BIT manuscript No.
(will be inserted by the editor)

Computation of the unit in the first place (ufp) and the unit in the last place (ulp) in precision-p base β

Siegfried M. Rump

Received: date / Accepted: date

Abstract There are simple algorithms to compute the predecessor, successor, unit in the first place, unit in the last place etc. in binary arithmetic. In this note equally simple algorithms for computing the unit in the first place and the unit in the last place in precision-p base-β arithmetic with $p \geqslant 1$ and with $\beta \geqslant 2$ are presented. The algorithms work in the underflow range, and numbers close to overflow are treated by scaling. The algorithms use only the basic operations with directed rounding. If the successor (or predecessor) of a floating-point number is available, an algorithm in rounding to nearest is presented as well.

Keywords Unit in the first place, unit in the last place, floating-point arithmetic, precision-p, base-β, predecessor, successor, INTLAB

Mathematics Subject Classification (2000) 65G99

1 Notation and main result

Let $\mathbb{F}_{\mathcal{N}}$ denote the set of normalized precision-p base-β floating-point numbers

$$\mathbb{F}_{\mathcal{N}} := \{ \pm m\beta^e \text{ with } \beta^{p-1} \leqslant m \leqslant \beta^p - 1 \text{ and } E_{\min} \leqslant e \leqslant E_{\max}\}, \quad (1.1)$$

and denote by

$$\mathbb{F}_{\mathcal{D}} := \{ \pm m\beta^{E_{\min}} \text{ with } 1 \leqslant m < \beta^{p-1}\} \quad (1.2)$$

the set of denormalized numbers. Then $\mathbb{F} := \mathbb{F}_{\mathcal{N}} \cup \mathbb{F}_{\mathcal{D}} \cup \{0\}$ is the set of all precision-p base-β floating-point numbers.[1] Set $\mathbb{F}^* := \mathbb{F} \cup \{-\infty, \infty\}$ and let an

S. M. Rump
Institute for Reliable Computing, Hamburg University of Technology, Am Schwarzenberg-Campus 3, Hamburg 21073, Germany, and Visiting Professor at Waseda University, Faculty of Science and Engineering, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan E-mail: rump@tuhh.de

[1] Note that our definition of $E_{\min}$ and $E_{\max}$ differs by $p-1$ from that in [13,8].

arithmetic on $\mathbb{F}^*$ following the IEEE 754 standard [7,8] be given. That means in particular that in RoundToNearest all floating-point operations have minimal error, bounded by the relative rounding error unit $\mathbf{u} := \frac{1}{2}\beta^{1-p}$. Moreover, different rounding modes are available, also with best possible result.

In [19] we introduced the "unit in the first place" (ufp) which is defined by

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) := \beta^{\lfloor \log_\beta |r| \rfloor}$$

and $\mathrm{ufp}(0) := 0$. For all real $r \in \mathbb{R}$ it is the value of the left-most nonzero digit[2] in the base β-representation.

In contrast, the often used "unit in the last place" (ulp) depends on the precision of the floating-point format in use. For a nonzero finite base-β string it is the magnitude of its least significant digit, or in other words, the distance between the floating point number and the next floating point number of greater magnitude [6][3]. There are several other definitions of the unit in the last place, in particular for real $r \notin \mathbb{F}$, cf. [12,13,2]. We use the definition above, namely $\mathrm{ulp}(r) = \beta^e$ for $r \in \mathbb{F}_{\mathcal{N}}$ according to (1.1), $\mathrm{ulp}(r) = \beta^{E_{\min}}$ for $r \in \mathbb{F}_{\mathcal{D}}$, and $\mathrm{ulp}(0) = 0$. All definitions have in common that they depend not only on the basis β but also on the precision of the floating-point format in use.

We invented the unit in the first place in [19] because it was very helpful if not mandatory to formulate complicated proofs of the validity of our new floating-point algorithms for accurate summation and dot products. We developed a small collection of rules using ufp, so that based on that no further understanding of the many properties of the IEEE 754 floating-point arithmetic was necessary to follow the proofs.

The main difference in the definition of ulp compared to ufp is to separate the use of the basis and of the precision. First, ufp is defined for a general real number, only depending on the basis β, and second precision-related results use ufp and the relative rounding error unit, i.e., the precision p. That separation was useful to formulate our proofs in [19] and following papers.

There are simple algorithms to compute the predecessor, successor, unit in the first place, unit in the last place etc. in binary arithmetic [14,10,5,11,3, 13], but apparently no method is known to compute the unit in the first place in a base-β arithmetic. Jean-Michel Muller [15] proposed a method based on the results in [9], however, it needs up to $\log_2(\beta)$ iterations. We are interested in a flat, loop-free algorithm with few operations.

Recently we wrote a toolbox for an IEEE 754 precision-p base-β arithmetic with specifiable exponent range[18] as part of INTLAB [16], the Matlab/Octave toolbox for reliable computing. As part of this we present in this note a simple algorithm to compute the unit in the first place for a precision-p base-β arithmetic with $p \geqslant 1$ and $\beta \geqslant 2$. The algorithm works correctly in the underflow range, where numbers close to overflow are treated by scaling.

[2] Using a finite representation if possible, i.e., avoiding infinitely many trailing $\beta-1$-digits.

[3] Harrison notes this in [6], which is different from what is called "Harrison's-ulp" [13,2]

That algorithm requires a directed rounding, i.e., RoundToZero, RoundUp or RoundDown; we could not construct a simple algorithm in RoundToNearest.

In addition, as a reply to suggestions by the referees, we present some additional algorithms to compute ufp and ulp. Those require a specific directed rounding mode and/or access to the predecessor/successor of a floating-point number. Since these algorithms are pretty obvious and the proofs of correctness are trivial, we banned them into the appendix.

We formulate our algorithm to compute the unit in the first place in the rounding mode RoundToZero and call the corresponding mapping $\mathrm{fl}_\diamond : \mathbb{R} \to \mathbb{F}$. It follows that the result of a floating-point operation with positive real result x is $\max\{f \in \mathbb{F} : f \leqslant x\}$, and that operations cannot cause overflow.

The predecessor and successor of $x \in \mathbb{R}$ in $\mathbb{F}^*$ is defined by

$$\begin{aligned} \mathrm{pred}(x) &:= \max\{g \in \mathbb{F}^* : g < x\} \\ \mathrm{succ}(x) &:= \min\{g \in \mathbb{F}^* : x < g\}, \end{aligned}$$

respectively. In precision-p base-β arithmetic we have

$$E_{\min} < k \leqslant E_{\max} \;\Rightarrow\; \mathrm{pred}(\beta^k) = (1-\beta^{-p})\beta^k \tag{1.3}$$

$$0 < f \in \mathbb{F}_{\mathcal{N}} \text{ and } f \neq \mathrm{ufp}(f) \;\Rightarrow\; \mathrm{pred}(f) = f - \beta^{1-p}\mathrm{ufp}(f) \tag{1.4}$$

$$0 < f \in \mathbb{F}_{\mathcal{N}} \;\Rightarrow\; \mathrm{succ}(f) = f + \beta^{1-p}\mathrm{ufp}(f) \tag{1.5}$$

Note that (1.5) includes the case $p = 1$, $\beta = 2$ for which $\mathbb{F}$ is the set of powers of 2, i.e., $f = \mathrm{ufp}(f)$ for all $f \in \mathbb{F}$, and $\mathrm{succ}(f) = f + \beta^{1-p}\mathrm{ufp}(f) = 2f$. Among the properties of ufp [19] is

$$0 \neq f \in \mathbb{F} \;\Rightarrow\; \mathrm{ufp}(f) \leqslant |f| \leqslant \beta(1-\beta^{-p}) \cdot \mathrm{ufp}(f). \tag{1.6}$$

Next we present in Figure 1.1 our algorithm to compute $\mathrm{ufp}(f)$ for $f \in \mathbb{F}$ in precision-p base-β arithmetic and RoundToZero or RoundDown. It is obvious how to adapt the algorithm for RoundUp. We assume that `subrealmin`, the smallest positive denormalized floating-point number equal to $\beta^{E_{\min}}$, is available. Overflow is easily avoided by proper scaling, but we omit that technical detail. Note that in a practical implementation, the constants `p1` and `phi` in lines 2 and 3 of Algorithm `ufp` would be stored rather than calculated, and the extra input parameters `p` and `beta` would be omitted.

Fig. 1.1: Algorithm ufp in RoundToZero or RoundDown

```
1   function S = ufp(f,p,beta)  % precision-p, base-beta
2     p1 = 1 - subrealmin;      % p1 = pred(1)
3     phi = beta^(p-1) + 1;
4     q = phi*abs(f);           % result in the normalized range
5     S = q  - p1*q;            % S = ufp(f)
```

Theorem 1.1 *Let S be the result of Algorithm* `ufp` *applied to $f \in \mathbb{F}$, where $E_{\min} \leqslant -1 < p \leqslant E_{\max}$. Suppose that all operations are executed in precision-p base-β floating-point arithmetic following the IEEE 754 standard with $p \geqslant 1$ and $\beta \geqslant 2$ in RoundToZero or RoundDown, and that $|f| < \beta^{E_{\max}-p+1}$. Then S is equal to* $\mathrm{ufp}(f)$.

Remark 1.1 The usual problems in the denormalized range are avoided because $q \in \mathbb{F}_{\mathcal{N}}$, so that the multiplication in line 5 are in the normalized range. The result of the final subtraction may be in the denormalized range but is error-free because of Sterbenz' lemma [21].

Proof The result is correct for $f = 0$, so henceforth we assume $f \neq 0$. We first verify that the used constants `p1` and `phi` are in $\mathbb{F}$. The rounding RoundToZero or RoundDown implies that p_1 in line 2 is the predecessor of 1, and (1.3) and $E_{\min} \leqslant -1$ yield $p_1 = 1 - \beta^{-p}$. Moreover, $\varphi \in \mathbb{F}$ follows by $\beta^{p-1} + 1 \leqslant \beta^p \leqslant \beta^{E_{\max}}$. Note that this includes the case $\varphi = 2$ for $p = 1$.

The input f is used only in line 4, and since $\mathrm{ufp}(f) = \mathrm{ufp}(|f|)$ we may henceforth assume without loss of generality that $f > 0$. The monotonicity of the rounding, (1.6) and (1.5) imply

$$\begin{aligned}\varphi f &\leqslant (\beta^{p-1} + 1)\beta(1 - \beta^p) \cdot \mathrm{ufp}(f) = (\beta^p + \beta - 1 - \beta^{1-p}) \cdot \mathrm{ufp}(f) \\ &< (1 + \beta^{1-p})\beta^p \mathrm{ufp}(f) = \mathrm{succ}(\beta^p \mathrm{ufp}(f)),\end{aligned}$$

so that the rounding mode implies $q = \mathrm{fl}_\diamond(\varphi f) \leqslant \beta^p \mathrm{ufp}(f)$. Therefore,

$$\beta^{p-1}\mathrm{ufp}(f) \leqslant \mathrm{ufp}(q) \leqslant \beta^p \mathrm{ufp}(f). \tag{1.7}$$

Hence q is always in the normalized range $\mathbb{F}_{\mathcal{N}}$ and $f < \beta^{E_{\max}-p+1}$ yields $\mathrm{ufp}(f) \leqslant \beta^{E_{\max}-p}$ and $q \leqslant \beta^p \mathrm{ufp}(f) \leqslant \beta^{E_{\max}}$.

We distinguish two cases. First, assume $\mathrm{ufp}(q) = \beta^p \mathrm{ufp}(f)$, which implies that $q = \beta^p \mathrm{ufp}(f)$ is a power of β. Then $q \geqslant \beta^p \beta^{E_{\min}} > \beta^{E_{\min}}$ and (1.3) yield

$$r := \mathrm{fl}_\diamond((1 - \beta^{-p})q) = \mathrm{pred}(q) = (1 - \beta^{-p})q$$

and therefore $S = \mathrm{fl}_\diamond(q - r) = \mathrm{fl}_\diamond(\beta^{-p}q) = \mathrm{fl}_\diamond(\mathrm{ufp}(f)) = \mathrm{ufp}(f)$. According to (1.7) it remains the second case

$$\mathrm{ufp}(q) = \mathrm{ufp}(\mathrm{fl}_\diamond((\beta^{p-1} + 1)f)) = \beta^{p-1}\mathrm{ufp}(f). \tag{1.8}$$

Note that $p = 1$ and $\beta = 2$ belongs to the first case $\mathrm{ufp}(q) = \beta^p \mathrm{ufp}(f)$. Next $\beta^{p-1} f \in \mathbb{F}_{\mathcal{N}}$ and (1.5) give

$$\begin{aligned}q &= \mathrm{fl}_\diamond((\beta^{p-1} + 1)f) = \mathrm{fl}_\diamond((1 + \beta^{1-p})\beta^{p-1} f) \\ &\geqslant \mathrm{fl}_\diamond(\beta^{p-1} f + \beta^{1-p}\mathrm{ufp}(\beta^{p-1} f)) = \mathrm{succ}(\beta^{p-1} f) \\ &\geqslant \mathrm{succ}(\beta^{p-1}\mathrm{ufp}(f)) = \mathrm{succ}(\mathrm{ufp}(q)).\end{aligned}$$

The monotonicity of the rounding, (1.6), $q > \mathrm{ufp}(q)$ and (1.4) yield

$$\begin{aligned}q &= \mathrm{fl}_\diamond(q) > \mathrm{fl}_\diamond((1 - \beta^{-p})q) =: r \\ &\geqslant \mathrm{fl}_\diamond(q - \beta^{1-p}(1 - \beta^{-p})\mathrm{ufp}(q)) \geqslant \mathrm{fl}_\diamond(q - \beta^{1-p}\mathrm{ufp}(q)) \\ &= \mathrm{pred}(q),\end{aligned}$$

and therefore $r = \mathrm{pred}(q) = q - \beta^{1-p}\mathrm{ufp}(q) = q - \mathrm{ufp}(f)$. Hence $S = \mathrm{fl}_\diamond(q - r) = \mathrm{fl}_\diamond(\mathrm{ufp}(f)) = \mathrm{ufp}(f)$. The theorem is proved. □

Fig. 1.2: Algorithm ufp in executable INTLAB code

```
function S = ufp(f)
  feature('setround',0)           % rounding RoundToZero
  p = flbetainit;                 % precision p
  p1 = 1 - subrealmin('flbeta');  % predecessor of 1
  phi = flbeta(1,p-1) + 1;        % beta^(p-1) + 1
  q = phi*abs(f);                 % result in the normalized range
  S = q  - p1*q;                  % ufp(f)
```

Algorithm `ufp` will part of the `flbeta` toolbox in INTLAB. Executable INTLAB code, which is almost identical to the one given in Figure 1.1, is shown in Figure 1.2, Here `flbeta` is a user-defined data type, where the precision $p \geqslant 1$, the base $\beta \geqslant 2$ as well as the exponent range $(E_{\min}, E_{\max})$ can be specified through initialization by `flbetainit`. As in every operator concept, an operation is executed in `flbeta`-arithmetic if at least one of the operands is of type `flbeta`. The `flbeta` toolbox respects the rounding mode; in line 2 it is switched to RoundToZero using the internal Matlab command `feature`.

The result of `p = flbetainit` as in line 3 without input and with one output argument is the precision p in use. The constructor `flbeta(m,e)` generates the `flbeta` constant $m\beta^e$. Otherwise the code is self-explaining.

Finally we want to mention that the `flbeta` toolbox was very useful for testing in different precisions p, different bases β and exponent ranges $E_{\min}, E_{\max}$. Frankly speaking, we found Algorithm `ufp` experimentally when playing around with different possibilities. However, we did not find a simple algorithm in the nearest rounding RoundTiesToEven.

We close the main part of this note with some open problems. As has been mentioned, we did not succeed to find a simple algorithm to compute ufp solely in rounding to nearest. Here "simple" means few operations without loop.

Problem 1.1 Given a precision-p base-β arithmetic following IEEE 754, find a simple algorithm to compute the unit in the first place (ufp) in rounding to nearest.

The problem is solved [17] in binary for $p \geqslant 1$.

Problem 1.2 Given a precision-p base-β arithmetic following IEEE 754, find a simple algorithm to compute the unit in the last place (ulp) in rounding to nearest.

Concerning units of a floating-point number, there is a third quantity of interest, namely, the magnitude of the least nonzero digit in a finite base-β representation. Historically [15], Shewchuk [20] uses this quantity implicitly for defining his "nonoverlapping expansion", with the notation $\omega(f)$ it appears in [4], and in [1] the notation $\mathrm{uls}(f)$ (unit in the least significant place) is used. For example, in a precision-3 decimal arithmetic and $f = 42$ we have $\mathrm{ufp}(f) = 10$, $\mathrm{ulp}(f) = 0.1$ and $\mathrm{uls}(f) = 1$.

Problem 1.3 Given a precision-p base-β arithmetic following IEEE 754, find a simple algorithm to compute the unit in the least significant place (uls) in any rounding mode.

2 Appendix

We add some more algorithms to compute ufp and ulp requiring specific rounding modes, namely RoundDown, RoundUp and/or RoundToNearest.

To compute ulp(f) in RoundUp [or RoundDown] we can just follow the definition ulp(f) = succ($|f|$) $-$ $|f|$ for nonzero $f \in \mathbb{F}$..

Fig. 2.1: Algorithm ulp in RoundUp

```
1   function S = ulp(f)
2     f = abs(f);
3     s = f + subrealmin;       % succ(abs(f))
4     S = s - f;                % S = ulp(f)
```

The result is correct in precision-p base-β floating-point arithmetic for any nonzero floating-point number f with $|f| < (\beta^p - 1)\beta^{E_{\max}}$, i.e., with absolute value not equal to the largest representable floating-point number `realmax`. If there is a possibility to obtain the successor of a floating-point number, then replacing line 3 by `s = succ(f);` produces correct code in any rounding mode because the computation in line 4 is error-free.

In RoundDown or RoundToZero, a little more effort is necessary to compute ulp(f). The algorithm in Figure 2.2 works for vector or matrix input as well. That is, by the way, also true for the previous algorithms.

Fig. 2.2: Algorithm ulp in RoundDown or RoundToZero

```
1   function S = ulp(f,beta)
2     f = abs(f);
3     p = f - subrealmin;            % pred(abs(f))
4     S = f - p;                     % S = ulp(f) if f is not power of beta
5     index = find( f+S == f );      % f is power of beta
6     if any(index(:))               % S = ulp(f)/beta
7       S(index) = S(index) * beta;  % correct S
8     end
```

The computed `S` in line 4 is correct for positive $f \in \mathbb{F}$ except powers of β in the normalized range. Otherwise, `S` is corrected in lines 5-8. The result is correct for nonzero $f \in \mathbb{F}$ with $|f| <$ `realmax`.

Sometimes if-statements may cause quite some computational overhead. The algorithm in Figure 2.3, working for nonzero $f \in \mathbb{F}$ with $|f| <$ `realmax`, closes that gap. If `f` is not a power of β, then `f + S` is the successor of `f`, so

that `d = 0` in line 5. Hence `S` is not changed in line 6. Otherwise, if `f` is a power of β, then `S = ulp(f)/beta` and `f + S` is equal to `f` in floating-point in the chosen rounding modes. Hence `d = -S = -ulp(f)/beta`, and the computed `S` is corrected into ulp(f) because `d` is a power of β and the computation in line 6 is error-free. However, the algorithm in Figure 2.3 is about twice as slow as the previous one in Figure 2.2.

Fig. 2.3: Algorithm ulp in RoundDown or RoundToZero without if-statement

```
1   function S = ulp(f,beta)
2     f = abs(f);
3     p = f - subrealmin;            % pred(abs(f))
4     S = f - p;                     % S = ulp(f) if f is not power of beta
5     d = ( ( f + S ) - f ) - S;     % d=0 <=> f is not a power of beta
6     S = S - (beta-1)*d;            % correction of S
```

Finally, if there is a possibility to obtain the successor of a floating-point number, then ufp can be calculated in any rounding mode by the algorithm in Table 2.4. The algorithm works, as for Theorem 1.1, correctly for nonzero $f \in \mathbb{F}$ satisfying $|f| < (\beta^p - 1)\beta^{E_{\max}-p+1}$ except that f must be nonzero.

Fig. 2.4: Algorithm ufp in any rounding mode if successor is available

```
1   function S = ufp(f,p)
2     f = abs(f)*beta^(p-1);         % scaled f
3     s = succ(f);
4     S = s - f;
```

That means the posed Problem 1.1 to find a simple algorithm to compute the unit in the first place in RoundToNearest is solved if such an algorithm for the successor is available. In [17] we presented a simple algorithm for binary arithemtic but only estimates for precision-p base-β.

3 Compliance with Ethical Standards

There is no conflict of interest.

Acknowledgements Many thanks to Christoph Lauter and Jean-Michel Muller for their constructive comments to an earlier version of this note. Moreover, my dearest thanks to two unknown referees for many fruitful remarks. Their thoughtful questions led to the algorithms presented in the appendix.

References

1. S. Boldo, M. Joldes, J.-M. Muller, and V. Popescu. Formal Verification of a Floating-Point Expansion Renormalization Algorithm. In *8th International Conference on Interactive Theorem Proving (ITP'2017)*, vol. 10499 of *Lecture Notes in Computer Science*, Brasilia, Brazil, 2017.
2. S. Boldo, C. Lauter, and J.-M. Muller. Emulating round-to-nearest ties-to-zero "augmented" floating-point operations using round-to-nearest ties-to-even arithmetic. *IEEE Transactions on Computers*, 70(7):1046–1058, 2021.
3. R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA, 2010.
4. M. Daumas. Multiplications of floating-point expansions. In Koren and Kornerup, editors, *Proc. of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, 250–257. IEEE Computer Society Press, Los Alamitos, CA, April 1999.
5. T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.
6. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, editors, *Theorem Proving in Higher Order Logics*, 113–130. Springer Berlin Heidelberg, 1999.
7. IEEE. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1–20, 1985.
8. IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 1–84, 2019.
9. C.-P. Jeannerod, J.-M. Muller, and P. Zimmermann. On various ways to split a floating-point number. In *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*, 53–60. IEEE, 2018.
10. W. Kahan. Further Remarks on Reducing Truncation Errors. *Communications of the ACM*, 8(1):40, 1965.
11. D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1969.
12. J.-M. Muller. On the definition of ulp(x). [Research Report] RR-5504, LIP RR-2005-09, INRIA, LIP. ffinria-00070503f.
13. J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, R. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2nd edition, 2018.
14. O. Møller. Quasi double precision in floating-point arithmetic. *BIT Numerical Mathematics*, 5:37–50, 1965.
15. J.-M. Muller. private communication, 2022.
16. S. M. Rump. INTLAB – INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Springer Netherlands, Dordrecht, 1999.
17. S.M. Rump, P. Zimmermann, S. Boldo, and G. Melquiond. Computing predecessor and successor in rounding to nearest. *BIT*, 49(2):419–431, 2009.
18. S.M. Rump. IEEE-754 precision-p base-β arithmetic implemented in binary. *ACM Trans. Math. Software*, to appear.
19. S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.
20. J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.
21. P.H. Sterbenz. *Floating-point computation*. Prentice Hall, Englewood Cliffs, NJ, 1974.