Using Multiple-Precision Arithmetic

David M. Smith

Loyola Marymount University

There are many different kinds of problems for which high precision calculation can be useful. The most common situation involves a computation that is numerically unstable, so that using double precision is not sufficient to get the final result as accurately as it is needed.

We will use the term "significant digits" to mean the number of equivalent decimal digits of precision, as opposed to the actual number of digits carried when the base is not ten. For most current machines, double precision is the highest accuracy provided in hardware, and this is 53 digits in base 2, or around 16 significant digits. Being able to do a calculation with 30 or 40 significant digits will often overcome the instability and give adequate accuracy for the final result.

Even though the term "multiple-precision" may first bring to mind applications such as the computation of billions of digits of π, most actual applications use precisions of a few tens of digits, rather than millions or even hundreds.

As an example of such a calculation, let us look at the computation of the Bessel function $J_1(x)$ for $|x|$ up to a few hundred. If we actually needed to compute $J_1(x)$, there are existing libraries that could be used, but we might have a similar formula that is not one of these standard functions.

For small values of x, there is a convergent series,

$$J_1(x) = \sum_{k=0}^{\infty} \frac{(-1)^k \, x^{2k+1}}{2^{2k+1} \, k! \, (k+1)!}.$$

This formula is easy to understand and easy to compute, but it becomes more unstable as $|x|$ gets larger. There are often other algorithms available. For $J_1(x)$ there is an asymptotic series that is stable for large x, and a backward recurrence that might bridge the gap between the convergent and asymptotic series.

If we were developing a library routine for $J_1(x)$, we would need to include many of these different formulas and select the best one depending upon x. But if we merely need to compute a few (or a few thousand) values for small and moderate sized x, it may be more cost-effective to code just the simple formula and use multiple-precision to control the instability. And even the developers of double precision library routines often use multiple-precision to check their results.

Consider this calculation when $x = 35.3$. Program 1 uses Fortran and double precision on a 32-bit computer to sum the series. The term $x^{2k+1}/2^{2k+1}$ is done in the loop as $(x/2)^{2k+1}$, but the other power functions are left in the loop, to keep some resemblance of the code to the original formula. If we wanted to tune the code for more speed, the remaining power functions could be replaced with a single multiplication by $-(x/2)^2$ each time through the loop. Printing out the partial sums of the series every few terms shows the instability.

Program 1

```
PROGRAM BESSEL
DOUBLE PRECISION ::  X,S,FACT
INTEGER ::  K
S = 0
X = 35.3D0/2
FACT = 1
DO K = 0, 70
   S = S + (-1)**K * X**(2*K+1) / ( FACT * FACT * (K+1) )
   FACT = FACT * (K+1)
   IF (MOD(K,5) == 0) THEN
       WRITE (*,"(A,I3,A,E25.15)") " K = ",K,"    S = ",S
   ENDIF
ENDDO
END PROGRAM BESSEL
```

Output from this program:

```
 K =   0    S =     0.176500000000000E+02
 K =   5    S =    -0.545194332007876E+09
 K =  10    S =     0.764675659761547E+12
 K =  15    S =    -0.896891165457883E+13
 K =  20    S =     0.433357476151880E+13
 K =  25    S =    -0.191563479309724E+12
 K =  30    S =     0.124759447806175E+10
 K =  35    S =    -0.164218062418216E+07
 K =  40    S =     0.547220520808295E+03
 K =  45    S =    -0.481654815068169E-01
 K =  50    S =     0.648516866704781E-02
 K =  55    S =     0.648330340078305E-02
 K =  60    S =     0.648330342495567E-02
 K =  65    S =     0.648330342495554E-02
 K =  70    S =     0.648330342495554E-02
```

Since the final result of the sum is about 15 orders of magnitude less than some of the partial sums, we don't have much confidence in any of the digits of that result.

Next, let us translate this program so it will use the FM multiple-precision package (FM stands for Floating-point Multiple-precision). The FM package is written in Fortran. The code and a set of sample programs can be found at `http://myweb.lmu.edu/dmsmith/FMLIB.html`, along with papers describing the organization of the package and the algorithms used.

Before Fortran-90, translating an existing program to use a multiple-precision package was fairly arduous. The multiple-precision operations consist of basic arithmetic, type conversions, mathematical functions, and so on. Each one is performed by a separate subroutine, so translating required converting the Fortran commands into calls to the appropriate subroutines.

A short Fortran statement like "`C = A + B`" might be translated to "`CALL FM_ADD(A,B,C)`". Longer statements like the formula in Program 1 turn into many lines of code, since each call performs only a single arithmetic operation or function.

Beginning with Fortran-90 in the early 1990s, derived types and operator overloading became part of the language. Now it is possible to write interface modules that tell the compiler how to automatically convert Fortran arithmetic expressions into calls to the various multiple-precision routines. This makes the conversion process much easier, since most of the original code can remain unchanged.

These are the changes made to Program 1 to get Program 2, the FM version.

1. Put "`USE FMZM`" at the top of each routine. This tells the compiler where to find all the rules for replacing arithmetic operations and intrinsic function calls (like $**$) by FM calls.

2. Change the variable declarations. "`DOUBLE PRECISION`" becomes "`TYPE (FM)`".

   Add a new character variable, `ST1`, used to format FM numbers for output. The integer declarations do not need to be changed, since we don't need multiple-precision integers here.

3. Set the FM precision level by calling `FM_SET` with the number of significant digits we want. In this case, we prompt the user to enter this value.

4. Change double precision constants to FM constants. `35.3D0` becomes `TO_FM("35.3")`.

5. Change the `WRITE` statement to handle FM numbers. Routine `FM_FORM` takes a character string defining the format and an FM number, and returns a character string as the formatted result to be printed by the `WRITE` statement.

Program 2

```
      PROGRAM BESSEL
      USE FMZM
      TYPE (FM) ::  X,S,FACT
      INTEGER ::  K,SD
      CHARACTER(80) ::  ST1
      WRITE (*,*) " Enter the number of significant digits to use."
      READ (*,*) SD
      CALL FM_SET(SD)
      S = 0
      X = TO_FM("35.3")/2
      FACT = 1
      DO K = 0, 70
         S = S + (-1)**K * X**(2*K+1) / ( FACT * FACT * (K+1) )
         FACT = FACT * (K+1)
         IF (MOD(K,5) == 0) THEN
             CALL FM_FORM("E25.15",S,ST1)
             WRITE (*,"(I3,A,I3,A,A)") SD," significant digits:      K = ",  &
                                       K,"    S = ",TRIM(ST1)
         ENDIF
      ENDDO
      END PROGRAM BESSEL
```

Final line of output from this program when run with different precision levels:

```
    15 significant digits:      K =  70    S =  .433726793549236M-2
    20 significant digits:      K =  70    S =  .433726598367081M-2
    25 significant digits:      K =  70    S =  .433726598367081M-2
    30 significant digits:      K =  70    S =  .433726598367077M-2
    40 significant digits:      K =  70    S =  .433726598367077M-2
```

Comparing the output from Program 1 to these higher-precision results shows that none of the digits computed in double precision are correct. When we ask for 15 significant digits from FM, six digits are correct at the end of the sum.

When FM uses `E` format for output, as with the variable `S` above, `"M"` is used in place of `"E"` to signify the output of an FM number. If this output needs to be read as input to another program not using multiple-precision, we can have FM use `"E"` or `"D"` instead.

Understanding some of the internal details of how FM does arithmetic can help us interpret these results. For better speed, the default base for arithmetic is large (10^7 on this machine). So computing one more digit in base 10^7 gives seven more significant digits of precision.

For the number π, normalization in base 10^7 makes 3 the first digit, 1415926 the second, and so on. This means carrying four digits in base 10^7 provides about $1 + 3 * 7 = 22$ significant digits. Increasing precision to five digits corresponds to about 29 significant digits.

When we ask for 15 significant digit precision in the first FM run, the program assumes we need a few guard digits and sets the number of base 10^7 digits at the next higher value, four. This means we really have around 22 significant digits during that calculation. After losing 16 of them to cancellation error in the sum, we are left with about six correct significant digits.

When converting constants for the FM version, if we leave the line as `X = 35.3D0/2`, instead of using the conversion function `TO_FM("35.3")`, then `X` is correct only to double precision accuracy. As expressed in base 2, the number 35.3 has an infinite repeating binary expansion (100011.010011001100110011...). Even before we use the constant `35.3D0`, the act of converting it from the base 10 form in our program to the base 2 double precision internal form creates an error in about the 16th significant digit. The "`X =`" part of the statement later causes the double precision value to be converted to FM precision, but the accuracy has already been lost.

For the same reason, `TO_FM(35.3D0)` is also accurate only to double precision. Sending the number to FM in character form insures that it is converted directly to an FM number with no intermediate rounding. For cases where all floating-point constants in a large program might be converted automatically, writing `TO_FM("35.3D0")` is allowed, and will give full FM precision.

There are actually several different versions of conversion functions like `TO_FM`, depending on whether the argument is integer, single precision, double precision, or character. The user interface modules invoked by the `USE FMZM` statement determine which individual conversion routine to use, based on the type of the input argument.

It is not necessary to replace the 2 by `TO_FM("2")` in this expression. Integers have no rounding errors like 35.3, and the 2 will convert exactly. In fact, it is better to leave integer terms alone when multiplying or dividing an FM number by an integer. There are separate subroutines for those operations, and the special cases where one argument is an integer can be done more quickly.

An interesting feature in the Program 2 output is that we get the same result when we ask for either 20 or 25 digits. This is because FM uses five base 10^7 digits in each case. In practice, when we want to compare two FM runs at different precision levels, we need to increase the requested number of base 10 digits by enough to force FM to choose a greater precision level.

If we compare just the results from the runs with 20, 30, and 40 digits, we know that they are done at three different FM precision levels. The agreement in the last two indicates that we have 15 digit accuracy for $J_1(35.3)$.

As the example above shows, someone using software for high precision needs to know some of the details about how the arithmetic works. Next we will examine a few more of the internal workings of the FM package. For more information, consult references [1–4]. These and other related papers can also be found at `http://myweb.lmu.edu/dmsmith/FMLIB.html`.

An FM number is stored as a list of integers in an array. This defines a floating-point number in base b having n digits of precision. The user can set the values of b and n. If the elements of this list are denoted x_i, then x_1 contains the exponent, x_2 contains the first digit, and so on until x_{n+1} contains the last digit. The position of the radix point is assumed to be before x_2, so the number is

$$\left(\frac{x_2}{b} + \frac{x_3}{b^2} + \frac{x_4}{b^3} + \cdots + \frac{x_{n+1}}{b^n}\right) \times b^{x_1}.$$

The digits are said to be normalized if $1 \le x_2 < b$, and $0 \le x_i < b$ for $i > 2$. The sign of the number is carried on another list element, which we ignore for these examples. Thus, if $b = 10$ and $n = 8$, the number π is represented as $\{1, 3, 1, 4, 1, 5, 9, 2, 7\}$.

The algorithm for adding two of these numbers is fairly straightforward. Check the exponents to see if one of the numbers needs to be shifted so that the decimal points are correctly aligned, then add corresponding digits to form the sum. This takes about n steps as we loop over the individual digits in the two lists. Then there is another loop with n more steps required to normalize the digits in the sum, since they might be as large as $2b - 2$.

This means the total number of operations for adding two n-digit numbers is proportional to n, although there are several special cases that may add to this total. For example, if there is a carry in the first digit before we normalize the digits (i.e., $x_2 \ge b$), then the other digits must essentially be shifted right to make room for the new first digit and the exponent must be adjusted.

Actually, more than n digits of the sum are computed so that there will be some guard digits to allow the sum to be correctly rounded back to n digits when returning the result. Even so, the total effort to add the two numbers is no more than some small constant times n, so we say that the work for addition is of order n, written $O(n)$.

Next, consider the algorithm for multiplication. The classical method is given in Knuth [5], and multiplies each digit of the first number by each digit of the second, for a total of n^2 basic operations. Then these terms are added in the right order to produce the $2n$ digits of the product. As with addition, the digits must be normalized, rounding done, and special cases handled, but the work for multiplication is $O(n^2)$.

Knuth also shows that some algorithms can multiply two n-digit numbers in $O(n \log(n))$ steps, but these methods are so complicated that they are not faster than the classical method unless n is very large. If the classical method takes $3n^2$ time units and the “fast” method takes $150n \log(n)$

time units, then the second method will be better for large n, but the first will be better for $n = 10$.

This analysis of efficiency explains why it is better to choose a large base b for the arithmetic. Suppose we want to compute xy with an accuracy of 30 significant digits. In base ten we have $n = 30$, so the multiplication will take a few times n^2 steps, which means several thousand steps. But using $b = 10^7$ means seven significant digits are stored in each x_i, except that the first digit, x_2, may have only one significant digit because of normalization. So $n = 6$ gives at least 30-digit accuracy with this large base, and now the multiplication takes a few times 6^2 steps. Doing the multiplication with the large base makes it over ten times faster.

One other consideration in choosing the base involves converting the numbers for input or output. With standard 32-bit hardware, FM uses double precision arrays for the lists of integer digits. In order to guarantee that these integers are represented exactly (no rounding) in double precision, they must be less than $2^{53} = 9 \times 10^{15}$. The product of two digits in base b could be as large as $(b-1)^2$, so in FM the base can be any integer from 2 to $\sqrt{9 \times 10^{15}} = 9.5 \times 10^7$.

Input or output operations require converting the FM numbers between base 10 and base b. In general, this is as hard as multiplication and takes $O(n^2)$ steps. However, for the special case where b is a power of ten this conversion can be done much more quickly, in $O(n)$ steps. For example, compare the work to convert $\pi = \{$ 1, 3, 1415926, 5358979, 3238463 $\}$ from base 10^7 to base ten for output, with the work to convert $\pi = \{$ 1, 3, 13438029, 85761677, 92784528 $\}$ from base $94,906,265$ to base ten. There is little loss of efficiency for the other arithmetic operations when the base is reduced from 9.5×10^7 to 10^7, and the faster input/output conversion more than makes up for it.

Getting back to some problems where multiple-precision can be useful, another case is one where the double precision version of a program fails because of overflow. For example, what is the probability of getting exactly 10,000 heads in 20,000 tosses of a fair coin?

The standard formula for the answer involves binomial coefficients,

$$p = \binom{20,000}{10,000} (0.5)^{20,000} = .00564182531222042$$

The difficulty here is not that we have lost accuracy, but that the two terms in the product are about `2.2E+6018` and `2.5E-6021`. For most current machines, the largest number that can be represented in double precision is `1.8E+308`, so the first term overflows and the second underflows.

As usual, there are ways an expert can overcome this overflow/underflow problem in double precision. But since the FM overflow threshold is more than `1.0E+400000000`, and there is a built-in binomial coefficient routine, we can do it in one line:

```
P = BINOMIAL( TO_FM(20000) , TO_FM(10000) ) * TO_FM("0.5") ** 20000
```

Sometimes FM is used to simulate the arithmetic of a particular computer. Suppose we have developed an algorithm and we have tested it on a computer using 53-bit base 2 rounded arithmetic (standard IEEE double precision). If a colleague needs to run our program on a different computer with 14-digit base 16 chopped arithmetic (such as an IBM mainframe computer), we may want to run some test cases on our machine to simulate the other one.

To run FM with a small base sacrifices some speed, but for cases like this one we can set the number of digits, base, overflow/underflow threshold, and rounding mode. There is an automatic tracing feature in FM that can also be useful in this type of application, if we want to see a log where the input and result of each operation is listed.

For exception handling FM uses five special symbols, called ±`OVERFLOW`, ±`UNDERFLOW`, and `UNKNOWN`. Functions and arithmetic operations are defined to handle these symbols and to return non-exceptional values whenever possible.

For example, $2 +$ `UNDERFLOW` returns 2, `EXP(-UNDERFLOW)` returns 1, 3 / `UNDERFLOW` returns `OVERFLOW`, and `ATAN(-OVERFLOW)` returns (approximately) $-\pi/2$.

Cases where we don't know the exception category or representable FM number of the result are returned as `UNKNOWN`, as with 1/0, `SQRT(-2)`, and `LOG(OVERFLOW)`. `OVERFLOW`/1.01 returns `UNKNOWN`, since the true result might be a representable number or `OVERFLOW`. But `OVERFLOW`/0.99 returns `OVERFLOW`.

Having the `UNDERFLOW` category means FM results never become zero as a result of underflow. Here is part of a program comparing FM and IEEE single precision when some intermediate results are near the underflow threshold. FM has been set to use 24 digits base 2, with the maximum FM exponent set to give underflow below 2^{-150}, to agree with the IEEE underflow threshold.

Program 3

```
DO J = 60, 90
   X   = 1.5
   YFM = 1.5
   DO K = 1, J
      X1 = 3.14159 + (J/900.0 + K/1000.0)
      X   =   X/X1
      YFM = YFM/X1
   ENDDO
   DO K = 1, J
      X1 = 3.14159 + (J/900.0 + K/1000.0)
      X   =   X*X1
      YFM = YFM*X1
   ENDDO
```

```
      CALL FM_FORM("F10.7",YFM,ST1)
      WRITE (*,"(A,I2,16X,F10.7,4X,A)") " J = ",J,X,TRIM(ST1)
   ENDDO
```

In this program X and X1 are single precision and YFM is TYPE (FM). We start with 1.5 and divide by J different values, then re-trace our steps by multiplying by those same values. Except for rounding and underflow effects, we expect to get back to 1.5 at the end. Here is some of the output:

```
J = 60                  1.4999998     1.4999998
J = 61                  1.5000001     1.5000001
...
J = 85                  1.5444297     1.5000004
J = 86                  1.6308042     1.4999999
J = 87                  2.0562468     1.4999998
J = 88                  0.0000000       UNKNOWN
J = 89                  0.0000000       UNKNOWN
J = 90                  0.0000000       UNKNOWN
```

Early on, both single precision and FM give good approximations to 1.5, but as J increases some intermediate results come closer to the underflow threshold. In the hardware arithmetic, accuracy degrades because of gradual underflow. After J = 87, the calculation has encountered complete underflow and the result is flushed to zero.

In the last column, the FM results show good accuracy through J = 87. After that, there is underflow, but the result is not flushed to zero. Then when we try to multiply that value by numbers greater than one, FM gives UNKNOWN.

For this example, the FM exception handling strategy works well. The FM numbers printed are accurate, and when we have lost track of the calculation due to underflow, the UNKNOWN result tells us so, instead of giving a potentially dangerous value of zero.

In addition to multiple-precision real numbers, FM provides multiple-precision integers and complex numbers. These are declared as TYPE (IM) and TYPE (ZM) respectively.

High precision integers are used in cryptography and number theory applications. These often require large prime numbers. Consider two 70-digit integers.

$a = 5468317884572019103692012212053793153845065543480825746529998049913559$,

$b = a + 2$.

Neither of these numbers has any prime factors less than 10 million, so they might be prime.

Fermat's theorem says $x^{p-1} \bmod p = 1$ when p is prime and x is not a multiple of p. If we find that $x^{p-1} \bmod p$ gives 1 for some p with several different x's, then it is very likely that p is prime (but it is not certain until further tests are done).

FM has function `POWER_MOD(A,B,C)` for type IM integers that computes `A ** B mod C`. We can quickly find

$2^{a-1} \bmod a = 398331661014267117637834640027001742827067752281578629275624495229 6938$

so a is definitely not prime. But several calculations like

$2^{b-1} \bmod b = 1,$

$3^{b-1} \bmod b = 1,$

$314159^{b-1} \bmod b = 1,$

make it quite likely that b is prime.

For an example using complex arithmetic, let us return to the Bessel function and compute $J_1(35.3 + 2.8\,i)$. These are the changes made to Program 2 to get a program that uses complex FM arithmetic.

1. Change "`TYPE (FM)`" to "`TYPE (ZM)`".
2. Change the initialization of `X` to `X = TO_ZM(" 35.3 + 2.8 i ")/2`
3. Change `CALL FM_FORM("E25.15",S,ST1)` to `CALL ZM_FORM("E21.15","E21.15",S,ST1)`

As before, asking for 30 and then 40 digits gives 15 significant digit agreement in the results:

```
 30 significant digits:  K = 70  S = -.767913653991677M-2 - .109761037849060M+1 i
```

The speed is not too bad for high precision, although it is much slower than the arithmetic that is built into the hardware. On a typical current desktop computer, if the precision is no more than 40 significant digits, we get around a million add or subtract operations per second, with multiply and divide 2 or 3 times slower. Elementary functions run at 5,000 or 10,000 calls per second, and special functions like gamma or binomial coefficients are more like 1,000 calls per second. But each function has a set of special cases for which the speed is much greater. For example, binomial coefficients are much faster when both arguments are integers less than 100.

These examples show that with the compiler doing most of the work, translating programs to use multiple-precision is not hard. There is great flexibility in using the package — we can set the precision, base, rounding mode, and other parameters that control the arithmetic. There are lots of problems where doing some high precision computing can turn a tricky calculation into an easy one, and computer power is now great enough that it is feasible to use multiple-precision for many of these applications.

References

1. Smith, D.M. Efficient Multiple-Precision Evaluation of Elementary Functions. *Math. Comp. 52* (January, 1989) 131–134.

2. Smith, D.M. A Fortran Package for Floating-Point Multiple-Precision Arithmetic. *ACM Trans. Math. Softw. 17*, (June, 1991), 273–283.

3. Smith, D.M. Multiple Precision Complex Arithmetic and Functions. *ACM Trans. Math. Softw. 24*, (December, 1998), 359–367.

4. Smith, D.M. Multiple-Precision Gamma Function and Related Functions. *Transactions on Mathematical Software 27* (December, 2001), 377 – 387.

5. Knuth, D.E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, third edition. Addison Wesley, Reading, Mass., 1998.