A Multiple-Precision Division Algorithm

By David M. Smith

Abstract. The classical algorithm for multiple-precision division normalizes digits during each step and sometimes makes correction steps when the initial guess for the quotient digit turns out to be wrong. A method is presented that runs faster by skipping most of the intermediate normalization and recovers from wrong guesses without separate correction steps.

1. Introduction. Knuth [3] described classical algorithms for multiplication and division using t digits of precision with base b arithmetic. These methods require $O(t^2)$ operations. Knuth also surveyed several algorithms with faster asymptotic running time. Bailey's multiple-precision package [1] included routines based on FFT methods for use at very high precision, as well as $O(t^2)$ routines. The FFT methods are faster asymptotically, but because the algorithms are complicated, the classical $O(t^2)$ methods are more efficient when the precision is less than a few thousand digits.

For multiplication, Brent's package [2] improved on the classical methods by restricting b so that integer overflow could not occur for numbers less than $8\,b^2$. This allowed his routine to do eight steps of the multiplication before normalizing, resulting in significantly faster execution. The FM multiplication routine [7] was based on this idea and reduced the number of normalizations still further. Here "normalizing" refers to the act of restoring the individual digits of an intermediate result to the range $[0, b-1]$.

It is more difficult to skip normalization during division, because the quotient digit q used in each "multiply and subtract" step is a guess. Early routines ([3], [4], [5], [6], [8]) used integer arithmetic to estimate q. Bailey used double precision to estimate q, which made the probability of an incorrect q much smaller. Brent used Newton iteration and multiplication to avoid the long-division algorithm, but this made division much slower than multiplication.

The early routines normalized the partial result at each step so that cases where q was wrong could be recognized and corrected. This correction took the form of an extra "addback" step that was done so the value of q would be correct before the next multiply and subtract step was taken.

Since a large base was usually used, the correction step was needed only rarely. This means that eliminating the correction step makes the division routine only slightly faster. The time spent normalizing partial results was often greater than the time spent generating them, so minimizing

1985 *Mathematics Subject Classification.* Primary 65D15.

the normalization is important.

In version 1.0 of the FM package, the division routine normalized only a few leading digits of the partial result so that q could be verified or a correction step taken. Bailey's division routine eliminated the correction step, but normalized each partial result.

This paper presents a division algorithm that eliminates most of the intermediate normalization and does no separate correction steps. This algorithm still requires $O(t^2)$ steps, but is often two or three times faster than previous $O(t^2)$ methods ([1], [2], [7]).

2. Examples. The method is similar enough to the standard long-division algorithm that the differences can be seen using the example quotient π/e. A floating-point division will be done, although the method works in the same way for multiple-precision integer division.

Commonly used bases are in the range $[10^4, 10^8]$ for most machines. The examples use $b = 10^4$, $t = 5$, and compute six digits of the quotient to simulate the guard digits carried by division routines.

To begin, the numerator is copied into a work array, W, with zero digits padded to the end to leave room for guard digit calculations. The numbers are stored in a floating-point format where the first word is the exponent (as a power of b) followed by the digits of the fraction part.

n =	1	3	1415	9265	3589	7932
d =	1	2	7182	8182	8459	0452

As the division proceeds, the "active" part of the work array shifts to the right, and most routines store the quotient digits in the left part of the array. The first word of W is the exponent of the quotient.

W =	1	0	3	1415	9265	3589	7932	0	0	0

In this example, four words of each number are used to generate the floating-point approximations. Since they will be divided to get the quotient digits, both can be scaled by any convenient factor. The numerator value, x_n, will be scaled to be an integer, and the denominator value, x_d, is 271828182.8459. The approximation to each quotient digit is $q = \lfloor x_n/x_d \rfloor$.

Step 1. Use words 2 through 5 for x_n. $x_n = 314159265.0$, $q = \lfloor x_n/x_d \rfloor = 1$.

Multiply and subtract: $\mathrm{W} \leftarrow \mathrm{W} - q * d$

W is displayed on two lines to show the current quotient and the active part of the array.

W =	1	1						
	1	–5767	1083	–4870	7480	0	0	0

Step 2. Use words 3 through 6 for x_n. $x_n = 423310825130.0$, $q = \lfloor x_n/x_d \rfloor = 1557$.

Multiply and subtract: $\mathrm{W} \leftarrow \mathrm{W} - q * d$

W =	1	1					
	1	–8881	–11181291	–12744244	–13163183	–703764	0	0

Word 3 is multiplied by b and added to word 4, then q is stored in word 3.

W = 1 1 1557

1119 –11181291 –12744244 –13163183 –703764 0 0

Step 3. Use words 4 through 7 for x_n. $x_n = 743444396817.0$, $q = \lfloor x_n/x_d \rfloor = 2734$.

W = 1 1 1557 2734

3241 –32379832 –35532771 –23830670 –1235768 0

Step 4. Use words 5 through 8 for x_n. $x_n = 2661448459330.0$, $q = \lfloor x_n/x_d \rfloor = 9790$.

W = 1 1 1557 2734 9790

10588 –105844551 –103932450 –84049378 –4425080

Step 5. Use words 6 through 9 for x_n. $x_n = 2505491450622.0$, $q = \lfloor x_n/x_d \rfloor = 9217$.

W = 1 1 1557 2734 9790 9217

17015 –170128944 –159462872 –82391683

Step 6. Use words 7 through 10 for x_n. $x_n = 510888888317.0$, $q = \lfloor x_n/x_d \rfloor = 1879$.

W = 1 1 1557 2734 9790 9217 1879

17298 –172957850 –97765661

Now the result can be rounded to five digits and the value 1.1557273497909217 returned.

This example shows that the active part of W can be kept unnormalized. When x_n is computed using double precision arithmetic, four words of W are essentially normalized during the calculation.

No attempt is made to check that the quotient digit q is correct. The next example shows what would happen if q were one too high in step 3 above. The first two steps are the same as before.

Step 3. Use $q = 2735$.

W = 1 1 1557 2735

3239 –32387014 –35540953 –23839129 –1236220 0

Step 4. Use words 5 through 8 for x_n. $x_n = -56833369129.0$, $q = \lfloor x_n/x_d \rfloor = -210$.

W = 1 1 1557 2735 –210

3406 –34032733 –22120909 –540170 –94920

Step 5. Use words 6 through 9 for x_n. $x_n = 2505491450622.0$, $q = \lfloor x_n/x_d \rfloor = 9217$.

W = 1 1 1557 2735 –210 9217

8833 –88317403 –74873324 –77871683

Step 6. Use words 7 through 10 for x_n. $x_n = 510888888317.0$, $q = \lfloor x_n/x_d \rfloor = 1879$.

W = 1 1 1557 2735 –210 9217 1879

8839 –88368302 –93245661

This shows that an incorrect q leads to unnormalized quotient digits later, so a final pass over the result is needed to normalize before returning the result. This gives the same value as in the first example.

Step 4 does the missing correction from step 3 at the same time as the normal multiply and subtract operation from step 4. The negative $q = -210 = -10,000 + 9790$ can be interpreted as subtracting 1 from the previous q while also using $q = 9790$ for the current step. The final normalization pass moves the -1 to the correct word.

3. Details of the Algorithm.

The algorithm will be given using Mathematica [9]. This version does not handle various special cases such as zero denominator, and has not been tuned for efficiency. Both arguments are assumed to be positive, with length at least five. No provision is made for normalizing the digits of W to avoid overflow. If rounding is to be fairly accurate, the base b should be at least 100. For a full implementation, see the Fortran subroutine in FMLIB 1.1.

```
RealW[ j_ ] := N[ ((W[[j-1]]*b + W[[j]])*b + W[[j+1]])*b +
               If[ Length[W]>=j+2 , W[[j+2]] , 0 ] ];

subtract[ q_ , d_ , ka_ , kb_ ] := Module[ { },
      Do[ W[[j]] = W[[j]] - q*d[[j-ka+2]] , {j,ka,kb} ];
      Return[ W ] ];

normalize[  ka_ , q_  ] := Module[ { },
      W[[ka]] = W[[ka]] + W[[ka-1]]*b;
      W[[ka-1]] = q;
      Return[ W ] ];

FinalNorm[ kb_ ] := Module[ {carry},
      Do[ carry = If[ W[[j]]<0 , Floor[ (-W[[j]]-1)/b] + 1 ,
                      If[ W[[j]]>=b , -Floor[ W[[j]]/b] , 0 ] ];
          W[[j]] = W[[j]] + carry*b;
          W[[j-1]] = W[[j-1]] - carry
      , {j,kb,3,-1} ];
      Return[ W ] ];

divide[ n_ , d_ ] := Module[ {last,laststep,q,result,round,xd,xn},
      W = Table[ 0 , {Length[n]+4} ];
      W[[1]] = n[[1]] - d[[1]] + 1;
      Do[ W[[j+1]] = n[[j]] , {j,2,Length[n]} ];
```

```
xd = N[ (d[[2]]*b + d[[3]])*b + d[[4]] + d[[5]]/b  ];
laststep =  t + 2;
Do[ xn = RealW[ step+2 ];
    q = Floor[ xn/xd ];
    last = Min[ step+t+1 , Length[W] ];
    W = subtract[ q , d , step+2 , last ];
    W = normalize[ step+2 , q ]
, {step,1,laststep} ];
W = FinalNorm[ laststep+1 ];
laststep = If[ W[[2]]==0 , laststep , laststep-1 ];
round = N[ W[[laststep+1]]/b ];
W[[laststep]] = W[[laststep]] + If[ round>=0.5 , 1 , 0 ];
result = If[ W[[2]]==0 , Table[ W[[j+1]] , {j,1,t+1} ],
                         Table[ W[[j]] , {j,1,t+1} ] ];
result[[1]] = If[ W[[2]]==0 , W[[1]]-1 , W[[1]] ];
Return[ result ] ]
```

The base b, precision t, and working array W are global symbols in this version. The following command performs the division shown in the previous section.

```
b = 10000;
t = 5;
n = { 1, 3, 1415, 9265, 3589, 7932 };
d = { 1, 2, 7182, 8182, 8459, 0452 };
divide[ n , d ]
```

In practice, W may be of type integer or double precision. x_n and x_d are computed in double precision. Let T be the threshold below which integers can be represented exactly in double precision and rounding errors do not affect arithmetic operations with integer arguments and results. The base is chosen so that $b \geq 2$ and so that b^2 can be represented exactly in W. This implies $b^2 < T$.

Using four words of W to compute x_n gives good accuracy for q. Suppose n is the exact value of the active part of W, scaled so that x_n is an integer. Then during step k,

$$n = \mathrm{W}_{k+1}\, b^3 + \mathrm{W}_{k+2}\, b^2 + \mathrm{W}_{k+3}\, b + \mathrm{W}_{k+4} + \frac{\mathrm{W}_{k+5}}{b} + \cdots .$$

The goal is to approximate n/d, where d is the scaled denominator

$$d = \mathrm{d}_2\, b^2 + \mathrm{d}_3\, b + \mathrm{d}_4 + \frac{\mathrm{d}_5}{b} + \frac{\mathrm{d}_6}{b^2} + \cdots .$$

Because d is normalized, $d \geq b^2$.

x_n and x_d are defined as the first four terms of these sums. We will bound the error in the estimate q.

Suppose that the value q used in step $k-1$ is no more than one in error. Then the correct quotient digit $q^* = \lfloor n/d \rfloor$ for step k satisfies $-b \leq q^* \leq 2\,b$. So $q^* \leq n/d \leq q^* + 1$ implies $|n/d| \leq 2\,b + 1$.

After step k, $|\mathrm{W}_i| < k\,b^2$. This means $|n - x_n| < 2\,k\,b$. Since $|d - x_d\,| < 1\,/\,b \leq d/b^3$ we have

$$\left|\frac{x_n}{x_d}\right| < \frac{|n| + 2\,k\,b}{d\,(1 - b^{-3})} < 2\,b + 3 + \frac{3\,k}{b}.$$

The error in q is

$$\left|\frac{n}{d} - \frac{x_n}{x_d}\right| = \left|\frac{x_d\,(n - x_n) - x_n\,(d - x_d)}{d\,x_d}\right| \leq \left|\frac{n - x_n}{d}\right| + \left|\frac{x_n}{x_d}\right| \left|\frac{d - x_d}{d}\right| < \frac{2\,k}{b} + \frac{2\,k + 4}{b^2}.$$

When a large base is used ($b^4 > T$), it often happens that x_n and x_d are not as accurate as assumed above, because of double precision rounding errors. In this case, we can assume that the two values actually computed, y_n and y_d, agree with x_n and x_d to about double precision accuracy. Then

$$\left|\frac{x_n - y_n}{x_n}\right| < \frac{1}{b^2} \qquad \text{and} \qquad \left|\frac{x_d - y_d}{x_d}\right| < \frac{1}{b^2}.$$

Here the quotient digit used is $q = \lfloor\, y_n/y_d\,\rfloor$. Then

$$\left|\frac{y_n}{y_d} - \frac{x_n}{x_d}\right| \leq \left|\frac{x_n}{y_d}\right| \left|\frac{y_n - x_n}{x_n}\right| + \left|\frac{x_n}{y_d}\right| \left|\frac{y_d - x_d}{x_d}\right| < \frac{4}{b} + \frac{5\,k + 13}{b^2}.$$

The inequalities above assume $b \geq 2$, and the bound on the error in q could be made slightly smaller for the case where b is large. For the usual case where $b \geq 10^4$, having $|\,q - q^*\,| \leq 1$ in step $k-1$ implies that the quotient digit used in step k is also no more than one in error.

This indicates that when b is large, using the unnormalized values in W gives an estimate q that is never more than one in error, and the probability of q being wrong is reasonably small.

The algorithm is still correct when the base is small, even though quotient digits may be used that are more than one in error. The final normalization pass takes longer because more unnormalized digits are generated initially. Another possible strategy for dealing with a small base is to use more than four words to generate x_n and x_d.

The other practical matter that must be addressed in a routine using this algorithm involves avoiding overflow or loss of precision. This can occur when the unnormalized intermediate results become too large. Suppose W is implemented as an integer array, $b = 10^4$, and the overflow threshold is $2^{31} - 1 = 2,147,483,647$. Since $21\,b^2$ will not overflow and each q is less than $2\,b$, 10 steps can be taken before normalizing the active part of W. As implemented in [7], the routine uses

the digits being multiplied to get a sharper upper bound on the elements of W. The average q is about $b/2$, so there are approximately 40 steps between normalization.

In a more recent version of [7], W is a double precision array. Then the threshold for representing integers exactly in double precision is $T = 2^{53} - 1 \approx 9.01 \times 10^{15}$ on most 32-bit machines. In this case the commonly used base is 10^7, so about 180 steps can be done before normalizing.

The main reason for guaranteeing that q is within one of the correct value is to avoid cases where the next q cannot correct an error in the previous step. Suppose in the double precision case that $b = 10^7$ and q is 100 in error. Then in the next step q will be about $100\,b$ in magnitude. Since the elements of d to be multiplied by q may be as large as $b - 1$, these products might exceed T. This could result in some products not being exact. To normalize W at this point would not solve the problem — a more complicated correction step would be needed. When smaller bases are used for the arithmetic, correspondingly larger errors in q can be tolerated by the algorithm.

4. Conclusion. This technique for eliminating most of the intermediate digit normalization operations can be used to perform multiple-precision division faster than previous methods. A routine implementing this method was compared to the division algorithm used in FMLIB 1.0, which normalizes several digits of the intermediate result at each step. Running on a 68040 Macintosh, the new version was 2.2 times faster at 40 significant digits, and 1.8 times faster at 500 digits. Similar speed improvements were found in testing on other types of computer.

The theoretical complexity for division is the same as that for multiplication, but in most multiple-precision packages division is significantly slower, even at high precision. In tests of the algorithm described above, the ratio of running times for division and multiplication ranged from about 1.6 at 40 decimal digits to 1.14 at 1,000 digits. Since the intermediate results are not normalized often, the main difference in running time is due to the double precision calculation of x_n and q at each step. This means that at high precision division is essentially as fast as multiplication.

Mathematics Department

Loyola Marymount University

Los Angeles, CA 90045

1. D.H. Bailey, "Multiprecision Translation and Execution of FORTRAN Programs." *ACM Trans. Math. Softw. 19*, 3 (September 1993), 288–319.

2. R.P. Brent, "A Fortran Multiple-Precision Arithmetic Package." *ACM Trans. Math. Software 4*, 1 (March 1978), 57–70.

3. D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, second edition. Addison Wesley, Reading, Mass., 1981.

4. E.V. Krishnamurthy and S.K. Nandi, "On the Normalization Requirement of Divisor in Divide-and-Correct Methods." *Comm. ACM 10*, 12 (December 1967), 809–813.

5. C.J. Mifsud, "A Multiple-Precision Division Algorithm." *Comm. ACM 13*, 11 (November 1970), 666–668.

6. D.A. Pope and M.L. Stein, "Multiple Precision Arithmetic." *Comm. ACM 3*, 12 (December 1960), 652–654.

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

8. M.L. Stein, "Divide-and-Correct Methods for Multiple Precision Division." *Comm. ACM 7*, 8 (August 1964), 472–474.

3. S. Wolfram, *Mathematica: A System for doing Mathematics by Computer*, second edition. Addison Wesley, Redwood City, Ca., 1991.