THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of Computer Programming

VOLUME 2

Seminumerical Algorithms

Third Edition

DONALD E. KNUTH

This page intentionally left blank

```
28          J1N   N2              If not, normalize.
29          JMP   2F
30    1H    J1P   N2
31    2H    SRC   5               |rX| ↔ |rA|.
32          DECX  1               (rX is positive.)
33          STA   TEMP            (The operands had opposite signs;
34          STA   HALF(0:0)         we must adjust the registers
35          LDAN  TEMP              before rounding and normalization.)
36          ADD   HALF
37          ADD   HALF            Complement the least significant portion.
38          SRC   5               Jump into normalization routine.
39          JMP   N2
40    HALF  CON   1//2            Half of the word size (Sign varies)
41    FU    CON   0               Fraction part f_u
42    FV    CON   0               Fraction part f_v
43    NORM  JAZ   ZRO             N1. Test f.
44    N2    CMPA  =0=(1:1)        N2. Is f normalized?
45          JNE   N5              To N5 if leading byte nonzero.
46    N3    SLAX  1               N3. Scale left.
47          DEC2  1               Decrease e by 1.
48          JMP   N2              Return to N2.
49    N4    ENTX  1               N4. Scale right.
50          SRC   1               Shift right, insert "1" with proper sign.
51          INC2  1               Increase e by 1.
52    N5    CMPA  =BYTE/2=(5:5)   N5. Round.
53          JL    N6              Is |tail| < (1/2)b?
54          JG    5F
55          JXNZ  5F              Is |tail| > (1/2)b?
56          STA   TEMP            |tail| = (1/2)b; round to odd.
57          LDX   TEMP(4:4)
58          JXO   N6              To N6 if rX is odd.
59    5H    STA   *+1(0:0)        Store sign of rA.
60          INCA  BYTE            Add b^{-4} to |f|. (Sign varies)
61          JOV   N4              Check for rounding overflow.
62    N6    J2N   EXPUN           N6. Check e. Underflow if e < 0.
63    N7    ENTX  0,2             N7. Pack. rX ← e.
64          SRC   1
65    ZRO   DEC2  BYTE            rI2 ← e − b.
66    8H    STA   ACC
67    EXITF J2N   *               Exit, unless e ≥ b.
68    EXPOV HLT   2               Exponent overflow detected
69    EXPUN HLT   1               Exponent underflow detected
70    ACC   CON   0               Floating point accumulator ▮
```

The rather long section of code from lines 26 to 40 is needed because MIX has only a 5-byte accumulator for adding signed numbers while in general $2p+1 = 9$ places of accuracy are required by Algorithm A. The program could be shortened to about half its present length if we were willing to sacrifice a little bit of its accuracy, but we shall see in the next section that full accuracy is important. Line 58 uses a nonstandard MIX instruction defined in Section 4.5.2. The running

time for floating point addition and subtraction depends on several factors that are analyzed in Section 4.2.4.

Now let us consider multiplication and division, which are simpler than addition, and somewhat similar to each other.

Algorithm M (*Floating point multiplication or division*). Given base b, excess q, p-digit, normalized floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$, this algorithm forms the product $w = u \otimes v$ or the quotient $w = u \oslash v$.

M1. [Unpack.] Separate the exponent and fraction parts of the representations of u and v. (Sometimes it is convenient, but not necessary, to test the operands for zero during this step.)

M2. [Operate.] Set

$$\begin{aligned} e_w &\leftarrow e_u + e_v - q, \quad & f_w &\leftarrow f_u\, f_v & &\text{for multiplication;} \\ e_w &\leftarrow e_u - e_v + q + 1, \quad & f_w &\leftarrow (b^{-1} f_u)/f_v & &\text{for division.} \end{aligned} \tag{9}$$

(Since the input numbers are assumed to be normalized, it follows that either $f_w = 0$, or $1/b^2 \le |f_w| < 1$, or a division-by-zero error has occurred.) If necessary, the representation of f_w may be reduced to $p+2$ or $p+3$ digits at this point, as in exercise 5.

M3. [Normalize.] Perform Algorithm N on (e_w, f_w) to normalize, round, and pack the result. (*Note:* Normalization is simpler in this case, since scaling left occurs at most once, and since rounding overflow cannot occur after division.) ∎

The following `MIX` subroutines, intended to be used in connection with Program A, illustrate the machine considerations that arise in Algorithm M.

Program M (*Floating point multiplication and division*).

```
01  Q      EQU  BYTE/2        q is half the byte size
02  FMUL   STJ  EXITF         Floating point multiplication subroutine:
03         JOV  OFLO          Ensure that overflow is off.
04         STA  TEMP          TEMP ← v.
05         LDX  ACC           rX ← u.
06         STX  FU(0:4)       FU ← ± f f f f 0.
07         LD1  TEMP(EXP)
08         LD2  ACC(EXP)
09         INC2 -Q,1          rI2 ← e_u + e_v − q.
10         SLA  1
11         MUL  FU            Multiply f_u times f_v.
12         JMP  NORM          Normalize, round, and exit.
13  FDIV   STJ  EXITF         Floating point division subroutine:
14         JOV  OFLO          Ensure that overflow is off.
15         STA  TEMP          TEMP ← v.
16         STA  FV(0:4)       FV ← ± f f f f 0.
17         LD1  TEMP(EXP)
18         LD2  ACC(EXP)
19         DEC2 -Q,1          rI2 ← e_u − e_v + q.
```

```
20        ENTX 0
21        LDA  ACC
22        SLA  1          rA ← f_u.
23        CMPA FV(1:5)
24        JL   *+3        Jump if |f_u| < |f_v|.
25        SRA  1          Otherwise, scale f_u right
26        INC2 1               and increase rI2 by 1.
27        DIV  FV         Divide.
28        JNOV NORM       Normalize, round, and exit.
29 DVZRO  HLT  3          Unnormalized or zero divisor  ▮
```

The most noteworthy feature of this program is the provision for division in lines 23–26, which is made in order to ensure enough accuracy to round the answer. If $|f_u| < |f_v|$, straightforward application of Algorithm M would leave a result of the form "$\pm 0 f f f f$" in register A, and this would not allow a proper rounding without a careful analysis of the remainder (which appears in register X). So the program computes $f_w \leftarrow f_u/f_v$ in this case, ensuring that f_w is either zero or normalized in all cases; rounding can proceed with five significant bytes, possibly testing whether the remainder is zero.

We occasionally need to convert values between fixed and floating point representations. A "fix-to-float" routine is easily obtained with the help of the normalization algorithm above; for example, in MIX, the following subroutine converts an integer to floating point form:

```
01 FLOT STJ  EXITF  Assume that rA = u, an integer.
02      JOV  OFLO   Ensure that overflow is off.
03      ENT2 Q+5    Set raw exponent.                    (10)
04      ENTX 0
05      JMP  NORM   Normalize, round, and exit.  ▮
```

A "float-to-fix" subroutine is the subject of exercise 14.

The debugging of floating point subroutines is usually a difficult job, since there are so many cases to consider. Here is a list of common pitfalls that often trap a programmer or machine designer who is preparing floating point routines:

1) *Losing the sign.* On many machines (not MIX), shift instructions between registers will affect the sign, and the shifting operations used in normalizing and scaling numbers must be carefully analyzed. The sign is also lost frequently when minus zero is present. (For example, Program A is careful to retain the sign of register A in lines 33–37. See also exercise 6.)

2) *Failure to treat exponent underflow or overflow properly.* The size of e_w should not be checked until *after* the rounding and normalization, because preliminary tests may give an erroneous indication. Exponent underflow and overflow can occur on floating point addition and subtraction, not only during multiplication and division; and even though this is a rather rare occurrence, it must be tested each time. Enough information should be retained so that meaningful corrective actions are possible after overflow or underflow has occurred.

It has unfortunately become customary in many instances to ignore exponent underflow and simply to set underflowed results to zero with no indication of error. This causes a serious loss of accuracy in most cases (indeed, it is the loss of *all* the significant digits), and the assumptions underlying floating point arithmetic have broken down; so the programmer really must be told when underflow has occurred. Setting the result to zero is appropriate only in certain cases when the result is later to be added to a significantly larger quantity. When exponent underflow is not detected, we find mysterious situations in which $(u \otimes v) \otimes w$ is zero, but $u \otimes (v \otimes w)$ is not, since $u \otimes v$ results in exponent underflow but $u \otimes (v \otimes w)$ can be calculated without any exponents falling out of range. Similarly, we can find positive numbers a, b, c, d, and y such that

$$\begin{aligned} (a \otimes y \oplus b) \oslash (c \otimes y \oplus d) &\approx \tfrac{2}{3}, \\ (a \oplus b \oslash y) \oslash (c \oplus d \oslash y) &= 1 \end{aligned} \tag{11}$$

if exponent underflow is not detected. (See exercise 9.) Even though floating point routines are not precisely accurate, such a disparity as (11) is certainly unexpected when a, b, c, d, and y are all *positive*! Exponent underflow is usually not anticipated by a programmer, so it needs to be reported.*

3) *Inserted garbage.* When scaling to the left it is important to keep from introducing anything but zeros at the right. For example, note the '`ENTX 0`' instruction in line 21 of Program A, and the all-too-easily-forgotten '`ENTX 0`' instruction in line 04 of the `FLOT` subroutine (10). (But it would be a mistake to clear register X after line 27 in the division subroutine.)

4) *Unforeseen rounding overflow.* When a number like .999999997 is rounded to 8 digits, a carry will occur to the left of the decimal point, and the result must be scaled to the right. Many people have mistakenly concluded that rounding overflow is impossible during multiplication, since they look at the maximum value of $|f_u f_v|$, which is $1 - 2b^{-p} + b^{-2p}$; and this cannot round up to 1. The fallacy in this reasoning is exhibited in exercise 11. Curiously, it turns out that the phenomenon of rounding overflow *is* impossible during floating point division (see exercise 12).

* On the other hand, we must admit that today's high-level programming languages give the programmer little or no satisfactory way to make use of the information that a floating point routine wants to provide; and the `MIX` programs in this section, which simply halt when errors are detected, are even worse. There are numerous important applications in which exponent underflow is relatively harmless, and it is desirable to find a way for programmers to cope with such situations easily and safely. The practice of silently replacing underflows by zero has been thoroughly discredited, but there is another alternative that has recently been gaining much favor, namely to modify the definition that we have given for floating point numbers, allowing an unnormalized fraction part when the exponent has its smallest possible value. This idea of "gradual underflow," which was first embodied in the hardware of the Electrologica X8 computer, adds only a small amount of complexity to the algorithms, and it makes exponent underflow impossible during addition or subtraction. The simple formulas for relative error in Section 4.2.2 no longer hold in the presence of gradual underflow, so the topic is beyond the scope of this book. However, by using formulas like $\text{round}(x) = x(1-\delta)+\epsilon$, where $|\delta| < b^{1-p}/2$ and $|\epsilon| < b^{-p-q}/2$, one can show that gradual underflow succeeds in many important cases. See W. M. Kahan and J. Palmer, *ACM SIGNUM Newsletter* (October 1979), 13–21.

There is a school of thought that says it is harmless to "round" a value like .999999997 to .99999999 instead of to 1.0000000, since this does not increase the worst-case bounds on relative error. The floating decimal number 1.0000000 may be said to represent all real values in the interval

$$[1.0000000 - 5 \times 10^{-8} \;..\; 1.0000000 + 5 \times 10^{-8}],$$

while .99999999 represents all values in the much smaller interval

$$(.99999999 - 5 \times 10^{-9} \;..\; .99999999 + 5 \times 10^{-9}).$$

Even though the latter interval does not contain the original value .999999997, each number of the second interval is contained in the first, so subsequent calculations with the second interval are no less accurate than with the first. This ingenious argument is, however, incompatible with the mathematical philosophy of floating point arithmetic expressed in Section 4.2.2.

5) *Rounding before normalizing.* Inaccuracies are caused by premature rounding in the wrong digit position. This error is obvious when rounding is being done to the left of the appropriate position; but it is also dangerous in the less obvious cases where rounding is first done too far to the right, followed by rounding in the true position. For this reason it is a mistake to round during the "scaling-right" operation in step A5, except as prescribed in exercise 5. (The special case of rounding in step N5, then rounding again after rounding overflow has occurred, is harmless, however, because rounding overflow always yields ± 1.0000000 and such values are unaffected by the subsequent rounding process.)

6) *Failure to retain enough precision in intermediate calculations.* Detailed analyses of the accuracy of floating point arithmetic, made in the next section, suggest strongly that normalizing floating point routines should always deliver a properly rounded result to the maximum possible accuracy. There should be no exceptions to this dictum, even in cases that occur with extremely low probability; the appropriate number of significant digits should be retained throughout the computations, as stated in Algorithms A and M.

C. Floating point hardware. Nearly every large computer intended for scientific calculations includes floating point arithmetic as part of its repertoire of built-in operations. Unfortunately, the design of such hardware usually includes some anomalies that result in dismally poor behavior in certain circumstances, and we hope that future computer designers will pay more attention to providing the proper behavior than they have in the past. It costs only a little more to build the machine right, and considerations in the following section show that substantial benefits will be gained. Yesterday's compromises are no longer appropriate for modern machines, based on what we know now.

The `MIX` computer, which is being used as an example of a "typical" machine in this series of books, has an optional "floating point attachment" (available at extra cost) that includes the following seven operations:

• `FADD`, `FSUB`, `FMUL`, `FDIV`, `FLOT`, `FCMP` (C = 1, 2, 3, 4, 5, 56, respectively; F = 6). The contents of rA after the operation '`FADD V`' are precisely the same as the

contents of rA after the operations

```
STA ACC;  LDA V;  JMP FADD
```

where FADD is the subroutine that appears earlier in this section, except that both operands are automatically normalized before entry to the subroutine if they were not already in normalized form. (If exponent underflow occurs during this pre-normalization, but not during the normalization of the answer, no underflow is signalled.) Similar remarks apply to FSUB, FMUL, and FDIV. The contents of rA after the operation 'FLOT' are the contents after 'JMP FLOT' in the subroutine (10) above.

The contents of rA are unchanged by the operation 'FCMP V'. This instruction sets the comparison indicator to LESS, EQUAL, or GREATER, depending on whether the contents of rA are "definitely less than," "approximately equal to," or "definitely greater than" V, as discussed in the next section. The precise action is defined by the subroutine FCMP of exercise 4.2.2–17 with EPSILON in location 0.

No register other than rA is affected by any of the floating point operations. If exponent overflow or underflow occurs, the overflow toggle is turned on and the exponent of the answer is given modulo the byte size. Division by zero leaves undefined garbage in rA. Execution times: $4u$, $4u$, $9u$, $11u$, $3u$, $4u$, respectively.

• FIX (C = 5; F = 7). The contents of rA are replaced by the integer "round(rA)", rounding to the nearest integer as in step N5 of Algorithm N. However, if this answer is too large to fit in the register, the overflow toggle is set on and the result is undefined. Execution time: $3u$.

Sometimes it is helpful to use floating point operators in a nonstandard way. For example, if the operation FLOT had not been included as part of MIX's floating point attachment, we could easily achieve its effect on 4-byte numbers by writing

```
FLOT STJ  9F
     SLA  1
     ENTX Q+4
     SRC  1
     FADD =0=
9H   JMP  *
```
(12)

This routine is not strictly equivalent to the FLOT operator, since it assumes that the 1:1 byte of rA is zero, and it destroys rX. The handling of more general situations is a little tricky, because rounding overflow can occur even during a FLOT operation.

Similarly, suppose MIX had a FADD operation but not FIX. If we wanted to round a number u from floating point form to the nearest fixed point integer, and if we knew that the number was nonnegative and would fit in at most three bytes, we could write

```
FADD  FUDGE
```