Engineering and Numerical Programming

This article first appeared in the November 2001 issue of *C/C++ Users Journal*. The article is reprinted with permission of *C/C++ Users Journal* Copyright © 2001.

MAPM, A Portable Arbitrary Precision Math Library in C

Michael C. Ring

Frustrated by the finiteness of fixed-size arithmetic? This math library gives you the precision you need.

Sometime ago I had to solve what is a fairly common problem in numerical and scientific programming. I needed to curve-fit a set of x,y data samples to a polynomial equation. This problem eventually led me to write my own arbitrary precision math library. Of course I knew at the time that other libraries existed, but they were lacking some features I considered important. This article describes some of the added features of MAPM, and some of the implementation details that readers probably don't think about every day; such as how to multiply two numbers together in less than $O(N^2)$ time. I hope that readers will find MAPM interesting and useful for their own applications. In particular, thanks to a C++ wrapper class contributed by Orion Sky Lawlor, I believe MAPM is very easy to use.

Why We Need Arbitrary Precision

The project that inspired MAPM had specific requirements related to the maximum error allowed at each data point. In theory, by increasing the order of the polynomial used in the curve fit, it is possible to minimize the error of the data samples. After viewing a plot of the data samples, it became obvious that a high-order polynomial curve fit would be required. I used a least-squares curve-fitting algorithm. For a 10th-order polynomial, the algorithm required a summation of the x data samples raised to the 20th power. In general, for an Nth order polynomial, least-squares will require computing data samples to the 2N power.

At first, everything was working as expected. As I increased the order of the curve fit, the error in the fit compared to the raw data became smaller. This was true until I modeled an 18th-order polynomial. At this point, the solution did not improve. Higher order curve fits

failed to yield improvement as well. I then tried the same input data on three different computers (with different operating systems and compilers) and generated three very different solutions. This is when I realized that a more fundamental problem was coming into play.

As you have probably already guessed, the accumulation of the round-off errors in the floating-point math was the culprit. I had reached the limits of Standard C's **double** data type, at least as implemented on the machines that were available to me. I realized I would need an arbitrary precision math package to complete my calculations. An arbitrary precision math package allows math to be performed to any desired level of precision.

For example, consider the two numbers **N1 = 98237307.398797975997** and **N2 = 87733164872.98273499749**. **N1** has 20 significant digits; **N2** has 22. If you assign these to C **double**s, each variable will maintain only 15-16 digits of precision [1]. If you then multiply these numbers, the result will be precise only to the 15-16 most significant digits.

The *true* multiplication result would contain 42 significant digits. An arbitrary precision math library will do the precise math and save the full precision of the multiplication. Addition and subtraction errors are also eliminated in an arbitrary precision math library. If you add **1.0E+100** to **1.0E-100** in normal C (or any other language), the small fractional number is lost. In arbitrary precision math, the full precision is maintained (~200 significant digits in this example).

After converting the curve-fitting algorithm to use arbitrary precision math, the three different computers all computed byte-for-byte identical results. This held true to well beyond 30th-order polynomials. In case you are curious, my accuracy requirements were satisfied with a 24th-order polynomial.

MAPM Feature Set

When I searched for an arbitrary precision math library to use, I noticed some common traits among the available libraries. First, most of them seemed to have a preference for integer-only math. And second, none of the arbitrary precision C libraries could perform the math functions typically found in **math.h**, such as **sqrt**, **cos**, **log**, etc.

It was at this point that I decided to write my own library. The basic requirements for my library were that it provide natural support for floating-point numbers and that it perform the most common functions found in **math.h**.

MAPM will perform the following functions to any desired precision level: **Sqrt**, **Cbrt** (cube root), **Sin**, **Cos**, **Tan**, **Arc-sin**, **Arc-cos**, **Arc-tan**, **Arc-tan2**, **Log**, **Log10**, **Exp**, **Pow**, **Sinh**, **Cosh**, **Tanh**, **Arc-sinh**, **Arc-cosh**, **Arc-tanh**, and also Factorial. The full **math.h** is not duplicated, though I think the functions listed above are most of the important ones. (My definition of what's important is what I've actually used in a real application.) MAPM also has a random number generator with of period of **1.0E+15**.

MAPM has proven to be very portable. It has been compiled and tested under x86 Linux, HP-UX, and Sun Solaris using the GCC compiler. It has also been compiled and tested

under DOS/Windows NT/Windows 9x using GCC for DOS as well as (old) 16- and 32-bit compilers from Borland and Microsoft. Makefiles are included in the online source archive (available at <www.cuj.com/code>) for the most common compilers under various operating systems.

The MAPM C++ Wrapper Class

Orion Sky Lawlor (**olawlor@acm.org**) has added a very nice C++ wrapper class to MAPM.

Using the C++ wrapper allows you to do things such as the following:

```
// Compute the factorial of the integer n

MAPM factorial(MAPM n)
{
   MAPM i;
   MAPM product = 1;

   for (i=2; i <= n; i++)
      product *= i;

   return product;
}
```

The syntax is the same as if you were just writing normal code, but all the computations will be performed with the high precision math library, using the new data type **MAPM**.

See Listing 1 for another C++ sample. Note the use of literal character strings as constants. This allows the user to specify constants that cannot be represented by a standard C data type, such as a number with 200 digits or a number with a very large or small exponent (e.g., **6.21E-3714**).

Algorithms for Implementing MAPM

Since the MAPM contains over 30 functions, it is not practical to discuss all the algorithms used in the library. I will, however, discuss the more interesting ones.

Multiplication

In an arbitrary precision math library, the multiplication function is the most critical. Multiplication is normally an **O(N^2)** operation. When you learned to multiply in grade school, you multiplied each digit of each number and did the final addition after all the multiplies were completed. To visualize this, multiply by hand a four-digit number by a four-digit number. The intermediate math requires 16 multiplies (4^2). This method works and is very easy to implement; however it becomes prohibitively slow when the number of digits becomes large. So a 10,000-digit multiply will require 100 million individual

multiplications!

Faster Multiplication

The next multiplication algorithm discussed is commonly referred to as the divide-and-conquer algorithm.

Assume we have two numbers (**a** and **b**) each containing **2N** digits:

$$\text{let: } a = (2^N)*A_1 + A_0,\ b = (2^N)*B_1 + B_0$$

where $\mathbf{A_1}$ is the "most significant half" of **a** and $\mathbf{A_0}$ is the "least significant half" of **a**. The same applies for $\mathbf{B_1}$ and $\mathbf{B_0}$.

Now use the identity:

$$ab = (2^{2N} + 2^N)A_1B_1 + 2^N(A_1-A_0)(B_0-B_1) + (2^N + 1)A_0B_0$$

The original problem of multiplying two **2N**-digit numbers has been reduced to three multiplications of **N**-digit numbers plus some additions, subtractions, and shifts.

This multiplication algorithm can be used in a recursive process. The divide-and-conquer method results in approximately $\mathbf{O(N^{1.585})}$ growth in computing time as **N** increases. So a 10,000-digit multiply will result in only approximately 2.188 million multiplies. This represents a considerable improvement over the generic $\mathbf{O(N^2)}$ method.

Really Fast Multiplication

The final multiplication algorithm discussed utilizes the FFT (Fast Fourier Transform). An FFT multiplication algorithm grows only at $\mathbf{O(N*Log_2(N))}$. This growth is far less than the normal $\mathbf{N^2}$ or the divide-and-conquer's $\mathbf{N^{1.585}}$. An FFT tutorial is beyond the scope of this article, but the basic methodology can be discussed.

First, perform a forward Fourier transform on both numbers, where the digits of each number are regarded as the samples being input to the FFT. This yields two sets of coefficients in the "frequency" domain. Each set of coefficients will contain as many samples as the number of digits in the original number. (Also, each coefficient will be a complex number.) Second, multiply the two sets of coefficients together. That is, if the numbers being multiplied are **a** and **b**, perform $\mathbf{c_{a0}*c_{b0}}$, $\mathbf{c_{a1}*c_{b1}}$, etc., where $\mathbf{c_{an}}$, $\mathbf{c_{bn}}$ are the frequency-domain coefficients of **a** and **b**. This pair-wise multiplication in the frequency domain is equivalent to convolution in the "time" or sample, domain. This yields the correct result, because when you multiply two **N**-digit numbers together, you are really performing a form of convolution across their digits. Third, compute the inverse transform on the product sequence. Lastly, convert the real part of the inverse FFT to integers and also release all your carries in the process.

The FFT used in MAPM is from Takuya Ooura. This FFT is fast, portable, and freely

distributable.

To really see the differences in these three multiplication algorithms, compare these run times for multiplying two 1,000,000-digit numbers:

```
FFT Method          : 40 seconds
Divide-and-Conquer  : 1 hr, 50 min
Ordinary N2         : 23.9 days*

*projected!
```

The MAPM library uses all three multiplication algorithms. For small input numbers, the normal **O(N^2)** algorithm is used. (After all, you don't need a special algorithm to multiply 43 x 87.) Next, the FFT algorithm is used. FFT-based algorithms do reach a point where the floating-point math will overflow, so at this point the divide-and-conquer algorithm is used. Once the divide-and-conquer algorithm has divided down to the point where the FFT can handle it, the FFT will finish up.

Division

Two division functions are used.

For dividing numbers less than 250 digits long, I used Knuth's division algorithm from *The Art of Computer Programming, Volume 2* [2] with a slight modification. I determine right in step D3 whether **q-hat** is too big, so step D6 is unnecessary. I use the first three (base 100) digits of the numerator and the first two digits of the denominator to determine the trial divisor, instead of Knuth's use of two and one digits, respectively.

For dividing numbers longer than 250 digits, I find **a/b** by first computing the reciprocal of **b** and then multiplying by **a**. The reciprocal **y = 1/x** is computed by using an iterative method such that:

$$y_{n+1} = y_n(2 - xy_n)$$

where $\mathbf{y_n}$ is the current best estimate for **1/x**. This calculation is performed repeatedly until $\mathbf{y_{n+1}}$ stops changing to within a predetermined tolerance.

Exponential, Sine, and Cosine

These three functions are all computed by using a series expansion. Translations of the input are required to efficiently calculate these quantities. For more detail, see the accompanying sidebar, "Practical Series Expansions for Sine, Cosine, and Exponentials"

Random Number Generator

The random number generator is also taken from Knuth [3]. Assuming the random number is **X**, compute (using all integer math):

```
X = (a*X + c) MOD m
```

where **x MOD y** represents **x modulo y**. From Knuth, **m** should be large, at least 2^{30}. MAPM uses **1.0E+15**. The **a** coefficient should be between **0.01*m** and **0.99*m** and not have a simple pattern of digits. The **a** coefficient should not have any large factors in common with **m** and (since **m** is a power of 10 in this case) if **a MOD 200 = 21**, then all **m** different possible values will be generated before **X** starts to repeat. MAPM uses **a = 716805947629621**, so **a MOD 200** does equal 21, and **a** is also a prime number. There are few restrictions on **c**, except that **c** can have no factor in common with **m**, hence I have set **c = a**. On the first call, the system time is used to initialize **X**.

Newton's Method (Also Known as Newton-Raphson)

Unlike sine and cosine, which have convenient series expansions, not all math functions can be computed with a closed-form equation. Instead, they must be computed iteratively. These functions in MAPM (reciprocal, square root, cube root, logarithm, arc sine, and arc cosine) all use Newton's method to calculate the solution. (Although there are series expansions for logarithm, arc sine, and arc cosine, they all converge very slowly, so they are not practical for calculating these functions.) In general, the use of Newton's method in this application depends on the existence of an inverse for the function being computed [4]. An initial guess is made for **f(x)**, where **f** is the function being computed and **x** is the input value. Call this initial guess **y**. This initial guess is then plugged into the equation:

$$y_{n+1} = y_n - (g(y_n) - x)/g'(y_n)$$

where:

```
x is the input value
y_n = current "guess" for the solution to
      f(x)
g(y_n) = inverse function of f(x)
         evaluated at y_n
g'(y_n) = first derivative of the inverse
          of f(x) evaluated at y_n
y_n+1 = refined estimate of f(x)
```

This calculation is iterated until $\mathbf{y_{n+1}}$ stops changing to within a certain tolerance.

Newton's method is quadratically convergent. This results in doubling the number of significant digits that must be maintained during each iteration. MAPM uses the corresponding Standard C run-time library function to provide the initial guess.

As an example, I will demonstrate how to implement the square root function using Newton's method. Newton's method essentially asks: if the current guess for **sqrt(x)** is **y**,

how close does squaring **y** come to producing **x**?

The equation for the inverse function **g(y)** is derived as follows:

```
f(x) = y = sqrt(x)
g(y) = y^2 = x, the inverse function of
             f(x)
g'(y) = 2y, the derivative of the
        inverse function
```

Set up the iteration:

$$y_{n+1} = y_n - (g(y_n) - x)/g'(y_n)$$
$$y_{n+1} = y_n - (y_n^2 - x)/2y_n$$

and after a little algebra:

$$y_{n+1} = 0.5*[y_n + x/y_n]$$

To calculate the square root of 8.3, suppose I start with an initial guess of 1.0. (In actual use, I would generate a better initial guess.)

$$y_{n+1} = 0.5*[y_n + 8.3/y_n]$$

The iterations of $\mathbf{y_n}$ are as follows:

iteration	y
0	1.00000000000000000000
1	4.65000000000000000000
2	3.21747311827956989247
3	2.89856862531072654536764
4	2.88102547060539111310782
5	2.88097205867270336382209
6	2.88097205817758669914416
7	2.88097205817758669910162
8	2.88097205817758669910162

As you can see, the iteration is converging to a constant value, the square root of 8.3. Similar iteration loops are used to compute the reciprocal, cube-root, logarithm, arc sine, and arc cosine functions. The library actually uses a better **sqrt** iteration:

$$y_{n+1} = 0.5*y_n*[3 - x*y_n^2]$$

This iteration actually finds **1/sqrt(x)**. It is preferable since there is no division operation.

Division is slower than multiplication and is best avoided when possible.

Summary

MAPM is a portable arbitrary math library that is easy to use, especially when used with the C++ wrapper class. It supports most of the math functions encountered in a typical application, so it can be used in many applications that require arbitrary precision. MAPM uses an FFT-based algorithm for multiplication, which is typically the performance bottleneck for arbitrary precision math libraries.

Notes and References

[1] This assumes that **double**s are implemented as 64-bit words, which is the case for most general purpose computers available today.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms, Third Edition* (Addison-Wesley, 1998), pages 270-273.

[3] ibid., pages 184-185.

[4] The reciprocal function is its own inverse, which might seem to pose a problem in using Newton's method. Fortunately, it can be factored out so that it does not actually appear in the algorithm.

Further Reading

Bjarne Stroustrup. *The C++ Programming Language, Third Edition* (Addison-Wesley, 1997).

Takuya Ooura. "A General Purpose FFT (Fast Fourier/Cosine/Sine Transform) Package", 1996-1999.

D. H. Bailey. "Multiprecision Translation and Execution of Fortran Programs", *ACM Transactions on Mathematical Software*, September 1993, p. 288-319.

Press, Teukolsky, Vetterling, and Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition* (Cambridge University Press, 1988-1992).

William H. Beyer. *CRC Standard Mathematical Tables, 25th Edition* (CRC Press, 1978).

Johnson and Riess. *Numerical Analysis* (Addison-Wesley, 1977).

Rule, Finkenaur, and Patrick. *FORTRAN IV Programming* (Prindle, Weber, and Schmidt, 1973).

Michael C. Ring *has a B.S. in Electrical Engineering from the University of Minnesota. He is currently a Principal Systems Engineer at Honeywell working system design and test on inertial navigation systems. He can be reached at* ***mike.ring@honeywell.com****.*

Listing 1: Using the C++ wrapper class with MAPM

```
//
//    Example MAPM C++ code with output.
//
//    Note the use of literal character strings as constants.
//
//    This allows the user to specify constants that cannot be
//    represented by a standard C datatype, such as a number with
//    200 digits or a number with a very large or small exponent
//    (i.e., 6.21E-3714).
//

char  outbuf[256];
MAPM  u, v, w, x, y, z;         // arbitrary precision datatype

m_apm_cpp_precision(50);        // set MINIMUM precision level
                                // for all calculations

x = 9.34231;
y = -21;
z = "-8.98234924982982492147982492479234792 1E-17";

w = (82.30421 + sin(x / "21.11")
              / exp(y * 0.0726426)) * "4.32917E-2" / z;
v = "3.742416" * log(-w);
u = sqrt(v) + cbrt(v);

x.toString(outbuf, 50);   printf("x = [%s] \n",outbuf);
y.toString(outbuf, 50);   printf("y = [%s] \n",outbuf);
z.toString(outbuf, 50);   printf("z = [%s] \n",outbuf);
w.toString(outbuf, 50);   printf("w = [%s] \n",outbuf);
v.toString(outbuf, 50);   printf("v = [%s] \n",outbuf);
u.toString(outbuf, 50);   printf("u = [%s] \n",outbuf);

//
//    end sample code
//


/*
Output from above code :

x = [9.3423100000000000000000000000000000000000000000000E+0]
```

```
y = [-2.10000000000000000000000000000000000000000000000000E+1]
z = [-8.98234924982982492147982492479234792100000000000000E-17]
w = [-4.06165846135776503301770738763183914451375637893923E+16]
v = [1.43121038693447060414449698088102300924570797933606E+2]
u = [1.71941170788317776083850581271655444461072359345223E+1]
*/

End of Listing
```

Practical Use of Series Expansions

The series expansion for the exponential, sine, and cosine functions are as follows:

$$exp(x) = 1 + x + x^2/2! + x^3/3! + x^4/4! + ...$$

$$sin(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9!$$

$$cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8!$$

Now assume you want to calculate **exp(543.7)**. (If you are curious, the answer is **1.33631797683075214970870991011 4E+236**.) You could put 543.7 into the above formula and evaluate the series until the final term reached the precision you desire. However, this method will take numerous iterations to converge. Since **|x| > 1**, the numerator will continue to grow larger, and the series won't converge until the factorial in the denominator can overtake the numerator. Even though this formula is numerically correct, it is not a practical method to calculate the exponential. What is needed is a translation of **x** so the magnitude of **x** is less than one. (Hopefully, **x** will be significantly smaller than one.) The smaller the magnitude of **x**, the faster the above series will converge to the desired precision. The translation of **x** is provided by David H. Bailey's MPFUN software package. This is a multiple precision math library written in Fortran. Dr. Bailey's algorithm uses a modification of the series expansion:

$$exp(t) = (1 + r + r^2/2! + r^3/3! + ...)^{q*2^n}$$

where **q = 256**, **r = t'/q**, **t' = t - n*Log(2)**, and where **n** is chosen so that **-0.5*Log(2) < t' <= 0.5*Log(2)**. Reducing **t mod Log(2)** and dividing by 256 insures that **-0.001 < r <= 0.001**, which accelerates convergence in the above series.

After the series expansion, you must raise the result to the 256th power. This may seem at first a daunting task, but remember that you can simply square the result the required number of times.

$$x^4 = (x^2)^2$$
$$x^8 = ((x^2)^2)^2$$
$$x^{16} = (((x^2)^2)^2)^2)$$
$$x^{256} = ((((x^2)^2 \text{ ... 8 times}$$

Also note that this method requires the value of **log(2)**. The value of **log(2)** in the MAPM library is accurate to 128 decimal places. If you want an exponential calculation that is accurate to greater than 128 decimal places, the library will re-compute **log(2)** on the fly as necessary to be as precise as the exponential precision specified.

Now assume you want to calculate the **sin(10000.0)**, where **10000.0** is in radians. This is certainly a valid argument to the **sin** function, though passing 10,000 to the series expansion will be quite inefficient due to the magnitude of the numerator being greater than one, as was previously discussed for the exponential function. You need a translation of **x** such that **|x| < 1**. In the MAPM library, two translations are performed first, and the series expansion is done in the third step. The first obvious translation is to limit the input angle to +/- pi radians. This is simply an **x MOD 2*pi** operation. Although this operation limits the input argument to +/- pi, pi is greater than one, so the series will still take too long to converge.

The second translation uses the multiple angle identity for **sin(5x)**. This identity is:

$$\sin(5x) = 16*\sin^5(x) - 20*\sin^3(x) + 5*\sin(x)$$

If you can calculate **sin(0.5)**, you could just use the above identity to compute **sin(2.5)**. If you desire **sin(3)**, calculate **sin(0.6)** and use the identity. By using this identity, the worst-case **x** input to the series expansion will be **pi/5** or 0.6283. This is less than one, so the series will converge significantly faster than before. However, you can do better. To calculate **sin(0.6283)**, you can use the multiple angle identity a *second* time to decrease the worst-case number to **0.6283/5** or 0.1257. You could (in theory) perform this operation numerous times, but you do reach the point of diminishing returns. The MAPM library uses this identity three times, so the worst-case input to the series expansion is **pi/(5*5*5)**, or 0.0251. This is small enough so that the sine series does converge quite rapidly to the desired precision.

I have not seen the multiple angle identity translation used in this context before, so I believe the MAPM library is the first arbitrary precision library to use this technique.

The cosine function also uses a multiple angle identity to minimize the value of the number passed to the series expansion. The following algorithm summarizes the process. It is essentially the same in concept to the algorithm used for computing **sin(x)**.

Step 1. Limit input argument to +/- pi.

Step 2. Use the multiple angle identity for **cos(4x)**:

$$\cos(4x) = 8*[\cos^4(x) - \cos^2(x)] + 1$$

Use this identity recursively three or four times as needed; that is, multiply the input angle by $\mathbf{1/(4^3)}$ or $\mathbf{1/(4^4)}$ respectively. If **|x|** is less than one radian, recurse three times. If **|x| >= 1** radian, recurse four times.

This step yields a worst-case **|x| = ~0.0156 (1/64 > pi/256)**.

Step 3. Apply traditional series expansion.