SIAM J. NUMER. ANAL.
Vol. 24, No. 2, April 1987

LEVEL-INDEX ARITHMETIC OPERATIONS*

C. W. CLENSHAW† AND F. W. J. OLVER‡

Abstract. In a recent paper the authors described a system for the internal representation of numbers in a computer, based on repeated exponentiations. The main objective in introducing this system is to eradicate the problems of overflow and underflow. The present paper supplies algorithms for performing the four basic arithmetical operations in the new system. The algorithms are accompanied by error analyses, which show that the algorithms can be executed with fixed-point arithmetic. Illustrative examples are included.

Key words. binomial probability distribution, computer arithmetic, Euclidean norm, generalized exponential, generalized logarithm, generalized precision, overflow, underflow

AMS(MOS) subject classifications. 65-04, 65G99

1. Introduction. Recently the authors introduced a way of representing numbers internally in a computer, based on the idea of repeated exponentiation [2]. If F is any nonnegative real number, then we write

$$F = e^{e^{\cdot^{\cdot^{e^f}}}}, \tag{1.1}$$

where $0 \leqq f < 1$ and the process of exponentiation is performed l times, l being a nonnegative integer. In particular, when $l = 0$, $F = f$. For a given value of F the numbers l and f are determined uniquely. They are called the *level* and *index* of F, respectively, and (1.1) is referred to as the *level-index* (li) form of F.

The level l and index f are somewhat analogous to the exponent ε and mantissa (or fractional part) μ in the standard floating-point representation

$$F = r^{\varepsilon}\mu, \tag{1.2}$$

in which $1/r \leqq \mu < 1$ and r is the internal arithmetic base of the computer. The main advantage of the representation (1.1) over (1.2) is the abolition of underflow and the virtual abolition of overflow. Other advantages discussed in [2] include the ability of the li system to function as a fixed-point system (when $l = 0$) or a floating-point system (when $l = 1$), the elimination of "wobbling precision" and the adoption of the absolute error in the index f as a generalized measure of precision.

In the present paper we show how to carry out arithmetic operations in the li system. Given the levels and indices of two nonnegative numbers F and G, say, we supply algorithms to find the level and index of the difference, sum, product and quotient of F and G. Except at low levels this computation cannot be done simply by finding the explicit fixed- or floating-point forms of F and G by repeated exponentiations, performing the required arithmetic operation and then converting the answer back to li form by repeatedly taking logarithms. The numbers involved may be too large. (Furthermore, if it were always possible to proceed in this manner then there would be little point in introducing the li system.) Instead, we shall construct algorithms that involve only ordinary arithmetic (in fact, fixed-point arithmetic).

* Received by the editors December 27, 1984; accepted for publication (in revised form) March 31, 1986.

† Department of Mathematics, The University of Lancaster, Lancaster, LA1 4YL, England.

‡ Institute for Physical Science and Technology, The University of Maryland, College Park, Maryland 20742. The research of this author has been supported, in part, by the U.S. Army Research Office, Durham, under contract DAAG 29-84-K-0022, the U.S. National Science Foundation under grant MCS 81-11725, and the United Kingdom Science and Engineering Research Council.

In the concluding sections we show, by means of two well-known examples, that the use of the new number system can lead to considerable simplifications in the construction of robust computing algorithms.

2. Arithmetic operations: algorithms.

2.1. Notation. As in [2], when $F \geqq 0$ we may express the li representation (1.1) concisely in either of the forms

$$F = [l/f] = \phi(l+f), \tag{2.1}$$

where $\phi(x)$ is the *generalized exponential function* defined by

$$\phi(x) = x, \quad 0 \leqq x < 1; \qquad \phi(x) = e^{\phi(x-1)}, \quad x \geqq 1. \tag{2.2}$$

Thus if $F = \phi(x)$, then l and f are, respectively, the integer and fractional parts of x.

The function that is inverse to $F = \phi(x)$ is the *generalized logarithm* $x = \psi(F)$, defined by

$$\psi(F) = F, \quad 0 \leqq F < 1; \qquad \psi(F) = \psi(\ln F) + 1, \quad F \geqq 1. \tag{2.3}$$

Explicitly

$$\psi(F) = l + \ln^{(l)} F, \qquad F \geqq 0, \tag{2.4}$$

where $\ln^{(l)} F$ denotes the lth repeated logarithm of F and the nonnegative integer l is determined uniquely by the condition

$$0 \leqq \ln^{(l)} F < 1. \tag{2.5}$$

Both $\phi(\cdot)$ and $\psi(\cdot)$ are increasing C^1 functions in the interval $[0, \infty)$.

Now let F and G be two numbers such that $F \geqq G \geqq 0$. We suppose that they are given in li form

$$F = [l/f] = \phi(x), \qquad x = l + f, \tag{2.6}$$

$$G = [m/g] = \phi(y), \qquad y = m + g, \tag{2.7}$$

and we seek their difference, sum, product or quotient H, say, also in li form

$$H = [n/h] = \phi(z), \qquad z = n + h. \tag{2.8}$$

We note that because $\psi(\cdot)$ is an increasing function we have $x \geqq y$, and hence $l \geqq m$.

2.2. Subtraction. Here $H = F - G$, so that

$$\phi(x) - \phi(y) = \phi(z). \tag{2.9}$$

Suppose first that $l = 0$. Then $m = 0$ and

$$n = 0, \qquad h = f - g. \tag{2.10}$$

Otherwise, we proceed as follows.

We compute three sequences $\{a_j\}$, $\{b_j\}$ and $\{c_j\}$, the members of which are defined by

$$1 = a_j \phi(x - j), \quad b_j = a_j \phi(y - j), \quad c_j = a_j \phi(z - j). \tag{2.11}$$

The a_j's are required for $j = l-1, l-2, \cdots, 0$. The b_j's are required for $j = m-1, m-2, \cdots, 0$, except that we need b_0 also when $m = 0$. The sequence c_j, $j = 0, 1, \cdots$, is terminated as soon as $c_j < a_j$ or $j = l-1$, whichever comes first.

Recurrence relations and initial values for generating the sequences are given by

(2.12) $$a_{j-1}=e^{-1/a_j}, \qquad a_{l-1}=e^{-f},$$

(2.13) $$b_{j-1}=e^{-(1-b_j)/a_j}, \quad b_{m-1}=a_{m-1}\,e^{g} \quad (m\geqq 1), \quad b_0=a_0 g \quad (m=0),$$

(2.14) $$c_j=1+a_j \ln c_{j-1}, \qquad c_0=1-b_0.$$

If c_j $(0\leqq j\leqq l-1)$ is the first member of its sequence to satisfy $c_j<a_j$, then

(2.15) $$n=j, \qquad h=\frac{c_j}{a_j}.$$

However, if $c_j\geqq a_j$ for $j=0,1,\cdots,l-1$, then

(2.16) $$n=l, \qquad h=f+\ln c_{l-1}.$$

The relations (2.12) to (2.16) are readily derivable from (2.2), (2.11) and (2.6) to (2.9). It should be noted that the recurrence relations for the b_j and c_j are the same but they are applied in opposite directions.

2.3. Addition. Here $H=F+G$,

(2.17) $$\phi(x)+\phi(y)=\phi(z).$$

Suppose first that $l=0$. We compute $h_0=f+g$. Then

(2.18) $$n=0, \quad h=h_0 \quad \text{if } h_0<1,$$

or

(2.19) $$n=1, \quad h=\ln h_0 \quad \text{if } h_0\geqq 1.$$

When $l>0$ we again define a_j, b_j and c_j by (2.11). The only modifications to the procedure of § 2.2 are as follows. First, a_0 is not needed when $m>1$. Second, the second equation in (2.14) is replaced by $c_0=1+b_0$; this is the consequence of the replacement of (2.9) by (2.17). Third, the computation of the c_j's always proceeds to $j=l-1$. Subsequently we form h_l, where

(2.20) $$h_l=f+\ln c_{l-1};$$

then

(2.21) $$n=l, \qquad h=h_l \quad \text{if } h_l<1,$$

or

(2.22) $$n=l+1, \qquad h=\ln h_l \quad \text{if } h_l\geqq 1.$$

2.4. Multiplication and division. For multiplication $H=FG$ and

(2.23) $$\phi(x)\phi(y)=\phi(z).$$

First, if $m>0$ then $n>0$ and on taking logarithms in (2.23) we obtain

(2.24) $$\phi(x-1)+\phi(y-1)=\phi(z-1).$$

The values of n and h are found by applying the addition algorithm to the sum on the left-hand side and increasing the level of the answer by unity.

Secondly, if $l>0$ and $m=0$ then $\phi(y)=g$. The sequences $\{a_j\}$ and $\{c_j\}$ are computed exactly as in § 2.2, except that the starting value for $\{c_j\}$ is $c_0=g$. The values of n and h are as in (2.15) or (2.16). (The sequence $\{b_j\}$ is not required here.)

Thirdly, if $l=m=0$ then $n=0$ and $h\equiv fg$ is found by ordinary fixed-point multiplication.

For division the procedures are similar. The first case to consider is given by $H=F/G$, with $F\geqq G>0$. If $m>0$ then we take logarithms and use the subtraction algorithm. Secondly, if $l>0$ and $m=0$ then

$$\phi(z-1)=\phi(x-1)+\ln(1/g). \tag{2.25}$$

Since z necessarily exceeds unity in this case, we may start the computation of the c_j's from

$$c_1=1+a_1\ln(1/g).$$

Thirdly, if $l=m=0$ then

$$\phi(z-1)=\ln f+\ln(1/g) \tag{2.26}$$

and is nonnegative. If necessary, we continue taking logarithms until we obtain a number in the interval $[0,1)$.

The other case of division is given by $H=G/F$, with $F>G\geqq 0$. Necessarily $n=0$. When $m>0$ we compute $\phi(x-1)-\phi(y-1)$ by the subtraction algorithm, stopping at c_0; then $h=e^{-c_0/a_0}$. When $l>0$ and $m=0$ we compute $a_0\equiv 1/\phi(x)$ as in the subtraction (or addition) algorithm and then form $h=a_0g$. When $l=m=0$ we compute $h=g/f$ by ordinary fixed-point division.

It will be observed that some available precision may be lost in multiplication and division when the result is at level zero, just as in fixed-point multiplication and division. Often, of course, this loss is undesirable, and a natural way to avoid it is to represent level-zero numbers by the li forms of their reciprocals. This modification is discussed briefly in § 5 and more fully in [3].

2.5. Extensions. The subtraction/addition algorithm may be adapted to calculations involving several li numbers. Suppose, for example, that we wish to find z, where

$$\phi(x_1)\pm\phi(x_2)\pm\cdots\pm\phi(x_p)=\pm\phi(z), \tag{2.27}$$

and $x_1\geqq x_s\geqq 0$ for $s=2,3,\cdots,p$. We first compute the sequence $\{a_j\}$ defined by the first of (2.11) with $x=x_1$. Secondly, for each $s=2,3,\cdots,p$ we compute the sequence $\{b_j^{(s)}\}$ defined by

$$b_j^{(s)}=a_j\phi(x_s-j).$$

Thirdly, we generate the sequence $\{c_j\}$ defined by the third of (2.11). The recurrence relations for the a_j, $b_j^{(s)}$ and c_j are just as in (2.12), (2.13) and (2.14), with[1] $f=\mathrm{fr}[x_1]$ and $g=\mathrm{fr}[x_s]$, but with the starting value c_0 replaced by

$$\pm c_0=1\pm b_0^{(2)}\pm b_0^{(3)}\pm\cdots\pm b_0^{(p)},\qquad c_0\geqq 0. \tag{2.28}$$

It may happen that $c_{l-1}\geqq a_{l-1}$, as is always the case in the original addition algorithm. In this event we proceed as in (2.20), (2.21) and (2.22), except that it might also happen that $\ln h_l\geqq 1$; if so, we would compute the generalized logarithm of h_l and adjust the level accordingly.

[1] fr ≡ fractional part.

The advantage of this procedure is that we need only one sequence $\{a_j\}$ and, especially, only one sequence $\{c_j\}$. If parallel processing facilities are available, then the sequences $\{a_j\}$ and $\{b_j^{(s)}\}$ may all be computed simultaneously. In these circumstances the total computing time will be almost independent of p: only the comparatively rapid fixed-point summation (2.28) will be affected by the actual value of p.

Similar considerations apply to the computation of rational fractions of the form

$$\frac{\phi(x_1)\phi(x_2)\cdots\phi(x_p)}{\phi(y_1)\phi(y_2)\cdots\phi(y_q)}$$

and it is unnecessary to supply details.

3. Arithmetic operations: error analysis.

3.1. Effects of inherent errors. The object of this section is to ascertain how many guard digits need be retained in the implementation of the subtraction algorithm of § 2.2 and the addition algorithm of § 2.3 in order that the cumulative effect of the abbreviation errors[2] on the final result z be comparable with the unavoidable error in z stemming from abbreviations in the data x and y. In order to arrive at a solution to this problem we shall use linearized perturbation theory. In terms of this theory, the change δz in z caused by inherent errors δx in x and δy in y is given by

$$\delta z = \frac{\phi'(x)}{\phi'(z)}\delta x \mp \frac{\phi'(y)}{\phi'(z)}\delta y, \tag{3.1}$$

the upper sign being taken for subtraction and the lower sign for addition; compare (2.9) and (2.17).

The derivatives in (3.1) may be found with the aid of the chain rule

$$\phi'(x) = \phi(x)\phi(x-1)\cdots\phi(x-l+1), \tag{3.2}$$

with similar expressions for $\phi'(y)$ and $\phi'(z)$. *Here and elsewhere we follow the usual convention that empty products are assigned the value unity and empty sums are assigned the value zero.* Thus when $l=0$ we have $\phi'(x)=1$.

In the subtraction algorithm, as the value of y approaches that of x, the value of z tends to zero, which may cause the quotients $\phi'(x)/\phi'(z)$ and $\phi'(y)/\phi'(z)$ to become large. (This is of course, simply a manifestation of the inevitable loss of precision on subtracting two nearly equal numbers.) Furthermore, $\phi'(x) \geqq \phi'(y)$, since $x \geqq y$. In consequence, we shall seek to express the change in z stemming from the abbreviation errors as a multiple of $\phi'(x)/\phi'(z)$.

In the addition algorithm, we have[3]

$$1 \geqq \phi'(x)/\phi'(z) \geqq \{2(1+\ln 2)\}^{-1} = 0.2953\cdots. \tag{3.3}$$

In this case there is no need to include the factor $\phi'(x)/\phi'(z)$ in the bound for the change in z.

3.2. Working precisions; computed values. We first note that the members of the sequences $\{a_j\}$, $\{b_j\}$ and $\{c_j\}$ satisfy the following relations:

$$0 < a_0 < a_1 < \cdots < a_{l-1} \leqq 1, \tag{3.4}$$

$$0 \leqq b_0 \leqq b_1 \leqq \cdots \leqq b_{m-1} \leqq 1 \quad (m \geqq 1), \quad 0 \leqq b_0 \leqq 1 \quad (m=0), \tag{3.5}$$

$$0 \leqq c_0 \leqq c_1 \leqq \cdots \leqq c_{n-1} \leqq 1 \quad \text{(subtraction)}, \tag{3.6}$$

$$1 \leqq c_{l-1} \leqq c_{l-2} \leqq \cdots \leqq c_0 \leqq 2 \quad \text{(addition)}. \tag{3.7}$$

[2] That is, rounding or chopping errors [10].

[3] The lower bound in (3.3) is attained when $x = y = 2$.

(We also have $0 \leqq c_n \leqq 1$ when $n \leqq l-1$ in the case of subtraction.) Inequalities (3.4) follow from (2.12) and the fact that $e^{-1/t} < t$ when $t > 0$. To prove (3.5) in the case $m \geqq 1$, we observe from (2.11) and the assumption $x \geqq y$ that each b_j is bounded by unity. Furthermore,

$$\frac{b_{j-1}}{b_j} = \left(\frac{e^{t_1}}{t_1}\right) \Big/ \left(\frac{e^{t_2}}{t_2}\right), \quad t_1 = \phi(y-j), \quad t_2 = \phi(x-j),$$

and e^t/t is increasing when $t > 1$. The proof of (3.6) and (3.7) is similar.

Let r continue to denote the internal arithmetic base of the computer, and d_0 be the number of r-nary places to which f and g are stored. Bearing in mind (3.4) to (3.7), we assume that each member of the sequence $\{a_j\}$ is stored to d_1 r-nary places, and each member of the sequences $\{b_j\}$ and $\{c_j\}$ is stored to d r-nary places, where

$$d_0 < d \leqq d_1. \tag{3.8}$$

(The reason for the distinction in the number of places to which $\{a_j\}$, $\{b_j\}$ and $\{c_j\}$ are stored will become clear in § 3.6.) For brevity we write

$$\gamma_0 = r^{-d_0}, \quad \gamma = r^{-d}, \quad \gamma_1 = r^{-d_1}, \tag{3.9}$$

so that $\gamma_0 \geqq 2\gamma \geqq \gamma_1$.

In this section we distinguish computed values of quantities from their true values by adding overbars. In general, of course, $\bar{a}_j \neq a_j$, $\bar{b}_j \neq b_j$ and $\bar{c}_j \neq c_j$. Again, bearing in mind (3.4) to (3.7) we require the computed values to satisfy the following inequalities in order to avoid failure of the algorithms:

(i) $0 \leqq \bar{a}_j \leqq 1$, $0 \leqq \bar{b}_j \leqq 1$, $0 \leqq \bar{c}_j \leqq 1$ (subtraction), $1 \leqq \bar{c}_j \leqq 2$ (addition), $\bar{c}_0 = 1 - \bar{b}_0$ (subtraction), $\bar{c}_0 = 1 + \bar{b}_0$ (addition).

(ii) If $m = l$ and $\bar{b}_{l-1} = 1$, then $\bar{b}_j = 1$, for every j. Furthermore, in the case of subtraction $\bar{c}_0 = 0$, $\bar{n} = 0$ and $\bar{h} = 0$.

(iii) If (ii) does not apply, then $\bar{b}_j < 1$, for every j, $\bar{c}_0 > 0$, and $\bar{c}_j > 0$, $1 \leqq j \leqq n-1$. If, also, $\bar{a}_j = 0$ when $j \geqq 1$, then $\bar{a}_{j-1} = \bar{a}_{j-2} = \cdots = \bar{a}_0 = 0$ and $\bar{b}_0 = 0$.

3.3. Computation of a_j, b_j and c_j.

Since we are considering the effects of abbreviation errors, we may suppose in the remainder of this section that x and y are exact.

We assume that for given values of a_j, b_j and c_{j-1} the quantities $\exp(-1/a_j)$, $\exp\{-(1-b_j)/a_j\}$ and $1 + a_j \ln c_{j-1}$ in equations (2.12), (2.13) and (2.14) are generated correct to absolute precisions γ_1, γ and γ, respectively. Accordingly, if δa_j, δb_j and δc_j are the errors in the computed values of a_j, b_j and c_j, respectively, then on neglecting terms of the second and higher orders in the Taylor-series expansions, we have[4]

$$|\delta a_{j-1}| \leqq a_j^{-2} e^{-1/a_j} |\delta a_j| + \gamma_1, \tag{3.10}$$

$$|\delta b_{j-1}| \leqq e^{-(1-b_j)/a_j} \left(\frac{|\delta b_j|}{a_j} + \frac{1-b_j}{a_j^2} |\delta a_j| \right) + \gamma, \tag{3.11}$$

$$|\delta c_j| \leqq (a_j / c_{j-1}) |\delta c_{j-1}| + |\ln c_{j-1}|\, |\delta a_j| + \gamma. \tag{3.12}$$

We assume also that e^{-f} and $a_{m-1}e^g$ or $a_0 g$ in (2.12) and (2.13) are evaluated to absolute precisions γ_1 and γ, respectively. In consequence of this assumption, and

[4] In (3.10)-(3.12) it needs to be emphasized that the symbols a_j, b_j and c_{j-1} denote the true values. Thus in case (iii) of § 3.2 if $\bar{a}_j = 0$ with $j \geqq 1$, then $\bar{a}_{j-1}(=0)$ and $\bar{b}_{j-1}(=0)$ are obtained exactly from (2.12) and (2.13); hence (3.10) and (3.11) would also apply without the terms γ_1 and γ on their right-hand sides.

also § 3.2(i), the starting values for the recurrences (3.10), (3.11) and (3.12) are subject to the conditions

(3.13) $|\delta a_{l-1}| \leqq \gamma_1$; $\quad |\delta b_{m-1}| \leqq \gamma, \quad m \geqq 1$; $\quad |\delta b_0| \leqq \gamma, \quad m=0$; $\quad \delta c_0 = \delta b_0$.

Consider first the δa_j. The function $t^{-2}\, e^{-1/t}$ vanishes as $t \to 0$, rises to a maximum value of $4e^{-2} = 0.5413 \cdots$ at $t = \frac{1}{2}$, then decreases to zero as $t \to \infty$. Setting $j = l-2$ in (3.10) and using the first of (3.13), we obtain

$$|\delta a_{l-2}| \leqq (4e^{-2}+1)\gamma_1.$$

Next, because $a_{l-1} \leqq 1$ it follows from (2.12) that $a_{l-2} \leqq e^{-1}$ $(<\frac{1}{2})$. Hence

$$a_{l-2}^{-2}\, e^{-1/a_{l-2}} \leqq e^2 e^{-e}.$$

Setting $j = l-3$ in (3.10) and substituting by means of the last two results, we find that $|\delta a_{l-3}|$ is bounded by $\lambda \gamma_1$, where

$$\lambda = (4+e^2)\, e^{-e} + 1 = 1.7515 \cdots. \tag{3.14}$$

This argument may be continued, but it is easy to see (and also to prove by induction) that the bounds obtained for $|\delta a_{l-4}|$, $|\delta a_{l-5}|, \cdots$, are all smaller than that obtained for $|\delta a_{l-3}|$. Thus

$$|\delta a_j| \leqq \lambda \gamma_1, \qquad j = 0, 1, \cdots, l-1. \tag{3.15}$$

In the case of the δb_j suppose first that $m \geqq 1$. Using (3.15) and observing that the functions e^{-t} and te^{-t} are bounded by 1 and e^{-1}, respectively, when $t \geqq 0$, we see that (3.11) may be replaced by

$$|\delta b_{j-1}| \leqq \frac{1}{a_j}\left(|\delta b_j| + \frac{\lambda \gamma_1}{e}\right) + \gamma.$$

Applying this inequality recursively, beginning with the second of (3.13), we derive

$$|\delta b_j| \leqq \left(\frac{1}{a_{j+1}a_{j+2}\cdots a_{m-1}} + \frac{1}{a_{j+1}a_{j+2}\cdots a_{m-2}} + \cdots + \frac{1}{a_{j+1}}\right)\left(\gamma + \frac{\lambda\gamma_1}{e}\right) + \gamma.$$

Setting $j = 0$ and using the fact that $m \leqq l$, we obtain

$$|\delta b_0| \leqq \left(\frac{1}{a_1 a_2 \cdots a_{l-1}} + \frac{1}{a_1 a_2 \cdots a_{l-2}} + \cdots + \frac{1}{a_1}\right)\left(\gamma + \frac{\lambda\gamma_1}{e}\right) + \gamma.$$

This inequality may be rewritten in the form

$$|\delta b_0| \leqq \frac{\phi'(x)}{\phi(x)}\left\{1 + \frac{1}{\phi'(x-l+1)} + \frac{1}{\phi'(x-l+2)} + \cdots + \frac{1}{\phi'(x-2)} + \frac{1}{\phi'(x-1)}\,\frac{\gamma}{\gamma + e^{-1}\lambda\gamma_1}\right\} \times (\gamma + e^{-1}\lambda\gamma_1);$$

compare (2.11) and (3.2). Hence

$$|\delta b_0| < \{\phi'(x)/\phi(x)\}\rho(\gamma + e^{-1}\lambda\gamma_1), \tag{3.16}$$

where

$$\rho = 1 + \sum_{j=1}^{\infty} \frac{1}{\phi'(j)} = 2.3921 \cdots. \tag{3.17}$$

The result (3.16) has been derived on the assumption that $m \geqq 1$, but from the third of (3.13) it is true trivially when $m = 0$.

Lastly, consider the c_j. In the subtraction algorithm, if we set aside the trivial case $y=x$ (compare § 3.2 (ii)) we have $y \leqq x-\gamma_0$. From (2.11) it follows that $b_0 \leqq \phi(x-\gamma_0)/\phi(x)$. Since $l \neq 0$ we have $x \geqq 1$, and it is easily verified that with this condition the maximum value of $\phi(x-\gamma_0)/\phi(x)$ is $\exp(-\gamma_0)$. Therefore

$$c_0 \equiv 1-b_0 \geqq 1-e^{-\gamma_0} \geqq \gamma_0 - \tfrac{1}{2}\gamma_0^2 \geqq \gamma.$$

From (3.6) it now follows that $\gamma \leqq c_j \leqq 1$, $1 \leqq j \leqq n-1$. Hence in (2.14) and (3.12) we have

$$|\ln c_{j-1}| \leqq \ln(1/\gamma). \tag{3.18}$$

And obviously, from (3.7), this inequality also holds for the addition algorithm.

Substituting in (3.12) by means of (3.15) and (3.18) we arrive at

$$|\delta c_j| \leqq (a_j/c_{j-1})|\delta c_{j-1}| + \gamma_2, \tag{3.19}$$

where we have written (temporarily)

$$\gamma_2 = \gamma + \lambda\gamma_1 \ln(1/\gamma). \tag{3.20}$$

Recursive use of (3.19) and reference to (3.13) then yields the required bound in the form

$$|\delta c_j| \leqq \left(1+\frac{a_j}{c_{j-1}}+\frac{a_ja_{j-1}}{c_{j-1}c_{j-2}}+\cdots+\frac{a_ja_{j-1}\cdots a_2}{c_{j-1}c_{j-2}\cdots c_1}\right)\gamma_2+\frac{a_ja_{j-1}\cdots a_1}{c_{j-1}c_{j-2}\cdots c_0}|\delta b_0|, \tag{3.21}$$

valid for all admissible j (that is, for $0 \leqq j \leqq \min(n, l-1)$).

3.4. Error bounds for subtraction. On multiplying each side of the inequality (3.21) by $c_0c_1\cdots c_{j-1}a_{j+1}a_{j+2}\cdots a_{l-1}$ and using the fact that each c_j is bounded by unity, we obtain

$$\begin{aligned} &c_0c_1\cdots c_{j-1}a_{j+1}a_{j+2}\cdots a_{l-1}|\delta c_j| \\ &\quad \leqq (a_{j+1}a_{j+2}\cdots a_{l-1}+a_ja_{j+1}\cdots a_{l-1}+\cdots+a_2a_3\cdots a_{l-1})\gamma_2 \\ &\quad + a_1a_2\cdots a_{l-1}|\delta b_0|. \end{aligned}$$

From (2.11) and (3.2) it is seen that the left-hand side of this inequality equals $\phi'(z)|\delta c_j|/\{\phi'(z-j)\phi'(x)a_j\}$. Next, since $j \leqq l-1$ the right-hand side is bounded by

$$(1+a_{l-1}+a_{l-2}a_{l-1}+\cdots+a_2a_3\cdots a_{l-1})\gamma_2+a_1a_2\cdots a_{l-1}|\delta b_0|, \tag{3.22}$$

and therefore by $\rho\gamma_2+\{\phi(x)/\phi'(x)\}|\delta b_0|$, where ρ is again defined by (3.17). Thus we have

$$\frac{|\delta c_j|}{a_j} \leqq \frac{\phi'(z-j)}{\phi'(z)}\{\phi'(x)\rho\gamma_2+\phi(x)|\delta b_0|\},$$

or, on substituting by means of (3.16),

$$\frac{|\delta c_j|}{a_j} < \frac{\phi'(z-j)}{\phi'(z)}\phi'(x)\rho(\gamma_2+\gamma+e^{-1}\lambda\gamma_1). \tag{3.23}$$

Suppose first that $n \leqq l-1$. Then from (2.15) $h = c_n/a_n$. Assuming that this quotient is computed to absolute precision γ, we obtain

$$|\delta z| = |\delta h| \leqq (|\delta c_n|/a_n)+(c_n/a_n^2)|\delta a_n|+\gamma. \tag{3.24}$$

For the first term on the right-hand side we set $j=n$ in (3.23) and use the identity $\phi'(z-n)=1$. For the second term,

$$\frac{c_n}{a_n^2}|\delta a_n| < \frac{|\delta a_n|}{a_n} \leqq \phi(x-n)\lambda\gamma_1 \leqq \frac{\phi'(x)}{\phi'(z)}\lambda\gamma_1,$$

where we have used (2.11), (3.15) and the fact that

$$\frac{\phi'(z)}{\phi'(x)}\phi(x-n)$$
$$=\frac{\phi(z)\phi(z-1)\cdots\phi(z-n+1)}{\phi(x)\phi(x-1)\cdots\phi(x-n+1)}\frac{1}{\phi(x-n-1)\phi(x-n-2)\cdots\phi(x-l+1)}\leqq 1.$$

Substituting in (3.24) by means of these results and (3.20), we arrive at the required bound in the form

$$|\delta z|\leqq\frac{\phi'(x)}{\phi'(z)}\left\{(2\rho+1)\gamma+\left(e^{-1}\rho+1+\rho\ln\frac{1}{\gamma}\right)\lambda\gamma_1\right\}. \tag{3.25}$$

Alternatively suppose that $n=l$. Assuming that the expression (2.16) for h is computed to absolute precision γ, we have

$$|\delta z|=|\delta h|\leqq(|\delta c_{l-1}|/c_{l-1})+\gamma.$$

Then applying (3.23), with $j=l-1$, we see that (3.25) holds a fortiori in this case.

3.5. Error bounds for addition. In the addition algorithm the c_j are bounded below by unity. Hence from (3.21), with $j=l-1$, we see that $|\delta c_{l-1}|$ is bounded by the quantity (3.22), and hence

$$|\delta c_{l-1}|\leqq\rho(\gamma_2+\gamma+e^{-1}\lambda\gamma_1).$$

In (2.20) and (2.22) assume that $f+\ln c_{l-1}$ and $\ln h_l$ are evaluated to absolute precision γ. Then

$$|\delta h|=|\delta h_l|\leqq(|\delta c_{l-1}|/c_{l-1})+\gamma\leqq|\delta c_{l-1}|+\gamma \quad \text{if } n=l,$$

or

$$|\delta h|\leqq(|\delta h_l|/h_l)+\gamma\leqq|\delta h_l|+\gamma\leqq|\delta c_{l-1}|+2\gamma \quad \text{if } n=l+1.$$

On combining the above results and substituting for γ_2 by means of (3.20), we find that in all cases δz satisfies

$$|\delta z|=|\delta h|\leqq(2\rho+2)\gamma+\left(e^{-1}\rho+\rho\ln\frac{1}{\gamma}\right)\lambda\gamma_1. \tag{3.26}$$

3.6. Guard digits. At this stage we need to specify the relation between γ and γ_1; compare § 3.2. Inspection of (3.25) and (3.26) suggests that one advantageous way would be to restrict γ_1 to be bounded by a constant multiple of $\gamma/\ln(1/\gamma)$. Suppose, for example, we choose this multiple to be unity, that is, suppose that we require γ_1 to satisfy

$$\gamma_1\leqq\gamma/\ln(1/\gamma). \tag{3.27}$$

On substituting by means of this bound and replacing λ, ρ and e by their numerical values (see (3.14), (3.17)), we find that the results (3.25) and (3.26) reduce to

$$|\delta z|\leqq\{\phi'(x)/\phi'(z)\}[9.98+3.30\{\ln(1/\gamma)\}^{-1}]\gamma \quad \text{(subtraction)}, \tag{3.28}$$

$$|\delta z|\leqq[10.98+1.55\{\ln(1/\gamma)\}^{-1}]\gamma \quad \text{(addition)}. \tag{3.29}$$

In practice, the contributions of the terms involving $\{\ln(1/\gamma)\}^{-1}$ here will be quite small. With the modest assumption that we are working with at least 8 bits, so that $\gamma\leqq 2^{-8}$, we have

$$|\delta z|\leqq\{\phi'(x)/\phi'(z)\}\times 10.58\gamma \text{ (subtraction)}, \qquad |\delta z|\leqq 11.26\gamma \text{ (addition)}. \tag{3.30}$$

The results just obtained show that if f and g are stored to d_0 digits in base r, then we should retain $\log_r 12$ guard digits in the sequences $\{b_j\}$ and $\{c_j\}$ and $\log_r 12 + \log_r \{(d_0 + \log_r 12) \ln r\}$ guard digits in the sequence $\{a_j\}$ to render the cumulative effects of the abbreviation errors comparable to the propagated effects of the inherent errors. Sample values of the smallest integers that are not less than $\log_r 12$ and $\log_r 12 + \log_r \{(d_0 + \log_r 12) \ln r\}$ are 4 and 9 for $r = 2$, $d_0 = 32$; 4 and 10 for $r = 2$, $d_0 = 64$; 2 and 8 for $r = 10$, $d_0 = 1{,}000{,}000$. However, in designing a hardware processor for implementing the algorithms, extra guard digits should be included to ensure that the effect of the abbreviation errors is quite negligible compared with that of the inherent errors—apart from, of course, the final rounding or chopping of the guard digits themselves. This precaution is analogous to the use of guard digits in the accumulator register for ordinary floating-point arithmetic operations.

4. Arithmetic operations: numerical examples.

4.1. Example 1. We compute $F - G$ and $F + G$, where

$$F = \phi(3.4546), \qquad G = \phi(1.9999).$$

In the notation of § 2.1

$$l = 3, \quad f = 0.4546, \quad m = 1, \quad g = 0.9999.$$

We shall compute the sequences $\{b_j\}$ and $\{c_j\}$ to eight decimal places, using the rounding mode of abbreviation. In the notation of § 3.2 we have $r = 10$, $d = 8$.

$$\gamma = 10^{-8}, \qquad \gamma / \ln(1/\gamma) = 0.5428 \cdots \times 10^{-9}.$$

Accordingly, it is ample to compute the sequence $\{a_j\}$ to ten decimal places.

For $F - G$ the computations in the subtraction algorithm proceed as follows. From (2.12) we find that[5]

$$a_2 = 0.63470\,17979, \quad a_1 = 0.20689\,51699, \quad a_0 = 0.00795\,96869.$$

From the second of (2.13) we obtain $b_0 = 0.02163\,451$. Lastly, from (2.14) we obtain

$$c_0 = 0.97836\,549, \quad c_1 = 0.99547\,480, \quad c_2 = 0.99712\,133.$$

Since $c_2 > a_2$, we apply (2.16). Thus $n = 3$, $h = 0.45171\,718$, and the required result is given by

$$F - G = \phi(\mathbf{3.45171\,718}).$$

In the addition algorithm, the sequences $\{a_j\}$ and $\{b_j\}$ are the same but the starting value for $\{c_j\}$ is given by $c_0 = 1 + b_0$. We find that

$$c_0 = 1.02163\,451, \quad c_1 = 1.00442\,834, \quad c_2 = 1.00280\,447.$$

From (2.20) we obtain $h_3 = 0.45740\,054$. Since $h_3 < 1$ equations (2.21) apply. Therefore

$$F + G = \phi(\mathbf{3.45740\,054}).$$

In this example it is possible to check the results by "unwinding," that is, by computing F and G in fixed-point or floating-point form. Repeated exponentiations yield

$$F = 125.63308\,252 \cdots, \qquad G = 2.71801\,001 \cdots.$$

[5] For simplicity we omit overbars from symbols in this section; compare § 3.2.

Hence

$$F-G=122.91507\,251\cdots,\qquad F+G=128.35109\,254\cdots,$$

$$\ln^{(3)}(F-G)=0.45171\,71752\cdots,\qquad \ln^{(3)}(F+G)=0.45740\,05475\cdots,$$

in adequate agreement with the results given above.

4.2. Example 2. We compute $F-G$ in the cases (i) $F=\phi(3.70001)$, $G=\phi(3.70000)$; (ii) $F=\phi(4.70001)$, $G=\phi(4.70000)$; (iii) $F=\phi(5.70001)$, $G=\phi(5.70000)$.

We use the same working precisions as in Example 1. In all three cases $f=0.70001$, $g=0.70000$. The computations are as follows.

(i) Here $l=m=3$, $a_2=0.49658\,03380$, $a_1=0.13348\,41086$, $a_0=0.00055\,77897$; $b_2=0.99999\,000$, $b_1=0.99997\,986$, $b_0=0.99984\,913$; $c_0=0.00015\,087$. Since $c_0<a_0$ we apply (2.15). Thus $n=0$, $h=0.27047\,828$ and

$$F-G=\phi(\mathbf{0.27047\,828}).$$

(ii) Here $l=m=4$, $a_3=0.49658\,03380$, $a_2=0.13348\,41086$, $a_1=0.00055\,77897$, $a_0=0.00000\,00000$; $b_3=0.99999\,000$, $b_2=0.99997\,986$, $b_1=0.99984\,913$, $b_0=0.76301\,447$; $c_0=0.23698\,553$, $c_1=0.99919\,692$, $c_2=0.99989\,276$, $c_3=0.99994\,674$. Since $c_3>a_3$ we apply (2.16). Thus $n=4$, $h=0.69995\,674$ and

$$F-G=\phi(\mathbf{4.69995\,674}).$$

(iii) Here $l=m=5$. On computing the a_j we find that $a_1=0.00000\,00000$. Also, $b_4=0.99999\,000$. Hence § 3.2 (iii) applies, and we have immediately $b_0=0$, $c_0=c_1=c_2=c_3=c_4=1$, $n=5$, $h=f$ and

$$F-G=\phi(\mathbf{5.70001\,000}).$$

The results in Cases (i) and (ii) may be checked by unwinding as in Example 1. In Case (i) we find that

$$F=1792.79022\,880\cdots,\quad G=1792.51978\,870\cdots,\quad F-G=0.27044\,010\cdots.$$

In Case (ii),

$$F=3.97103\,375\cdots\times10^{778},\qquad G=3.03007\,191\cdots\times10^{778},$$

$$F-G=0.94096\,184\cdots\times10^{778},\qquad \ln^{(4)}(F-G)=0.69995\,6737\cdots.$$

The only significant discrepancy occurs in Case (i). It is accounted for by the sensitivity of the answer z to the data x and y, and therefore, also, to abbreviation errors introduced during the computation. With the aid of (3.2), we find the following values for the coefficient of δx in (3.1):

$$\phi'(x)/\phi'(z)=2.704\cdots\times10^4\quad\text{in Case (i)},$$

$$\phi'(x)/\phi'(z)=4.224\cdots\quad\text{in Case (ii)}.$$

Therefore the actual discrepancies lie well within the projections of the error analysis of § 3.

In Case (iii) F and G are too large to be handled by unwinding in ordinary floating-point form.

4.3. Remark. In each example we computed the sequences $\{a_j\}$, $\{b_j\}$ and $\{c_j\}$ in order. In a software or hardware implementation, execution time could be saved by computing the a_j and b_j in parallel; compare § 2.5.

5. The symmetric level-index system. As we have noted in § 1, the li system functions as a fixed-point system for numbers in the interval $[0, 1]$: this is in the sense that the set of machine-representable numbers in this interval is equispaced. However, many algorithms entail the formation of products of very large and very small numbers, and if we wish to preserve relative precision in these cases then we need a set of machine-representable numbers that is increasingly dense as the origin is approached. The floating-point system achieves this by introduction of negative exponents. A corresponding modification of the li system is the *symmetric level-index* (sli) system, one notation for which is as follows.

In place of the mapping $F = \phi(x)$ and its inverse $x = \psi(F)$, we use $F = \Phi(x)$ and $x = \Psi(F)$, where

(5.1) $$\Phi(x) = 1/\phi(1-x), \quad x<0; \qquad \Phi(x) = \phi(1+x), \quad x \geqq 0;$$

(5.2) $$\Psi(F) = 1-\psi(1/F), \quad 0<F<1; \qquad \Psi(F) = \psi(F)-1, \quad F \geqq 1.$$

Since $\phi(x)$ is an increasing C^1 function on the interval $[0, \infty)$ and $\phi(1) = \phi'(1) = 1$, it follows that $\Phi(x)$ is an increasing C^1 function on the interval $(-\infty, \infty)$. Similarly $\Psi(F)$ is an increasing C^1 function on the interval $(0, \infty)$.

Clearly arithmetic operations in the sli system can be constructed either by direct combination, or by modification, of the arithmetic operations given in § 2 for the li system. These procedures are examined in [3] and will not be discussed further here.

6. Applications. In this section we show, by means of two examples, that by eliminating overflow and underflow the li and sli systems not only make computing algorithms more robust, but may simplify their construction considerably.[6] These simplifications will save human effort in devising and programming algorithms, and they may recover some of the extra machine time that is needed to implement li or sli arithmetic.

6.1. Binomial probability distribution. The first example is the evaluation of the binomial probability distribution defined by

$$I(n, r, p) = \sum_{s=0}^{r} \binom{n}{s} p^s q^{n-s},$$

where $q = 1-p$, $0 \leqq p \leqq 1$ and $0 \leqq r \leqq n$. This distribution has been tabulated by the National Bureau of Standards [9] for $p = 0.00(0.01)0.50$ and n up to 49.[7] In his well-known book on floating-point computation Sterbenz [12] devotes over six pages to the difficulties caused by overflow and underflow in programming the computation of $I(n, r, p)$. The problem has also been considered by Matsui and Iri [8]. The numerator $n(n-1)\cdots(n-s+1)$ and denominator $s!$ of the binomial coefficient overflow, and the powers p^s and q^{n-s} underflow, for quite moderate values of n, s and p. Of course, the final sum $I(n, r, p)$ always lies in the interval $[0, 1]$.

Sterbenz shows how to overcome these difficulties by careful rescaling and includes two programs for implementing his modified algorithms. In contrast, virtually any algorithm can be executed in sli arithmetic. This includes, for example, the following

[6] That overflow and underflow problems may be more common than is generally assumed has been demonstrated recently by Feldstein and Turner [4].

[7] Actually reference [9] tabulates $\sum_{s=r}^{n} \binom{n}{s} p^s q^{n-s}$, that is, $1 - I(n, r-1, p)$. Individual terms of the sum are also tabulated.

naïve algorithm:

```
$q=1-p, u=1, v=1, w=1, x=1$
for $s=1$ through $n$
    $x \leftarrow qx$
$y=x$
$z=y$
for $s=1$ through $r$
    $u \leftarrow (n+1-s)u$
    $v \leftarrow sv$
    $w \leftarrow pw$
    $x \leftarrow x/q$
    $y \leftarrow (u/v)wx$
    $z \leftarrow y+z$
$I(n, r, p)=z.$
```

Thus at the close of the computations we have

$$u=n(n-1)\cdots(n-r+1), \quad v=r!, \quad w=p^r, \quad x=q^{n-r},$$

$$y=\binom{n}{r}p^rq^{n-r}, \qquad z=I(n,r,p).$$

The algorithm just given has been programmed and executed in sli arithmetic. Indices were rounded to 27 binary places at the conclusion of each arithmetic operation: this corresponds to $8\frac{1}{2}$ decimal places, approximately. Results for $p=0.1$, $n=2000$, $r=200$ (the case considered by Sterbenz) are as follows:

$$u=[3.68842667], \quad v=[3.64769156], \quad w=[-3.59530170],$$

$$x=[-3.50519488], \quad y=[-2.22893471], \quad z=[-0.65619393].$$

The square brackets signify that the numbers within are the sli images of the corresponding variables; for example,

$$u=\exp\exp\exp\exp(0.68842667), \qquad w=1/\exp\exp\exp\exp(0.59530170).$$

(This use of square brackets differs from that of § 2.1.) Approximate values of the corresponding floating-point (flp) numbers are given by

$$u=0.54144\times10^{656}, \quad v=0.78879\times10^{375}, \quad w=0.99996\times10^{-200},$$

$$x=0.433058\times10^{-82}, \quad y=0.29724761\times10^{-1}, \quad z=0.51882226.$$

Accordingly, only the final results y and z are representable in single-precision IEEE standard floating-point arithmetic [6]. The intermediate quantities u and v would overflow, and w and x would underflow. (In fact, u and v also overflow in the double-precision version of the IEEE standard.)

It needs to be observed that although sli arithmetic cannot fail because of overflow or underflow, there is some loss of accuracy because of the naïveté of the algorithm. On recomputation with double precision in the sli images, the following results were obtained:

$$u=[3.68842666], \quad v=[3.64769155], \quad w=[-3.59530169],$$

$$x=[-3.50519492], \quad y=[-2.22894905], \quad z=[-0.65619750].$$

The loss of one decimal place in the original value of x stems from the large number of recurrence steps: 2200. The more severe loss of $3\frac{1}{2}$ decimal places in the value of y and 3 decimal places in the value of z stem from cancellation in the computation of y from the formula $y=(u/v)wx$. In effect, a very large number is being divided by another very large, and almost equal, number.[8] It is important to realize, however, that if higher accuracy is needed, then it could be achieved quite simply by executing the sli program in double, or higher, precision: *there is no need to recode the algorithm.* In contrast, when flp arithmetic fails, as it does in the present example, the algorithm being used may need to be modified extensively, or even abandoned.

6.2. Euclidean norm. Our second example is the evaluation of the Euclidean norm of a vector:

$$\|x\|=(x_1^2+x_2^2+\cdots+x_n^2)^{1/2}.$$

The direct algorithm proceeds as follows:

```
$y=0$
for $i=1$ through $n$
    $y \leftarrow y+x_i^2$
$\|x\|=y^{1/2}$.
```

When executed in floating-point arithmetic the algorithm fails when any of the squares x_i^2, or their sum, overflows or underflows, even though the required answer $\|x\|$ may lie between the smallest and largest machine-representable numbers. To avoid these failures careful rescalings are needed, and Blue [1] has devised a fairly robust, but rather complicated, algorithm on these lines. The kernel of Blue's algorithm consists of 19 lines of FORTRAN, including 6 machine-dependent constants.

With sli arithmetic there is no need for Blue's algorithm. The direct algorithm will always yield accurate answers with no possibility of overflow or underflow.

A numerical illustration is supplied in Tables 1 and 2. In Table 1 the x_i, their squares x_i^2 and partial sums $\sum x_i^2$ are represented as normalized decimal floating-point numbers with five decimal places in the mantissae; for example

$$x_1=0.51515\times 10^{19}, \qquad x_6=0.22261\times 10^{-22}.$$

TABLE 1
Euclidean norm in flp.

i	x_i	x_i^2	$\sum x_i^2$
1	0.51515 +19	0.26538 +38	0.26538 +38
2	0.31416 +17	0.98697 +33	0.26539 +38
3	0.26658 +2	0.71065 +3	0.26539 +38
4	0.14142 +23	0.20000 +45	0.20000 +45
5	0.98765 +22	0.97545 +44	0.29755 +45
6	0.22261 −22	0.49555 −45	0.29755 +45
7	0.12345 +23	0.15240 +45	0.44995 +45
8	0.88088 +11	0.77595 +22	0.44995 +45

$\|x\|=0.21212\ \ +23.$

[8] The same kind of cancellation occurs, but in more subtle manner, in flp arithmetic; see [7]. This reference also contains a further discussion of the present example.

TABLE 2
Euclidean norm in sli.

i	x_i	x_i^2	$\sum x_i^2$
1	[3.2816147]	[3.4016766]	[3.4016766]
2	[3.2555873]	[3.3823005]	[3.4016767]
3	[2.1729316]	[2.6322989]	[3.4016767]
4	[3.3141689]	[3.4262379]	[3.4262379]
5	[3.3128548]	[3.4252394]	[3.4267863]
6	[−3.3183110]	[−3.4293888]	[3.4267863]
7	[3.3136730]	[3.4258610]	[3.4273541]
8	[3.1583057]	[3.3119641]	[3.4273541]

$\|x\| = [3.3156371]$.

In forming the squares and their partial sums each mantissa has been rounded to five decimal places immediately after computation. For computers that adhere to the IEEE floating-point standard, in single-precision the value of x_i^2 overflows for $i = 4, 5, 7$ and underflows for $i = 6$; furthermore, the partial sums overflow for $i \geqq 4$.

In Table 2 the x_i, x_i^2 and $\sum x_i^2$ are represented by their sli images to seven decimal places.[9] In forming the squares and their partial sums each sli entry has been rounded to seven decimal places immediately after computation. For comparison of the final results we have

$$\Phi(3.3156371) = 0.21211 \cdots \times 10^{23}.$$

This example would run successfully with Blue's program. However, if we were to increase the flp exponents of any or all the x_i's by 500, say, then this would no longer be the case. This is because these x_i's would be too large to be accommodated even in double-precision IEEE standard arithmetic.

7. Summary and conclusions.

7. Summary and conclusions. In § 2 we constructed algorithms for the operations of subtraction, addition, multiplication and division in the level-index number system. In § 3 we showed, by linearized perturbation analysis, that these algorithms can be implemented in fixed-point arithmetic. In § 4 we illustrated the algorithms by numerical examples in base 10. In § 5 we described briefly the symmetric level-index number system. In § 6 we showed how the use of the li and sli systems can simplify programming by means of two examples: the binomial probability distribution and the Euclidean norm.

The next stages in the development of the li and sli systems will include the following. (i) Fast processes are needed for evaluating the requisite exponentials and logarithms. Given numbers a, b and c in fixed-point form, we need to be able to form e^{-a}, $e^{-1/a}$, $e^{-(1-b)/a}$ and ln c, for example, to fixed-point precision. On microcomputers or programmable calculators algorithms of the CORDIC type originated by J. E. Volder might be useful for this purpose; see, for example, [11], [14]. For mainframe computers, however, digital parallel algorithms of the kind used for fixed- and floating-point arithmetic operations will probably be more effective; see, for example, [5, § 3.2]. Considerable progress on these lines has been made by P. R. Turner [13]. (ii) The

[9] As in the previous example, it would take us too far afield to give a complete error analysis. As one might expect, however, an flp number with l digits in its exponent and d decimals in its mantissa, and an li or sli number with $l+d$ decimals in its fractional part are roughly to comparable precision [7].

linearized perturbation theory of § 3 needs to be strengthened by analysis that will yield strict (as well as realistic) error bounds.

Acknowledgments. The authors are pleased to acknowledge valuable criticisms of earlier versions of this paper by A. Feldstein, D. W. Lozier, P. R. Turner and the referees. Dr. Lozier also undertook the task of programming li and sli arithmetic and the algorithm of § 6.1.

REFERENCES

[1] J. L. BLUE, *A portable Fortran program to find the Euclidean norm of a vector*, ACM Trans. Math. Software, 4 (1978), pp. 15-23.

[2] C. W. CLENSHAW AND F. W. J. OLVER, *Beyond floating point*, J. Assoc. Comput. Mach., 31 (1984), pp. 319-328.

[3] C. W. CLENSHAW AND P. R. TURNER, *The symmetric level-index system*, manuscript.

[4] A. FELDSTEIN AND P. R. TURNER, *Overflow, underflow, and severe loss of significance in floating-point addition and subtraction*, IMA J. Numer. Anal., 6 (1986), pp. 241-251.

[5] K. HWANG AND F. A. BRIGGS, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[6] IEEE STANDARD 754, *Binary floating-point arithmetic*, The Institute of Electrical and Electronics Engineers, New York, 1985.

[7] D. W. LOZIER AND F. W. J. OLVER, *Closure and precision in computer arithmetic*, manuscript.

[8] S. MATSUI AND M. IRI, *An overflow/underflow-free floating-point representation of numbers*, J. Inform. Process., 4 (1981), pp. 123-133.

[9] NATIONAL BUREAU OF STANDARDS, *Tables of the Binomial Probability Distribution*, Applied Mathematics Series No. 6, U.S. Government Printing Office, Washington, D.C., 1950.

[10] F. W. J. OLVER, *Further developments of rp and ap error analysis*, IMA J. Numer. Anal., 2 (1982), pp. 249-274.

[11] C. W. SCHELIN, *Calculator function approximation*, Amer. Math. Monthly, 90 (1983), pp. 317-325.

[12] P. H. STERBENZ, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[13] P. R. TURNER, *Towards a fast implementation of level-index arithmetic*, manuscript.

[14] J. S. WALTHER, *A unified algorithm for elementary functions*, AFIPS Conference Proc., 38 (1971), pp. 379-385.