracies in an ADC. That is partly a matter for the equipment designer to address. It may be part of the job of the algorithm to remove as much as possible of these sorts of error, by smoothing or, for example, filtering the data.

We are concerned here with the third type, that due to errors in processing. We want these to be small compared with the variations in the data. Otherwise, the information in new data is difficult to extract. The exact computation errors can only be known if we know the exact data and sequence of computations. If we can predict the error we can correct for it; for example, as for bias.

At this point we have not discussed signal-to-noise ratios (SNRs), a topic to be addressed in Section 5.8. In a general sense “noise” is the net effect of the errors in measurement and processing. In this document, noise refers only to processing errors. Processing errors may be measured by running the same data through the same algorithm with a higher precision calculation. This can take the form of double-precision floating point (emulated in the scalar processor). In some cases, single-precision floating point may suffice.

5.2.2. Propagation of Errors

We can characterize the accumulation of errors through a multi-step functional relationship by the propagation of errors technique. If x is a measured variable with error Δx, and $z = c\,x$, where c is a constant, then the error in z is Δz

$= c\,\Delta x$. If the relationship is not linear, $z = f(x)$, we can derive a relationship assuming approximate linearity in the vicinity of the observed value of x,

$$\Delta z = \frac{df}{dx}\Delta x \quad . \qquad \textit{Eq.(5)}$$

This is reasonably accurate provided Δx is small compared with the scale of non-linearities in f. If f is a function of multiple variables (x_i, x_j, …), then we can similarly propagate the errors in the input variables:

$$V(z) = \sum \frac{\partial f}{\partial x_i}\frac{\partial f}{\partial x_j} V(x_i, x_j) \quad . \qquad \textit{Eq.(6)}$$

All derivatives and the variance-covariance (usually abbreviated as covariance) matrix $V(x_i, x_j)$ are evaluated at the observed values x_i, x_j, …. The "diagonal" terms $V(x_i, x_i)$ are usually written as $V(x_i)$, representing the variance of x_i. With this relation we incorporate covariances (the off-diagonal elements) among the measured values x, if any. If we have multiple functions of a vector $x = (x_1, \ldots, x_{N-1})$, $z_1 = f(x)$, $z_2 = g(x)$, …, the variance of each is derived from Eq.(6) (replacing f by g as appropriate). In general, in vector/matrix notation, if $z = (z_1, z_2, \ldots)$ and $V(z)$ is the covariance matrix of the vector z, then

$$V(z_m, z_n) = \sum \frac{\partial z_m}{\partial x_i}\frac{\partial z_n}{\partial x_j} V(x_i, x_j) \quad . \qquad \textit{Eq.(7)}$$

Here, $V(z_m, z_n)$ is the covariance between z_m and z_n.

The covariance matrix is positive definite with positive diagonal elements $V(z_m)$ greater than or equal to the absolute values of all off-diagonal elements in the same rows and columns. This allows us to normalize the variables to their variances and define the correlation

$$\rho_{m,n} = \pm\sqrt{\{V(z_m, z_n) / [V(z_m)V(z_n)]\}} \quad . \qquad \textit{Eq.(8)}$$

The correlation $-1 \le \rho_{m,n} \le 1$ takes the sign of the covariance and expresses the linear variability of one variable with respect to variations in the other. As is well-known in statistical practice, a non-zero correlation does not necessarily signify causality in either direction. That is, causality signifies that variations in one are the cause, or partial cause, of variation in the other. However, both z_m and z_n can be influenced by other variables that cause them to vary together. Zero correlation means that there is no net linear dependence of one variable with respect to the other. It does not necessarily mean that the variables are independent of each other; there could be a non-linear dependence with an average value of zero.

Propagation of errors is widely used in data analysis to propagate the uncertainties in original measurements into the final results of a computation. It should be understood that the linearity assumptions are sometimes only very approximate. As long as the errors Δx_i are small compared with the values x_i this approximation is usually fairly reliable.

Be aware that cases exist where the approximation breaks down, and a more sophisticated error analysis is required. If $f(x)$ is simply $1/x$, for example, and if the distribution of x allows the value 0 with non-zero probability or probability density, a mathematical analysis easily shows that the expected mean of $z = 1/x$ is infinite, even though the mean of x may be finite. Infinite mean implies infinite variance and Eq.(6) through Eq.(8) break down completely.

In our discussion here, we are not interested in the errors in the measurements x_i, but the errors in the operations of the algorithm. Section 8.1 illustrates the use of propagation of errors in the propagation of computational errors through an algorithm.

5.3. Sources of Error in a Fixed-Point Architecture

Overflow conditions in fixed-point arithmetic can result in catastrophic alteration of results. To avoid overflow, fixed-point numbers may be shortened prior to calculations, transferring the loss to the ULP(s). If we assume that overflows, as discussed in Section 3.1, "Fixed-Point Overflow," have been avoided, then we can concentrate on the pervasive loss of the least significant bits due to the finite word size. Results of multiplication of 16-bit fixed-point integers are stored in a 40-bit MAC so that no precision is lost at that stage. However, when the results are stored into 16-bit

words for later use, 15 bits or more can be lost. Because of the size of the MACs that stage can often be postponed until many terms have been accumulated. In that case, the intermediate sums are computed at full precision.

Meyer[8] reports that errors in fixed-word-length processors arise predominantly from the affects of this type of quantization. For two's complement arithmetic, wordlength reductions can be done by truncation, rounding, or convergent rounding. "Truncation" corresponds to rounding toward negative infinity, "rounding" to rounding to nearest with ties always rounded toward zero, and "convergent rounding" to rounding to nearest with ties rounded to the nearest even number. In the absence of a specific hardware implementation, a shift operation followed by truncation often yields the fastest execution. Truncation is the C-language standard and has been adopted by the Intrinsity processors. Other techniques may be implemented in software.

Therefore, to illustrate these errors, we analyze the truncation error. Assume that a computer word contains b significant bits including the sign bit. For the Intrinsity processors b = 32 for a full word, 16 for a half word. We assume that fixed-point calculations are done and that scaling is used. Figure 1 and Figure 2 diagram the simple illustrative case of b = 5. One-bit truncation means that the ULP is set to 0. This can occur when the number is right-shifted by 1 bit (multiplication by 1/2), for example to prepare for an addition. This also shifts the radix point and has the same numerical effect as if the right-most bit (the LSB) is replaced by 0, as depicted, for both positive and negative numbers.

We can therefore analyze the truncation based on the net effect, i.e., as though the only change is to zero the LSB. The symbols x1 and x2 are bits whose values will not be affected. Thus, truncation has the effect of forcing the number in line (a) to be converted to the number in line (b), for both positive and negative numbers. The number in line (b) is unchanged under truncation. For a positive number the truncation error is either $-2^{-(b-1)}$ or 0 (Figure 1). Note that in two's complement arithmetic with sign extension for negative numbers, the number in line (b) of Figure 2 is more negative than the number in line (a). Thus the error is the same, with the same sign, for negative numbers as for positive numbers.

The scaling factor may change with each step and the error in the final result is also scaled so that it stays in proportion.

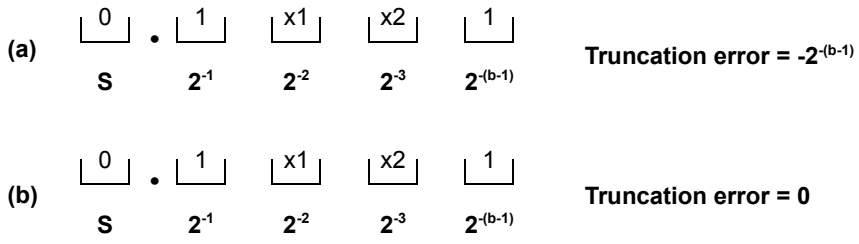

Figure 1: Error for a positive number if the last bit is truncated in a b = 5 bit fixed-point word

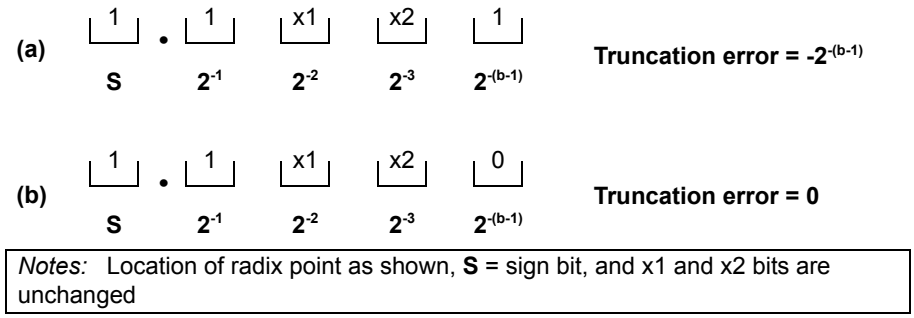

Figure 2: Error for a negative number if the last bit is truncated in two's complement arithmetic with b = 5

If we assume that the low-order bits of the numbers take on the values 0 or 1 with equal probability, then the average truncation error for a positive number is $-2^{-(b-1)}/2 = -2^{-b}$ and exactly the same for a negative number. Thus, there is a bias. Clearly the variance equals the square of this since the only possible outcomes are equidistant above and below this average for either sign. Since the variance does not depend on the sign of the number, we do not need to make any assumptions about the distribution of signs. The MSE therefore includes equal contributions from variance and squared bias.

In these examples we have discussed truncation of a single bit. If more than one bit is lost the maximum truncation error is larger. As mentioned above, this may occur when two b-bit words are multiplied and the 2b-bit result must be fit into a b-bit word. Division can similarly result in the loss of more than 1 bit in the quotient.

Meyer [8] provides the basis for the results (given in Table 1) of the mean and variance of errors in the three types of operations mentioned above: simple truncation, rounding, and convergent rounding. Since convergent rounding rounds ties up and down more or less equally, less bias is expected than for simple rounding.

Multiplication by a scaling factor to fit the result of an operation back into a b-bit word is implicit. Although convergent rounding results in no mean error under the assumptions above, this does not mean that there is no error. In fact, for a single bit of loss the convergent rounding variance is the largest of the three methods. In this case, the variance equals the MSE: $2^{-2b+1} = 2^{-2(b-2)}/8$. For multi-bit losses the MSE is $2^{-2b} \times 16/12 = 2^{-2(b-2)}/12$. The factor 1/12 comes from a presumption that the accurate value of the computation is uniformly distributed between computer-allowed values. The variance of a uniform distribution is 1/12 of its range. It can be shown in advanced theory that this presumption of uniformity is only an approximation. For truncation the 1-bit loss MSE is $2(2^{-2b})$, because the variance and squared bias terms contribute equally, and this is equal to the MSE for convergent rounding; for >>1 bits of loss the truncation MSE is larger than that for convergent rounding by the square of the bias. Since convergent rounding is computationally expensive in software, some signal processors offer convergent rounding in hardware in an effort to reduce errors. Although this reduces bias, the MSE may not be greatly reduced and, if the truncation bias can be reduced by other means (perhaps by correcting the result on average by the bias), convergent rounding may not be an improvement.

The sign of the average error not only varies with the quantization technique. It also depends on the specific algorithm and at what points in the algorithm the quantizations are done. For example, for an FFT Meyer[8] demonstrates that the mean error depends upon the output frequency index. However he also reports that the variance of the error is insensitive to the frequency index. Therefore, in this case, the bias and the MSE must both depend on the frequency.

In floating-point math, 1 ULP is the least significant bit, the value of which depends on the exponent. Note that there is also quantization, and the sign-magnitude convention for negative numbers means that truncation always moves the result toward 0. Therefore, there is a sign-dependent bias: negative results are increased and positive results are decreased. Accumulated over a number of calculations, this can amount to a significant effect. We cannot assume that the number of negative values is equal to the number of positive values.

If we express the error by its variance as distinct from its bias, we may find corrections for the bias. With sufficient data we can apply the central limit theorem of statistics to the variance. Then the average errors are approximately Gaussian with width equal to the standard deviation, the square root of the variance. The variance of the sum of independent computations is the sum of the individual variances. This facilitates estimation of the effects of sequential operations. Note, however, that where various terms containing errors are combined with additions and multiplications there may be strong correlations in the resultant errors. They may not be independent.

The importance of a bias depends on the mix of single-bit *versus* multiple-bit quantizations. For truncations, if the multi-bit truncations dominate then this bias may dominate the variance as a contributor to the MSE.

Table 2: Errors from quantization to fit into a b-bit word length, following Meyer [8]

Quantization Method		Lose 1 Bit (E.g., Simple Sums)	Lose >>1 Bit (E.g., Products)
Truncation	Mean error (bias)	-2^{-b}	$-2^{-(b-1)}$
	Variance	2^{-2b}	$\sim 2^{-2b} \times 16/12$
Rounding	Mean error	2^{-b}	0
	Variance	2^{-2b}	$\sim 2^{-2b} \times 16/12$
Convergent rounding	Mean error	0	0
	Variance	2^{-2b+1}	$\sim 2^{-2b} \times 16/12$

5.4. Condition Numbers and Posedness

Accuracy of an algorithm is sometimes described in terms of condition numbers. If there is computational error the results may be equivalent to a perfect calculation with different input. The amount of change in the input that would be necessary is characterized by the condition number. A problem with a large condition number is that it is very sensitive to errors.

Let the N-vector $x = (x_0, \ldots, x_{N-1})$ be one complete set of input variables for processing by an algorithm. Let $f(x)$ be the set of outputs; $f(x)$ may also be a vector of not necessarily the same length. If the computation features infinite precision then there is no quantization error in $f(x)$. Let us express the effect of quantization error in a realistic computation as equivalent to a perfect computation with slightly different input, $f(X)$. The necessary perturbation in the input is $\Delta x = x - X$. Then the output error due to quantization is approximately given by the first term of a Taylor's series expansion,

$$f(x) - f(X) = \sum_{n=0}^{N-1} \left\{ \frac{\partial}{\partial x_n} f(x) \right\} (\Delta x) + \ldots \quad , \qquad \textit{Eq.(9)}$$

where the partial derivatives express the first-order sensitivity of the algorithm to input changes. If $f(x)$ is multiple-valued, Eq.(9) applies to each separately. Eq.(9) expresses the propagation of the vector of perturbations Δx through the operation of the algorithm. Truncating the Taylor's series after the first term is reasonable for small perturbations. The relative propagation error is

$$\varepsilon_f = \frac{\Delta f}{f} = \left(\frac{1}{f(x)} \right) \sum_{n=0}^{N-1} x_n \cdot \left[\frac{\partial}{\partial x_n} f(x) \right] \cdot \frac{\Delta x}{x} \quad , \qquad \textit{Eq.(10)}$$

where we assume $f(x) \neq 0$. Finally, we define the condition number κ_n as

$$\kappa_n = \frac{x_n}{f(x)} \cdot \left[\frac{\partial}{\partial x_n} f(X) \right] \quad . \qquad \textit{Eq.(11)}$$

The condition number is an amplification factor on the relative data errors Δx. It helps us understand the effect of a given relative error. If a condition number is 100, then a relative error of 10^{-6} in the input or in the computation loses two decimal places in the relative error of the output: 10^{-4}. A 100-ULP net quantization error is equivalent to a 1-ULP input error. Whether or not that is acceptable depends on the requirements. A problem is "ill conditioned" if some of the condition numbers are too large to satisfy the requirements of the problem. In that case small errors in the input can result in large errors in the output, and quantization errors can be dangerous. Note that if $f(x)$ is close to zero the

condition number is ill-defined (depending on the value of the derivative), and many use an "absolute condition number" defined by the partial derivative only.

This definition of condition number is a result of backward error analysis, which propagates an output error back through the algorithm to an equivalent input error. Since κ_n relates to a relative error, not an absolute error, it may be that large condition numbers for some of the output values are acceptable. This is the case if the affected output value is an unimportant part of the final result.

If the problem admits of a unique solution that is continuous with the input data, it is considered to be "well posed". An example of an ill-posed problem is to find the real roots of a polynomial of order >1 as the coefficients vary. For certain coefficient values there may be no real roots; the number of real roots can change discontinuously with continuous changes in the parameters. In the neighborhood of such discontinuities the problem is ill-posed. If the parameters are randomly varying data, this problem may exhibit numerical instability. An ill-posed problem cannot be solved by changing the algorithm, but only by changing the problem (some problems can be "regularized" as a well-posed problem that yields an approximate solution). An ill-conditioned problem may be well posed. In that case, well-conditioned solutions may be possible by rewriting the problem into an equivalent problem with the same solutions. For more discussion see Ref. [3].

In a problem with multiple *f(x)* terms there are multiple condition numbers. In addition, there may be terms *f(x)* for which the value is near zero. Therefore it is sometimes more convenient to adopt other definitions of the condition number, perhaps involving a suitable norm of the Δx, so that there is only one or a small number of condition numbers that describe the whole problem.

Note: Some authors define the condition number in terms of the supremum of a choice of suitable norms of the terms. In the definition we adopted, the condition number could be negative.

Use of condition numbers can highlight problems that sometimes arise, for example, in iterative algorithms. A small numerical error that arises in an early stage can become magnified when carried through each subsequent step. For example, suppose the first step computes y_0 with a certain quantization error, ε_0. If the next step to compute y_1 requires y_0 be multiplied by a factor $f > 1$, then the error in y_0 adds $f\varepsilon_0$ to the quantization error in y_1, and so on. It may be possible to recast the algorithm, for example to conduct the iterations in the opposite direction. Then an error ε_1 in y_1 would contribute ε_1/f to y_0, and the computation is much more stable and has a much smaller condition number.

5.4.1. Relative and absolute errors and condition numbers for elementary operations

We evaluate the relative and absolute errors in elementary arithmetic operations involving x_1 and x_2 in terms of errors in x_1 and x_2. This may help the developer understand the growth of errors due to errors from earlier steps. The error due to the operation itself depends on whatever quantization occurs at that step. For this section, the notation $Dx = \Delta x/x$ refers to the relative error in x. In all error expressions, higher order terms have been neglected and it should be understood that the equations are approximate.

5.4.1.1. Sum

If $y = x_1 + x_2$, then

$$\kappa_1 = \frac{x_1}{x_1 + x_2} \quad ; \quad \kappa_2 = \frac{x_2}{x_1 + x_2} \quad . \qquad \textit{Eq.(12)}$$

Then the relative error $Dy = \kappa_1 Dx_1 + \kappa_2 Dx_2$.

From propagation of errors, the variances add. If x_1 and x_2 are independent, $V(y) = V(x_1) + V(x_2)$. Note that the condition numbers are between 0 and 1 if x_1 and x_2 have the same sign. Then this problem is well-conditioned; errors in x_1 and x_2 contribute less than their full values to the error in y.

However, if x_1 and x_2 have opposite signs, the condition numbers can become very large. This is, perhaps, the best-known numerical analysis result: the relative error of the difference of two nearly-equal numbers can be very large. Though the variance of y remains the sum of the variances of x_1 and x_2, and therefore the average absolute error σ_y =

$\sqrt{V(y)}$ does not depend on the sign of x_1 or x_2, we observe that σ_y/y may be large. The developer is well-advised to avoid such cancellation of terms unless the sum of x_1 and x_2 is a small part of the whole.

5.4.1.2. Products

If $y = x_1 x_2$, then $\kappa_1 = \kappa_2 = 1$ and the relative error $Dy = Dx_1 + Dx_2$. The relative error increases but does not explode and a product is therefore stable.

From propagation of errors,

$$V(y) = \left(\frac{\partial y}{\partial x_1}\right)^2 V(x_1) + \left(\frac{\partial y}{\partial x_2}\right)^2 V(x_2)$$
$$= x_2^2 V(x_1) + x_1^2 V(x_2) \quad , \qquad \textit{Eq.(13)}$$

assuming independence of x_1 and x_2. The importance to the absolute error in y of an error in one of the multiplicands depends on the square of the value of the other.

5.4.1.3. Quotient

If $y = x_1/x_2$, $x_2 \neq 0$, then $\kappa_1 = 1$ and $\kappa_2 = -1$. Therefore, $Dy = Dx_1 - Dx_2$ and the quotient is stable. The variance in y is

$$V(y) = \left(\frac{1}{x_2}\right)^2 V(x_1) + \left(\frac{x_1^2}{x_2^4}\right) V(x_2)$$
$$= \left(\frac{y}{x_1}\right)^2 V(x_1) + \left(\frac{y}{x_2}\right)^2 V(x_2) \quad , x_1, x_2 \neq 0 \ . \qquad \textit{Eq.(14)}$$

5.4.1.4. Powers

If $y = x^p$, $p > 0$, $x > 0$, then $\kappa = p$ and $\Delta y = p\,\Delta x$. If $p < 1$, i.e., the power is a root, then the errors are well-conditioned. If $p > 1$, the conditioning may still be fine for moderate p but is possibly unacceptable for large p.

The variance of y is $V(y) = (p\,x^{p-1})^2\,V(x)$ and the average absolute error $\sigma_y = p\,x^{p-1}\,\sigma_x$ could grow large quickly if $p > 1$. Therefore, caution is advised when evaluating a polynomial. Horner's rule [9] may be a better scheme for an algorithm than a brute-force power-multiplication-sum approach.

5.5. Cancellation and Damping of Errors

If the errors in the data are not independent, or if they contribute in the same ways to a calculation, there can be a cancellation of errors that can actually improve the accuracy above the precision of any of the input. For example, let x be a measured value with an error Δx and $a = \cos(x)$, $b = \sin(x)$, and $y = a + b$. Then from simple calculus we may show that the maximum value of $\Delta y = \Delta a + \Delta b$ is $\Delta x\,\sqrt{2}$, at $x = 7\pi/4$ (315°), and the minimum is the negative of that at $x = 3\pi/4$ (135°). However, at $x = \pi/4$ and $5\pi/4$ (45° and 225°) a first-order calculation (i.e., accurate to the first derivative of the Taylor's series) would give an error of 0, i.e., perfect cancellation of errors. This follows immediately from propagation of errors: $V(y) = [\cos(x) - \sin(x)]^2\,V(x)$. At these values of x the error in one corrects for the error in the other, since both depend only on the error in x and have opposite signs at these angles. Higher-order terms would, of course, contribute a non-zero error estimate. The actual error as a function of Δx can always be calculated analytically and curves drawn, if desired.

Even more dramatic is the error in $y = x^3 - 3x^2 + 3x$ near $x = 1$. Although y is cubic in x, both the first and second derivatives are 0 at $x = 1$ ($x = 1$ is a saddle point). Therefore the first contribution to the propagation of errors Taylor's series is from the third derivative, and a Δx of 0.1 contributes a Δy of only 0.001 at $x = 1$. The second and third terms of the expression approximately cancel the error of the first at that particular point and, near there, the fact that y is cubic in x only creates a misleading impression of error magnification.

The errors can also be damped. For example, if one of the terms in a sum has a large relative error, but the term itself is small compared to the others, its large relative error is damped - it doesn't contribute much to the error in the sum. The size of its absolute error in relation to the final sum is what counts. In this case the condition number is small, so the large relative error $Dx = \Delta x/x$ does not do much damage.

Both cancellation and damping of errors depend on the values of the input and output. Therefore it is difficult to draw general conclusions for a given problem unless the ranges of input and output are known. A simulation may be the best general approach.

5.6. Dithering

Dithering is a technology originally developed for the audio industry for overcoming errors of quantization. Low-power audio signals may have an amplitude comparable to the size of the ULP. An analog sine wave with that amplitude is normally digitized, with the size of the quantization playing a major role. The quantized digital amplitude shows only a series of steps up and steps down, giving audible pulses at the input frequency instead of continuous, smooth sound. With the dithering technique, you can add high-frequency noise of amplitude carefully calculated, which is usually close to 1 ULP. Paradoxically, adding noise smooths the sound by randomizing the steps. Within one period of the input sine the high frequency noise causes many steps up and down. The probability of a step depends on the instantaneous amplitude of the sound. The digitized bits are mostly high when the amplitude is high and mostly low when the amplitude is low. These flip rapidly back and forth at in-between amplitudes. The ear smooths out the high-frequency fluctuations and senses the average amplitude as a fairly accurate rendering of the original intent. The smooth variations of the original signal are replaced by many probabilistic steps instead of relatively long duration steps.

Dithering is also used in signal detection in the presence of a sensor threshold. A signal below the threshold of the sensor is randomly amplified by the addition of high-frequency noise of a carefully calculated amplitude and distribution. Where the signal is maximum the probability that signal plus noise exceeds the threshold is maximum, and vice-versa. Over sufficient periods at the signal frequency a Fourier analysis finds the desired signal, even though it may never have exceeded the threshold of detection. By adding carefully chosen noise to the signal, the signal-to-noise ratio is improved for these types of very small signals.

Dithering may be used in the digital calculation arena as well. By randomly adding a very small amount to the signal, for example, you can randomly affect truncation error and remove the bias. That is, although truncation always forces a bias, by adding random noise that is *oppositely* biased this can be removed on average. However, the MSE is not lowered; it can only be increased.

In most cases, the dithering noise is usually chosen to be Gaussian, uniform, or triangular in distribution. Calculation or simulation can verify the amount and the distribution of noise for optimal results. Although adding noise obviously increases the noise floor, in many applications the high frequency noise can subsequently be removed by filtering.

5.7. Summary of Calculational Errors

The computational error depends the number of bits of precision and can be expressed in terms of 1 ULP. As mentioned at the start of Section 5, because fixed-point numbers are equally spaced, 1 ULP has a fixed size. It is an absolute error of size dependent only on the position of the radix point. If the numbers are scaled by shifting, the ULP is scaled inversely. For example, if we down-scale by right-shifting z bits we up-scale 1 ULP by the same number of bits.

Floating-point numbers are not equally spaced, and 1 ULP depends on the value of the number. An error of 1 ULP is (approximately) a purely relative error. Programmer scaling of floating-point numbers is rarely needed; the scaling is automatic.

If we assume that overflow does not occur we can now estimate errors from calculations. For example, if $c = a + b$, the computational errors come from quantization after carrying out the addition operation, not from any measurement uncertainties in a or b.

In a process that requires many steps we can treat calculational errors as random variables with means and variances, as given in Table 2 on page 15, and trace their effects on the output. For example, in a sum over n variables there are n - 1 sums, each of which may have an error of as much as 1 ULP. If the variances of each process are independent, then the variance of a sum of n terms, if each sum is individually quantized before the next sum is performed, is $(n - 1)\,V(\oplus)$. The notation $\oplus$ signifies that the variance is due to quantization from the sum and is not related to the variance of the terms being summed. Understand that many algorithms do not quantize each operation individually. For example, the FastMATH MACs may make it possible to avoid that. Understand also that some algorithms, such

as an FFT, realize their efficiency by reusing partial sums. Therefore, the errors repeat and independence is not satisfied; correlations occur.

To illustrate, let us estimate the expected calculational errors in the two sums

$$s_1 = (b + c + d) \quad \text{and} \qquad \textit{Eq.(15)}$$

$$s_2 = (a + b + c) \quad . \qquad \textit{Eq.(16)}$$

These sums have $b + c$ in common and differ in the additional terms, d or a. Assume truncation, and that exactly 1 bit is lost in each quantization. Then $V(s_1) = V(\oplus) = 2 \times 2^{-2b} = V(s_2)$, because there are two sums. This is solely the variance due to the quantization of the sums. If $s_0 = b + c$ is performed first and subsequently d is added for s_1 and a for s_2, the variances are the same but there is a covariance term, signifying a correlation between the calculational errors in the two sums. Adopt the notation that ε_1 refers to the quantization error from adding the d term and ε_2 similarly for the a term. Then $V(s_0) = V(\oplus) = 2^{-2b}$. The covariance term is obtained from Eq.(7), propagation of errors:

$$\begin{aligned} V(s_1, s_2) = & \left(\frac{\partial s_1}{\partial \varepsilon_1}\right)\left(\frac{\partial s_2}{\partial \varepsilon_1}\right) V(d) + \left(\frac{\partial s_1}{\partial \varepsilon_2}\right)\left(\frac{\partial s_2}{\partial \varepsilon_2}\right) V(a) \\ & + \left(\frac{\partial s_1}{\partial \varepsilon_1}\right)\left(\frac{\partial s_2}{\partial \varepsilon_2}\right) V(\varepsilon_1, \varepsilon_2) + \left(\frac{\partial s_1}{\partial \varepsilon_2}\right)\left(\frac{\partial s_2}{\partial \varepsilon_1}\right) V(\varepsilon_1, \varepsilon_2) \\ & + \left(\frac{\partial s_1}{\partial s_0}\right)\left(\frac{\partial s_2}{\partial s_0}\right) V(s_0) = V(s_0) \quad . \end{aligned} \qquad \textit{Eq.(17)}$$

Note that all terms but the last drop out. The first two terms drop out because either s_1 or s_2 does not depend upon the error term in one of the specified derivatives, and the next two drop out because the covariance $V(\varepsilon_1, \varepsilon_2) = 0$ (ε_1 and ε_2 are assumed independent). From Eq.(17) therefore, the $V(s_1, s_2)$ covariance is 2^{-2b} and the correlation between s_1 and s_2 is
$+\sqrt{\{2^{-2b}/(4 \times 2^{-2b})\}} = \sqrt{1/4} = 1/2$.

> **Note:** The FastMATH processor provides for direct addition of two matrix registers with the result directed to a matrix register. If overflow is possible, scaling must be done before the addition. If multiple terms are to be summed, overflow in intermediate terms can be avoided by taking advantage of the FastMATH 40-bit MACs. To do addition and direct the result to a MAC, do it as a multiply-add with a multiplication factor of 1. A sequence of accumulations can be done this way, preserving full precision in the intermediate results (up to the limit of 40 bits total). Then scaling and possible truncation only needs to be done once, when the final result is moved back into a register.

5.8. Effects of Signal Strength and SNR

It is generally assumed that the importance of errors is in proportion to their fraction of the size of the signal. In a comparison test of one algorithm against another, we need a figure of merit. If the computational errors can be measured in test cases by comparison with a more precise calculation, such as one with double-precision, floating point math, then we can estimate the signal-to-noise ratio (SNR). It is customary to express SNR as a ratio of powers, signal to noise, expressed in decibels (dBs):

$$SNR = 10\log\left\{\frac{\sum[Re(x_n)]^2 + \sum[Im(x_n)]^2}{\sum[\Delta Re(x_n)]^2 + \sum[\Delta Im(x_n)]^2}\right\} \quad \text{dB.} \qquad \textit{Eq.(18)}$$

In Eq.(18), the variables x_n are complex calculated result values from an algorithm applied to typical data, the Δ values are their (absolute) computational errors, and the summations are over a number of such values. All logarithms are to base 10. For real-valued computations, the imaginary parts are, obviously, not present. This relation expresses the ratio of the sum of the squared result values with the sum of the squared errors. It does not express the importance of any particular error to any particular result value. That could be a weakness in this definition of SNR if the relative error, result value by result value, is important. The summations could be over a number of results from a single application of the algorithm, for example, for an FFT. For an algorithm with a small number of output values, such as

a dot product, you may prefer to sum over a number of applications of the algorithm with random input data. We note that, since the terms are added in quadrature, one anomalously large value of x_n or Δx_n could have a large effect on SNR. This affects the variance of SNR, as we subsequently show.

It is useful to have a few SNR values for comparison. A fixed-point word in (s15.0) format has a maximum absolute value of $2^{16} - 1 \approx 2^{16}$. A 1-ULP error in a word with this value has SNR = $10 \log[(2^{16})^2] = 320 \log 2 = 96.3$. Other reference values of SNR for a 1-ULP error are given in Table 3. These are, of course, optimum values, not easily obtainable in practice.

Table 3: SNR for representative cases; 1-ULP error to maximum value

Word Size (Significand)	SNR for 1-ULP Error
15 bits + sign	96.3
31 bits + sign	192.7
16 bits, unsigned	102.4
32 bits, unsigned	195.7
24 bits (single-precision float)	150.5
53 bits (double-precision float)	325.1

Since both results and computational errors are computed from random data, they are themselves random variables with some distribution. The SNR is therefore a statistic, since it is computed from random data plus known quantities. It also has a probability density function. In order to estimate how much data we need to get a reliable estimate for the SNR we need to have an estimate of the size of the fluctuations in SNR, as the data varies. We proceed to get a feel for what this distribution is.

To be a useful performance metric, we must assume that the mean of x_n is 0. Otherwise, we can conceive a procedure to add an arbitrary constant to all x_n, without changing Δx_n, and obtain arbitrarily good SNR.

Let us assume that the output signals $\mathrm{Re}(x_n)$ and $\mathrm{Im}(x_n)$ and their computational errors are all Gaussian-distributed. Over a sufficiently large number of occurrences this may at least be approximately satisfied. Assume that we can introduce overall scale factors in numerator and denominator, such that the variance of each x_n and each Δx_n is unity. Thus, if the data x_n has variance σ_d^2 and the errors Δx_n have variance σ_e^2, then we multiply the term in curly brackets in Eq.(18) by σ_e^2/σ_d^2 as normalization. We construct a random variable y equal to this normalized ratio, so that we recover SNR = $10 \log(y\sigma_d^2/\sigma_e^2)$. The summations cover N terms. First, we look at the probability density function for y. Each normalized sum in the numerator and denominator is distributed as $\chi^2(N)$, that is, as a chi-squared distribution with N degrees of freedom, and the sum of squares of real and imaginary parts is then $\chi^2(M)$, with $M = 2N$. If the x_n terms happen to be correlated, then, as long as the covariance matrix is of full rank (i.e., none of the x_n can be expressed as a linear combination of the others), we can transform (rotate in x_n space) to a set of independent variates and proceed from there. These rotated variates probably require different scaling to achieve unit variance. This change in scaling expresses the effect on SNR of the correlations.

The primary source of the fluctuations of the output signal (the numerator) is the random variations in the data, coupled with all the other sources of fluctuations mentioned in Section 5. The power of the signal derives from its fluctuations, since with 0 mean, in the absence of fluctuations all $x_n = 0$. For the computational error terms in the

denominator, the source is the quantization errors. The probability density of y, a ratio of $\chi^2(M)$ variates, is well-known in the statistical literature [6] to follow the F density[9] with (M, M) degrees of freedom:

$$f_{M,M}(y) = \frac{y^{\frac{M}{2}-1}}{B\left(\frac{M}{2},\frac{M}{2}\right)(1+y)^{M}} \quad , \qquad 0 < y. \qquad \textit{Eq.(19)}$$

If $M = 2$ the density has a finite mode (peak) at $y = 0$ and if $M = 1$, it has an infinite mode at $y = 0$. The B function is a normalization constant given by

$$B\left(\frac{M}{2},\frac{M}{2}\right) = \frac{\Gamma\left(\frac{M}{2}\right)\Gamma\left(\frac{M}{2}\right)}{\Gamma(M)} \quad , \qquad \textit{Eq.(20)}$$

where Γ is the gamma function[1]; $\Gamma(X) = X!$, if X is integer. This density extends over the positive real axis. If $M > 2$, it rises steeply at first from $y = 0$ to a single mode at $y = [M(M - 2)]/[M(M+2)]$. It falls off slowly from there. Example curves are given in [6]. The effect of an anomalously large squared value in the numerator is seen in the long tail of this density to infinity. We note from Section 5.3 that the mean of Δx_n is not zero in all cases. We can correct for that by adding the appropriate constant to the denominator of the ratio in Eq.(18), or by using a non-central F distribution[6] [10].

The F density can be defined in more generality for a different number of degrees of freedom in numerator and denominator. Because our problem involves the same number of terms in numerator and denominator, we can realize some simplifications.

The mean of y is infinite if $M \le 2$. Otherwise, $E(y) = M/(M - 2)$. The variance of $f_{M,M}(y)$ is infinite for $M \le 4$, in which case no measurement of SNR can be considered reliable. For $M > 4$,

$$V(y) = \frac{4M^2(M-1)}{M(M-2)^2(M-4)} \quad . \qquad \textit{Eq.(21)}$$

Thus, the number of measurements [terms in the sums in Eq.(18)] should be >> 4 to achieve a good value for y and therefore SNR. As long as $\sigma(y) = \sqrt{V(y)}$ is small compared with y, then the standard error in SNR = $10 \log(y\sigma_d^2/\sigma_e^2)$ is found approximately by propagation of errors (Section 5.2.2):

$$\sigma_{SNR} = \frac{10\sigma_y}{y \cdot \sigma_d^2/\sigma_e^2} \ \text{dB}, \quad M > 4. \qquad \textit{Eq.(22)}$$

Obtaining this expression was the object of the exercise. We recommend that M be chosen sufficiently large that σ_{SNR} << SNR. Otherwise a reliable measurement is not obtained. Note from Eq.(21) that, for large M, $\sigma_y = \sqrt{V(y)} \to 2/\sqrt{M}$.

The random variate

$$z = \frac{\log(y)}{2} = \frac{\text{SNR}}{20} - \frac{\log\left(\frac{\sigma^2_d}{\sigma^2_e}\right)}{2} \qquad \textit{Eq.(23)}$$

is known in the literature [6, 10] as Fisher's z-statistic (Fisher's z also appears in slightly different forms elsewhere). In general it is characterized by a more symmetric probability density function than that of y; in particular, all moments are finite. For our case, with both numerator and denominator in Eq.(18) having the same number of degrees of freedom, the z distribution is exactly symmetric about $z = 0$ with a single mode there. This is a consequence of both numerator and denominator being built from Gaussian variates with mean 0. Therefore, the mean of z is 0, implying from Eq.(23) that the mean SNR is $10 \log(\sigma_d^2/\sigma_e^2)$.

9 Sometimes known as the variance ratio density or Snedcor's F density (but also credited to R. A. Fisher).

Again, if the x_n or Δx_n are not of zero mean then y obeys a non-central F distribution. See the statistical literature for details on using that distribution.

5.8.1. Relation to Variances of Signal and Errors

In many signal processing problems random errors other than computational errors can also be characterized by a SNR, with the appropriate errors in the denominator. For example, if the error comes from an ADC the error may be a relatively predictable fraction of the signal strength—linear if the ADC and the effects of the algorithm are linear. Then the ratio of the input signal to the output errors usefully identifies the constant of proportionality.

However, for computational errors, because of the effects of scaling the error may not be strongly dependent upon the maximum signal strength, but rather upon its variability. There is also no reason to expect it to show a linear dependence on the signal. Thus a signal with twice the power may have the same size computational errors, so an SNR estimated for one signal may not apply to another, even for the same algorithm. For signals with similar power, SNR is of value in studies of an algorithm. For different algorithms the most reliable comparative measure of accuracy is to directly compare the errors themselves using identical input signals or to average over input signals possessing identical distributions. To determine if an algorithm satisfies accuracy requirements, you may try different inputs spanning the range of powers and distributions expected. As we have seen above, both power and error distributions may approximately obey the $\chi^2(M)$ distribution and therefore have long tails. However, for large M, the χ^2 distribution is very similar to a Gaussian, where the tails, though long, have rapidly decreasing probability. Therefore, it is important to average over enough data, even if you are only separately interested in either signal power or error power.

The shape of the signal may have an effect on the computational errors due to the fact that scaling factors for overflow protection depend on the maxima. If large scaling is required, small signal values may lose precision, i.e., have many leading zeros. If the signal does not vary much, structuring it so that the mean is 0 will cause it to concentrate near 0. Hence, the average signal power may be low and the SNR correspondingly low.

The power is related to the variability of the data. Note that, by Eq.(3), and temporarily allowing for non-zero mean values,

$$V[Re(x_n)] = E\{[Re(x_n)]^2\} - \{E[Re(x_n)]\}^2 \quad . \qquad \textit{Eq.(24)}$$

For a complex signal, if the real and imaginary parts of the signal are independent and from the same distribution they have equal variances. If their mean values are now set to zero that term drops out. Then we can simplify to

$$V\{|x_n|\} = E\{|x_n|^2\} = E\{[Re(x_n)]^2\} + E\{[Im(x_n)]^2\} \quad . \qquad \textit{Eq.(25)}$$

If there are enough terms in the sum it is a reasonable approximation to replace the expectation value by the average of the observed value. The error term in the denominator of SNR is computed similarly. Hence,

$$SNR = 10\log\left\{\frac{\sum V(|x_n|)}{\sum V(|\Delta x_n|)}\right\} \quad . \qquad \textit{Eq.(26)}$$

Hence, for signals and errors normalized to zero mean the power is directly related to the variance, and the SNR can be calculated from the ratio of variances.

6. Example: 32-bit Multiplication in (s15.16) Format

The FastMATH processor supports 16-bit SIMD multiplications and multiply-add operations with native single-instruction operations. These are straightforward to implement. We focus here on 32-bit multiplications, which are necessary if more than 16 bits of precision are needed. We must break each 32-bit multiplicand into two 16-bit parts, multiply all the parts together, and recombine the results. Note (Table 1) that multiplication of two signed 32-bit numbers (sign + 31 bits) results in up to 62 significant digits plus a sign, independent of the position of the radix point. Further operations involving additions can increase that, due to carries from the additions. This requires that a 64^+-bit word exists to contain the result. The FastMATH multiply-accumulator contains 40 bits. Therefore, we deal with the

products of the 16-bit halfwords separately. At least 256 additions of such products are then possible before risk of overflow. Therefore, if a sequence of operations is necessary, we can treat each of the partitions separately, multiply and accumulate each term contributing to that partition, and thereby take advantage of the 40-bit accumulators. The dual accumulators enable two such computation streams to be done in parallel.

We must understand how to fit the final combined result into a signed 32-bit word while maintaining the correct radix point. Clearly, many bits will be lost. The secret to designing fixed-point code is to select a fixed-point representation and a scaling of the data and/or intermediate results so that there is no overflow and yet a minimum amount of significance is lost.

We analyze the multiplication of two (s15.16) words A and B as an example. Refer to Figure 3. Divide each into two halves, A_h, A_l, B_h, and B_l. Only the high halfwords carry a sign. The low halfwords are treated as unsigned 16-bit integers. The full result requires $A_h \times B_h$, $A_h \times B_l$, $A_l \times B_h$, and $A_l \times B_l$. Each product is shifted and added into 32-bit portions of a 64-bit product, as depicted. The fixed-point format of the result determines which 32 of those bits are retained in the final result; (s15.16) is shown. The high-high product goes into the upper 32 bits. The high-low and low-high products go into the middle 32 bits, and the low-low products go into the lowest 32 bits.

The order in which these halves are multiplied is immaterial and may be performed in such a way as to optimize performance. All that is required is that the products be shifted into the final result and added, as shown. For example, if a sum of products is being computed, as for an finite impulse response filter (FIR), you can separately sum all the high-high terms into one of the accumulators, and similarly sum one of the other product terms into the other. Repeat the process for the other two sets of products. Shifting and adding for the final result can be done at the end of each sequence of such operations, paying attention to possible overflows from the summations.

Note the location of the radix point in the final result. The low-low product is shifted a halfword to the right before being added to the result. The high-high product is shifted a halfword to the left before summing. The high-order 16 bits and the low-order 16 bits are discarded. At that time, the developer must ensure that no bits other than sign bits are present in the high-order 16 bits. Otherwise, the most significant part of the result is lost (overflow). Any bits other than sign bits in this halfword could come from the high-high product or from carries from the “cross” terms, low-high or high-low, or from the sum of the four contributions

Note: It is possible that a carry bit, even one from adding the low-low term, can propagate up to the high-order 16 bits.

FastMATH C-language code for this 32-bit multiplication, for 16 numbers in parallel, is shown in Figure 4. The Intrinsity matrix language instruction set is implemented in the C language as a set of intrinsic instructions, as Figure 4 illustrates; the mnemonics are straightforward. In the FastMATH implementation, each term, A and B, is a 16-element matrix. All steps apply to all elements in parallel. We follow the prescription for the radix point location as shown in Figure 3. We retain only the 32 bits of the result that symmetrically surround the radix point. At this point, the developer must note that this implementation has assumed that the high-high product ($A_h \times B_h$) has no more than 15 significant bits plus the sign. Those bits would otherwise be lost when that product is shifted 2 bytes to the left to align the radix points. This also assumes that the sum of the $A_h \times B_l$ and $A_l \times B_h$ products does not overflow. If the data cannot be scaled so that these products have no bits lost to these steps, then the location of the radix point should be changed. Section 6.1.1 discusses this.

If either or both of the products is negative, the diagrammed process works automatically in two’s complement arithmetic with no additional steps. Recall that only the high halfwords are signed. The low halfwords are unsigned 16-bit integers. Following the two’s-complement standard, the Intrinsity multiply-add instructions treat a signed halfword consisting of all Fs (FFFF) as a -1 in the LSB position. In (s15.16) fixed-point representation, -1.0 becomes (FFFF.0000). For negative numbers, an arithmetic right shifts fills the vacated bits at the left with 1s, and these sign-extended 1-bits are part of the calculation. For example, $A000_{16} = 1010\ 0000\ 0000\ 0000_2 = -(2^{13} + 2^{14}) = -24\ 576_{10}$, when arithmetic right-shifted 4 bits, becomes $FA00_{16} = 1111\ 1010\ 0000\ 0000_2 = -(2^9 + 2^{10}) = -1\ 536_{10}$. The sign bit has been extended four positions to the right, each traversed position filled with 1, and the significand is shifted right four bits.

6.1. Simple Example: A 1-Byte Machine

Let us work an example for a 1-byte (2 hexadecimal digits) machine. This demonstrates all the steps of multiplication by halfword. We wish to multiply hexadecimal 12 (decimal 18) by -1. The -1 is stored in the 1-byte word as FF. Then the product is obtained as shown in Figure 5, assuming a 2-byte accumulator that accommodates the largest possible products.

We multiply the low-low halfwords, the low-high and high-low (the 12_{16} is listed first, the -1 second, in these partial products as shown in Figure 5), and the high-high halfwords, and shift and add as shown.

The first product row is the low-low product $2_{16} \times F_{16}$, where both the 2 and the F are treated as unsigned integers because they are located in the low nybbles (a nybble (4 bits) is a halfword in a 1-byte machine). Therefore, in decimal, that product is $2_{10} \times 15_{10} = 30_{10} = 1E_{16}$. The low-high and high-low products are listed next, shifted one nybble to the left as in an ordinary multiplication by hand. The low-high product, for example, has become $2_{16} \times (-1)_{16}$, since the high halfword of the multiplicands carries the sign; therefore, the F represents -1, instead of 15_{10}, as it was in the first product. Finally, the high-high product is shifted one byte to the left. In the final summation there is a carry bit carried forward from the second hexadecimal digit all the way off the left side, where it is discarded. This is necessary to complete the final product as a negative number. We are left with the 2-byte product FFEE, which is $-12_{16} = -18_{10}$.

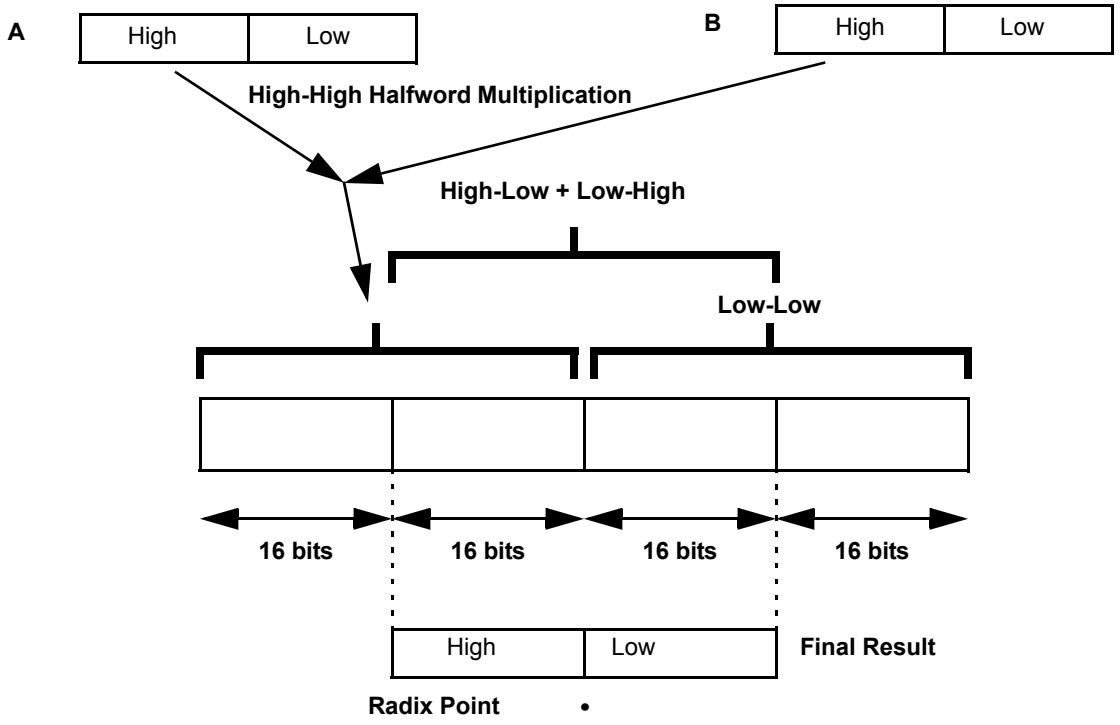

Figure 3: (s15.16) 32-bit multiplication flow demonstrating breakdown into 16-bit halves and reconstruction after multiplication

Key in this computation is that the two's complement sign bits are fully included in the multiplication. For example, in the high-low product (third row) the unsigned F from the low nybble of the upper multiplier appears intact in the product, multiplied by 1, and the result in the third row appears as a positive number. Products involving signed F, the high nybble of the upper product, which are nothing more than the sign-extension bits of the -1, appear sign-extended in the second and fourth rows.

6.1.1. Radix Point

We can now illustrate the radix-point issue with this simple example (Figure 5). We place the radix point between the halfwords, as was previously done in Figure 3, but for the 1-byte example, and do the same multiplication. Now our representation is (s3.4). Note that $1.2_{16} = 0001.0010_2 = 1 + 2^{-3} = 1.125_{10}$. Note also that -1.0 is represented by F.0 in (s3.4). Then the product becomes $1.2_{16} \times F.0 = FE.E0 = -1.2_{16}$, or -1.125_{10}. If we are able to retain only one byte, we retain the same (s3.4) representation that we started with and retain $E.E = -1.2_{16}$. No overflow has occurred and no precision has been lost. However, this won't be the case in general, as we now show.

If both numbers are scaled such that the radix point is at the left (s0.7), then the same pattern of bits as in Figure 5 makes the product ($1.111\ 1111 = -2^{-7}$) × ($0.001\ 0010 = +0.140\ 625_{10}$) = ($1.111\ 1111\ 1110\ 1110 = -0.001\ 098\ 6\ldots_{10}$). Recall that in (s0.7) format, the bit to the left of the radix point is purely a sign bit, and in two's complement arithmetic the string of 1-bits following the radix point are extensions of this sign. What if we remove the sign-extended 1-bits following the radix point by scaling before the result is stored in a 1-byte register? That is, we might be tempted to scale by left-shifting by 1 byte, storing the negative number 1.110 1110 and later removing the scale factor of 2^8 before the result is published. However, this might be inadvisable: the fact that the byte of 1s removed is not needed is a consequence only of the fact that the two multiplicands are small numbers in absolute value. Larger multiplicands would have filled both bytes of the result and scaling by 2^8 removes the most significant bits. In general, we might retain only the upper two nybbles, unless it is known in advance that the dynamic range of the products fits into a different group of bits. This illustrates how choosing the scaling and the radix point position depends on the dynamic range issues.

```
/* We begin with the high-low portion of the product, directed into MAC 1.*/
ma1 = MatrixMulLowUnsignedHigh(Ah, Bl);
/* Add the low-high portion into the same accumulator.*/
ma1 = MatrixMulAddLowUnsignedHigh(Al, Bh);
/* Perform the high-high halfword product into the other accumulator.*/
ma0 = MatrixMulHighHigh(Ah, Bh);
/* Move the results into two matrix registers for temporary storage.*/
/* For simplicity, we will show recovery of only the low-order 32 bits. An exact calculation
   would note the possibility that ma1 could have an overflow due to the sum.*/
matrixT1 = MatrixMoveFromLow(ma1);
matrixT2 = MatrixMoveFromLow(ma0);
/* Finish with the low-low portion of the product.*/
ma0 = MatrixMulLowUnsignedLowUnsigned(Al, Bl);
matrixT3 = MatrixMoveFromLow(ma0);
/* Now we have 3 partial products. We will shift and sum and store in a 32-bit matrix element,
following the prescription of Figure 3 for the radix point location.*/
matrixFinal = MatrixShiftRightLogicalByteImmed(matrixT3, 2); /*Shift low-low 2 bytes right.*/
matrixFinal = MatrixAdd(matrixT1, matrixFinal); /*No shift for matrixT1 needed.*/
/* Developers note: next step will lose any high-order bits from the high-high product.*/
matrixT4 = MatrixShiftLeftByteImmed(matrixT2, 2); /*Shift high-high 2 bytes left.*/
matrixFinal = MatrixAdd(matrixT4, matrixFinal); /*Final answer. */
```

Figure 4: FastMATH C-language code showing actual implementation of 16 parallel 32-bit × 32-bit multiplications in the matrix unit

The same considerations apply to the 32-bit multiplication diagrammed in Figure 3 and coded in Figure 4. We might retain the high 32 bits instead of the middle 32 bits. This might be necessary in any case, if the high order bits of the high-high portion of the product contain significant information. This means that the low-low product illustrated in the figures contributes nothing to the final product outside of its single carry bit, which might be 0. If high speed is more important than maximum accuracy, one could decide not to perform the low-low multiplication step at all, sacrificing only about 1/2 ULP in the final 32-bit result on average.

```
   FF
 × 12
------
 001E    Multiply low-low
 FFE0    Multiply low-high
 00F0    Multiply high-low
 FF00    Multiply high-high
------
 FFEE
```

Figure 5: Example multiplication of signed hexadecimal 1-byte numbers

7. Block Floating Point

Most signal processing applications can be performed in fixed-point format. For those that require an extended dynamic range without losing accuracy, the block floating-point technique, credited to Wilkinson[11], is available. Although there is a cost in performance, it may make it possible to more efficiently use the available bits. The savings in cost and power by using faster fixed-point rather than floating-point math may more than compensate.

Scaling of fixed-point numbers may be done to reduce the dynamic range to prevent overflow or to increase the number of bits available for the math to increase resolution. The programmer must keep track of the scale factors in order to restore correct values at some stage. If an algorithm is designed for input with known range of values, the scaling and its subsequent removal may be hard-coded for the calculation. For example, the data may be pre-scaled to a desired range. This is also known as gain control, in that the power of the input data is adjusted. In algorithms with multiple steps the intermediate results may also require scaling. An example would be an FFT, in which the results may require a different scale after each stage. For an algorithm designed to accomodate an arbitrary range of data, the "corner cases" may require very conservative prescaling, as well as scaling of intermediate results, to prevent overflow. This can degrade the accuracy for less extreme data sets.

Dynamic scaling is also possible, to recover some of the range-of-values flexibility of floating-point math. Thus, the resolution is adapted to the range of the data. Data with a smaller range (which often is the majority of the data) may then be processed with more bits than data with the maximum range. That is, if the scaling is adjusted to accomodate a few large values, then smaller values will have many leading zeros, losing precision. Dynamic scaling has the cost that scale factors must be determined "on the fly" and may incur a performance reduction. If input data is slowly-varying, with known maximum rate of change, it may be necessary to reassess the scale factor only at preset intervals.

We sometimes observe that certain entire blocks of a data set may require different scaling than others. This can be, for example, data from certain devices, from certain runs of the data, or from certain portions of the calculation, such as for the FFT. In this case we may be able to determine overall scaling factors for a block of data from calibration constants or with coding that does not need to look at the entire block. Therefore, this might be amortized over a large amount of computation. In cases such as this, the developer may be able to dynamically set up scale factors that apply to the entire block of numbers.

This is known as block floating point because it features a scale factor for a block of numbers. The stored scale factor has the same characteristics as a floating-point exponent, except that it is stored in its own memory location, and that it applies to an entire block of numbers rather than being tailored for each number separately. In the FastMATH pro-

cessor it can be stored in a scalar register. This frees up all 16 or 32 bits for significand with a substantial potential accuracy gain. Of course, the trade-off with the number of cycles needed to determine the proper scaling, to apply it and subsequently remove it, and to store the scale factor, needs to be evaluated.

An example of an application well-suited to block floating point might be filters. The data can be scaled as one block and the tap coefficients as another block. Even if the filter is adaptive, with time-varying coefficients, if the dynamic ranges of the coefficients are known in advance it may not be necessary to dynamically compute the block scalings.

8. Error Study Examples

We demonstrate techniques for error analysis with two problems, a finite impulse response filter, and a Cholesky decomposition algorithm. The first illustrates a relatively simple function that lends itself readily to mathematical analysis. The second illustrates a much more complex case, for which the mathematics lends useful guidance, but carrying through a complete analysis may not be the most practical approach. In cases like that we may find it more useful to do a numerical analysis or simulation on the computer, relying on the math primarily only to understand, and verify, the qualitative features.

8.1. Error Study Example: Filtering

Proakis and Manolakis [9] have analyzed the effects of quantization error on digital filters. Both finite impulse response (FIR) and infinite impulse response (IIR) filters are discussed. Proakis and Manolakis find that these errors, if not carefully managed, can lead to significant non-linearities in a filter designed to be linear. Certain filter designs may exhibit oscillations in their output as a direct result of computational errors. This is especially problematical for filters that involve small numbers of bits in their coefficients and/or data.

An FIR filter with M taps can be described by the difference equation

$$y(n) = \sum_{k=0}^{M-1} a_k x(n-k) \quad , \qquad \textit{Eq.(27)}$$

where $y(n)$ is the output at time n and $x(n - k)$ is input at time $n - k$. This sum features M multiplications and $M - 1$ additions. Both additions and multiplications suffer from quantization. If we lump the quantization errors from each multiplication and addition together with a net variance σ^2, and neglect the fact that there are not quite as many additions as multiplications, then the variance in $y(n)$ is given approximately by

$$V(y) = \sum_{k=0}^{M-1} \sigma^2 = M\sigma^2 \quad . \qquad \textit{Eq.(28)}$$

Here we assume that the errors in each multiplication-addition are independent. Now we can take advantage of the FastMATH architecture, which is structured such that a multiply-add is accumulated in one of the 40-bit accumulators with no quantization error until the final result is moved into a register. For a sequence of multiply-adds in the accumulator, as for example in Eq.(27), the factor M in Eq.(28) is replaced by 1, but more bits are lost in the quantization.

The filter coefficients a_k also likely suffer quantization errors. If the filter is adaptive, with a_k being recalculated at intervals, then there is an additional calculational error in the a_k. There are likely also quantization errors in the input values x. Include these errors in any general error analysis.

If the calculations also suffer a bias, as in truncation or rounding systems (Table 2), then the biases can be treated the same way. If b is the total computational bias in one multiply-add step, the overall biases add to form

$$b(y) = Mb \quad . \qquad \textit{Eq.(29)}$$

The frequency response of this filter is

$$H(\omega)= \sum_{k=0}^{M-1} a_k e^{-j\omega k} \quad . \qquad \textit{Eq.(30)}$$

If each a_k has error ε, then the error in the frequency response is

$$\Delta H(\omega)= \varepsilon \sum_{k=0}^{M-1} e^{-j\omega k} \quad . \qquad \textit{Eq.(31)}$$

The variance is then, assuming independent average errors σ_a in each a_k, $V[H(\omega)] = M\sigma_a^2$. We have so far neglected the errors in computing $e^{-j\omega k} = \cos(\omega k) - j \sin(\omega k)$, and in the multiplication by a_k and summation. The quantization errors in the multiplications and summation can be reduced by use of the FastMATH MACs, as described at the start of this section. Errors in computing the trigonometric functions must be evaluated by the developer for the algorithm at hand. Note that, from Eq.(31), the net error $\Delta H(\omega)$ may be reduced to a small value as the phase in the sum cycles from $\omega k = 0$ to $(M - 1)\ \omega$ (cancellation of errors is discussed in Section 5.5). Be aware, however, that the relative error might be large if all the a_k have similar values because of the possible cancellations of terms (see Section 5.4.1).

Proakis and Manolakis [9] show in considerable detail that certain treatments of FIR or IIR filters are less sensitive to quantization errors. They recommend a form known as a cascade form to minimize these problems, which sometimes are severe. The FastMATH processor, with its 40-bit MACs, can sometimes be used to great advantage in this type of algorithm.

8.2. Error Study Example: Cholesky Decomposition

We demonstrate error study with a sophisticated algorithm coded on the FastMATH processor. Cholesky decomposition[2] is a technique incorporated into the solution of linear equations of the form $\boldsymbol{Ax = b}$, where $\boldsymbol{x}$ is a vector of unknown values to be determined, $\boldsymbol{b}$ is a vector of known (e.g., measured) values, and $\boldsymbol{A}$ is a known non-singular symmetric positive-definite[10] matrix expressing the relation between the unknowns and the measurements. More efficient than finding the inverse of $\boldsymbol{A}$ and then using $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$, the ("rootless" form of) Cholesky decomposition finds a lower-triangular matrix $\boldsymbol{L}$ and a diagonal matrix $\boldsymbol{D}$ that satisfy $\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$, where the T signifies transpose. Such a decomposition always exists for $\boldsymbol{A}$ as specified. The elements of $\boldsymbol{D}$ are all positive. Intrinsity has implemented a complex fixed-point version of Algorithm 4.29 of Ref. [2] for the FastMATH processor. SIMD operation allows the computation of 16 decompositions in parallel. The solution of the linear equations follows the decomposition in two more steps. The total time is less than obtaining the inverse and performing the matrix-vector multiplication.

We restrict our attention to the decomposition, i.e., finding $\boldsymbol{L}$ and $\boldsymbol{D}$. The algorithm begins by solving for the upper left corner of these matrices and proceeds down $\boldsymbol{L}$ and $\boldsymbol{D}$ together by rows to the complete solution. Each subsequent row requires additional computation because it spans more columns in the lower-triangular format of $\boldsymbol{L}$. Input is (s0.15), achieved by pre-scaling if needed, and output is (s15.16). The decomposition requires one division for each row. This is what determines the fixed-point format of the output: 32 bits and symmetric about the radix point (see Section 4, "Methodology: Fixed-Point Implementation"). All math is done in (s15.16) format, with multiplications as described in Section 6, "Example: 32-bit Multiplication in (s15.16) Format."

A few terms from the $\boldsymbol{L}$ matrix are given in Eq.(32). The lowercase a terms are elements of $\boldsymbol{A}$. All denominators in the divisions are real variables, a consequence of the fact that the $\boldsymbol{A}$ matrix is positive definite. The subscript notation refers to row and column, in that order. We number the rows and columns following the convention of Ref [2], beginning with row 1, column 1. The * notation signifies complex conjugate. We observe that all terms for column 1 are similar; a corresponding pattern continues for each column.

10 Under certain conditions[2], $\boldsymbol{A}$ need not be positive definite.

$$L_{21} = a_{21}/a_{11}$$
$$L_{31} = a_{31}/a_{11}$$
$$L_{41} = a_{41}/a_{11}$$
$$L_{32} = \frac{(a_{32} - a_{31}^{*} a_{21}/a_{11})}{(a_{22} - a_{21}^{*} a_{21}/a_{11})} = \frac{(a_{32} - a_{31}^{*} L_{21})}{(a_{22} - a_{21}^{*} L_{21})} \quad . \qquad \textit{Eq.(32)}$$

The quantization errors for the first column arise entirely from the division. For the second and subsequent columns it is much more complicated. If we wish, we can derive an expression for the variance of $\boldsymbol{L_{32}}$ from Eq.(32) using propagation of errors. Both numerator and denominator feature a subtraction, a complex multiplication (requiring two multiplications and an addition or subtraction for the real part and imaginary part, separately), and a division.

It is already clear from the few terms shown here that each subsequent column requires more terms. In the absence of strong correlations, our first expectation is that this implies that the computational errors increase as the column number increases; variances due to independent computations add. But be aware that most efficient algorithms are recursive. The recursive nature of the algorithm may require care with possible correlations. This example demonstrates that effect.

The operations to calculate $\boldsymbol{L_{32}}$ are not independent. That is, both numerator and denominator include the error from $\boldsymbol{L_{21}}$, which is apparent from the second form in Eq.(32). You might consider evaluating $\boldsymbol{L_{32}}$ from the first form in Eq.(32). Then, each division is done independently and a correlation in the calculational errors can be expected to disappear. However, the algorithm is recursive; it does not recompute terms that have been computed previously. Therefore, the second is the relevant form for the algorithm and the correlation between numerator and denominator will contribute to the final variance. As the error in $\boldsymbol{L_{21}}$ varies, it affects both numerator and denominator similarly, on average.

We can understand general principles from this sort of theoretical analysis. However, because the expressions are extremely complicated, especially as the order of columns and rows increases, we choose to make the evaluation numerically. Note also that the elements of $\boldsymbol{A}$ are correlated because of its symmetric, positive-definite nature. In the design phase, an algorithm can be coded in scalar floating point code (double precision, if necessary) and scalar fixed-point code, and the errors evaluated from the differences before the SIMD coding is performed. In that way, the accuracy of the choice of algorithm can be evaluated before the parallelization is done.

Average discrepancies (in absolute value) between the fixed-point and a floating-point implementation for each individual element of the real parts of $\boldsymbol{L}$ are shown in Figure 6 (figure prepared using MatLab). These data are averages over 1000 8×8 input matrices with random elements (satisfying the requirements of being symmetric and positive definite). The vertical axis of the plot is truncated at 4.0 to better show the variations in the low-lying columns; the last element (row 8, column 7) has a value of 14. The values may be compared to $\boldsymbol{L}$ matrix elements that average around a few $\times 10^4$ for these examples. The average overall SNR from the real part of the $\boldsymbol{L}$ matrix is 65. It is immediately clear from the figure that the results are remarkably constant within columns of $\boldsymbol{L}$, differing only from column to column, confirming the prediction of the theory. The standard deviations (not shown) of these discrepancy values are at least as large as the average discrepancies, ranging from one to three times as large. Since the distribution of the discrepancies (in absolute value) cannot go below 0.0. this indicates that the discrepancy distributions have a very long tail to high values.

We also note the general trend toward larger errors as the column number increases, again as predicted. If we had coded the algorithm to begin at the lower right and work upward, we would expect this trend to be reversed, but with the same overall SNR. An exception to the general trend is column 2. The error for column 2 is much lower than for either neighboring column. This must be due to the correlations mentioned above, which have had a factor of 2 effect on the error for that column. In this case, the efficiency of the algorithm is further rewarded by an improvement in the errors. A detailed analysis of errors for larger matrices (not shown) indicates a further oscillation in the error; even in the regime of column number where the SNR is showing a general increase, the rate of increase oscillates due to correlations.

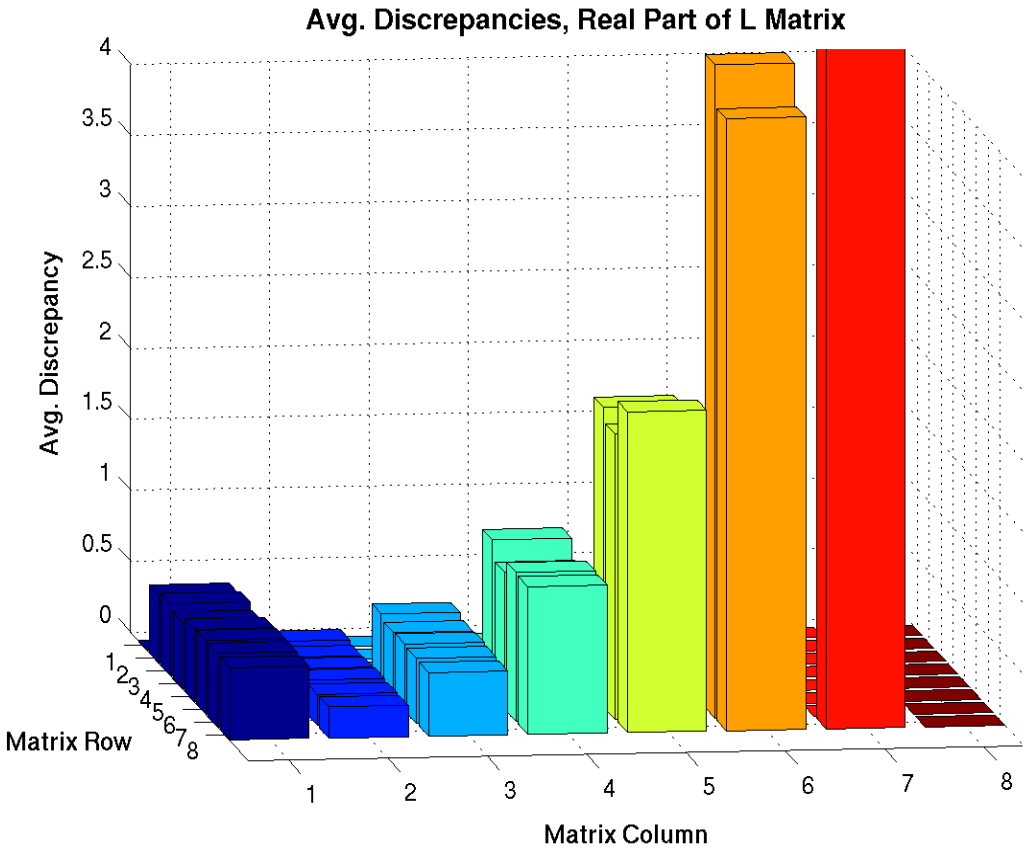

Figure 6: Average absolute discrepancies for real part of lower-triangular matrix *L* from Cholesky decomposition with random input; 8 × 8 example

If some matrix elements are more important than others, the developer can choose to reorganize the problem so those are in the positions with smallest error. These kinds of statistics illustrate the process of understanding the errors in complex algorithms from actual measurements on the computer. The magnitudes of the results shown in Figure 6 depend on the algorithm for generating the random input, the scaling of the input, and other factors that vary with the problem. The developer needs to make an effort to understand the specific details of the problem at hand.

9. Conclusions

Nearly any algorithm can be adapted to run on a fixed-point machine to take advantage of its normally faster instructions/clock cycle and reduced power features. There is a clean methodology for coding an algorithm on a fixed-point machine. The FastMATH machine offers a high degree of parallelism combined with a 2-GHz cycle speed (1-GHz in the low-power FastMATH-LP processor) and a 1-Mbyte L2 cache that directly feeds the SIMD processor. It can offer faster execution with less power consumption than most competing floating-point machines.

Computational errors can affect the choice of an algorithm and its implementation, for both floating- and fixed-point machines. The range of the data may determine the necessity for scaling to avoid overflows or to preserve accuracy. On the other hand, in some cases with limited data range it may be possible to increase the range by appropriate scaling, to improve the resolution. The FastMATH 40-bit MAC can be used to avoid quantization errors in intermediate steps.

Intermediate results in the calculations also need to be understood, and may require a different scaling. Before proceeding with the code the developer may find it advantageous to study the floating-point implementation to understand the range of the data and the range of the intermediate results, if any. Once this is understood then one can design the location of the radix point is possible at each stage of the algorithm, as well as the appropriate scale factor(s) to fit within that design.

10. Acronyms

ADC	analog-to-digital converter
BFP	block floating point
DMA	direct memory access
FFT	fast Fourier transform
FIR	finite impulse response filter
GHz	gigahertz
GOPS	giga operations per second
IIR	infinite impulse response filter
LSB	least significant bit
MAC	multiply-accumulator register
MSE	mean square error
rms	root mean square
SIMD	single instruction, multiple data
SNR	signal-to-noise ratio
ULP	unit of least precision

11. References

[1] Abramowitz, Milton and Irene A. Stegun, *Handbook of Mathematical Functions* (New York: Dover Publications, Inc., 1972).

[2] Engeln-Müllges, Gisela and Frank Uhlig, *Numerical Algorithms with FORTRAN*, (New York: Springer-Verlag, 1996).

[3] Garg, Hari Krishna. *Digital Signal Processing Algorithms: Number Theory, Convolution, Fast Fourier Transforms, and Applications*, CRC Press, New York (1998).

[4] Institute of Electronics and Electrical Engineers, *IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985; R1990)*. IEEE, http://www.ieee.org/ (accessed September 30, 2003).

[5] Intrinsity, Inc., *Intrinsity Software Application Writer's Manual*, version 0.3 (beta), http://www.intrinsity.com/ (accessed September 30, 2003).

[6] Johnson, Norman L., Samuel Kotz, and N. Balakrishnan, vol. 2 of *Continuous Univariate Distributions* (New York: John Wiley & Sons, Inc., 1995).

[7] Knuth, D. E., *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, 3rd ed. (New York: Addison-Wesley, 1998).

[8] Meyer, Raimund, "Error Analysis and Comparison of FFT Implementation Structures," *Proceedings of ICASSP'89*, (1989): 888–891.

[9] Proakis, John G. and Dimitris G. Manolakis, *Digital Signal Processing*, 3rd ed. (New Jersey: Prentice Hall, 1996).

[10] Stuart, Alan and J. Keith Ord, vol. 1 of *Kendall's Advanced Theory of Statistics*, (New York: Oxford University Press, 1987).

[11] Wilkinson, J. H., *Rounding Errors in Algebraic Processes* (Prentice-Hall, Englewood Cliffs, NJ, 1963).

12. Revision History

Revision	Date	Brief Description
1.0	10/9/2003	Initial release