

Where Did All My Decimals Go?

Chuck Allison
Utah Valley State College

January 2006 (Updated September 2022)

Abstract

It is tremendously ironic that computers were invented with number crunching in mind, yet nowadays most CS graduates leave school with little or no knowledge of the intricacies of numeric computation. This paper surveys what every CS graduate should know about floating-point arithmetic, based on experience teaching a recently-created course on modern numerical software development.

[Note: This paper was first delivered to the Consortium for Computing Sciences in Colleges (CCSC), Rocky Mountain Region, October 2005]

Missiles miss their targets. Spacecraft malfunction. Control systems lose control. Why? Because someone didn't do their numerical homework. Today's CS curricula have broadened their scope to the point that the *raison d'être* for automatic computation has practically been forgotten. While most jobs in computing do not require in-depth knowledge of floating-point number systems, every CS graduate should be sufficiently fluent with numerics to produce accurate results.

Consider the following expression, expressed in C/Java syntax:

$$1.0f - 0.2f - 0.2f - 0.2f - 0.2f - 0.2f$$

(The “f” suffix denotes the float type, a 32-bit floating-point number.) Students are shocked to discover that the value resulting from this expression is not zero. The value returned by an IEEE-compliant system is -1.49012×10^{-8} . Is this good enough? Well, it depends on how you define “good enough.” Not if you're positioning a satellite or guiding a deep space vehicle—small errors can cause big problems. Even in common applications, seemingly benign errors can propagate to the point of rendering results useless at best and downright dangerous at worst. If computer scientists and programmers don't understand what's going on here, who will?

Number Systems

The key to numeric success is in understanding the structure and behavior of the number system being used. Most programmers understand integers quite well. They know that they can overflow but otherwise cause few surprises. Integers are a special case of a fixed-point number system. In such a system the number of digits before and after the radix point is fixed. Consider as an example the system of base-10 numbers with three digits before the decimal and one after. This system represents the following 19,999 evenly-spaced numbers:

$$\{-999.9, -999.8, \dots, 0, \dots, 999.8, 999.9\}$$

Additions and subtractions of fixed-point numbers that do not result in overflow are exact, a very handy feature in financial calculations. Multiplications and divisions can result in numbers with more decimals than can be stored, so a minimal amount of rounding error occurs for these operations. In a round-to-nearest scheme the error of each multiplication is never more than half the unit in the last place ($\frac{1}{2}$ of .1, or .05, in this case).

Quite often we are more interested in percentage error, or *relative error*, instead of absolute error. Consider the error that ensues when representing the number $x = 865.54$. The closest machine number to x is 865.5, which represent an absolute error of .04, and a relative error of $\frac{865.54-865.5}{865.54} = .00005$. Now using the same five digits, consider the number .86554 whose closest machine companion is .9. This time the absolute error is .03446, but the relative error is $\frac{.9-.86554}{.86554} = .04$. Hence the relative error is affected by the magnitude of the number, not the digits used. This is counter-intuitive to people accustomed to using scientific notation, which is the basis for floating-point number systems.

In a floating-point number system, the number of digits is fixed, but the decimal point is not (it “floats”). This requires storing both the mantissa (the significant digits) and the exponent, but results in a much larger range of numbers as compared to a fixed-point number system using a similar amount of storage. Consider the floating-number system defined by the following four parameters:

- $B = 10$ the base (decimal, in this case)
- $p = 4$ the precision (number of digits in the mantissa)
- $m = -2$ the minimum exponent allowed
- $M = 3$ the maximum exponent allowed

Numbers in this system are of the form $d_0.d_1d_2d_3 \times 10^e$ where the exponent, e , ranges from -2 to 3. We typically require that d_0 be positive (otherwise the bit patterns for some numbers turn out not to be unique, which impedes developing efficient algorithms for floating-point arithmetic; a system requiring $d_0 \neq 0$ is called a *normalized* system), so the total number of numbers available in this system is $9 \times 10 \times 10 \times 10 \times 6 \times 2 = 108,000$, the smallest positive magnitude is .01 and the largest is 9,999. The numbers in this system are no longer evenly spaced (it’s obvious why, but will be clarified shortly), but “cover” the same range as the fixed-point system seen earlier.

Now compare the relative rounding error of storing in this system the same two numbers used earlier. The relative error of 865.54 is $\frac{865.54-865.5}{865.54} = .00005$, and the error representing .86554 turns out to be the very same quantity (namely $\frac{.86554-.8655}{.86554} = .00005$). This is a very nice feature of floating-point number systems: the relative rounding error is uniform throughout the system. To summarize, the advantages of floating-point number systems over fixed-point systems are twofold: 1) a greater range of numbers is obtained with similar storage capacity, and, 2) the relative error due to rounding is independent of the magnitude of the number.

Arbitrary-precision Arithmetic

If the idea of any rounding error at all bothers you, you’re not alone. This is the impetus behind arbitrary-precision packages, such as Java’s `BigDecimal` class, and the many arbitrary-precision

packages available for C and C++. If ultimate precision is crucial, then by all means use such a package, but there remains the ever-present trade-off between accuracy and efficiency. Real-time applications and large number-crunching tasks require the speed that only fixed-size floating-point hardware can provide. This is why most scientific applications use floating-point arithmetic, and why it is crucial to train CS students to compute responsibly.

Spacing in Floating-point Systems

Since the logical radix point in a floating-point number system is not fixed, the contribution of the digit in the last place varies, depending on the magnitude of the number (which in turn determines the mantissa and the exponent being used). For example, the number 865.5 in our toy system is actually represented as 8.655×10^2 . The next number to the right in this system is 8.656×10^2 , representing a spacing of $.001 \times 10^2 = .1$. Therefore, numbers in this system between 10^2 and 10^3 are .1 apart, and the spacing decreases by a factor of 10 as you move to the next range of powers to the left (10^1 to 10^2). In general, for a number system using a base, B , and a precision, p , the spacing between numbers in the range $[B^e, B^{e+1})$ is B^{1-p+e} . So while the absolute rounding error in fixed-point systems is constant and the relative error changes, the situation is reversed for floating-number systems.

To be more precise, the relative error between *consecutive* floating-point numbers isn't really uniform—it actually falls within a fixed range, known as the system *wobble*. Consider the sequence of numbers between successive powers of the system base, $[B^e, B^{e+1})$:

$$\{B^e, B^e + B^{1-p+e}, B^e + 2 \times B^{1-p+e}, B^e + 3 \times B^{1-p+e}, \dots, B^{e+1}\}$$

As you consider the relative spacing, r , between consecutive numbers in this range, you find that it falls in the range

$$B^{-p} \leq r \leq B^{1-p}$$

(This reveals why binary systems are the best—the wobble is determined by a factor of B , and 2 is the smallest possible base). The quantity B^{1-p} is the upper bound of such error and is called *machine epsilon* (ε), and is considered the single most important measure of a floating-point number system. Using ε we can rewrite the system wobble as

$$\frac{\varepsilon}{B} \leq r \leq \varepsilon$$

A little algebra also reveals that the *absolute* spacing, s , between floating-point numbers in the neighborhood of a number, x , obeys

$$\frac{\varepsilon|x|}{B} \leq s \leq \varepsilon|x|$$

This all may seem like so much mathematical meandering, outside the interest of the typical CS student, but it is crucial for crafting correct numerical algorithms. Suppose, for instance, that it is required to write a root-finding algorithm, and the user is allowed to specify a desired accuracy in the result. A C++ specification for such a function might look like the following.

```
double bisect(double tol, double a, double b, double f(double));
```

The first parameter, `tol`, is the user's desired error tolerance. This particular procedure, as its name reveals, does a bisection reduction of the interval $[a, b]$ until either a zero of the function $f(x)$ is found or the interval has been reduced to a width of `tol`. But what if the spacing between numbers in the neighborhood of the solution is greater than `tol`? Then the algorithm, which likely has a loop condition like the following, will never terminate:

```
while (b-a > tol)
```

The algorithm needs to be tuned so that the requested tolerance is not unreasonable. The following modification brings the algorithm back into reality:

```
double eps = numeric_limits<double>::epsilon();
tol = max(tol, eps*abs(a));
tol = max(tol, eps*abs(b));
while (b-a > tol) { . . .
```

The first C++ statement above obtains the machine epsilon for the running environment. Then the tolerance is adjusted to be no smaller than the spacing between numbers in the interval $[a, b]$, using the spacing formula developed earlier.

ULPS

Another important measure of the accuracy of a computation considers the number of floating-point numbers between two machine numbers. For example, the bisection algorithm for root finding can actually guarantee “maximum accuracy” by reducing the interval $[a, b]$ to an interval that contains no floating-point numbers at all (in other words, the final interval, $[a', b']$, is such that a' and b' are consecutive floating-point numbers). Here is a C++ program that does exactly that:

```
void bisect(double& a, double& b, double f(double)) {
    if (a >= b || sign(f(a)) == sign(f(b)))
        throw logic_error("Bad interval");

    // Bisect until zero found or interval is 1 ulp
    for (;;) {
        // Loop invariants
        assert(b > a);
        double fa = f(a);
        int sign_a = sign(fa);
        assert(sign_a != sign(f(b)));

        double c = (a+b)/2.0;

        // At 1 ulp?
        if (c <= a || c >= b)
            break;
```

```

double fc = f(c);
if (fc == 0.0) {
    // Stumbled across a zero.
    a = c;
    b = c;
    break;
}
else if (sign_a == sign(fc))
    a = c;
else {
    assert(sign(f(b)) == sign(f(c)));
    b = c;
}
}
}

```

This program maintains an interval that brackets a zero of the function f , provided that the original interval $[a, b]$ does. It continues until either it stumbles across a zero at the bisection point, c , or until there are no floating-point numbers between the endpoints of the current interval. The latter condition is detected by the condition:

```

if (c <= a || c >= b)
    break;

```

When there are no floating-point numbers between the current endpoints a and b , then c , the average of a and b , will be equal to either a or b . In either case, it's time to stop. We say that a and b are “1 ulp apart”, where “ulp” stands for “unit in the last place”, or equivalently, “unit of least precision”. In other words, b is obtained from a by just incrementing a 's last mantissa digit by 1. At this point it will do to report either endpoint as the root. Formally, we say that the real numbers a and b are u ulps apart if the number of floating-point intervals (including fractions thereof) between a and b is u .

It is possible to approximate the numbers of ulps between two numbers, x and y , by computing the number of floating-point intervals between their corresponding machine numbers, x' and y' . This is usually left as an exercise to the students, so the procedure will not appear here, but it is easy to calculate the maximum number of bisections that will occur in root finding exactly, when machine numbers a and b are between the same consecutive powers of the floating-point base. Suppose a binary base is used, and that the system precision is 24. Given a starting interval of $[1, 2]$, we can compute that $\text{ulps}(1,2)$ is the interval width $2 - 1 = 1$ divided by the spacing of numbers between 1 and 2, which we determine by the formula B^{1-p+e} :

$$\frac{2 - 1}{2^{1-24+0}} = \frac{1}{2^{-23}} = 2^{23}$$

Therefore, no more than 23 bisections are required using this number system to reduce the interval $[a,b]$ to an interval 1 ulp in width. There is no risk of the procedure failing, we get the best accuracy possible from our machine, and we don't even have to specify an error tolerance!

Sources of Numeric Error

We have already illustrated simple rounding error (aka roundoff), which is the unavoidable error that results from approximating a real number that is not representable within a floating-point number system (it falls in the gap between two successive machine numbers). This error can grow astronomically if we're not careful. A typical case of concern occurs when subtracting two numbers nearly equal in magnitude. Most of the leading digits are equal, so they disappear after the subtraction. The result is that the natural rounding error that taints the digit in the last place gets promoted to a more significant decimal place. For example, consider the following subtraction:

$$.123456789 - .123456788$$

You might be content calling this 1×10^{-8} , but considering that the last digits of machine numbers are always suspect because of roundoff (these numbers may have had many more decimals to the right which the floating-point system in use could not store), the difference of those two last digits in the pair of numbers above is also suspect, but now it has been promoted to the most significant digit position we have! The difference can be totally wrong (maybe the true error is really closer to 2—an error of 100%). This is known as catastrophic cancellation, and the numeric developer must be ever on guard for it.

Sometimes cancellation can be avoided by rewriting a formula. When writing a quadratic equation solver, blindly using the quadratic formula will sometimes be a mistake. Here is such a program, which solves the equation $x^2 - 100,000 + 1$:

```
// quad1a.cpp (uses single precision)
// Solves the equation x^2 + bx + c = 0 (i.e., a == 1)
#include <cfloat>
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    cout.precision(FLT_DIG-1);
    cout.setf(ios::showpoint);
    cout.setf(ios::scientific);
    float b = -1.0e5f;
    float c = 1.0f;
    float d = b*b -4.0f*c;

    if (d >= 0) {
        // Real roots
        float e = sqrt(d);
        float r1 = (-b - e) / 2.0f;
        float r2 = (-b + e) / 2.0f;
        cout << "r1 = " << r1 << ", r2 = " << r2 << endl;
    }
    else {
```

```

        // Complex roots
        float p = -b / 2.0f;
        float q = sqrt(-d) / 2.0f;
        cout << "r1 = " << p << " + " << q << "i\n";
        cout << "r2 = " << p << " - " << q << "i\n";
    }
}

/* Output:
r1 = 0.00000e+00, r2 = 1.00000e+05
*/

```

The actual roots to 11 digits are 99999.99999 and 0.0000100000000001. Again, you might consider this to be just fine, but we can do better. Because b is so large relative to the constant term, the quantities b and e (which is the discriminant $\sqrt{b^2 - 4ac}$) are nearly equal, so the expression $(b - e)$ above suffers from cancellation, skewing the result. We can take advantage of the fact that the product of the roots in a quadratic equation is always the constant term, c . In the following version of the program, we first compute the root that will not risk cancellation (by checking signs), and divide that root into c to get the other root.

```

// quad2a.cpp (uses single precision)
// Solves the equation x^2 + bx + c = 0 (i.e., a == 1)
// This version avoids potential catastrophic cancellation
// and computes r2 as c/r1.

#include <cfloat>
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    cout.precision(FLT_DIG-1);
    cout.setf(ios::showpoint);
    cout.setf(ios::scientific);
    float b = -1.0e5f;
    float c = 1.0f;

    float d = b*b - 4.0f*c;
    if (d >= 0) {
        // Real roots
        float e = sqrt(d), r1;
        if (b >= 0)
            r1 = (-b - e) / 2.0f;
        else
            r1 = (-b + e) / 2.0f;
        float r2 = c/r1;
    }
}

```

```

    cout << "r1 = " << r1 << ", r2 = " << r2 << endl;
}
else {
    // Complex roots
    float p = -b / 2.0f;
    float q = sqrt(-d) / 2.0f;
    cout << "r1 = " << p << " + " << q << "i\n";
    cout << "r2 = " << p << " - " << q << "i\n";
}
}

/* Output:
r1 = 1.00000e+05, r2 = 1.00000e-05
*/

```

This time the other root is much closer to the true answer.

The interaction of roundoff with the subtraction of numbers of vastly differing magnitudes can also wreak havoc. To illustrate, recall that elementary functions such as e^x and $\sin x$ have simple Taylor Series expansions (which is how they're computed by software libraries). The Taylor Series expansion for e^x about $x = 0$ is:

$$\sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} \dots$$

The terms shrink in magnitude as i increases, so for ultimate accuracy we can just keep adding until the i th term is so small it makes no difference in the result, as the following function illustrates.

```

float myexp(float x) {
    float sum1 = 1;           // First term, and partial sum
    float sum2 = 1 + x;      // Second partial sum
    float term = x;          // Second term
    int i = 1;               // Index already included in sum
    while (sum2 != sum1) {
        sum1 = sum2;
        term = term*x/++i;    // Compute next term
        sum2 += term;
    }
    return sum2;
}

```

The answers for positive x are just fine, but for negative x the results aren't even close. For example, computing $f(-55.5)$ yields a negative number (-2.90487×10^{15} vs. the true answer of $7.88235979060085 \times 10^{-25}$)!

The problem here is that the initial terms of the Taylor Series are much larger in magnitude than the true answer. The Taylor Series requires subtraction in this case to whittle its sum down to the true, minuscule answer, but the normal roundoff in the last place of the initial terms is in a more significant decimal position than the digits of the expected answer, so we were doomed from

the start! In this case there is an easy work-around (compute $\frac{1}{e^{-x}}$ for negative x instead), but that's just pure luck.

The base chosen for a floating-point number system also affects accuracy. For example, in base-2, the fraction .1 is an infinite repeating decimal (.000110011). Just as the fraction $\frac{1}{3}$ is always .3 ulps off in a decimal system, it is impossible to represent .1 exactly in a binary system, so some rounding error is inevitable. This explains why subtracting $0.2f$ from $1.0f$ five times as seen in the beginning of this paper did not yield zero. Even using double precision (double, in C or Java) the result is not “zero,” but instead comes back as $5.55111512312578 \times 10^{-17}$.

Overflow can occur when you least expect it. The bisection algorithm illustrated above routinely calculates the midpoint of its interval. But what if the interval endpoints are greater than half the largest representable floating-point number? Then the sum $a + b$ will overflow. An alternative midpoint formula, $a + \frac{b-a}{2}$, can be used instead (although this formula is not so good when a and b are both small). The point of all this discussion is that one cannot always implement formulas directly—you must take into account the context and consequences of the fixed precision of floating-point number systems.

The IEEE Floating-point Standards

Writing portable numeric software has been very tedious, if not practically impossible, because different computing platforms have used different floating-point number systems. Bases of 2, 10, and even 16 have been used. Calculators and financial software packages use base 10 to avoid the roundoff that occurs when changing bases. Since 1985, however, most computing systems used for scientific computation have used IEEE formats as specified in the standards IEEE 754 and IEEE 854. The latter pertains mostly to calculators and financial packages, so this section will discuss the binary formats found in IEEE 754.

The IEEE 754 specification describes two base-2 floating-point systems, known as *single precision* and *double precision*. These correspond respectively to the implementations of `float` and `double` in most languages, and behave identically across platforms. Single precision is a 32-bit format that uses 1 bit for the sign, 8 for the exponent, and 24 for the mantissa. You may immediately notice that these numbers add up to 33, not 32. That's because IEEE single precision is a normalized binary system—the leading bit is always 1—so it is just assumed and not stored. (This is a second reason why 2 is the optimal base for a floating-point system—you get a free bit of storage for each number). The corresponding field sizes for double precision are 1 for the sign, 11 for the exponent, and 53 for the mantissa (52 stored). The exponents are stored in a *biased* format, meaning that they are offset by a fixed value (127) so that 0 is the smallest exponent stored, and 255 is the largest (all bits are 1), but the actual exponents they represent could conceivably range from -127 to 128. For example, the number 6.5 is represented in 32 bits as:

0 10000001 1010000000000000000000

(Spaces have been inserted to separate the three logical parts of the number for readability; on a little endian machine the bytes would of course be reversed.) The sign bit is 0, so this number is non-negative. The exponent field is 129, so this represents a logical exponent of 2. The fraction is .625, so the full mantissa is 1.625, giving the number

$$1.625 \times 2^2 = 6.5$$

The offset for double precision exponents is 1023.

But what about the number zero? If the leading bit of the mantissa is always 1, zero can never be represented. To represent zero, the stored exponent field of all 0-bits (logical exponent value of -127) is reserved. If the exponent field *and* the fraction are all zeroes, then the number represented is zero. Since the sign bit is independent of the exponent field, both a positive and negative zero are representable, but they both correspond to the pure number zero in floating-point computations.

Since the logical exponent value -127 is now off limits for normal purposes, it is also used to define *subnormal* floating-point numbers. If the exponent field is all zeroes, but the fractional part is not, then the number is formed by assuming a leading bit in the one's place of 0, not 1, and the logical exponent is assumed to be -126 (not -127; otherwise we'd be skipping numbers). Such numbers are called subnormal because they are not normalized (the leading mantissa bit is 0 instead of 1), and they are smaller in magnitude than the normalized values. The smallest normalized magnitude is:

$$0\ 00000001\ 000000000000000000000000 = 2^{-126} \approx 1.17549 \times 10^{-38}$$

The largest subnormal magnitude is:

$$0\ 00000000\ 111111111111111111111111111111 = (1 - 2^{-23}) \times 2^{-126} = 2^{-126} - 2^{-149}$$

and the smallest subnormal magnitude above zero is 2^{-149} . The spacing between positive, single-precision subnormals is 2^{-149} all the way down to zero (hence the relative distance between consecutive subnormals exceeds the system wobble).

IEEE also defines special numbers that are very handy for corner cases in numeric calculations. For example, dividing by zero shouldn't always be considered a bad thing. Every calculus student knows, for example, that

$$\lim_{x \rightarrow 0} \frac{1}{x} = \infty, \lim_{x \rightarrow \infty} \frac{1}{x} = 0$$

Often when infinity pops up we throw up our hands and quit, but this isn't always necessary. A well-known formula from electronics, for example, looks something like:

$$\frac{1}{\frac{1}{r_1} + \frac{1}{r_2}}$$

The output of this formula is well defined when the resistances are either 0 or "infinity". For example, when r_1 is 0, the denominator tends to infinity, so the final result is zero. Likewise, if r_2 is infinity, then the answer is r_1 . To accommodate these reasonable but exceptional cases, IEEE 754 incorporates the notion of infinity. A floating-point number is interpreted as infinity if its exponent bits are all set, and its fraction is zero. This means that the physical exponent 255 (logical 128) is out of commission for normalized numbers, therefore, the largest single-precision normalized number is

$$0\ 11111110\ 111111111111111111111111111111 = \frac{2^{24} - 1}{2^{23}} \cdot 2^{127} \approx 3.40282 \times 10^{-38}$$

When the exponent field is all 1's but the fraction is not zero, the number is the special value "Not a Number," or NaN for short. A NaN occurs when you try some invalid operation, such as taking the square root of a negative number. Once a NaN is introduced into a computation, any ensuing computation results in a NaN. A NaN is the only floating-point value that does not compare equal to itself, but that's okay—it's not a number! The following program illustrates all of these concepts.

```

// ieee.cpp: Illustrates IEEE special values
#include <cassert>
#include <cmath>
#include <iostream>
#include <limits>
#include <typeinfo>
using namespace std;

float f(float r1, float r2) {
    return 1.0f / (1.0f/r1 + 1.0f/r2);
}

int main() {
    cout.setf(ios::boolalpha);
    int radix = numeric_limits<float>::radix;
    int digits = numeric_limits<float>::digits;
    cout << "radix = " << numeric_limits<float>::radix << endl;
    cout << "fraction bits = " << numeric_limits<float>::digits
        << endl;
    cout << "smallest normal = " << numeric_limits<float>::min()
        << endl;
    cout << "smallest subnormal = "
        << numeric_limits<float>::denorm_min() << endl;
    cout << "largest number = " << numeric_limits<float>::max()
        << endl;
    float zero = 0.0f;
    float inf = 1.0f/zero;
    cout << inf << endl;
    assert(inf == numeric_limits<float>::infinity());
    cout << "f(0,1) = " << f(0.0f, 1.0f) << endl;
    cout << "f(inf,2) = " << f(inf, 2.0f) << endl;
    float nan = sqrt(-1.0f);
    cout << nan << endl;
    cout << "nan + 2 = " << nan + 2 << endl;
    cout << (nan == nan) << endl;
}

```

```

/* Output:
radix = 2
fraction bits = 24
smallest normal = 1.17549e-38
smallest subnormal = 1.4013e-45
largest number = 3.40282e+38
Inf
f(0,1) = 0

```

```
f(Inf,2) = 2
NaN
nan + 2 = NaN
false
*/
```

The cleverness of the IEEE format also makes computing `ulps(x,y)` simple: subtract their bit representations as integers:

```
auto xint = *reinterpret_cast<unsigned long long*>(&x);
auto yint = *reinterpret_cast<unsigned long long*>(&y);
auto ulps = yint - xint;
```

This works as long as x and y are on the same side of zero. Otherwise `ulps(x,y)` is undefined.

Summary

Computer science students don't need to be full-fledged numerical analysts, but they may be called upon to write mathematical software. Indeed, scientists and engineers use tools like Matlab and Mathematica, but who *implements* these systems? It takes the expertise that CS graduates have to write such sophisticated software. Without knowledge of the intricacies of floating-point computation, they will make a mess of things. In this paper I have surveyed the basics that every CS graduate should have mastered before they can be trusted in a workplace that does any kind of computing with real numbers. I began teaching a course on this topic at the junior level in 2005, and it has been well received. Students come to the course with a second semester of calculus behind them and in-depth experience with a modern programming language, preferably C++. Programming assignments include the following:

1. Write a function, `ulps(x, y)`, that computes the number of full floating-point intervals between any two positive machine numbers. Write it generically to transparently accommodate both `float` and `double`.
2. Write functions to extract information from IEEE floating-point numbers, such as `sign`, `exponent`, `fraction`, `mantissa`, and whether the number is normal, subnormal, infinity, or NaN. Also, write a function, `ulp(x)`, to return the spacing of machine numbers in the vicinity of x . All functions should work regardless of endian-ness of platform.
3. Write a library-quality $\sin x$ function. Use modern argument-reduction and economization techniques to efficiently derive an accurate result.
4. Implement a root-finding algorithm that finds the root of a function in an interval where a root is known to exist. Use a hybrid of the secant and bisection methods to obtain maximum machine accuracy and efficiency.
5. Implement an *adaptive* numerical integration scheme that refines the sample Riemann grid only when necessary.

6. Solve a system of linear equations using *partial pivoting* (to minimize roundoff) and the LU decomposition, to preserve storage and to minimize computation by being able to solve the system with multiple right-hand sides but only performing a single Gaussian elimination.
7. Use Monte Carlo methods to solve problems intractable by direct, analytical methods.