Environment Parameters and Basic Functions for Floating-Point Computation

W. S. BROWN AND S I. FELDMAN
Bell Laboratories

This paper presents a language-independent proposal for environment parameters and basic functions for floating-point computation, and suggests a specific representation in terms of generic functions for Fortran 77 The environment parameters were introduced in 1967 by Forsythe and Moler, who attributed the essentials of their theory to Wilkinson These parameters are also used in the PORT mathematical subroutine library, with precise definitions in terms of a more recent model of floating-point computation, and a similar set has been proposed by the IFIP Working Group on Numerical Software. Three of the basic functions are taken from another proposal by this group, but redefined in terms of the parameters and the model to provide a firm theoretical foundation The other three basic functions can be expressed in terms of these, but we feel they should be provided separately for convenience

The proposed parameters specify the base of the floating-point number system, the maximum precision consistent with the representational and operational accuracy of the machine, and the maximum exponent range within which one can compute without fear of overflow, underflow, or unexpected loss of accuracy The proposed functions separate a floating-point number into its constituents, reconstruct a number from those constituents, scale a number by a power of the base, and determine the absolute or relative spacing (in the sense of the model) between numbers in the vicinity of a given number.

Key Words and Phrases: environment parameters, floating-point arithmetic, software portability
CR Categories: 4 22, 5.11, 6 32

1. INTRODUCTION

In programming languages that offer floating-point computation, there is a widely recognized need for *environment parameters* to describe the vital floating-point characteristics of the host computer, and for *basic functions* to analyze, synthesize, and scale floating-point numbers and to provide sharp measures of roundoff error. This paper presents a language-independent proposal for such parameters and functions, and suggests a specific representation in terms of generic functions for Fortran 77.

The inclusion of environment parameters and basic functions in programming languages was suggested by Naur [6] in 1964. For floating-point computation, however, he proposed only the symmetric range parameter (see Section 2) and the absolute-spacing function (see Section 4).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Authors' address. Bell Laboratories, Murray Hill, N.J 07974.
© 1980 ACM 0098-3500/80/1200-0510 $00.75

The environment parameters that we propose have been in common use since the early days of automatic computing. Their sufficiency for the analysis of machine-independent algorithms is implicit in Wilkinson's theory [9] and is made explicit in the restatement of that theory by Forsythe and Moler [4]. These parameters are also used in the PORT mathematical subroutine library [5], and a similar set has been proposed [3] by the IFIP Working Group on Numerical Software (WG 2.5), though this group does not suggest including the parameters in programming languages. In Wilkinson's theory it is assumed that all arithmetic operations yield correctly rounded or correctly chopped results in the same floating-point number system. By contrast, WG 2.5 and the PORT group have broadened their definitions to include many computers that are less simple or less accurate than this assumption implies. However, the WG 2.5 definitions depend solely on the static number representation of the host computer, while the PORT definitions, which we adopt in this paper, use Brown's model of floating-point computation [1] to reflect its dynamic behavior as well.

While the proposed basic functions appear to provide all the needed capabilities, there is little precedent for the choice of any particular set. Three of our functions are taken from another WG 2.5 proposal [7] but are redefined in terms of our parameters and Brown's model to provide a firm theoretical foundation. The other three basic functions can be expressed in terms of these, but we feel they should be provided separately for convenience.

Using the proposed parameters and functions, one can write portable and robust codes that deal intimately with the floating-point representation. Subject to underflow and overflow constraints, one can scale a number by a power of the floating-point radix inexpensively and without error. Similarly, one can take an approximate logarithm of a floating-point number very cheaply by extracting the exponent field, and one can readily implement algorithms (e.g., those for the logarithm and nth root functions) that operate separately on the exponent and fraction-part. The convergence of iterations is extremely important in numerical computation. While one often wants to relate the termination conditions to the accuracy of the host computer, it is essential to avoid demanding more accuracy than the computer can provide. Although a termination criterion can be formulated in terms of the environment parameters alone, it may be desirable to use the roundoff-measuring functions for finer control, especially when the floating-point radix is greater than 2.

We view it as essential to provide mechanisms for accomplishing these goals in any language that is used for scientific computing. Ideally, to facilitate translations from one language to another, these mechanisms ought to be provided in a similar manner in all such languages. Therefore, we present our proposal in a language-independent form, before suggesting a specific representation for Fortran. (We describe differences between our proposal and the WG 2.5 proposals in parenthetical comments.)

2. ENVIRONMENT PARAMETERS

In this section we present the environment parameters of Brown's model and review other key properties. First, for any given real number $x \neq 0$, we define the

(integer) *exponent*, e, and the (real) *fraction-part*, f, relative to a specified (integer) *base* $b \geq 2$, so that

$$x = fb^e, \qquad b^{-1} \leq |f| < 1, \tag{1}$$

and therefore

$$b^{e-1} \leq |x| < b^e. \tag{2}$$

Next, we introduce the parameters of the model—four basic integer parameters and three derived real parameters, all constants for a given floating-point number system. If a computer supports two or more such systems (e.g., single- and double-precision), then each has its own parameters. The basic parameters, all integers, are

(1) the *base*, $b \geq 2$;
(2) the *precision*, $p \geq 2$;
(3) the *minimum exponent*, $e_{\min} < 0$;
(4) the *maximum exponent*, $e_{\max} > 0$.

These define a system of *model numbers* consisting of zero and all numbers of the form

$$x = fb^e \tag{3}$$

where

$$f = \pm(f_1 b^{-1} + \cdots + f_p b^{-p}), \quad f_1 = 1, \ldots, b-1, \quad f_2, \ldots, f_p = 0, \ldots, b-1, \tag{4}$$

and

$$e_{\min} \leq e \leq e_{\max}. \tag{5}$$

The parameters must be chosen so that these model numbers are exactly representable in the machine, and so that operations on them behave according to the following simple and desirable rules, which are restated as axioms by Brown [1].

(1) The result of a *basic arithmetic operation* (addition, subtraction, multiplication, negation, or division by a power of the base) on model numbers must be no less accurate than the result that would be obtained by chopped arithmetic in the model system. (Chopping away from zero, as in the case of negative results in a two's complement system, is permitted.)
(2) The result of a division when both operands are model numbers but the divisor is not a power of the base may be less accurate than the model-chopped result by up to one unit in the last (pth) place.
(3) The result of a comparison of model numbers must be exact.
(4) The result of an arithmetic operation (or comparison) on operands between model numbers must be within the interval (or set) of permitted results of the same operation on the neighboring model numbers.

Note that if x is a model number, then so is $-x$, but $1/x$ need not be either in range (see (5)) or exactly representable (see (4)). On many computers $e_{max} + e_{min} \approx 0$, but on machines with the implicit radix point at the right, it is likely that $e_{max} + e_{min} \approx 2p$.

We do not assume that the computer actually uses a normalized sign-magnitude representation for floating-point numbers. In fact, the details of the hardware representation are of no concern to us. What we require is simply that the model numbers be possible values for program variables and that arithmetic operations be at least accurate enough to satisfy the axioms.

Usually the base is chosen to be whatever the manufacturer says it is, and the remaining parameters are chosen to make the range and precision as large as possible. In some cases the resulting model numbers coincide exactly with the machine numbers. However, anomalies in the behavior of a computer may require that the parameters be penalized to reduce the purported range or precision. For example, on some computers multiplication by 1.0 causes the last b-digit of the multiplicand to be replaced by 0. On such a computer a precision penalty of 1 is necessary to ensure that model numbers are not affected by the anomaly. Each penalty reduces the size of the set of model numbers, thus creating machine numbers that are not model numbers and need not behave quite so well in computation. Any anomalies that cannot be accommodated by modest penalties are usually recognized by the manufacturer as design errors and repaired in due course.

Since the model numbers with a given exponent e are equally spaced on an absolute scale, the relative spacing decreases as the magnitude of the fraction-part f increases. For error analysis the maximum relative spacing

$$\epsilon = b^{1-p} \tag{6}$$

is of critical importance. Also of interest throughout this paper are the smallest positive model number,

$$\sigma = b^{e_{min}-1}, \tag{7}$$

and the largest model number

$$\lambda = b^{e_{max}}(1 - b^{-p}). \tag{8}$$

From a theoretical viewpoint the integer parameters b, p, e_{min}, and e_{max} are fundamental; in practice, a programmer is more likely to want the real parameters ϵ, σ, and λ.

To use these environment parameters effectively in the development, analysis, and documentation of mathematical software, it is obviously crucial to determine their values for each target machine, to make them conveniently available to software developers from relevant programming languages, and to publish them in human-readable form for the benefit of the ultimate users. Unfortunately, there are many possible anomalies that may affect the parameters, and some of them are quite subtle.

While it may not be easy to determine the correct values of the environment parameters, it is necessary, and N. L. Schryer has developed a test program [8] to aid in the task. This program performs arithmetic operations and comparisons

on carefully chosen pairs of operands and tests whether the results conform to the axioms of the model. Obviously, such a test cannot be exhaustive because there are far too many pairs of numbers in any useful floating-point system to test them all. Nevertheless, Schryer's program exercises the hardware quite thoroughly.

(The WG 2.5 proposal includes b, p, ϵ, σ, and λ as independently defined quantities with different names. It omits $e_{\min}$ and $e_{\max}$ but introduces the *symmetric range*, $\tau \sim \min(\sigma^{-1}, \lambda)$, whose definition, unlike the others in their proposal, depends on the dynamic behavior of the host computer.)

3. ANALYSIS AND SYNTHESIS FUNCTIONS

As noted in Section 1, it is often necessary in numerical computation to scale a number by a power of the base, to break a number into its exponent and fraction-part, or to synthesize a number from these constituents. To provide convenient access to precise and efficient versions of these operations, we propose the following functions:

exponent(x) returns the (integer) exponent of x, as defined in (1). If $x = 0$, or if the exponent cannot be represented, the result is an unspecified integer.

fraction(x) returns the fraction-part of x, as defined in (1); if $x = 0$, the result is 0.

synthesize(x, e) returns *fraction*(x) $\cdot\, b^e$ if possible. If $x = 0$, the result is 0. Otherwise if $e > e_{\max}$ (or $e < e_{\min}$), an overflow (or underflow) may or may not be signaled, and the computation may or may not continue with an unspecified floating-point result.

scale(x, e) returns xb^e if possible. If $x = 0$, the result is 0. Otherwise, if $|xb^e| > \lambda$, (or $0 < |xb^e| < \sigma$), an overflow (or underflow) may or may not be signaled, and the computation may or may not continue with an unspecified floating-point result. However, if the computation continues in the underflow case, then the numerical result must either be zero or have the sign of x and magnitude not greater than σ.

Each of the four functions is defined for all nonzero real x and can be evaluated without error whenever the result is representable in the machine. In Section 6 we discuss the possibility that the fraction-part of the result in the machine's representation may be too precise to be representable, even though x is a machine number. Although we cannot guarantee exact results for all machine numbers, we do require exact results for all model numbers. When x is not a model number, each function must return the exact result for a possibly perturbed argument $\hat{x}$, which must not be outside the smallest model interval containing x. We also require consistency in the sense that the computed values of the functions must approximately satisfy (18) and (19) in Section 5 for all x, even in the neighborhoods of the discontinuities at powers of the base. Because of these discontinuities, the user must take care not to reevaluate an argument or change its precision between related calls.

To avoid surprises from the *synthesize* function, it is often necessary to take special measures to prevent x from being rounded into a discontinuity. Thus in the exponential function given below, we have $1 \le y < b$, but we must act to ensure that $1 \le \tilde{y} \le b - \epsilon$, where $\tilde{y}$ is the computed value of y.

Another possible danger with the *synthesize* function is that it might be invoked with e outside the range of permissible exponents. We considered demanding a check for this, but we want the function to be implemented by in-line code (see Section 6). The detection of an invalid exponent would be costly, and any really helpful response (e.g., triggering a trap or invoking an error handler) would be out of the question on most machines. Fortunately, if e is obtained from the *exponent* function, it can safely be passed to the *synthesize* function, and in any case it is easy for the user to check that $e_{\min} \leq e \leq e_{\max}$.

A more serious danger is that the *scale* function might be invoked with xb^e out of range. Despite the high price of doing so, our definition protects the user from random values when computation continues after an underflow. However, we see no response to overflow that would be both useful and generally feasible.

To illustrate the use of these functions, we present procedures for evaluating the logarithm and exponential functions and for scaling a vector. For each of the elementary functions we show the range reduction and the formation of the final result, but we leave the details of mathematical approximation to a subprocedure. While these procedures are reasonably (and provably) accurate on all computers that conform to the model, there are many computers on which it is possible to achieve greater accuracy. However, to do so, one must pay careful attention to machine-dependent details, as in the excellent treatise by Cody and Waite [2].

Our first example is a procedure *log* for the natural logarithm. Let $L(x)$ be a subprocedure that approximates $\log(x)$ in the interval $1 \leq x \leq 2$. For range reduction we use the formula

$$\log(fb^e) = (e - 1)\log(b) + \log(bf)$$

where $1 \leq bf < b$. To compute $\log(bf)$ we find an integer k, such that $bf = 2^k y$ with $1 \leq y < 2$, and then use the formula

$$\log(bf) = k \log 2 + L(y).$$

The constants $\log 2$ and $\log b$ are evaluated when the procedure is first invoked.

```
  procedure log(x)
  integer k
  real x, y, log2, logb
  if (x ≤ 0) error
initialization:
  logical first := true
  if (first)
    first := false
    log2 := L(2)
    y := b; k := 0
    while (y ≥ 2){y := y/2; k := k + 1}
    logb := k · log2 + L(y)
analysis:
  y := b · fraction(x); k := 0
  while (y ≥ 2){y := y/2; k := k +1}
synthesis:
  return ((exponent(x) + 1) · logb + k · log2 + L(y))
  end
```

Our next example is a procedure *exp* for the exponential function. Let $E(x)$ be

a subprocedure that approximates $\exp(x)$ in the interval $0 \leq x \leq \log b$. For range reduction, we find q and r such that

$$x = q \log b + r,$$

where q is an integer and $0 \leq r < \log b$, and then use the formula

$$\exp(x) = b^q \exp(r).$$

When $b > 2$, further range reduction will probably be needed, but we shall leave that to E. Since $1 \leq \exp(r) < b$, the exponent of $\exp(r)$ is 1 (see (1)), and so the exponent of $\exp(x)$ is $q + 1$. To protect the synthesis from overflow, underflow, and roundoff, the program compares $q + 1$ to the limits of the exponent range and forces $E(r)$, if necessary, into the mathematically correct interval. All but the overflow check could safely be omitted if we used *scale*(y, q) instead of *synthesize*$(y, q + 1)$ to construct the final result, since the *scale* function is continuous in y and includes its own underflow check.

```
  procedure exp(x)
  integer q
  real x, y, r, logb
initialization:
  logical first := true
  if (first){first .= false; logb .= log(b)}
analysis:
  q := ⌊x/logb⌋
  r := x − q · logb
  y := E(r)
synthesis.
  if (q + 1 > e_max) overflow
  if (q + 1 < e_min) underflow
  if (y ≤ 1)y := 1
  if (y ≥ b − ε)y := b − ε
  return (synthesize(y, q + 1))
  end
```

Our last example is a procedure *vecscale* to scale a nonzero vector $(x_1, \ldots, x_n)$ by a power of the base, so that its component of largest magnitude will be near ± 1. Our program sets the largest exponent to zero and scales all components accordingly. There is no risk of overflow, but the possible implications of underflow must be considered by the user.

```
procedure vecscale(x)
integer e, i, n
real s
real array (1:n)x
s := max_{i=1}^{n} (|x_i|)
if (s = 0) error
e .= exponent(s)
for i := 1, ..., n
        x_i .= scale(x_i, − e)
return
end
```

Although an optimizer might be able to remove some of the scaling code from the loop, the scaling may nevertheless be quite costly on some machines. To circum-

vent this cost, it would be tempting to synthesize the scale factor $t = b^{-e}$ and scale by multiplication. Unfortunately, however, we have no guarantee that b^{-e} is in range. Alternatively, one might synthesize the scale factor $u = b^{e-1}$, which certainly is in range, and scale by division. However, on some computers the division would involve multiplication by $u^{-1} = b^{1-e}$, which might be out of range.

Although the *synthesize* and *scale* functions are roughly interchangeable (see Section 5), each has its advantages. The *synthesize* function is closer to the number representation and hence usually more efficient, but its discontinuity makes roundoff in the first argument quite hazardous. The *scale* function is mathematically simpler and hence quite safe from roundoff, but its evaluation poses a greater threat of overflow, and its underflow check is often quite costly.

If the *scale* function were generally available as a machine instruction, we would be willing to drop the *synthesize* function. However, for the present, we reluctantly conclude that both are needed. (The WG 2.5 proposal includes the *exponent* and *synthesize* functions, with the names INTXP and SETXP, but omits the *fraction* and *scale* functions because they are not independent.)

4. PRECISION FUNCTIONS

To attain sharp control over the termination of an iteration, one needs to know the absolute or relative spacing of model numbers in the vicinity of a given number x. We have already shown (see Section 2) that if $x = fb^e$, the absolute spacing is b^{e-p}, and it follows that the relative spacing is $b^{-p}/|f|$. Unfortunately, if $|x| < \sigma/\epsilon = b^{e_{\min}+p-2}$, then the absolute spacing is less than σ and hence too small to be represented in the model. This suggests defining the *absolute-spacing* function

$$\alpha(x) = \begin{cases} b^{e-p}, & \text{if } |x| \geq \sigma/\epsilon \\ \sigma, & \text{if } |x| < \sigma/\epsilon \end{cases} \tag{9}$$

and the *relative-spacing* function

$$\rho(x) = \begin{cases} b^{-p}/|f|, & \text{if } x \neq 0 \\ \text{undefined}, & \text{if } x = 0. \end{cases} \tag{10}$$

Instead of including $\rho(x)$ in the basic set, we favor the *reciprocal-relative-spacing* function

$$\beta(x) = |f|\, b^p, \tag{11}$$

because it is simpler, faster, and more often wanted; see, for example, (15). Note that

$$\alpha(x)\beta(x) = |x|, \qquad \text{whenever} \quad |x| \geq \sigma/\epsilon, \tag{12}$$

and $\beta(x)\rho(x) = 1$ whenever $x \neq 0$. Also, $\beta(x)$ is the absolute value of the fraction-part of x in a representation with the radix point at the right. From (4) we see that $\beta(x)$ is an integer whenever x is a model number, while from (1), that $b^{p-1} \leq \beta(x) < b^p$ whenever $x \neq 0$.

Each of the basic precision functions, α and β, is defined for all real x and can be evaluated without error whenever the result is representable in the machine. In Section 6 we discuss the possibility that the fraction-part of $\beta(x)$ in the machine's representation may be too precise to be representable, even though x

is a machine number. Although we cannot guarantee exact results for all machine numbers, we do require exact results for all model numbers. When x is not a model number, each function must return the exact result for a possibly perturbed argument $\hat{x}$, which must not be outside the smallest model interval containing x. We also require consistency in the sense that the computed values of the functions must approximately satisfy (12), (20), and (21) for all x, even in the neighborhoods of the discontinuities at powers of the base. Because of these discontinuities the user must take care not to reevaluate an argument or change its precision between related calls.

The paradigm of an iteration can now be written in the form

$$x := x_0 \qquad (13)$$
$$\text{repeat}\{\delta := \phi(x);\ x := x + \delta\}\text{until}(|\delta| \le k\alpha(x))$$

where $\phi(x)$ is a function that computes the next correction, and k is a constant large enough to avoid cycles involving nearby floating-point numbers but small enough ($k\epsilon < 1$) to suggest at least some relative accuracy ($|\delta/x| < 1$) whenever $|x|$ is not too small. If the sum of all the neglected terms is less than the final $|\delta|$, then the error in the final x, including an allowance for roundoff or underflow in the last addition in (13), is bounded by $k + 1$ times the larger of σ and an *ulp* (that is, a *u*nit in the *l*ast *p*lace of the model-number representation).

To obtain maximum accuracy, possibly to within the larger of a single ulp or σ, at minimum risk, one can use (13) with k fairly large, to get into the region of convergence, and than continue iterating until $|\delta|$ stops decreasing. For this purpose we write

$$\delta_{\text{old}} := \delta;\ \delta := \phi(x) \qquad (14)$$
$$\text{while}(|\delta| < |\delta_{\text{old}}|)\{x := x + \delta,\ \delta_{\text{old}} := \delta,\ \delta := \phi(x)\}$$

immediately after (13). One cannot use (14) as a replacement for (13), because δ may behave erratically at the beginning of the process. However, in proceeding from (13) to (14) it may be possible to change the iteration function ϕ from one that converges slowly, but globally, to one that converges rapidly, but only locally.

Once the iteration terminates, one can estimate the relative error and the absolute error (in ulps) by writing

$$\begin{aligned}&\text{if } (|\delta| < \sigma)\delta := \sigma\\ &\text{if } (|\delta| \ge |x|)\{\textit{no relative accuracy}\}\\ &\text{else}\\ &\quad \textit{relative-error} := |\delta/x|\\ &\quad \textit{absolute-error} := \textit{relative-error} \cdot \beta(x)\end{aligned} \qquad (15)$$

If $|x| < \sigma/\epsilon$, so that an ulp is less than σ, and if the relative error is large, then it may be possible to get a better answer by rescaling the problem. On the other hand, since the true solution may be zero, a small relative error may be unachievable, and a small absolute error may be quite satisfactory.

(The WG 2.5 proposal includes the absolute-spacing function, defined differently and called EPSLN, but omits the relative-spacing and reciprocal-relative-spacing functions.)

5. SIDE RELATIONS

Although each of the environment parameters defined in Section 2 is useful, they do not form an independent set. The integer parameters can be generated from the real ones via the relations

$$\begin{aligned} b &= \mathit{scale}(1.0, 1) = \mathit{synthesize}(1.0, 2), \\ p &= 2 - \mathit{exponent}(\epsilon), \\ e_{\min} &= \mathit{exponent}(\sigma), \\ e_{\max} &= \mathit{exponent}(\lambda). \end{aligned} \tag{16}$$

Conversely, the real parameters can be generated from the integer ones via the relations

$$\begin{aligned} \epsilon &= \mathit{synthesize}(1.0, 2 - p) = \alpha(1.0) = 1/\beta(1.0), \\ \sigma &= \mathit{synthesize}(1.0, e_{\min}) = \alpha(0.0), \\ \lambda &= \mathit{synthesize}(b - \epsilon, e_{\max}). \end{aligned} \tag{17}$$

When using these identities in a program, we assume that integer constants and environment parameters are available without error. This assumption could be violated, for example, by a compiler with a faulty input conversion procedure. However, we believe that most language processors avoid such errors, and the rest could easily be corrected.

The basic functions are also interrelated in various ways. First, a number can be synthesized from its constituents:

$$\begin{aligned} x &= \mathit{scale}(\mathit{fraction}(x), \mathit{exponent}(x)) \\ &= \mathit{synthesize}(\mathit{fraction}(x), \mathit{exponent}(x)) \\ &= \mathit{synthesize}(x, \mathit{exponent}(x)). \end{aligned} \tag{18}$$

Next,

$$\begin{aligned} \mathit{fraction}(x) &= \mathit{synthesize}(x, 0), \\ \mathit{scale}(x, e) &= \mathit{synthesize}(x, \mathit{exponent}(x) + e). \end{aligned} \tag{19}$$

Note, however, that both sides of the formula for $\mathit{scale}(x, e)$ are undefined when the mathematical result, xb^e, is out of range, and the two unspecified values need not agree. Next, by (9),

$$\alpha(x) = \begin{cases} \mathit{synthesize}(1.0, \max(e_{\min}, 1 - p + \mathit{exponent}(x))), & \text{if } x \neq 0, \\ \sigma, & \text{if } x = 0. \end{cases} \tag{20}$$

Finally, by (11)

$$\beta(x) = \mathit{synthesize}(|x|, p). \tag{21}$$

6. IMPLEMENTABILITY

Each of the seven environment parameters is a well-defined constant for any given floating-point number system. Although it may be convenient to express these parameters as functions (see Section 7), a compiler should substitute the correct values rather than produce code to fetch them at run time. Values in single- and double-precision for several computers that are accessible to us have

Table I Tested Values of the Environment Parameters in Single- and Double-Precision (penalties are shown in parentheses)

	b	p	e_{min}	e_{max}	ϵ	σ	λ
Cray-1	2	47(1)	−8189(3)	8190(1)	1.42×10^{-14}	3.67×10^{-2466}	2.73×10^{2465}
	2	94(2)	−8099(93)	8190(1)	1.01×10^{-28}	4.54×10^{-2439}	2.73×10^{2465}
DEC VAX	2	24	−127	127	1.19×10^{-7}	2.94×10^{-39}	1.70×10^{38}
	2	56	−127	127	2.78×10^{-17}	2.94×10^{-39}	1.70×10^{38}
Honeywell	2	27	−127(1)	127	1.49×10^{-8}	2.94×10^{-39}	1.70×10^{38}
6000	2	63	−127(1)	127	2.17×10^{-19}	2.94×10^{-39}	1.70×10^{38}
IBM 370	16	6	−64	63	9.54×10^{-7}	5.40×10^{-79}	7.24×10^{75}
	16	14	−64	63	2.22×10^{-16}	5.40×10	
	16	14	−64	63	2.22×10^{-16}	5.40×10^{-79}	7.24×10^{75}
Interdata	16	6	−64	63	9.54×10^{-7}	5.40×10^{-79}	7.24×10^{75}
8/32	16	14	−64	63	2.22×10^{-16}	5.40×10^{-79}	7.24×10^{75}

Notes On the Cray-1, arithmetic is not always good to the last bit, so p must be reduced from 48 to 47 in single-precision and from 96 to 94 in double-precision. To avoid underflow during the comparison of double-precision model numbers, $e_{min}^{(dp)}$ must be increased by $p - 1 = 93$ from −8192 to −8099 Although single-precision comparisons are also implemented in software, the single-precision subtraction underflows to a tiny number with the corrrect sign, and hence there is no analogous penalty on $e_{min}^{(sp)}$ However, to avoid overflow in computing σ/σ, $e_{min}^{(sp)}$ must be increased to $2 - e_{max} = -8189$. Finally, to avoid overflow on multiplication by 1 0, e_{max} must be reduced from 8191 to 8190. Despite all the penalties, the model was not fully supported in double-precision at the time of our initial testing Due to an error in the Fortran compiler, two double-precision numbers were considered equal if their leading words were the same Fortunately, the compiler has since been repaired.

On the DEC VAX 11/780, the model is fully supported with no penalties.

On Honeywell 6000 Series computers, e_{min} must be increased from −128 to −127 because of the lack of symmetry of the two's complement number representation Even so, the model was not fully supported at the time of our initial testing On underflow the numerical result is supposed to be zero, but due to an error in the Fortran trap handler, the numerical result was sometimes equal to 2^{256} times the chopped mathematical result. Fortunately, the trap handler has since been repaired

On the IBM 370/168, the model is fully supported with no penalties.

On the Interdata 8/32, there are no penalties, but the model is not fully supported. Due to a design error in the floating-point processor, the computed value of $x - y$ may be zero when $x = 2^{256}y$. The error has been corrected in newer Interdata computers.

been verified by Schryer's test program and presented in Table I. If a penalty was imposed on one of the four basic parameters, it is shown in parentheses. The value that would be inferred from the space available in the computer's number representation exceeds the model value in magnitude by the amount of the penalty. For example, on the Cray-1, 48 bits are available for the fraction-part of a single-precision number, but a penalty of 1 reduces the precision to $p = 47$. During the testing, hardware and compiler bugs that could not be evaded by reducing the claimed precision and range were discovered on several machines. These are mentioned in the caption of Table I. Schryer will publish details of the testing later and in the meantime will call the bugs to the attention of the manufacturers.

In principle, the *exponent* and *synthesize* functions constitute a complete set. If they are available, the remaining four functions can be constructed with the aid of (19)-(21). While such high-level implementations could easily be made quite safe (by including appropriate tests for overflow and underflow), we would expect them to be too inefficient for most applications. Fortunately, each of the six basic functions is simple enough to permit a short in-line implementation on most machines. However, any differences between the model's representation of floating-point numbers (see (3)-(5)) and the machine's representation must be

Table II. Instruction Counts for In-Line Single-Precision Implementations of the Basic Functions (in each case operands are fetched from local variables, and result is left in a register)

	FPEXPN	FPFRAC	FPMAKE	FPSCAL	FPABSP	FPRRSP
Cray-1	5	6	9	15	9	7
DEC VAX	2	3	4	8	5	3
Honeywell 6000	6	5	9	12	11	5
IBM 370	3	4	7	11	7	4
Interdata 8/32	3	4	7	11	7	4

accounted for, and the resulting codes are rarely as short as we would like. We have implemented all six functions for several computers that are accessible to us, and the instruction counts for these code samples are presented in Table II.

Each of the analysis and synthesis functions typically requires only a few instructions to fetch or modify the exponent field of a floating-point number. It may also be necessary to shift the exponent into position, or to add or subtract a bias constant. Special care may be needed for negative x on machines where negation affects the exponent field. Also, on a two's-complement computer with the implicit b-point at the left, the machine fraction-part of a negative power of 2 is -1 rather than $-\frac{1}{2}$, and therefore the machine exponent of such a number must be adjusted by 1. For the precision functions it is convenient to proceed directly from (20) and (21), although one does not actually use the *exponent* or *synthesize* functions. In all cases special care may be required when $x = 0$. In particular, if the result is zero, it may be important to avoid setting the exponent to an unusual value.

As noted in Sections 3 and 4, each of the six basic functions is defined for all nonzero real x and can be evaluated without error whenever the result is representable in the machine. In some implementations the results are exact for all machine numbers. In other implementations, however, there may be circumstances in which the exact result is too precise for the location in which it is to be placed and must therefore be shortened by rounding or chopping. For example, the argument might be taken from a double-length accumulator when the result is to be placed in a single-length storage location. As a more exotic example, if we model a six-digit hexadecimal computer by setting $b = 2$ and $p = 21$, then there are many machine numbers that cannot exactly be doubled by invoking $scale(x, 1)$.

Since five of the six basic functions are discontinuous at powers of the base, there is a risk of inconsistencies on any machine that rounds or chops away from zero. An implementor can guard against this instability, if necessary, by shortening each extraprecise argument to one of the neighboring model numbers according to a consistent rounding or chopping rule.

7. FORTRAN REPRESENTATION

In all of the above we have carefully ignored the distinction between single- and double-precision numbers. The American National Standard Fortran language specifically has floating-point variables of these two precisions; some compilers recognize a third. There is talk of adding a mechanism to Fortran to permit specifying the number of digits of accuracy rather than the number of machine words. To avoid difficulties in this area, we propose using generic functions, for

which the compiler chooses the operation to be performed and/or the type of the result from the type of the first argument. Although we have taken some care to choose appropriate names, we recognize that others may have different tastes or better ideas. Like the conversion functions in Fortran 77, the proposed functions need not have specific names for the different types. The only restriction on such generic functions is that they cannot be passed as actual arguments.

The following seven generic functions, in which the prefix "EP" stands for "environment parameter," would provide the necessary parameters:

EPBASE(X) = b
EPPREC(X) = p
EPEMIN(X) = $e_{\min}$
EPEMAX(X) = $e_{\max}$
EPMRSP(X) = ϵ
EPTINY(X) = σ
EPHUGE(X) = λ

The first four of these functions return integers related to the precision of X. The last three return floating-point values with the same precision as X. The functions EPBASE, EPPREC, and EPHUGE should also be defined for integer arguments; an appropriate model of integer computation is outlined by Fox et al. [5].

We considered a number of ways of introducing these environment parameters. Adding new "dot constants" (analogous to .TRUE.) or preinitialized integer variables would work. However, a complete set of names would be needed for each possible type. For example, it would be necessary to define .HUGE., .DHUGE., and .IHUGE.. The solution in the PORT library of having one integer-valued function (I1MACH), one real-valued function (R1MACH), and one double-precision-valued function (D1MACH), is possible but rather clumsy to use. We consider our solution attractive because it requires only seven names and can accommodate any number of actual or potential types. Whatever the representation, a compiler should provide the values at compile-time, as discussed in Section 6. (The WG 2.5 proposal provides single- and double-precision names for its parameters but does not suggest any mechanism for including them in Fortran.)

For the six computational procedures we suggest

FPEXPN(X) = *exponent*(X)
FPFRAC(X) = *fraction*(X)
FPMAKE(X,E) = *synthesize*(X, E) = *fraction*(X)b^E
FPSCAL(X,E) = *scale*(X, E) = Xb^E
FPABSP(X) = $\alpha(X)$
FPRRSP(X) = $\beta(X)$

where the prefix "FP" stands for "floating point." FPEXPN returns the exponent of X, represented as an integer; the other five functions return floating-point values with the same precision as X. (Our FPEXPN, FPMAKE, and FPABSP correspond to the functions INTXP, SETXP, and EPSLN, respectively, in the WG 2.5 proposal.)

Altogether, we have proposed thirteen new generic functions for Fortran 77.

These come in two families, each marked by a mnemonic prefix to make the names distinctive and the relationships clear. It may seem odd to old-time Fortranners that some of the functions always return values of type integer and yet have names beginning with E or F. However, the list of intrinsic functions in Fortran 77 already includes both generic names (e.g., LOG and LOG10) and specific names (e.g., CHAR and LLT) that violate the implicit typing rules. The only real difficulty from the point of view of the American National Standard Fortran is that FPMAKE and FPSCAL have arguments of different types, while all current intrinsic functions have arguments of identical type.

ACKNOWLEDGMENTS

We thank N. L. Schryer for the tested parameter values in Table 1 and T. G. Szymanski for the Cray-1 functions cited in Table 2. We also thank the participants in stimulating discussions at the Los Alamos Portability Workshop (Los Alamos, New Mexico, May 1976), the Argonne National Laboratory Workshop on the Portability of Numerical Software (Oak Brook, Illinois, June 1976), the Mathematics Research Center Symposium on Mathematical Software (Madison, Wisconsin, March 1977), and several meetings of the IFIP Working Group on Numerical Software. Finally, we are indebted to numerous colleagues and several referees for helpful comments on earlier drafts.

REFERENCES

1. BROWN, W.S. *A Simple But Realistic Model of Floating-Point Computation.* To be published An earlier version of this paper was published in *Mathematical Software III.* John R. Rice, Ed., Academic Press, New York, 1977, pp 343-360.
2. CODY, W.J., AND WAITE, W M *A Software Manual for the Elementary Functions.* Prentice-Hall, Englewood Cliffs, N.J., 1980
3. FORD, B. Parameterization of the environment for transportable numerical software. *ACM Trans. Math Softw 4* (June 1978), 100-103.
4. FORSYTHE, G.E., AND MOLER, C.B. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1967, p 87.
5. FOX, P.A., HALL, A D., AND SCHRYER, N L. The PORT Mathematical Subroutine Library. *ACM Trans Math Softw. 4* (June 1978), 104-126
6. NAUR, P. Machine dependent programming in common languages. *BIT 7* (1967), 123-131. An earlier version of this paper was published in *ALGOL Bulletin,* No. 18, October 1964.
7. REID, J Functions for manipulating floating-point numbers. *ACM SIGNUM Newsletter 14,* 4 (Dec. 1979), 11-13.
8. SCHRYER, N L. UNIX™ as an environment for producing numerical software. *ACM SIGNUM Newsletter 14* (March 1979), pp. 49-52
9. WILKINSON, J.H. *Rounding Errors in Algebraic Processes* Prentice-Hall, Englewood Cliffs, N.J., 1963.

Received November 1978; revised March 1980; accepted May 1980.