Implementing Complex Elementary Functions Using Exception Handling

T. E. HULL and THOMAS F. FAIRGRIEVE

University of Toronto

and

PING TAK PETER TANG

Argonne National Laboratory

We develop algorithms for reliable and accurate evaluations of the complex elementary functions required in Fortran 77 and Fortran 90, namely cabs, csqrt, cexp, clog, csin and ccos. The algorithms are presented in a pseudo-code which has convenient exception handling facilities. A tight error bound is derived for each algorithm. Corresponding Fortran programs for an IEEE environment have also been developed to illustrate the practicality of the algorithms, and these programs have been tested very carefully to help confirm the correctness of the algorithms and their error bounds — the results of these tests are included in the paper, but the Fortran programs are not.

Categories and Subject Descriptors: G.1 [**Numerical Analysis**]: General—*error analysis, numerical algorithms;* Approximation—*elementary function approximation;* G.4 [**Mathematical Software**]: *Algorithm analysis; Reliability and robustness; Verification*

General Terms: Algorithms, design

Additional Key Words and Phrases: Complex elementary functions, implementation

1. INTRODUCTION

Our purpose is to develop algorithms, along with error bounds, for reliable and accurate evaluations of the complex elementary functions required in Fortran 77 [1] and Fortran 90 [4], namely cabs, csqrt, cexp, clog, csin, and ccos. These (seemingly oxymoronic) complex elementary functions can be expressed in terms of formulas involving only real arithmetic and real elementary functions. Complex arithmetic is not needed. If care is taken these formulas can usually be arranged so that serious numerical cancellation will not occur during their evaluation. (If this cannot be arranged, higher precision may be necessary at such critical points in the calculations.)

This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario, as well as by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38, and by the STARS Program Office, under AF Order RESD-632.
Authors' addresses: T. E. Hull and T. F. Fairgrieve, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4 (e-mail: {tehull,tff}@cs.utoronto.ca); Ping Tak Peter Tang, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL U.S.A. 60439-4801 (e-mail: tang@antares.mcs.anl.gov)

The main difficulty that remains in such evaluations is the possibility that overflow or underflow might occur at some intermediate stages in the calculation. Such exceptions are often "spurious", in the sense that the final mathematical result is within the range of machine representable numbers. In these circumstances, which normally occur only rarely, the algorithms must provide for alternative calculations, which may be more lengthy, but which are able to circumvent the spurious exceptional situations.

All of this suggests designing the algorithms with the help of exception handling facilities. Each algorithm would then begin with a direct evaluation of the original carefully arranged formulas. This would be efficient, and almost always successful. But if an exception should occur during such a calculation, it would cause control of the calculations to be transferred to an exception handler that would do what was needed to circumvent the difficulty, if possible, or, if not possible, to cause an appropriate exception to be returned by the function itself. To capture this essential feature of our algorithms, we present them in a pseudo-code which possesses a convenient exception handling facility.

Section 2 provides basic information on error bounds that we assume about the real arithmetic and real elementary functions used in evaluating the complex functions. Special attention is given to the real sine and cosine functions. Three functions needed later are introduced, and our conventions about exceptions returned by the complex functions are also specified precisely. Section 3 introduces the exception handling construct used in our pseudo-code. The pseudo-code algorithms are presented in Section 4, along with error analyses. Special implementations for testing are described in Section 5, and formulas for the error bounds derived in Section 4 are tabulated in Section 5, but extensive testing is done only for IEEE binary arithmetic [3] on Sun systems [6]. Issues to be considered in production implementations are discussed in Section 6, and concluding remarks are given in Section 7.

We should emphasize that we assume throughout that the function arguments are exact. If an argument z is only slightly in error, the induced error in the corresponding value of the function $f(z)$ is generally small. But in special situations, the error can also be very large. The relative error in $f(z)$, $\Delta f(z)/f(z)$, is approximately $zf'(z)/f(z)$ times the relative error in z, $\Delta z/z$. The *relative error magnification factor* $zf'(z)/f(z)$ can be very large. For example, in one extreme case, when $f(z)$ is $\log(z)$, the factor is $1/\log(z)$ which can be arbitrarily large when z is near 1.

Complex exponentiation is not included in this paper; it is difficult to develop an algorithm that does much better than simply evaluating $\text{cexp}(w*\text{clog}(z))$ to approximate $z**w$. We hope to consider this problem in a separate paper.

An earlier version of this paper appeared as an Argonne Preprint [2].

2. BASIC NUMERICAL OPERATIONS

The complex elementary functions described in Section 4 depend on real arithmetic operations and on real elementary functions. The main purpose of this section is to introduce a notation for the errors and error bounds associated with these real operations which can be used later to derive error bounds for the complex functions. Error bounds associated with the real sine and cosine functions are discussed further in a separate subsection, and the implementation of $\mathrm{log1p}(x) = \log(1+x)$ is also considered separately, since log1p is not always available as one of the standard real functions. Two special functions for manipulating exponents are also specified. Possible exceptions for complex functions are specified in a final subsection.

2.1 Errors and Error Bounds

We assume that input values for the complex elementary functions are normalized complex floating-point numbers. With this assumption, it can be arranged that all internal operations which might generate errors (such as rounding errors) will operate only on normalized real floating-point numbers. The associated error analyses are mostly relatively straightforward, and produce results which are simple and easy to use. (It would also be a straightforward matter to make provision for the input or output of other values, such as complex numbers whose components could be denormalized, but then the error analyses would be very much more complicated. We also do not distinguish between signed zeros; for more about the latter in this context, see Kahan [5].)

Except for cabs, whose output is to be a normalized real floating-point number (if overflow is not returned), the output in each example is to be a normalized complex floating-point number (if no exception is returned).

The main assumption we make about the real arithmetic is that, if x and y are normalized real floating-point numbers, and op is one of the four basic arithmetic operations, then there is a relative roundoff error bound E such that

$$fl\,(x \,\mathrm{op}\, y) = (x \,\mathrm{op}\, y)(1+\epsilon),$$

for some ϵ where $|\,\epsilon\,| \le E$, provided no exception occurs. Here $fl\,(x \,\mathrm{op}\, y)$ is the rounded floating-point approximation to $x \,\mathrm{op}\, y$ produced by the machine. And we assume that $fl\,(x \,\mathrm{op}\, y) = x \,\mathrm{op}\, y$ whenever $x \,\mathrm{op}\, y$ is machine representable.

We also assume that corresponding error bounds are known for the real elementary functions. For example, we assume there is a bound E_{sqrt} such that

$$fl\,(\sqrt{x}) = \sqrt{x}(1+\epsilon_{\mathrm{sqrt}}),$$

for some ϵ_{sqrt} where $|\,\epsilon_{\mathrm{sqrt}}\,| \le E_{\mathrm{sqrt}}$.

Similarly, we assume relative error bounds, namely E_{exp}, E_{sincos}, E_{arg}, E_{log}, E_{log1p}, E_{sinhcosh}, for the other real elementary functions used in Section 4. Each such bound should be only a small multiple of E, but the bound for sin and cos needs special attention and will be considered in more detail in the next subsection. (We also assume that there can be no underflow with sin or cos.)

With these assumptions we can derive bounds on the errors in evaluating expressions that arise in Section 4. For example, we can conclude that the program expression $x * y * sqrt(x)$, or $fl\,(xy\sqrt{x})$, has the value

$$xy\sqrt{x}(1+\epsilon)(1+\epsilon)(1+\epsilon_{\mathrm{sqrt}}),$$

and hence is correct to within a relative error bounded by $2E + E_{\mathrm{sqrt}}$, if we neglect terms which are small multiples of E^2, and provided no exception is returned. (The notation is convenient – note that different occurrences of ϵ are not necessarily the same.)

We use results like this to derive relative error bounds for the approximations we obtain to the complex functions in Section 4. In general, if f is a complex function with f_r and f_i its real and imaginary parts respectively, and f^c, f_r^c and f_i^c are the corresponding calculated approximations, then the magnitude of the relative error in f^c is

$$\begin{aligned} \left| \frac{f^c - f}{f} \right| &= \left| \frac{\dfrac{f_r^c - f_r}{f_r} f_r + i \, \dfrac{f_i^c - f_i}{f_i} f_i}{f_r + i \, f_i} \right| \\ &= \sqrt{\frac{\epsilon_r^2 f_r^2 + \epsilon_i^2 f_i^2}{f_r^2 + f_i^2}}, \end{aligned}$$

where ϵ_r and ϵ_i are the relative errors in f_r^c and f_i^c respectively. (We are assuming here that f_r and f_i are not zero. If either one is zero, such cases have to be treated separately.)

Let us denote bounds for $|\, \epsilon_r \,|$ and $|\, \epsilon_i \,|$ by E_r and E_i respectively. Then

$$\left| \frac{f^c - f}{f} \right| \leq \sqrt{\frac{E_r^2 f_r^2 + E_i^2 f_i^2}{f_r^2 + f_i^2}}.$$

This is bounded by $\max(E_r, E_i)$, which is a useful bound when E_r and E_i are not very different, which turns out to be the case with all our examples in Section 4 except for clog.

In two of the examples in Section 4 (csqrt and cexp) we examine in more detail the expression from which this bound is obtained and we are able to determine bounds which are somewhat smaller than $\max(E_r, E_i)$. In the case of clog, we are able to obtain a bound which is dramatically smaller than $\max(E_r, E_i)$.

2.2 Errors in Sine and Cosine Approximations

As stated in the previous subsection, we assume a relative error bound E_{sincos} for the sine and cosine functions. This means that we assume a bound E_{sincos} such that

$$fl\,(\sin(x)) = \sin(x)\,(1 + \epsilon_{\text{sin}}) \quad \& \quad fl\,(\cos(x)) = \cos(x)\,(1 + \epsilon_{\text{cos}})$$

where $|\, \epsilon_{\text{sin}} \,| \leq E_{\text{sincos}}$ and $|\, \epsilon_{\text{cos}} \,| \leq E_{\text{sincos}}$. This assumption is of no use unless we restrict the range of values of x for which it is to hold. In Section 5 we will give the result of measuring this value for $|\, x \,| < 10^6$, on the system we use for testing, but in the remainder of this subsection we will examine in more detail how such a result depends on the range for x, the accuracy with which the range reduction is done, the radix, and the precisions used in calculating the approximations.

We consider in detail only one relatively common case, namely the case where $\sin(x)$ and $\cos(x)$ are approximated by first finding x', where x' is approximately x mod $\pi/2$ and $|\, x' \,| < \pi/4$, and then using x' in some standard way to determine the final approximation to $\sin(x)$ or $\cos(x)$.

Suppose that the approximation to $\pi/2$ that is used in the range reduction is $PIby2$, where $PIby2 = (\pi/2)(1+\epsilon_{\pi/2})$, and we are expecting to choose the bound $E_{\pi/2}$ for $\epsilon_{\pi/2}$ to be very much smaller than E. We first determine n so that $|\, x - nPIby2\,| < \pi/4$ and let $x' = x - nPIby2$. Both $PIby2$ and x' will be in relatively high precision. (For a good discussion of how this might be accomplished, see Payne and Hanek [7].) The value of x' would now be rounded, to x'' say, where $x'' = x'(1+\epsilon'')$ and $|\,\epsilon''\,|$ is at least $\leq E$, and then x'' would be used as the argument for a sine or a cosine approximation valid over $(-\pi/4, \pi/4)$. (It is common practice to use (possibly simulated) high precision arithmetic in these computations, in which case ϵ'' would likely be bounded by a small multiple of E^2.)

The final approximation to $\sin(x)$ is then $(-1)^{n/2}\sin(x'')\ (1+\epsilon_{\sin(-\pi/4,\pi/4)})$ if n is even, or $(-1)^{(n-1)/2}\cos(x'')\ (1+\epsilon_{\cos(-\pi/4,\pi/4)})$ if n is odd, where the relative errors introduced at this stage are the errors committed in approximating the sine and cosine functions over the interval $(-\pi/4, \pi/4)$ and should be bounded by only small multiples of E. Similar expressions can be obtained to approximate $\cos(x)$.

If n is even, the calculated approximation to $\sin(x)$ is therefore

$$\begin{aligned} fl\,(\sin(x)) &= \sin(x'' + n\pi/2)(1+\epsilon_{\sin(-\pi/4,\pi/4)}) \\ &= \sin((x - nPIby2)(1+\epsilon'') + n\pi/2)(1+\epsilon_{\sin(-\pi/4,\pi/4)}) \\ &= \sin(x + (x - nPIby2)\epsilon'' - n(PIby2 - \pi/2))(1+\epsilon_{\sin(-\pi/4,\pi/4)}) \\ &= \sin(x)(\cos(A) + \cot(x)\sin(A))(1+\epsilon_{\sin(-\pi/4,\pi/4)}), \end{aligned}$$

where

$$\begin{aligned} A &= (x - nPIby2)\epsilon'' - n(PIby2 - \pi/2) \\ &= (x - n\pi/2)\epsilon'' - n(\pi/2)\epsilon_{\pi/2}\epsilon'' - n(\pi/2)\epsilon_{\pi/2}. \end{aligned}$$

We first require $PIby2$ to be accurate enough so that $\left|\,(n\pi/2)\epsilon_{\pi/2}\,\right| < E$. Then $n(\pi/2)\epsilon_{\pi/2}\epsilon''$ can be neglected, and $|\,A\,|$ is bounded by a small multiple of E, so that we can replace $\cos(A) + \cot(x)\sin(A)$ with $1 + \cot(x)A$. Thus, the relative error in $fl\,(\sin(x))$ becomes

$$(x - n\pi/2)\cot(x)\epsilon'' - (n\pi/2)\cot(x)\epsilon_{\pi/2} + \epsilon_{\sin(-\pi/4,\pi/4)},$$

neglecting only small multiples of ϵ^2. In this expression $(x - n\pi/2)\cot(x) = (x - n\pi/2)\cot(x - n\pi/2)$. Here $|\,x - n\pi/2\,| = \left|\,x - nPIby2 + n(\pi/2)\epsilon_{\pi/2}\,\right| < \pi/4 + E$, and it can be shown that $|\,u\cot(u)\,| \leq 1$ for $|\,u\,| < \pi/4 + E$. Thus the relative error in $fl\,(\sin(x))$ is bounded by

$$E'' + \max\left|\,(n\pi/2)\cot(x)\epsilon_{\pi/2}\,\right| + E_{\sin(-\pi/4,\pi/4)},$$

neglecting small multiples of E^2, and where max is taken over all machine representable values of x in the appropriate range. (Of course E_{sincos} can be expected to be somewhat smaller, but this expression shows clearly what factors need to be controlled to keep E_{sincos} reasonably small.)

This result is applicable when n is even. When n is odd, $\cot(x)$ must be replaced by $\tan(x)$ and $E_{\sin(-\pi/4,\pi/4)}$ by $E_{\cos(-\pi/4,\pi/4)}$. Similar results can be derived for approximations to $\cos(x)$. All of these results can be combined into one bound which is applicable in approximations to either $\sin(x)$ or $\cos(x)$, namely

$$E'' + \max\left(\mid (n\pi/2)\cot(x)\mid, \mid (n\pi/2)\tan(x)\mid\right) E_{\pi/2} + E_{\mathrm{sincos}(-\pi/4,\pi/4)},$$

neglecting small multiples of E^2, and where max is taken over all machine representable values of x in the appropriate range.

We now impose an even more stringent requirement on the accuracy of $PIby2$. It must be chosen so that the error contributions involving the tangent and cotangent terms in the above expression are bounded by a small multiple of E. The choice will depend on the radix and precision of the number system, as well as on the appropriate range of values of x in that number system.

To illustrate, let us consider just one example. If x is a 24-bit binary number, and $\mid x \mid \leq 10^6$, we have determined that $\max\left(\mid (n\pi/2)\tan(x)\mid, \mid (n\pi/2)\cot(x)\mid\right)$ is almost 7.775×10^{12}. If $PIby2$ is stored as accurately as possible in 66 bits (which is the default case in Sun systems [6, p. 53]), it turns out that $E_{\pi/2} < 1.288 \times 10^{-21}$, so that

$$\begin{aligned} \max\left(\mid (n\pi/2)\tan(x)\mid, \mid (n\pi/2)\cot(x)\mid\right) E_{\pi/2} &< 10.015 \times 10^{-9} \\ &< .168 \times 2^{-24} \\ &= .168E. \end{aligned}$$

Under these circumstances, the above bound for the relative errors in the sine and cosine functions is $E'' + .168E + E_{\mathrm{sincos}(-\pi/4,\pi/4)}$, which indicates that storing $PIby2$ in 66 bits is accurate enough when $\mid x \mid \leq 10^6$.

If we allow the larger range $\mid x \mid \leq 10^9$, $\max\left(\mid (n\pi/2)\tan(x)\mid, \mid (n\pi/2)\cot(x)\mid\right) E_{\pi/2}$ is almost $586E$, which suggests that a considerably more accurate value for $PIby2$ should be used in the range reduction. (As an option, the Sun system provides access to a very high precision approximation.)

We will not consider any further detail at this point. The above is enough to illustrate what sort of considerations should be taken into account in practice. Some of our bounds in Section 4 depend on E_{sincos}. It would be helpful if this quantity were to be carefully documented for the individual systems on which our algorithms might be implemented.

2.3 The LOG1P Function

In Subsection 4.4 we need to evaluate a real elementary log function in a situation where the argument of the log function must itself be evaluated and so is known only approximately. This introduces a serious problem when the value of that argument is near 1, say $1+x$ where x is small. It turns out that we can often calculate x to working precision, so that a real elementary function $\mathrm{log1p}(x)$ that approximates $\log(1+x)$, without explicitly adding 1 to x, is ideally suited to our purposes.

Since log1p is not always available, we present here a simple way to specify a reasonably accurate log1p function using the log function. (For a more accurate implementa-

```
function LOG1P (x : real ) : real
    possible exceptions (domainerror )
    real y
    real E ← appropriately initialized
    real oneoverE ← appropriately initialized
    if x ≤ −1 then
        return domainerror
    elsif x > oneoverE then
        return log(x)
    elsif | x | < E then
        return x
    else
        y := 1 + x
        return log(y) − ((y − 1) − x)/y
    endif
end LOG1P
```

Fig. 1. An implementation of log1p using log. After looking after three special cases, the program takes care to obtain an accurate approximation to $\log(1+x)$.

tion that does not rely on the log function, see Tang [8].) The program in Figure 1 first looks after the exceptional case when $x \leq -1$. Then, when x is so large that $\mathrm{log1p}(x)$ can be replaced by $\log(x)$, the program makes the replacement; it is sufficient to have $x > 1/E$, and this also ensures that the replacement is made in the one case where it is necessary to do so, which is when x is so large that $x+1$ would overflow, *i.e.,* when x is the largest machine representable number and the rounding mode is "round-up". Then, when $|\,x\,|$ is so small that $\mathrm{log1p}(x)$ can be replaced by x, the program makes the replacement; it is sufficient to have $|\,x\,| < E$, but is also necessary to make the replacement if the arithmetic happens to be truncated. (Otherwise, you could have, on a binary machine for example, $x = -E^3$, so that y would be $1 - E/2$ instead of 1, and then $\log(y)$ would be $-E/2$ plus possibly a small multiple of E^2 – depending on how accurate the log function is – and the returned value would be 0 or a small multiple of E^2, instead of $-E^3$; in either case the relative error is enormous.)

Having looked after all these special cases, the program assigns the rounded value of $1+x$ to y, and then recovers the error with $(y-1)-x$. The (approximate) relative error in y is then the rounded value of the quotient $((y-1)-x)/y$, which we denote here by *relerr.* Then we have

$$y(1 - relerr)$$

equal to $1+x$ to high accuracy.

Then $\log(1+x)$ equals

$$\begin{aligned} \log(y(1 - relerr)) &= \log(y) + \log(1 - relerr) \\ &= \log(y) - relerr \end{aligned}$$

to high accuracy.

We have tested the program only in the standard rounding mode of IEEE [3] arithmetic (where the two **elsif** clauses are not needed), but we believe the program is also valid for other arithmetic systems with reasonable rounding conventions, including truncation.

2.4 The logb and scalb Functions

We need two functions for manipulating exponents. The first returns an integer value. It is

$$\mathrm{logb}(x) = \lfloor \log_{radix}(|\, x \,|) \rfloor ,$$

if $x \neq 0$. We will not use this function when $x = 0$.

The second function is, for x real and n an integer,

$$\mathrm{scalb}(x, n) = (radix)^n x,$$

unless this overflows or underflows, in which case the appropriate exception is raised.

Both functions are exact, provided of course that there is no overflow or underflow with the second one.

2.5 Exceptions

In the examples of Section 4, the only exceptions that can be returned are *overflow, underflow* and *domainerror.*

We have adopted the convention that *overflow* will be returned whenever either one of the components of the result overflows, or if both overflow. We do not try to provide the component values themselves, but only minor modifications would be needed to provide values of $\pm\infty$, where appropriate, if that were considered to be desirable. An appropriate normalized value for a component that did not overflow or underflow could also be provided.

If only one component underflows, and the value of the other component is so much larger that setting the underflowed component to zero does not alter the error bound by more than a small multiple of E^2, we set that underflowed component to zero and do not return *underflow.* (In fact, this is the situation that occurs with csqrt, clog, csin and ccos.) If the underflowed component is smaller than the normal component by a factor of at most E (this is quite arbitrary), we still set the underflowed component to zero and do not return *underflow,* but we increase the bound as required – otherwise, if the criterion for setting the underflowed component to zero is not met, or if both components underflow, we return *underflow.* (What is described in the last sentence can only occur with cexp. The setting of the underflowed component to zero has been accounted for in our error analysis, and it increases the bound by only about 5% in the system we use for testing.) If *underflow* is returned, we do not try to provide component values themselves, but, once again, only minor changes would be needed to provide for special values, such as denormalized numbers.

3. PSEUDO-CODE

Our notation for the needed exception handling construct is shown in Figure 2. The calculations in the enable block will normally produce the required result, but, if an

```
enable
    - - - -
    - - - -
    - - - -
handle
    - - - -
    - - - -
    - - - -
end
```

Fig. 2. The exception handling construct. The enable block would normally produce the required result, but the handler takes over if an exception occurs in the enable block.

exception occurs during the course of these calculations, control is transferred to the handle block, or handler, where action is taken to circumvent or otherwise cope with the exceptional situation.

There are various ways in which such constructs can be implemented, depending on the programming language used. We will discuss some of these ways briefly in Sections 5 and 6. But for our present purposes it does not matter how such constructs are implemented. In particular, it does not matter whether the transfer of control takes place as soon as the first exception occurs, or whether, as is possible in an IEEE environment [3], the calculation continues to the end of the enable block where a test is made to determine whether or not an exception occurred. We do not make use of any intermediate results that might have been obtained in the enable block.

We assume that any indication that exceptions occurred in the enable block disappears on leaving the handler and also that exception handling constructs can be nested within handlers of other such constructs. However, we do not assume that any indication of the type of exception (overflow, or underflow, etc.) which caused the transfer of control is available in the handler. This seems appropriate considering the "impreciseness" of interrupts in pipelined machines, or in the presence of any parallelism.

We do not allow exits from exception handling constructs, except for possible returns from within handlers — whether they are returns of values or returns of exceptions.

Otherwise our pseudo-code is reasonably self-explanatory. It is intended to provide an easy-to-understand description of algorithms for calculating good approximations to the complex elementary functions. Implementing the programs can be more convenient in some languages than in others, a matter to which we return in Sections 5 and 6.

4. PSEUDO-CODE ALGORITHMS

In this section we present algorithms in the form of pseudo-code programs for each of the six complex elementary functions required in Fortran 77 and Fortran 90. The error bounds derived in this section are repeated in tabular form in Section 5. The term *precision* is used in the programs to denote the number of significant digits in the machine representations of real numbers — for example, in the IEEE [3] binary representation, single precision is 24.

4.1 Complex Absolute Value CABS

We first consider the absolute value function $|\, z\,|$, where $z = x + i\, y$. The program in Figure 3 for calculating an approximation to this function is based on the formula

$$|\, z\,| \;=\; \sqrt{x^2 + y^2}\,.$$

The result of any such calculation is a representable real value for any value of z, except that it can overflow in extreme cases when x and y are both very large.

The main difficulties in developing a program for this function are in dealing with the possibility of spurious overflows or underflows. Since these will occur only rarely in most calculations, a good strategy is to attempt to calculate immediately the required approximation directly on the basis of the formula, since this is both efficient and reasonably accurate – and it works most of the time – and then, if that approach fails, the program can take all the time it needs to look after the exceptional cases.

The handler first looks after the special cases where x or y is zero. Otherwise $\mathrm{logb}(x)$ and $\mathrm{logb}(y)$ both exist and can be used to determine whether or not x and y differ greatly in magnitude. If they do differ by enough, the smaller of $|\, x\,|$ and $|\, y\,|$ can be neglected. Otherwise they are close enough that they can be scaled so that the corresponding scaled result can be calculated without any spurious overflows or underflows. Finally, the scaled value of the result can be unscaled to provide the required result, but it might happen that this unscaling will itself produce a result which overflows, in which case *overflow* must be returned.

For the error analysis we first consider the case when no exception occurs. The analysis proceeds as follows:

$$fl\,(x^2) = x^2(1+\epsilon) \quad \text{and} \quad fl\,(y^2) = y^2(1+\epsilon)$$

and, since these are of the same sign,

$$fl\,(x^2) + fl\,(y^2) = (x^2 + y^2)\,(1+\epsilon),$$

so that

$$fl\,(x^2 + y^2) = (x^2 + y^2)\,(1+2\epsilon),$$

if ϵ^2 is neglected. Then

$$\sqrt{fl\,(x^2 + y^2)} = \sqrt{x^2 + y^2}\,(1+\epsilon),$$

so that

$$fl\left(\sqrt{x^2 + y^2}\right) = \sqrt{x^2 + y^2}\,(1+\epsilon+\epsilon_{\mathrm{sqrt}})$$

if small multiples of ϵ^2 are neglected.

Thus the relative error in the final result is bounded by $E + E_{\mathrm{sqrt}}$ if small multiples of E^2 are neglected. It is a straightforward matter to check that the error cannot exceed this bound in any of the paths through the handler, unless of course overflow occurs in the final unscaling. The bound is therefore valid for all values of the input argument, as long as *overflow* is not returned.

```
function CABS (z : complex ) : real
    possible exceptions (overflow )
    real x, y, answer, scaledx, scaledy, scaledanswer
    integer logbx, logby
    integer precision ← appropriately initialized
    x := z.realpart    y := z.imagpart
    enable  - - try the simplest formula - it will work most of the time
        answer := sqrt(x**2 + y**2)
    handle  - - overflow or underflow has occurred
        if x = 0 or y = 0 then
            answer := abs(x) + abs(y)
        else
            logbx := logb(x)    logby := logb(y)
            if 2 * abs(logbx − logby) > precision + 1 then
                  - - exponents are so different that one of x and y
                  - - can be ignored
                answer := max(abs(x), abs(y))
            else  - - scale - we scale so that abs(x) is near 1
                scaledx := scalb(x, −logbx)
                scaledy := scalb(y, −logbx)
                scaledanswer := sqrt(scaledx**2 + scaledy**2)
                enable  - - now unscale if possible - this might overflow
                    answer := scalb(scaledanswer, logbx)
                handle  - - must be overflow in scalb
                    return overflow
                end
            endif
        endif
    end
    return answer
end CABS
```

Fig. 3. This program for the absolute value function first attempts to approximate $\sqrt{x^2+y^2}$ directly, which is efficient and reasonably accurate, and is almost always successful. But, if overflow or underflow occurs in this attempt, the handler takes over and manages to avoid any spurious overflows or underflows, usually by scaling; however, the final result can still overflow in very exceptional cases when the scaling is undone – then *overflow* is returned.

```
function CABS2 (z : complex ) : real
    possible exceptions (overflow )
    real x, y, answer, maxmag, temp
    x := z.realpart     y := z.imagpart
    enable
        answer := sqrt(x**2 + y**2)
    handle  - - must be overflow or underflow
        maxmag := max(abs(x), abs(y))
        enable
            temp := sqrt((x/maxmag)**2 + (y/maxmag)**2)
        handle  - - must be underflow
            temp := 1
        end
        enable
            answer := maxmag * temp
        handle  - - must be overflow
            return overflow
        end
    end
    return answer
end CABS2
```

Fig. 4. An alternative to what is in Figure 3. With this approach the error bound is somewhat larger.

To conclude this section we will mention another, very different approach which leads to the alternative program shown in Figure 4. It should be noted that the error bound in this approach is somewhat larger, namely $2.25E + E_{\text{sqrt}}$.

4.2 Complex Square Root CSQRT

We now consider the square root function $\sqrt{z}$, where $z = x + i\,y$. If we write $z = re^{i\theta}$, we have

$$\begin{aligned}\sqrt{z} &= \sqrt{r}\,e^{i\theta/2} \\ &= \sqrt{r}\,\cos(\theta/2) + i\,\sqrt{r}\,\sin(\theta/2) \\ &= \sqrt{r}\,\sqrt{\tfrac{1+x/r}{2}} + i\,\sqrt{r}\,\sqrt{\tfrac{1-x/r}{2}} \\ &= \sqrt{\tfrac{r+x}{2}} + i\,\sqrt{\tfrac{r-x}{2}},\end{aligned}$$

where we will adopt the conventions that the real part is non-negative, and, if y is zero, the imaginary part is also non-negative but otherwise its sign is the same as the sign of y. Note that neither component can overflow, so the function cannot overflow. Furthermore, if one of the components underflows, this component can be set to zero without altering the error bound by more than a small multiple of E^2, so that, according to the criterion described in Subsection 2.5, the function cannot underflow.

To avoid loss of accuracy due to cancellation, we rewrite $\sqrt{r+x}$ as $|\,y\,|/\sqrt{r+|\,x\,|}$ when $x < 0$, and $\sqrt{r-x}$ as $y/\sqrt{r+x}$ when $x > 0$. Then, if we set $t = \sqrt{2\,(r+|\,x\,|)} = \sqrt{2\,(\sqrt{x^2+y^2}+|\,x\,|)}$, we finally obtain

$$\sqrt{z} = \begin{cases} t/2 + i\,y/t, & x > 0 \\ \sqrt{|\,y\,|}\,/\sqrt{2} + i\,\mathrm{sign}(y)\sqrt{|\,y\,|}\,/\sqrt{2}, & x = 0 \\ |\,y\,|/t + i\,\mathrm{sign}(y)\,t/2, & x < 0, \end{cases}$$

where $\mathrm{sign}(0) = +1$, but otherwise $\mathrm{sign}(y) = \pm 1$ according to the sign of y. In this form the mathematical specification satisfies the sign conventions stated above and it also avoids any possibility of loss of accuracy due to cancellation. By writing $\sqrt{|\,y\,|}/\sqrt{2}$ in place of $\sqrt{|\,y\,|/2}$ it also avoids the possibility of underflow in the latter form.

There are two remaining difficulties which are taken into account in the program in Figure 5. One is to avoid any spurious overflows or underflows in evaluating an approximation to t. This is done in a way analogous to what was done for cabs: if such an exception occurs, the handler first looks after the cases when x or y is zero; then, if their exponents differ by enough (depending on which has the larger exponent), the smaller of $|\,x\,|$ and $|\,y\,|$ can be ignored, and the corresponding expressions for t are quite simple; finally, in all other cases, scaling can be used to avoid the exceptions – but the scaling must be done in terms of an even exponent so that the final unscaling is in terms of an integer exponent. Because two square root operations are required, the final approximation for t cannot overflow, unlike the case for the final approximation in cabs.

```
function CSQRT (z : complex ) : complex
   possible exceptions (none )
   real x, y, t, scaledx, scaledy, scaledt, temp
   real sqrt2 ← appropriately initialized
   integer logbx, logby, evennearlogbx
   integer precision ← appropriately initialized
   complex answer
   x := z.realpart    y := z.imagpart
   enable
      t := sqrt(2 * (sqrt(x**2 + y**2) + abs(x)))
      if x > 0 then
         answer.realpart := t/2
         answer.imagpart := y/t
      elsif x < 0
         answer.realpart := abs(y)/t
         answer.imagpart := sign(y) * t/2
      else
         temp := sqrt(abs(y))/sqrt2
         answer.realpart := temp
         answer.imagpart := sign(y) * temp
      endif
   handle  - - overflow or underflow has occurred
      if x = 0 then
         temp := sqrt(abs(y))/sqrt2
         answer.realpart := temp
         answer.imagpart := sign(y) * temp
      elsif y = 0 then
         if x > 0 then
            answer.realpart := sqrt(x)
            answer.imagpart := 0
         else
            answer.realpart := 0
            answer.imagpart := sqrt(−x)
         endif
```

```
else  - - determine t
    logbx := logb(x)    logby := logb(y)
    if logby − logbx > precision then
        - - x can be ignored
        t := sqrt2 ∗ sqrt(abs(y))
    elsif 2 ∗ (logbx − logby) > precision + 1 then
        - - y can be ignored
        t := 2 ∗ sqrt(abs(x))
    else  - - scale and unscale - we scale so that
        - - abs(x) is near 1, with even exponent
        evennearlogbx := logbx + logbx mod 2
        scaledx := scalb(x, −evennearlogbx)
        scaledy := scalb(y, −evennearlogbx)
        scaledt := sqrt(2 ∗ (sqrt(scaledx∗∗2
                    +scaledy∗∗2) + abs(scaledx)))
        t := scalb(scaledt, evennearlogbx div 2)
    endif
    if x > 0 then
        answer.realpart := t/2
        enable
            answer.imagpart := y/t
        handle  - - underflow has occurred
            answer.imagpart := 0
        end
    elsif x < 0 then
        enable
            answer.realpart := abs(y)/t
        handle  - - underflow has occurred
            answer.realpart := 0
        end
        answer.imagpart := sign(y) ∗ t/2
    endif
  endif
 end
 return answer
end CSQRT
```

Fig. 5. This program for the square root function first attempts to approximate $\sqrt{2(\sqrt{x^2+y^2}+|x|)}$ directly, and, if successful, the final approximation to $\sqrt{z}$ is obtained according to whether $x > 0$, $x < 0$, or $x = 0$. But, if overflow or underflow occurs in this attempt, the handler takes over and manages to avoid any such overflow or underflow. There are two places in the handler where underflow might still occur – but if it does, the component involved is set to 0.

Once the approximation to t has been obtained, the approximations to the real and imaginary parts of $\sqrt{z}$ are easily obtained, depending on whether x is greater than or less than zero. The only remaining difficulty is that, in the handler only, the approximation to $\mid y \mid / t$ might underflow. If this should occur, the program in Figure 5 replaces the underflowed value with zero. It is safe to do this and still preserve the error bound obtained below, since in any such case, the other part of the approximation to $\sqrt{z}$ (real part if $x > 0$ and imaginary part if $x < 0$) is always well above the level required in Subsection 2.5 for such situations.

For the error analysis, we first consider the case when no exception occurs. We can begin with the final result for cabs, namely,

$$fl\left(\sqrt{x^2+y^2}\right) = \sqrt{x^2+y^2}(1+\epsilon+\epsilon_{\mathrm{sqrt}}).$$

Then

$$fl\left(\sqrt{x^2+y^2} + \mid x \mid\right) = (\sqrt{x^2+y^2} + \mid x \mid)(1+2\epsilon+\epsilon_{\mathrm{sqrt}}),$$

so that

$$fl\left(2(\sqrt{x^2+y^2} + \mid x \mid)\right) = 2(\sqrt{x^2+y^2} + \mid x \mid)\,(1+2\epsilon+\epsilon_{\mathrm{sqrt}}),$$

on binary machines (another ϵ is needed on non-binary machines). Then

$$\sqrt{fl\left(2(\sqrt{x^2+y^2} + \mid x \mid)\right)} = \sqrt{2(\sqrt{x^2+y^2} + \mid x \mid)}\,(1+\epsilon+0.5\epsilon_{\mathrm{sqrt}}),$$

so that

$$fl\,(t) = \sqrt{2(\sqrt{x^2+y^2} + \mid x \mid)}\,(1+\epsilon+1.5\epsilon_{\mathrm{sqrt}}),$$

on binary machines (another 0.5ϵ is needed on non-binary machines). Of course, all of the above is assuming that small multiples of ϵ^2 can be neglected.

On binary machines there is a further division by t in one of the components of the final result when $x \neq 0$, which means that the relative error in that component is $2\epsilon + 1.5\epsilon_{\mathrm{sqrt}}$. The error in each component when $x = 0$ is only $2\epsilon + \epsilon_{\mathrm{sqrt}}$, assuming the constant $sqrt2$ is initialized to be within $1+\epsilon$ of $\sqrt{2}$. It is also easily seen that the errors in any path through the exception handler cannot be greater. Thus we can conclude that, in the notation of Subsection 2.1, one of E_r and E_i can be $E + 1.5E_{\mathrm{sqrt}}$ while the other is $2E + 1.5E_{\mathrm{sqrt}}$, so that their maximum,

$$2E + 1.5E_{\mathrm{sqrt}},$$

is a bound for the relative error in the approximation to $\sqrt{z}$, neglecting small multiples of E^2, on binary machines.

This bound can be tightened somewhat by examining the error formula in more detail. From Subsection 2.1, we have

$$\left|\frac{f^c - f}{f}\right| \leq \sqrt{\frac{E_r^2 f_r^2 + E_i^2 f_i^2}{f_r^2 + f_i^2}},$$

which, for $x > 0$, gives

$$\left| \frac{f^c - f}{f} \right| \le \sqrt{\frac{(E + 1.5E_{\text{sqrt}})f_r^2 + (2E + 1.5E_{\text{sqrt}})f_i^2}{f_r^2 + f_i^2}}.$$

The right hand side of this inequality reaches its maximum value when $\mid f_i/f_r \mid$ is as large as possible. However, for $x > 0$, $\mid f_i/f_r \mid < 1$, so that

$$\left| \frac{f^c - f}{f} \right| \le \sqrt{2.5E^2 + 4.5EE_{\text{sqrt}} + 2.25E_{\text{sqrt}}^2},$$

which is less than $2E + 1.5E_{\text{sqrt}}$. The result is the same for $x < 0$. The bound is even smaller when $x = 0$. Thus we conclude that the relative error in the approximation to $\sqrt{z}$ is bounded by

$$\sqrt{2.5E^2 + 4.5EE_{\text{sqrt}} + 2.25E_{\text{sqrt}}^2},$$

neglecting small multiples of E^2, on binary machines.

On non-binary machines there is an extra 0.5ϵ in the bound for t. There is also the same further division by t in one component, as well as a division by 2 in the other component, when $x \ne 0$, so that the error bounds for both components of the result are equal. The final relative error bound for non-binary machines is therefore $2.5E + 1.5E_{\text{sqrt}}$, neglecting small multiples of E^2.

4.3 Complex Exponential CEXP

The complex exponential function of $z = x + i\,y$ is easily expressed in terms of real elementary functions of x and y. The relationship is

$$e^z = e^x \cos(y) + i\, e^x \sin(y),$$

and this form leads immediately to the enable block in the program of Figure 6.

In the handler, $\exp(x)$ is first tested for overflow. If it does, then *overflow* is returned. Admittedly, this ignores a narrow "fringe" of possible values of x and y where the overflow could be avoided, namely those values of x and y such that $\exp(x)$ overflows by such a small amount that multiplication by $\cos(y)$ and $\sin(y)$ would produce values that are just below the overflow level.

We have chosen to ignore these fringe values as not being worth the trouble to detect and reinstate. However, if one wishes to include them, one possibility would be to develop a special procedure to return separately the significand s and exponent e of $\exp(x)$, and to use the results of this procedure to determine $scalb(s * \cos(y), e)$ and $scalb(s * \sin(y), e)$. Then overflow can occur only if one or both components do actually overflow.

Otherwise the handler must cope with underflow. If $\exp(x)$ underflows, or even if only $\exp(x) * \max(\mid \cos(y) \mid, \mid \sin(y) \mid)$ underflows, then *underflow* must be returned. Otherwise both components do not underflow, so only one does, and our agreed upon

```
function CEXP (z : complex ) : complex
    possible exceptions (overflow , underflow )
    real   x, y, expx, cosy, siny, m, M, temp
    real   E ← appropriately initialized
    complex answer
    x := z.realpart     y := z.imagpart
    enable
        expx := exp(x)
        answer.realpart := expx * cos(y)
        answer.imagpart := expx * sin(y)
    handle  - - overflow or underflow has occurred
        enable
            expx := exp(x)
        handle
            if x > 0 then
                return overflow
            else
                return underflow
            endif
        end
        cosy := cos(y)     siny := sin(y)
        M := max(abs(cosy), abs(siny))
        enable
            temp := expx * M
        handle  - - both components underflow
            return underflow
        end
         - - only one component underflows
        m := min(abs(cosy), abs(siny))
        if m ≤ M * E then
             - - the underflowed component can be set to zero
            if abs(cosy) > abs(siny) then
                answer.realpart := expx * cosy
                answer.imagpart := 0
            else
                answer.realpart := 0
                answer.imagpart := expx * siny
            endif
        else
            return underflow
        endif
    end
    return answer
end CEXP
```

Fig. 6. In this program for the complex exponential function, the handler first deals with overflow or underflow of $exp(x)$. Then, if only one component underflows, it determines whether or not that one component can be safely set to zero.

test, which in this case is whether

$$\min(|\cos(y)|, |\sin(y)|) \leq \max(|\cos(y)|, |\sin(y)|)E,$$

is used to determine whether the result after setting the underflowed component to 0 can be accepted. Otherwise *underflow* is returned.

The error analysis is straightforward. In the absence of any exceptions, the relative error in the real part is $\epsilon + \epsilon_{\mathrm{exp}} + \epsilon_{\mathrm{cos}}$, and in the imaginary part it is $\epsilon + \epsilon_{\mathrm{exp}} + \epsilon_{\mathrm{sin}}$, which gives an overall bound of $E + E_{\mathrm{exp}} + E_{\mathrm{sincos}}$, neglecting small multiples of E^2. The bound is somewhat larger in the case where one underflowed part is set to zero. The relative error in such an underflowed part is 1, and the relative error in the overall approximation is then bounded by

$$\begin{aligned}
&\sqrt{\frac{(1)^2(e^x\min(|\cos(y)|, |\sin(y)|))^2 + (E + E_{\mathrm{exp}} + E_{\mathrm{sincos}})^2(e^x\max(|\cos(y)|, |\sin(y)|))^2}{(e^x\cos(y))^2 + (e^x\sin(y))^2}} \\
\leq\ &\sqrt{\frac{(1)^2(e^xE)^2 + (E + E_{\mathrm{exp}} + E_{\mathrm{sincos}})^2(e^x)^2}{(e^x)^2}} \\
=\ &\sqrt{E^2 + (E + E_{\mathrm{exp}} + E_{\mathrm{sincos}})^2},
\end{aligned}$$

neglecting small multiples of E^2.

Hence, in all cases, the overall relative error in the approximation of e^z is bounded by

$$\sqrt{E^2 + (E + E_{\mathrm{exp}} + E_{\mathrm{sincos}})^2},$$

neglecting small multiples of E^2, and provided that *overflow* or *underflow* is not returned. This bound turns out to be about 5% more than $E + E_{\mathrm{exp}} + E_{\mathrm{sincos}}$ on the system we use for testing.

4.4 Complex Natural Logarithm CLOG

The complex natural logarithm of $z = x + iy = re^{i\theta}$ can be expressed in terms of its components as follows:

$$\begin{aligned}
\log(z) &= \log(r) + i\theta \\
&= \tfrac{1}{2}\log(x^2 + y^2) + i\arg(z),
\end{aligned}$$

where $-\pi < \theta \leq \pi$. (The arg function can be approximated by Fortran's ATAN2(y, x).)

To evaluate the components of this function, it is convenient to first introduce M and m, the maximum and minimum of $|x|$ and $|y|$, respectively. Then, if $M = 0$, an exception must be returned, which is designated as *domainerror* in the program in Figure 7.

The imaginary part of the logarithm function is easily calculated. The arg function might underflow, but then the real part is simply $\log(M)$ and it can be shown that the underflowed part can be set to zero without any significant increase in the error bound.

The main difficulties with the complex log function arise in evaluating the real part.

The program first deals with the case when $m = 0$, and the real part is again simply $\log(M)$. Then the most serious difficulty occurs when $x^2 + y^2$ or, what is the same thing, $M^2 + m^2$, is near 1, for then an accurate approximation to the log function would require an accurate approximation to $M^2 + m^2 - 1$, and this cannot be obtained directly because of possible serious cancellation.

We will postpone dealing with this difficulty, for the time being, and first consider only those cases where M is not in the interval $(1/2, \sqrt{2})$. (This ensures that $M^2 + m^2$ is not in the interval $(1/2, 2)$.) Then the remaining difficulty is that spurious overflow or underflow may occur in the evaluation of $M^2 + m^2$. But we can deal with such exceptions in a way analogous to what we did for cabs and csqrt. As indicated in the program, m can be ignored if the exponents of M and m are sufficiently different – and the resulting real part is simply $\log(M)$. Otherwise, scaling can be used to avoid overflow or underflow. If $M^2 + m^2$ is scaled by a factor $radix^{-scale}$, the logarithm must be corrected by adding $scale \times \log(radix)$ to the logarithm of the scaled value of $M^2 + m^2$. It is important that the scaling be chosen so that there is no possibility of cancellation in this addition.

Now let us consider the case where $1/2 < M < \sqrt{2}$. In this case $1/4 < M^2+m^2 < 4$, and we want to calculate an accurate approximation to

$$M^2 + m^2 - 1 = (M - 1)(M + 1) + m^2.$$

The best way to do this is to evaluate the expression $(M^2-1)+m^2$ in double the working precision and then round the result back to working precision. If the evaluation is done in the working precision, the error analysis becomes more complicated, as we shall see, and it turns out that the error in the real part can be enormous, although the overall error bound turns out to be less than double the quite modest bound we obtain for the case when doubled precision is used. (If doubled precision is not available, it could be simulated, but, since the simulation is likely to be extremely slow, it may be worthwhile avoiding the simulation in those cases where there is no cancellation – *i.e.,* when $M \geq 1$, or where the cancellation is not very serious – *i.e.,* when $M^2 - 1$ and m^2 are not very close in magnitude.)

We turn now to an error analysis for the program in Figure 7, and at first we will assume that the arithmetic is binary. The error bound for the imaginary part is of course E_{arg}, unless the imaginary part underflows. If this underflow does occur, the imaginary part can safely be set to zero, as we have already indicated, and, since the real part is simply $\log(M)$, with an error bound of $E_{\log}$, an overall relative error bound is $\max(E_{\log}, E_{\text{arg}})$.

Otherwise, for the real part we consider first the case where $M \leq 1/2$ or $M \geq \sqrt{2}$, and no exception occurs. Here we have

$$\begin{aligned} fl\left(\tfrac{1}{2}\log(M^2+m^2)\right) &= 0.5\log((M^2+m^2)(1+2\epsilon))(1+\epsilon_{\log}) \\ &= 0.5\log(M^2+m^2)\left(1+\frac{2\epsilon}{\log(M^2+m^2)}+\epsilon_{\log}\right) \end{aligned}$$

if we neglect small multiples of ϵ^2. And, since $M^2 + m^2 \geq 2$ or $M^2 + m^2 \leq 1/2$, we know that $|\log(M^2 + m^2)| \geq \log 2$, so that the relative error bound for the real part

```
function CLOG ( z : complex ) : complex
    possible exceptions (domainerror)
    real x, y, M, m, scaledM, scaledm, scaledr
    integer scale
    complex answer
    real sqrt2, logradix ← appropriately initialized
    integer precision ← appropriately initialized
    x := z.realpart    y := z.imagpart
    M := max(abs(x), abs(y))    m := min(abs(x), abs(y))
    if M = 0 then
        return domainerror
    endif
    enable
        answer.imagpart := arg(z)
    handle  - - must be underflow of arg
        answer.imagpart := 0    answer.realpart := log(M)
        return answer
    end
    - - now determine the real part
    if m = 0 then
        answer.realpart := log(M)
    elsif M ≤ 1/2 or M ≥ sqrt2 then
        enable
            answer.realpart := 0.5 * log(M**2 + m**2)
        handle  - - must be overflow or underflow in M**2 + m**2
            if 2 * (logb(M) − logb(m)) > precision + 1 then
                - - m can be ignored
                answer.realpart := log(M)
            else - - scale so that exceptions are avoided, and the two
                - - terms in answer.realpart are of the same sign
                if M > 1 then  - - must have been overflow
                    scale := logb(M)
                else  - - must have been underflow
                    scale := logb(M) + 2
                endif
                scaledM := scalb(M, −scale)    scaledm := scalb(m, −scale)
                scaledr := scaledM**2 + scaledm**2
                answer.realpart := scale * logradix + 0.5 * log(scaledr)
            endif
        end
    else  - - 1/2 < M < sqrt2
        enable  - - use doubled precision if possible in evaluating the argument of log1p
            answer.realpart := 0.5 * log1p((M − 1) * (M + 1) + m**2)
        handle  - - must be underflow in m**2
            answer.realpart := log(M)
        end
    endif
    return answer
end CLOG
```

Fig. 7. This program for the complex logarithm function first looks after three special cases – when the argument is zero, when the imaginary part underflows, and when $m = 0$. The real part is then calculated in a way depending on whether $\max(|x|, |y|)$ is outside the interval $(1/2, \sqrt{2})$ or not; in the first case scaling may be needed to cope with spurious overflow or underflow; in the second case the accuracy of the final result can be very sensitive to how accurately the argument of log1p is calculated.

is $2E/\log 2 + E_{\log}$, which is bounded by $2.886E + E_{\log}$, if we neglect small multiples of E^2.

When an exception does occur this bound is obviously still valid when no scaling is required. When scaling is required this bound is valid for the second term in the expression for the real part, while the first term, *scale* $*$ *logradix*, is bounded by $2E$, if *logradix* is stored as accurately as possible; since the two terms are of the same sign, the error in their sum is bounded by $2.886E + E_{\log}$. Thus, for all cases when $M \leq 1/2$ or $M \geq \sqrt{2}$, an overall relative error bound is $\max(2.886E + E_{\log}, E_{\arg})$, neglecting small multiples of E^2.

We now consider the case where $1/2 < M < \sqrt{2}$. If an exception occurs in this case, the real part is $\log(M)$ and the error bound is $E_{\log}$. Otherwise we need to determine how errors in the evaluation of $(M-1)(M+1) + m^2$ affect log1p of this expression. If this expression can be evaluated to within a factor of $1 + \eta$, then we have

$$\begin{aligned}
&fl\left(\tfrac{1}{2}\mathrm{log1p}\left((M-1)(M+1) + m^2\right)\right) \\
&= \; 0.5\log\left(1 + \left((M-1)(M+1) + m^2\right)(1+\eta)\right)(1+\epsilon_{\mathrm{log1p}}) \\
&= \; 0.5\log\left(M^2 + m^2 + (M^2 + m^2 - 1)\eta\right)(1+\epsilon_{\mathrm{log1p}}) \\
&= \; 0.5\log\left((M^2+m^2)\left(1 + \frac{M^2+m^2-1}{M^2+m^2}\eta\right)\right)(1+\epsilon_{\mathrm{log1p}}) \\
&= \; 0.5\log(M^2+m^2)\left(1 + \frac{M^2+m^2-1}{(M^2+m^2)\log(M^2+m^2)}\eta + \epsilon_{\mathrm{log1p}}\right)
\end{aligned}$$

if we neglect small multiples of η^2 and $\epsilon\eta$.

We know that $1/4 < M^2 + m^2 < 4$, and we can show that $\left|\dfrac{u-1}{u\log(u)}\right| < 2.165$ for $1/4 < u < 4$, so that $2.165H + E_{\mathrm{log1p}}$ is an error bound for the real part, where H is a bound for $|\,\eta\,|$.

If doubled precision is used, $M - 1$ and $M + 1$ are evaluated exactly, and so are $(M-1)(M+1)$ and m^2. The sum of these two expressions will suffer a rounding error of at most ϵ^2, so that the final rounding to working precision presents log1p with an argument in which $\eta = \epsilon$. The final relative error in the real part when $1/2 < M < \sqrt{2}$ is therefore bounded by $2.165E + E_{\mathrm{log1p}}$, if we neglect small multiples of E^2.

The overall relative error, for all M, is therefore bounded by

$$\max(2.886E + E_{\log}, 2.165E + E_{\mathrm{log1p}}, E_{\arg}),$$

if we neglect small multiples of E^2 and provided of course that *domainerror* is not returned. This completes the error analysis of the program in Figure 7 when the argument of log1p near the end of the program is calculated to within a factor of $1 + \epsilon$, which is the case when doubled precision is used to evaluate that argument. (The 2.886 and 2.165 in this bound must be replaced by 3.886 and 3.165, respectively, for non-binary machines because of the multiplication by 0.5 in the expressions being evaluated.)

When doubled precision is not used, the situation is quite a bit more complicated since serious cancellation may occur in the evaluation of $(M-1)(M+1)+m^2$. If $M=1$ there is obviously no cancellation, and the argument for log1p is simply m^2, which will be accurate to within a factor of $1+\epsilon$, so that $\eta=\epsilon$, as was the case with doubled precision. If $M>1$, there is also no cancellation, but the argument is now accurate only to within a factor of $1+3\epsilon$ (note that $M-1$ is exact), so that $\eta=3\epsilon$, which leads us to a bound which replaces the 2.165 above with 6.495. If $M<1$, there can be cancellation, but, if $4m^2 \leq \mid M^2-1 \mid$, the cancellation is not very serious: it can be shown that in this case the argument is accurate to within a factor of $1+3\epsilon$, so that $\eta=3\epsilon$, and the bound is therefore the same as it was for $M>1$.

This leaves us with a situation where more serious cancellation can take place. For this, we derive the following:

$$
\begin{aligned}
&fl\left(\tfrac{1}{2}\mathrm{log1p}\left((M-1)(M+1)+m^2\right)\right)\\
&\quad= 0.5\mathrm{log1p}\left[\left((M^2-1)(1+2\epsilon)+m^2(1+\epsilon)\right)(1+\epsilon)\right](1+\epsilon_{\mathrm{log1p}})\\
&\quad= 0.5\log\left[M^2+m^2+(M^2-1)2\epsilon+m^2\epsilon+(M^2-1+m^2)\epsilon\right](1+\epsilon_{\mathrm{log1p}})\\
&\quad= 0.5\log\left[(M^2+m^2)\left(1+\frac{(M^2-1)2\epsilon+m^2\epsilon+(M^2-1+m^2)\epsilon}{M^2+m^2}\right)\right](1+\epsilon_{\mathrm{log1p}})\\
&\quad= 0.5\log(M^2+m^2)+0.5\left(\frac{(M^2-1)2\epsilon+m^2\epsilon+(M^2-1+m^2)\epsilon}{M^2+m^2}\right)\\
&\qquad\qquad+\ 0.5\log(M^2+m^2)\epsilon_{\mathrm{log1p}},
\end{aligned}
$$

if we neglect small multiples of ϵ^2. From the second term in this expression we see that the relative error in the real part could be enormous (and this is confirmed by our tests in the next section). But we continue towards finding a relative error bound in the overall result. From the above we see that the absolute error in the real part is bounded by

$$
\begin{aligned}
&0.5\frac{\mid M^2-1\mid 2E+m^2E+(\mid M^2-1\mid+m^2)E}{M^2+m^2}+0.5\mid\log(M^2+m^2)\mid E_{\mathrm{log1p}}\\
&\qquad=0.5\frac{3\mid M^2-1\mid E+2m^2E}{M^2+m^2}+0.5\mid\log(M^2+m^2)\mid E_{\mathrm{log1p}},
\end{aligned}
$$

and, since $\mid M^2-1\mid < 4m^2$, this in turn is bounded by

$$
\frac{7m^2E}{M^2+m^2}+0.5\mid\log(M^2+m^2)\mid E_{\mathrm{log1p}}.
$$

This expression is a bound for $E_r\mid f_r\mid$ which can be substituted into the overall relative error bound developed in Subsection 2.1, namely

$$
\left|\frac{f^c-f}{f}\right| \leq \sqrt{\frac{E_r^2f_r^2+E_i^2f_i^2}{f_r^2+f_i^2}}.
$$

We can proceed as follows:

$$\begin{aligned}\left|\frac{f^c - f}{f}\right| &\le \frac{E_r |\, f_r \,|}{\sqrt{f_r^2 + f_i^2}} + E_i \\ &\le \frac{7m^2E}{(M^2+m^2)|\, f_i \,|} + \frac{0.5|\log(M^2+m^2)\,|E_{\mathrm{log1p}}}{|\, f_r \,|} + E_{\mathrm{arg}},\end{aligned}$$

so that we are considering the first part of the bound for $E_r |\, f_r \,|$ relative to $|\, f_i \,|$, but the second part relative to $|\, f_r \,|$. This will lead to an overall error bound that is acceptably small.

The second term is obviously E_{log1p}. For a bound on the first term, we consider two cases, one where $|\, y \,| \ge |\, x \,|$, and the other where $|\, y \,| < |\, x \,|$.

The imaginary part is $\arg(z)$ and, in the first case, $|\arg(z)\,| \ge \frac{\pi}{4}$, so that the contribution of the first part to the relative error is bounded by

$$\begin{aligned}\frac{28m^2E}{(M^2+m^2)\pi} &\le \frac{14E}{\pi} \\ &< 4.457E,\end{aligned}$$

since $m^2/(M^2+m^2) \le 0.5$ for $m \le M$.

In the second case, $|\arg(z)\,| = \arctan(m/M)$, so the contribution of the first part to the relative error is

$$\frac{7m^2E}{(M^2+m^2)\arctan(m/M)}$$

which also turns out to be $\le 4.457E$.

Thus, we end up with an overall relative error bound of $4.457E + E_{\mathrm{log1p}} + E_{\mathrm{arg}}$, for the case when serious cancellation can take place.

Collecting together all the results we have obtained for the program in Figure 7 when only working precision is used throughout, we have established the following overall relative error bound for the clog function:

$$\max(2.886E + E_{\mathrm{log}}, 6.495E + E_{\mathrm{log1p}}, 4.457E + E_{\mathrm{log1p}} + E_{\mathrm{arg}}),$$

if we neglect small multiples of E^2, and provided of course that *domainerror* is not returned. (Here each of 2.886, 6.495 and 4.457 must be increased by 1 for non-binary machines, because of the multiplication by 0.5 in the expressions being evaluated.)

4.5 Complex Sine CSIN

The sine function of $z = x + iy$ can be represented in terms of real elementary functions as follows:

$$\sin(z) = \sin(x)\cosh(y) + i\,\cos(x)\sinh(y)$$

and the program in Figure 8 is based on this formula.

If overflow occurs in the evaluation of the real and imaginary parts of this function, $\cosh(y)$ or $\sinh(y)$ is too large in magnitude (probably both), and the handler returns *overflow.* As in the case of cexp some "fringe" values of z (for which the real and

```
function CSIN (z : complex ) : complex
    possible exceptions (overflow )
    real x, y, coshy, sinhy
    complex answer
    x := z.realpart     y := z.imagpart
    enable
        answer.realpart := sin(x) * cosh(y)
        answer.imagpart := cos(x) * sinh(y)
    handle
        enable
            coshy := cosh(y)
            sinhy := sinh(y)
        handle  - - must be overflow
            return overflow
        end
        answer.realpart := sin(x) * coshy
        enable
            answer.imagpart := cos(x) * sinhy
        handle  - - must be underflow
            answer.imagpart := 0
        end
    end
    return answer
end CSIN
```

Fig. 8. This program for the complex sine function is straightforward, although it should be acknowledged that *overflow* can be returned for some "fringe" values of z whose corresponding real and imaginary parts are actually slightly below the overflow threshold.

imaginary parts of $\sin(z)$ are slightly below the overflow threshold, even though at least one of $\cosh(y)$ and $\sinh(y)$ alone do overflow) are neglected here. But, as with cexp, an auxiliary procedure that computes the exponent and fraction part of $\exp(y)$ separately could be used to avoid this situation, since $\sinh(y)$ and $\cosh(y)$ both effectively equal $\exp(y)/2$ in magnitude when these functions overflow.

If underflow occurs it can only occur in the multiplication associated with the imaginary part. But when that happens the real part is so much larger that the underflowed part can be set to zero without any significant increase in the error bound.

An upper bound for the overall relative error in this function is

$$E + E_{\text{sincos}} + E_{\text{sinhcosh}},$$

neglecting small multiples of E^2, provided, of course, that *overflow* is not returned.

4.6 Complex Cosine CCOS

The cosine function of $z = x + iy$ can be represented in terms of real elementary functions as follows:

$$\cos(z) = \cos(x)\ \cosh(y) - i\ \sin(x)\ \sinh(y)$$

```
function CCOS (z : complex ) : complex
    possible exceptions (overflow )
    real x, y, coshy, sinhy
    complex answer
    x := z.realpart     y := z.imagpart
    enable
        answer.realpart := cos(x) * cosh(y)
        answer.imagpart := −sin(x) * sinh(y)
    handle
        enable
            coshy := cosh(y)
            sinhy := sinh(y)
        handle - - must be overflow
            return overflow
        end
        answer.realpart := cos(x) * coshy
        enable
            answer.imagpart := −sin(x) * sinhy
        handle - - must be underflow
            answer.imagpart := 0
        end
    end
    return answer
end CCOS
```

Fig. 9. This program for the complex cosine function is straightforward, although it should be acknowledged that *overflow* can be returned for some "fringe" values of z whose corresponding real and imaginary parts are slightly below the overflow threshold.

and the program in Figure 9 is based on this formula.

If overflow occurs $\cosh(y)$ or $\sinh(y)$ is too large in magnitude (probably both), and, as with csin in the previous section, the handler returns *overflow* . But also as with csin, some "fringe" values of z are neglected here, although they could be included with the help of an auxiliary procedure that returns the exponent and fraction part of $\exp(y)$ separately.

If underflow occurs it can only occur in the multiplication associated with the imaginary part. But when that happens the real part is so much larger that the underflowed part can be set to zero without any significant increase in the error bound.

An upper bound for the overall relative error in this function is

$$E + E_{\text{sincos}} + E_{\text{sinhcosh}},$$

neglecting small multiples of E^2, provided, of course, that *overflow* is not returned.

5. SPECIAL IMPLEMENTATIONS AND TESTING

We have implemented the algorithms presented in Section 4 on a Sun 4/40 in Fortran 77 (compiler version 1.4), in order to test their correctness, especially the correctness of their error bounds. These implementations are special in the sense that they are as

Table I. Observed error bounds for the single precision real elementary functions in the Sun library (version 1.4), in units of E, the relative error bound for single precision real arithmetic. The result given for E_{sincos} is for $|\,x\,| < 10^6$.

E_{sqrt}	E_{exp}	E_{sincos}	E_{arg}	E_{log}	E_{log1p}	E_{sinhcosh}
1.000	1.152	1.102	2.326	1.382	1.000	1.000

close as possible to the pseudo-code descriptions of Section 4, and in particular have not been modified in any way to improve their efficiency. (Efficiency and other production implementation issues will be discussed in the next section.)

Except for a portion of one version of clog, which was an alternative suggested in Subsection 4.4, the floating point operations in these implementations are in single precision. (Some care had to be taken to make sure this was the case! We examined the generated assembly language instructions to ensure that no extended precision was used in any intermediate calculations.) This enabled us to use the corresponding double precision results from the Sun system as the "true" results for test purposes. We assume that these "true" results are correct to single precision accuracy.

The exception handling construct was implemented by allowing the enable block to be executed and then testing for which exception flags had been raised, as is possible in an IEEE environment.

It would have been natural, with our interpretation, to use the "`ieee_handler`" trap-handling facility provided by the Sun system [6, p. 67] but it turned out to be both inefficient and somewhat difficult to use. Testing the floating point exception flags using the "`ieee_flags`" subroutine [6, p. 64] is also inefficient, so we instead accessed the flags by using the math library routine "`swapEX_`".

To test the correctness of the error bounds for the examples in Section 4, we must first determine what those bounds turn out to be for the system we are using, and this requires the determination of E_{sqrt}, E_{exp}, etc. It is convenient to present these in units of E, and the results are given in Table I. These relative error bounds were determined by examining all relevant single precision arguments, except that in the case of E_{sincos} we considered only values of the argument which were less than 10^6 in absolute value, and in the case of E_{arg} we determined the bound by adding E to E_{atan}, where the extra E allows for the additional error induced by using $fl\,(y/x)$ as the argument of atan in place of y/x. The boundaries of the arg function, where at least one of x or y is 0 or $\pm\infty$, were also considered. (The bounds for log1p and sinhcosh are $1.000E$ because the single precision versions of these functions use correctly rounded results from their double precision implementations. The bound for our version of log1p in Figure 1, which uses only single precision, is $2.198E$.)

The results in Table I are used to determine the theoretical bounds in column 3 of Table II. The observed bounds in column 4 of Table II were obtained by comparing the results from our implementations with the "true" results provided by the Sun system's double precision functions for a large number of mostly random input arguments. The IEEE single precision arguments at which these observed bounds occurred are given in hexadecimal form in column 5.

The random arguments were constructed from random real parts and random imag-

Table II. A comparison of theoretical and observed relative error bounds for Sun Fortran (version 1.4) implementations of the complex elementary function programs in Figures 3–9. Columns 3 and 4 are in units of E, the relative error bound for real arithmetic. SP and PDP stand for IEEE single precision and partial IEEE double precision, respectively. (For cexp, $\text{real}\,(z) < 10^6$, and for csin and ccos, $\text{imag}\,(z) < 10^6$.)

Function	Theoretical Bound $\longrightarrow$	Based on Table I	Observed Bound	at this argument
cabs	$E + E_{\text{sqrt}}$	2.000	1.962	$\text{5c80ad41} + i\text{da86776f}$
cabs2	$2.25E + E_{\text{sqrt}}$	3.250	2.495	$\text{e0723701} + i\text{60efff95}$
csqrt	$\sqrt{2.5E^2 + 4.5EE_{\text{sqrt}} + 2.25E^2_{\text{sqrt}}}$	3.042	2.980	$\text{3b11897f} + i\text{3f012faf}$
cexp	$\sqrt{E^2 + (E + E_{\text{exp}} + E_{\text{sincos}})^2}$	3.405	2.815	$\text{bea5ebce} + i\text{3f490421}$
clog				
(SP)	$\max\,(2.886E + E_{\log},$ $6.495E + E_{\text{log1p}},$ $4.457E + E_{\text{log1p}} + E_{\text{arg}})$	7.783	5.047	$\text{3f000005} + i\text{3c1a387b}$
(PDP)	$\max\,(2.886E + E_{\log},$ $2.165E + E_{\text{log1p}}, E_{\text{arg}})$	4.268	3.601	$\text{3fb505d3} + i\text{3ca34a89}$
csin	$E + E_{\text{sincos}} + E_{\text{sinhcosh}}$	3.102	3.016	$\text{40c0f2f4} + i\text{3bc59845}$
ccos	$E + E_{\text{sincos}} + E_{\text{sinhcosh}}$	3.102	3.035	$\text{48914bef} + i\text{3be55c78}$

inary parts, by generating random exponents and random significands within appropriate ranges. Ten thousand such random arguments were generated for each of the 4 semi-axes, and $100,000,000$ were generated for each of the 4 quadrants. The origin was also tested. The regions were restricted where necessary so that overflow and underflow would be avoided most of the time, while at the same time ensuring good coverage of the proper domain of each function. Many special cases were also tested, including many near the boundaries of the regions that separated points which would probably lead to exceptions being returned and points which would probably not lead to exceptions being returned. The most important special cases in terms of trying to observe large errors were those arguments where real and imaginary parts were chosen to maximize the errors in the relevant real elementary functions; in fact, these special cases produced most of the observed maximums — so much for random testing!

As can be seen from Table II, all non-exceptional results were in error by no more than what would be expected on the basis of the theoretical error bounds. In fact, the theoretical bounds are not much larger than the observed bounds. The sort of discrepancy shown in the table is not surprising, considering the kinds of reasoning used in determining theoretical error bounds, especially in the case of the two clog functions.

(It happens that most of the relatively large discrepancy in the case of cabs2 can be explained. The IEEE arithmetic in our system satisfies more than what we assumed in Subsection 2.1. It is accurate to within half a unit in the last place. This can make a significant difference to the error analysis in cabs2 because of the special form of an expression such as $1 + (y/x)^2$. It can be argued that the contribution from such an expression to the final relative error bound in cabs2 is a maximum when its value is just greater than 1.5. We have followed this argument through to obtain finally a relative error bound of only 2.650 for cabs2, in place of 3.250.)

Special mention should be made of the results for clog. The version that makes use of double precision in the computation of the argument for log1p is not very much more accurate in terms of its observed overall relative error bound than the version that uses only single precision. However the real part of the former is much more accurate than the real part of the latter — the observed bound is $3.604E$ in the former but approximately $5 \times 10^6 E$ in the latter! — this enormous error occurred at 3f7ffffc + i3a3504f3. (The observed error bounds in their imaginary parts were equal, only $2.292E$.)

Of course we do not claim that these experimental results actually prove the correctness of the theoretical error bounds, but we do believe the evidence is very convincing. However, correctness of the programs involves more than correctness of the error bounds for values of the argument which do not lead to exceptions being returned. It must also be true that exceptions are returned only when it is reasonable to do so. (Otherwise a program could be considered correct if it always returned an exception, no matter what the input argument!)

According to Subsection 2.5, overflow is to be returned when either component overflows. If one of our function programs does return overflow, for a particular value of the input argument, our test program considers this to be a correct return if at least one of the components of the "true" result is within the relative error bound of a true overflow, specifically if

$$\max(|f_r|, |f_i|)(1 + E_{\mathrm{f}}) \geq HUGE + ulpup/2,$$

where $HUGE$ is the largest machine representable number, and $ulpup$ is a unit in the last place of $HUGE$ in the direction of ∞.

The test for an underflow return to be correct is that either

(1) both components of the "true" result are within the relative error bound of a true underflow, specifically if

$$\max(|f_r|, |f_i|)(1 - E_{\mathrm{f}}) < TINY - ulpdown/2,$$

or

(2) one component underflows, but is nevertheless within an error bound of being greater than a value that is within an error bound of the other component, specifically if

$$\min(|f_r|, |f_i|)(1 - E_{\mathrm{f}}) < TINY - ulpdown/2$$

and

$$\min(|f_r|, |f_i|)(1 + E_{\mathrm{f}}) > \max(|f_r|, |f_i|)(1 - E_{\mathrm{f}})E,$$

where $TINY$ is the smallest positive machine representable number, and $ulpdown$ is a unit in the last place of $TINY$ in the direction of 0.

These criteria had to be modified slightly, in an obvious way, to allow for the fact that "fringe" areas were neglected in the algorithm for cexp, csin and ccos.

In each test case for which *overflow* or *underflow* was returned, our test program determined that the appropriate criterion was satisfied.

The exceptional return of *domainerror* in clog is a special case which was easy to check separately.

6. PRODUCTION IMPLEMENTATIONS

The exception handling construct in Ada is reasonably close to what we have used in this paper, except for the unfortunate fact that Ada does not recognize underflow as an exception. Our construct is simpler than Ada's in that we do not expect the handler to be told, in effect, what exceptions have occurred. A proposal somewhat like Ada's, but which did recognize underflow, was proposed for Fortran 90 — but was rejected in favor of having no exception handling facility at all! Our construct could be implemented in PL/1, but not very elegantly.

Extensions to existing languages often provide facilities for implementing our construct, for example by providing access to trapping and exception flags in IEEE environments — as we have already indicated in the preceding section.

In the absence of exception handling facilities, pretesting can be used. For example, in the case of cabs, $|\, x \,|$ and $|\, y \,|$ can be tested in advance to make sure that no overflow or underflow will occur in the evaluation of $x^2 + y^2$ and, if that is the case, evaluate the expression $\sqrt{x^2 + y^2}$, but otherwise carry out the calculations in the handler, using another pretest to determine whether or not the final unscaling will cause overflow. In such an implementation, the original pretesting will determine if conditions are satisfied which are *sufficient* to ensure that no spurious exceptions will occur. They may not be necessary, and in such cases this will cause the program to execute the handler more often than is necessary.

Of course, if higher precision is available, it may be possible to avoid some pretesting, as long as the exponent range in that higher precision is sufficiently broader. For example, again for cabs, the entire calculation of $\sqrt{x^2 + y^2}$ can be done in higher precision, and a test needs to be performed only when the result of this calculation is about to be coerced to the original precision. But this idea is of little use if the working precision is already the highest available.

Apart from trying to implement exception handling in a more efficient way, our implementations can be made slightly more efficient, mostly by writing the programs so that some of the intermediate "store" operations are avoided.

In addition, the floating point status register would have to be saved on entry to the function subprograms, and later restored and updated to include the exception flag, if any, returned by the function.

7. CONCLUDING REMARKS

We have presented algorithms for reliable and accurate evaluations of the complex elementary functions required in Fortran 77 and Fortran 90. It was convenient to describe these algorithms with the help of an exception handling construct.

Implementations in Fortran for Sun systems have been tested extensively. The observed error bounds were between 64% and 98% of the theoretical bounds – which is not only convincing evidence of the correctness of the theoretical bounds, but also indicates that the theoretical bounds are quite tight. (It was interesting to discover that choosing arguments near where we thought the largest error might occur usually led to observed bounds that were larger than those found even by very extensive random testing – in one case by over 13%!) In the tests it was also found that exceptions were returned when, and only when, it was reasonable to do so.

ACKNOWLEDGMENTS

Much of this work was inspired by discussions with members of the Ada Numerics Working Group under the chairmanship of Gil Myers. Jim Cody was particularly helpful during early stages of our investigation. We also wish to thank the referees for helpful suggestions.

REFERENCES

1. American National Standard Programming Language FORTRAN. *ANSI X3.9–1978*. American National Standards Institute, Inc., New York, 1978.

2. Hull, T. E., Fairgrieve, T. F., and Tang, P. T. P. Implementing complex elementary functions using exception handling. Argonne National Laboratory, 9700 S. Cass Ave., Argonne, Illinois 60439-4844. Preprint MCS-P338-1192 (Jan. 1993).

3. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Standard 754–1985*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1985.

4. ISO/IEC 1539 : 1991, Information technology – Programming languages – Fortran. International Standards Organization, Geneva, 1991.

5. Kahan, W. Branch cuts for complex elementary functions, or Much ado about nothing's sign bit. In *The State of the Art in Numerical Analysis: Proceedings of the joint IMA/SIAM conference*, A. Iserles and M. J. D. Powell, Eds. Clarendon Press, Oxford, 1987, pp. 165–211.

6. Numerical Computations Guide. Part Number: 800-5277-10, Revision A. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100. (Feb. 22, 1991).

7. Payne, M. N., and Hanek, R. N. Radian reduction for trigonometric functions. *ACM SIGNUM Newsletter 18*, 1 (Jan. 1983), 19–23.

8. Tang, P. T. P. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Trans. Math. Softw. 16*, 4 (Dec. 1990), 378–400.