BRANCH CUTS
for
COMPLEX ELEMENTARY FUNCTIONS,
or
MUCH ADO ABOUT NOTHING'S SIGN BIT.

by

W. Kahan
Elect. Eng. and Computer Science.
and Mathematics Departments,
University of California.
Berkeley, CA 94720.
May 17, 1987

Abstract

Zero has a usable sign bit on some computers, but not on others. This accident of computer arithmetic influences the definition and use of familiar complex elementary functions like $\sqrt{\ }$, arctan and arccosh whose domains are the whole complex plane with a slit or two drawn in it. The Principal Values of those functions are defined in terms of the logarithm function from which they inherit discontinuities across the slit(s). These discontinuities are crucial for applications to conformal maps with corners. The behavior of those functions on their slits can be read off immediately from defining Principal Expressions introduced in this paper for use by analysts. Also introduced herein are programs that implement the functions fairly accurately despite roundoff and other numerical exigencies. Except at logarithmic branch points, those functions can all be continuous up to and onto their boundary slits when zero has a sign that behaves as specified by IEEE standards for floating-point arithmetic; but those functions must be discontinuous on one side of each slit when zero is unsigned. Thus does the sign of zero lay down a trail from computer hardware through programming language compilers, run-time support libraries and applications programmers to, finally, mathematical analysts.

Prepared for the joint IMA/SIAM Conference on "The State of the Art in Numerical Analysis" held at the University of Birmingham, England, April 14 - 18, 1986, for which the proceedings have been published in 1987 by the Oxford University Press, edited by M. J. D. Powell and A. Iserles. This is an augmented and corrected version that supersedes the paper in the proceedings.

Preamble:

In 1946 a long working day could be consumed by the creation and numerical inversion of an 8x8 matrix on the computing machine of that era, an electro-mechanical desk-top contraption that carried ten decimal digits. A 100x100 matrix was out of the question. Twenty years later both matrices could be handled in a fraction of a minute, at a cost well under a dollar, by an electronic computer that filled a room, carried about eight sig. dec., and took an hour to program. Now, after another twenty years, the 8x8 matrix can be entered and inverted in a shirt-pocket calculator, carrying ten sig. dec., in a few minutes spent almost entirely on input and output; the big 100x100 matrix can be inverted in a desk-top computer, carrying over sixteen sig. dec., in a few seconds at a cost under a cent. Measured by the obvious metrics,- speed, price and precision,- scientific computation has come a long way. Were these the only metrics that mattered, I should have nothing to say.

Other aspects of computation must have some subtle influence upon our lives because the cost of computation has not dropped so fast in the past two decades as the price of computer arithmetic might suggest. Programming costs almost as much now as it ever did, and has come to dominate the thoughts of many a scientist and engineer. Considering how much time we spend thinking about what the computer will do for us, we should be surprised if its ways did not alter our ways of thought a little. But who would expect the computer's treatment of the sign of zero to influence our thinking? In fact, the ways computers perform arithmetic can affect the way we think profoundly, much though we may wish it were the other way around.

BRANCH CUTS FOR COMPLEX ELEMENTARY FUNCTIONS

Introduction

Conventions dictate the ways nine familiar multiple-valued complex elementary functions, namely

$\sqrt{\ }$, ln, arcsin, arccos, arctan, arcsinh, arccosh, arctanh, z^w ,

shall be represented by single-valued functions called "Principal Values" . These single-valued functions are defined and analytic throughout the complex plane except for discontinuities across certain straight lines called "slits" so situated as to maximize the reign of continuity, conserving as many as possible of the properties of these functions' familiar real restrictions to apt segments of the real axis. There can be no dispute about where to put the slits; their locations are deducible. However, Principal Values have too often been left ambiguous on the slits, causing confusion and controversy insofar as computer programmers have had to agree upon their definitions. This paper's thesis is that most of that ambiguity can and should be resolved; however, on computers that conform to the IEEE standards 754 and 854 for floating-point arithmetic the ambiguity should not be eliminated entirely because, paradoxically, what is left of it usually makes programs work better.

What has to be ambiguous is the sign of zero. In the past, most people and computers would assign no sign to zero except under duress, and then they would treat the sign as + rather than − . For example, the real function

$$\begin{aligned} \text{signum}(x) &:= +1 \quad \text{if} \quad x > 0 \quad , \\ &:= \ 0 \quad \text{if} \quad x = 0 \quad , \\ &:= -1 \quad \text{if} \quad x < 0 \quad , \end{aligned}$$

illustrates the traditional non-committal attitude toward zero's sign, whereas the Fortran function

$$\begin{aligned} \text{sign}(1.0, x) &:= +1.0 \quad \text{if} \quad x \geq 0 \quad , \\ &:= -1.0 \quad \text{if} \quad x < 0 \quad , \end{aligned}$$

must behave as if zero had a + sign in order that this function and its first argument have the same magnitude. Just as sign(1.0, x) is continuous at $x = 0+$, i.e. as x approaches zero from the right, so can each principal value above be continuous as its slit is reached from one side but not from the other. Sides can be chosen in a consistent way among all the elementary complex functions, as they have been chosen for the implementations built into the Hewlett-Packard hp-15C calculator that will be used to illustrate this approach.

The IEEE standards 754 and 854 take a different approach. They prescribe representations for both +0 and −0 that are distinguishable bit patterns treated as numerically equal: $+0 = -0$, so the ambiguity is benign. Rather than think of +0 and −0 as distinct numerical values, think of their sign bit as an auxiliary variable that conveys one bit of information (or misinformation) about any numerical variable that takes on zero as its value. Usually this information is irrelevant: the value of $3 + x$ is the same for $x := +0$ as for $x := -0$, and likewise for the functions signum(x) and sin x. Usually it is irrelevant, [illegible]

extraordinary arithmetic operations *must be* affected by zero's sign; for example 1/(+0) = +∞ but 1/(-0) = -∞ . To retain its usefulness, the sign bit must propagate through certain arithmetic operations according to rules derived from continuity considerations; for instance (-3)*(+0) = -0 , (-0)/(-5) = +0 , (-0)-(+0) = -0 , etc. These rules are specified in the IEEE standards along with the one rule that had to be chosen arbitrarily:

s-s := +0 for every string s representing a finite real number. Consequently when t = s , but 0 ≠ t ≠ ∞ , then s-t and t-s both produce +0 instead of opposite signs. (That is why, in IEEE style arithmetic, s-t and -(t-s) are numerically equal but not necessarily indistinguishable.) Implementations of elementary transcendental functions like sin(z) and tan(z) and their inverses and hyperbolic analogs, though not specified by the IEEE standards, are expected to follow similar rules; if f(0) = 0 < f'(0) , then the implementation of f(z) is expected to reproduce the sign of z as well as its value at z = ±0 . That does happen in several libraries of elementary transcendental libraries; for instance, it happens on the Motorola 68881 Floating-Point Coprocessor, on Apple computers in their Standard Apple Numerical Environment, in Intel's Common Elementary Function Libraries for the i8087 and i80287 floating-point coprocessors, in analogous libraries now supplied with the Sun III , with the ELXSI 6400 and with the IBM RT/PC, and in the C Math Library currently distributed with 4.3 BSD UNIX for machines that conform to IEEE 754. With a few unintentional exceptions, it happens also on the hp-71B hand-held computer, whose arithmetic was designed to conform to IEEE 854.

If a programmer does not find these rules helpful, or if he does not know about them, he can ignore them and, as has been necessary in the past, insert explicit tests for zero in his program wherever he must cope with a discontinuity at zero. On the other hand, if the standards' rules happen to produce the desired results without such tests, the tests may be omitted leaving the programs simpler in appearance though perhaps more subtle. This is just what happens to programs that implement or use the elementary functions named above, as will become evident below.

Where to put the slits.

Each of our nine elementary complex functions f(z) has a slit or slits that bound a region, called the "principal domain" , inside which f(z) has a *principal value* that is single valued and analytic (representable locally by power series), though it must be discontinuous across the slit(s). That principal value is an extension, with maximal principal domain, of a real elementary function f(x) analytic at every interior point of its domain, which is a segment of the real x-axis. To conserve the power series' validity, points strictly inside that segment must also lie strictly inside the principal domain; therefore the slit(s) cannot intersect the segment's interior. Let z* = x-iy denote the complex conjugate of z = x+iy ; the power series for f(x) satisfy the identity f(z*) = f(z)* within some complex neighborhood of the segment's interior, so the identity should persevere throughout the principal domain's interior too. Consequently complex conjugation must map the slit(s) to itself/themselves. The slit(s) of an *odd* function f(z) = -f(-z) must be invariant under reflection in the origin z = 0 . Finally, the slit(s) must begin and end at *branch-point(s)*; these are singularities around which

some branch of the function cannot be represented by a Taylor nor Laurent series expansion. A slit can end at a branch point at infinity.

Consequently the slit for $\sqrt{\ }$, $\ln$ and z^w turns out to be the negative real axis. Then the slits for arcsin , arccos and arctanh turn out to be those parts of the real axis not between -1 and $+1$; similarly those parts of the imaginary axis not between $-i$ and $+i$ serve as slits for arctan and arcsinh . The slit for arccosh , the only slit with a finite branch-point (-1) inside it, must be drawn along the real axis where $z \leq +1$. None of this is controversial, although a few other writers have at times drawn the slits elsewhere either for a special purpose or by mistake; other tastes can be accommodated by substitutions sometimes so simple as writing, say, $\ln(-1) - \ln(-1/z)$ in place of $\ln(z)$ to draw its slit along (and just under) the positive real axis instead of the negative real axis.

Why do Slits Matter?

A computer program that includes complex arithmetic operations must be a product of a deductive process. One stage in that process might have been a model formulated in terms of analytic expressions that constrain physically meaningful variables without telling explicitly how to compute them. From those expressions somebody had to deduce other complex analytic expressions that the computer will evaluate to solve the given physical problem. The deductive process entails transformations among which some may resemble algebraic manipulations of real expressions, but with a crucial difference:

> Certain transformations, generally valid for real expressions, are valid for complex expressions only while their variables remain within suitable regions in the complex plane.

Moreover, those regions of validity can depend disconcertingly upon the computer that will be used to evaluate the expressions in question. For example, simplifying the expression $\sqrt{(z/(z-1))}\,\sqrt{(1/(z-1))}$ to $\sqrt{(z)}/(z-1)$ seems legitimate in so far as they both describe the same complex function, one that is continuous everywhere except for a pole at $z = 1$ and a jump-discontinuity along the negative real axis $z < 0$. And when those two expressions are evaluated upon a variety of computers including the ELXSI 6400, the Sun III, the IBM RT/PC, the IBM PC/AT, PC/XT and PC using i80287 or i8087, and the hp-71B, they agree *everywhere* within a rounding error or two. But when the same expressions are evaluated upon a different collection of computers including CRAYs, the IBM 370 family, the DEC VAX line, and the hp-15C, those expressions take opposite signs along the negative real axis! An experience like this could undermine one's faith in some computers.

What deserves to be undermined is blind faith in the power of Algebra. We should not believe that the equivalence class of expressions that all describe the same complex analytic function can be recognized by algebraic means alone, not even if relatively uncomplicated expressions are the only ones considered. To locate the domain upon which two analytic expressions take equal values generally requires a combination of algebraic, analytical and topological techniques. The paradigm is familiar to complex analysts, but it will be summarized here for the sake of other readers, using the two expressions given above for concrete illustration.

How to decide where two analytic expressions describe the same function.

1. Locate the singularities of each constituent subexpression of the given expressions.

The singularities of an analytic function are the boundary points of its domain of analyticity. These will consist of poles, branch-points and slits in this paper; but more generally they would include certain contours of integration, boundaries of regions of convergence, etc. In general, singularities can be hard to find; in our examples the singularities are obviously the pole at $z = 1$, the branch-point $z = 0$, and respective slits $0 < z < 1$, $z < 1$ and $z < 0$ whereon the quantities under square root signs are negative real.

2. Taken together, the singularities partition the complex plane into a collection of disjoint connected components. Inside each such component locate a *small continuum* upon which the equivalence of the given two expressions can be decided; that decision is valid throughout the component's interior.

The "small continuum" might be a small disk inside which both expressions are represented by the same Taylor series; or it could be a curvilinear arc within which both expressions take values that can be proved equal by the laws of real algebra. Other possibilities exist; some will be suggested by whatever motivated the attempt to prove that the given expressions are equivalent. In our example, the two expressions are easily proven equal on that part of the real axis where $z > 1$, which happens to lie inside the one connected component into which the slits along the rest of the real axis divide the complex plane. Therefore the two expressions must be equivalent everywhere in the complex plane except possibly where $z \leq 1$. (When a complex variable satisfies this kind of inequality its value must be real.)

3. The singularities constitute loci in the plane upon which the processes in steps 1 and 2 above can be repeated, finally leaving isolated singular points to be handled individually. End of paradigm.

In our example, the slit along $z < 1$ is partitioned into two connected components by the branch-point at $z = 0$. Each component has to be handled separately. Whether the two expressions are equivalent on a component must depend upon the definition of complex $\sqrt{z}$ on its slit where $z < 0$; there diverse computers appear to disagree. That is what this paper is about.

More generally, programmers who compose complex analytic expressions out of the nine elementary functions listed at this paper's beginning will have to verify whether their expressions deliver the functions that they intend to compute. In principle, that verification could proceed without prior agreements about the functions' values on their slits if instead analysts and programmers were obliged to supply an explicit expression to handle every boundary situation as they intend. Such a policy seems inconsiderate (not to say unconscionable) considering how hard some singularities are to find and how easy to overlook; but that policy is not entirely heartless since verifying correctness along a boundary costs the intellect nearly as much as writing down a statement of intent about that boundary. The trouble with those statements is that they generally contain inequalities and tests

and diverse cases, and as they accumulate they burden proofs and programs with a dangerously enlarged capture cross-section for errors. And almost all of those statements become superfluous in programs after we agree upon reasonable definitions for the functions in question on their slits.

For instance, in our example above we had to discover whether the two expressions agreed on an interval $0 < z < 1$ that lies strictly inside the domain of the desired function's analyticity, not on its boundary. That interval turns out to be a *removable singularity*, and it does remove itself from all the computers mentioned above because they evaluate both expressions correctly on that interval; diverse computers disagree only on the boundary where the desired function is discontinuous. Perhaps that's just luck. (Unlucky examples do exist and one will be presented later.) Let us accept good luck with gratitude whenever it simplifies our programs.

Complex analytic expressions that involve slits and other singularities are intrinsically complicated, and they get more complicated when rounding errors are taken into account. Our objective cannot be to make complicated things simple but rather, by choosing reasonable values for our nine elementary functions on their slits, to make them no worse than necessary.

Principal values on the slits, IEEE style.

Since all the slits in question lie on either the real or the imaginary axis, every point z on a slit is represented in at least two ways, at least once with a $+0$ and at least once with a -0 for whichever of the real and imaginary parts of z vanishes. Benignly, ambiguity in z at a discontinuity of $f(z)$ permits $f(z)$ to be defined formally continuously, except possibly at the ends of some slits, by continuation from inside the principal domain. This continuity goes beyond mere formality. By analytic continuation, the domain of each of our nine elementary functions $f(z)$ extends until it fills out a *Riemann Surface*; think of this surface as a multiple covering wrapped like a bandage around the *Riemann Sphere* and mapped onto it continuously by f. To construct f's principal domain, cut the bandage along the slit(s) and discard all but one layer covering the sphere. That layer is a *closed* surface mapped by f continuously onto a subset of the sphere. The shadow of that layer projected down upon the sphere is the principal domain; it consists of the whole sphere, but with slit(s) covered twice. That is why we wish to represent slits ambiguously.

Here are some illustrative examples, the first of a real function that is recommended for any implementation of IEEE standard 754 or 854.

$copysign(x, y)$ has the magnitude of x but the sign bit of y, so
$copysign(1,+0) = +1 = \lim copysign(1, y)$ at $y = 0+$, and
$copysign(1,-0) = -1 = \lim copysign(1, y)$ at $y = 0-$.

$\sqrt{(-1 + i0)} = +0 + i = \lim \sqrt{(-1 + iy)}$ at $y = 0+$;
$\sqrt{(-1 - i0)} = +0 - i = \lim \sqrt{(-1 + iy)}$ at $y = 0-$,

Consequently, $\sqrt{(z^*)} = \sqrt{(z)}^*$ for every z, and $\sqrt{(1/z)} = 1/\sqrt{(z)}$ too. These identities persist within roundoff provided the programs used for square root and reciprocal are those, supplied in this paper, that would have been chosen anyway for their efficiency and accuracy.

$\arccos(2 + i0) = +0 - i\,\mathrm{arccosh}(2) = \lim \arccos(2 + iy)$ at $y = 0+$,
$\arccos(2 - i0) = +0 + i\,\mathrm{arccosh}(2) = \lim \arccos(2 + iy)$ at $y = 0-$,
An implementation of arccos that preserves full arbitrary in the imaginary part of $\arccos(2 + iy)$ when y is very tiny can be expected to get its sign right when $y = +0$ too without extra tests in the code; such a program is supplied later in this paper.

But the foregoing examples make it all seem too simple. The next example presents a more balanced picture.

Let function $a(x) := \sqrt{(x^2 - 1)}$ for real x with $x^2 \geq 1$, and let $b(x) := a(x)$ for real $x \geq 1$; note that $b(x)$ is not yet defined when $x \leq -1$. The principal values of the complex extensions of a and b following the principles enunciated above turn out to be

$$a(z) = \sqrt{(z^2 - 1)} = a(-z) \quad , \quad \text{and}$$
$$b(z) = \sqrt{(z-1)}\,\sqrt{(z+1)} = -b(-z) \quad .$$

Both a and b are defined throughout the complex plane and both have a slit on the real axis running from -1 to $+1$, but a has another slit that runs along the entire imaginary axis separating the right half-plane where $a = b$ from the left half-plane where $a = -b$. The functions are different because generally

$$\sqrt{(\xi)}\,\sqrt{(\eta)} = \sqrt{(\xi\,\eta)} \quad \text{when} \quad |\arg(\xi) + \arg(\eta)| < \pi \ ,$$
$$= -\sqrt{(\xi\,\eta)} \quad \text{when} \quad |\arg(\xi) + \arg(\eta)| > \pi \ ,$$
$$= \pm\sqrt{(\xi\,\eta)} \quad \text{(hard to say which)} \quad \text{when} \quad \xi\,\eta \leq 0 \ .$$

Both functions a and b are continuous up to and onto ambiguous boundary points in IEEE style arithmetic, as described above, only if that arithmetic is implemented carefully; in particular, the expression $z + 1$ should not be replaced by the ostensibly equivalent $z + (1+i0)$ lest the sign of zero in the imaginary part of z be reversed wrongly. (Generally, mixed-mode arithmetic combining real and complex variables should be performed directly, not by first coercing the real to complex, lest the sign of zero be rendered uninformative; the same goes for combinations of pure imaginary quantities with complex variables. And doing arithmetic directly this way saves execution time that would otherwise be squandered manipulating zeros.) When z is near ± 1 the expression $a(z)$ nearly vanishes and loses its relative accuracy to roundoff. Although this loss could be avoided by rewriting $a(z) := \sqrt{((z-1)\,(z+1))}$, doing so would obscure the discontinuity on the imaginary axis in a cloud of roundoff which obliterates $\mathrm{Re}(z)$ whenever it is very tiny compared with 1 as well as when it is ± 0 .

Also obscure is what happens at the ends of some slits. Take for example $\ln(z) = \ln(\rho) + i\theta$, where $\rho = |z|$ and $\theta = \arg(z)$ are the polar coordinates of $z = x + iy$ and satisfy

$$x = \rho \cos\theta \ , \quad y = \rho \sin\theta \ , \quad \rho \geq 0 \quad \text{and} \quad -\pi \leq \theta \leq \pi \ .$$

Evidently $\rho := +\sqrt{(x^2+y^2)}$, and when $0 < \rho < +\infty$ then

$$\theta := 2\arctan(\, y/(\rho+x)\,) \quad \text{if} \quad x \geq 0 \ , \quad \text{or}$$
$$:= 2\arctan(\, (\rho-x)/y\,) \quad \text{if} \quad x \leq 0 \ ,$$

At the end of the slit where $z = x = y = \rho = 0$ (and $\ln(\rho) = -\infty$) the value of θ may seem arbitrary, but in fact it must cohere with other almost arbitrary choices concerning division by zero and arithmetic with infinity. A reasonable choice is to interpose the reassignment

$$\text{if} \quad \rho = 0 \quad \text{then} \quad x := \mathrm{copysign}(1, x)$$

between the computations of ρ and θ above. More about that later.

The foregoing examples provide an unsettling glimpse of the complexities that have daunted implementers of compilers and run-time libraries who would otherwise extend to complex arithmetic the facilities they have supplied for real floating-point computation. These complexities are attributable to failures, in complex floating-point arithmetic, of familiar relationships like algebraic identities that we have come to take for granted in the arena of real variables. Three classes of failures can be discerned:

i) The domain of an analytic expression can enclose singularities that have no counterparts inside the domain of its real restriction. That is why $\sqrt{(z^2-1)} \neq \sqrt{(z-1)}\sqrt{(z+1)}$, for example.

ii) Rounding errors can obscure the singularities. That is why, for example, $\sqrt{(z^2-1)} = \sqrt{((z-1)(z+1))}$ fails so badly when either $z^2 = 1$ very nearly or when $z^2 \neq 0$ very nearly. To avoid this problem, the programmer may have to decompose complex arithmetic expressions into separate computations of real and imaginary parts, thereby forgoing some of the advantages of a compact complex notation.

iii) Careless handling can turn infinity or the sign of zero into misinformation that subsequently disappears leaving behind only a plausible but incorrect result. That is why compilers must not transform $z - i$ into $z - (1+i0)$, as we have seen above, nor $-(-x-x^2)$ into $x + x^2$, as we shall see below, lest a subsequent logarithm or square root produce a nonzero imaginary part whose sign is opposite to what was intended.

The first two classes are hazards to all kinds of arithmetic; only the third kind of failure is peculiar to IEEE style arithmetic with its signed zero. Yet all three kinds must be linked together esoterically because the third kind is not usually found in an application program unless that program suffers also from the second kind. The link is fragile, easily broken if the rational operations or elementary functions, from which applications programs are composed, contain either of the last two kinds of failures. Therefore, implementers of compilers and run-time libraries bear a heavy burden of attention to detail if applications programmers are to realize the full benefit of the IEEE style of complex arithmetic. That benefit deserves some discussion here if only to reassure implementors that their assiduity will be appreciated.

The first benefit that users of IEEE style complex arithmetic notice is that familiar identities tend to be preserved more often than when other styles of arithmetic are used. The mechanism that preserves identities can be revealed by an investigation of an analytic function $f(z)$ whose domain is slit along some segment of the real or imaginary axis; say the real (x) axis. When $z = x + iy$ crosses the slit, $f(z)$ jumps discontinuously as y reverses sign although $f(z)$ is continuous as z approaches one side of the slit or the other. Consequently the two limits

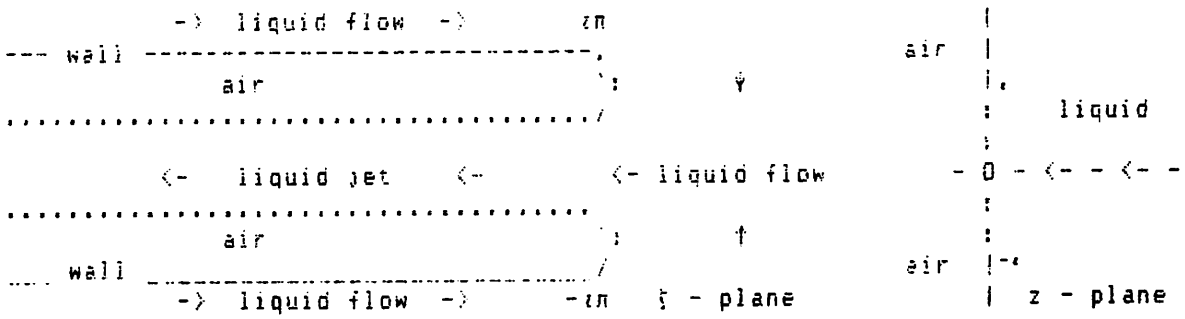
$$f(x + i0) := \lim f(x + iy) \text{ as } y \to 0+ \quad \text{and}$$
$$f(x - i0) := \lim f(x + iy) \text{ as } y \to 0-$$

both exist, but they are different when x has a real value inside the slit. Ideally, a subroutine $F(z)$ programmed to compute $f(z)$ should match these values: $F(x \pm i0) = f(x \pm i0)$ respectively should be satisfied within a small tolerance for roundoff. This normally happens in IEEE style

arithmetic as a by-product of whatever steps have been taken to ensure that $F(x + iy) = f(x + iy)$, within a similarly small tolerance, for all sufficiently small but nonzero $|y|$. To generate a discontinuity, the subroutine F must contain constructions similar to copysign(..., y) or arctan(1/y) possibly with "y" replaced by some other expression that either vanishes or tends to infinity as $y \to 0$. That expression cannot normally be a sum or difference like $\arctan(y-1) + \pi/4$ or $\exp(y) - 1$ that vanishes by cancellation, because roundoff can give such expressions values (typically 0) that have the wrong sign when $|y|$ is tiny enough. Instead, to preserve accuracy when $|y|$ is tiny, that expression must normally be a real product or quotient involving a power of y or $\sin(y)$ or some other built-in function that vanishes with y and therefore should inherit its sign at $y = \pm 0$. Thus does careful implementation of compiler and library combine with careful applications programming to yield correct behavior on and near the slit. And if two such carefully programmed subroutines $F(z)$, though based upon different formulas, agree within roundoff everywhere near the slit, then the foregoing reasoning implies that normally they have to agree on the slit too; this is the way IEEE style arithmetic preserves identities like $\sqrt{(z^*)} = (\sqrt{z})^*$ and $\sqrt{(1/z)} = 1/\sqrt{z}$ that would have to fail on slits if zero had no sign.

Of course, applications programmers generally have things more important than the preservation of identities on their minds. Here is a more typical and realistic example:

Pictures of Conformal Map $\zeta = f(z)$:



Conformal Map $\zeta = f(z)$ of Half-Plane to Jet with Free Boundary

Let $f(z) := 1 + z^2 + z\sqrt{(1+z^2)} + \ln(z^2 + z\sqrt{(1+z^2)})$, and construe the equation $\zeta := f(z)$ as a conformal map, from the plane of $z = x + iy$ to the plane of $\zeta = \xi + i\eta$, that maps the right half-plane $x \geq +0$ onto the region wetted by a liquid that is being forced by high pressure to jet into a slot. The walls of the slot, where $\xi < 0$ and $\eta = \pm\pi$, should be the images of those parts of the imaginary axis $z^2 < -1$ lying beyond $\pm i$. The free surfaces of the jet, curving forward from $\zeta = \pm i\pi$ and then back to $\zeta = -\infty \pm i\pi/2$, should be the image of that segment of the imaginary axis $-1 < z^2 < 0$ between $\pm i$.

The picture of $f(z)$ should be symmetrical about the real axis because $f(z^*) = f(z)^*$. As z runs up the imaginary axis, with $x = +0$ and y running from $-\infty$ through -1 toward -0 and then from $+0$ through $+1$ toward $+\infty$, its image $\zeta = f(z)$ should run from left to right along the lower wall and back along the lower free boundary of the jet, then from left to right along the jet's upper free boundary and back along the upper wall. This is just what happens when $f(z)$ is plotted from a one-line program on the hp-71B calculator, which implements the proposed IEEE standard 854. But when $f(z)$ is programmed onto the hp-15C, whose zero is unsigned, the lower wall disappears. Its pre-image, the lower part of the imaginary axis where $z/i < -1$, is mapped during the computation of $f(z)$ into the slit that belongs to $\sqrt{}$ and ln; the upper part $z/i > 1$ gets mapped onto the same slit. For lack of a signed zero, that slit gets attached to a side that is right for the upper wall but wrong for the lower wall, thereby throwing the pre-image of the lower wall away into a tiny segment of the upper wall. To put the lower wall back, x must be increased from 0 to a tiny positive value while y runs from $-\infty$ to -1. (How tiny should x be? That's a nontrivial question.)

The misbehavior revealed in the foregoing example $f(z)$ may appear to be deserved because $f(z)$ has slits on the imaginary axis $z^2 < -1$ beyond $\pm i$. Should mapping a slit to the wrong place be blamed upon the discontinuity there rather than upon arithmetic with an unsigned zero? No. Arithmetic with an unsigned zero can also cause other programs to misbehave similarly at places where the functions being implemented are otherwise well behaved. For example consider $c(z) := z - i\sqrt{(z+1)}\sqrt{(z-1)}$, whose slit lies in the imaginary axis $-1 < z^2 < 0$ between $\pm i$. Now $\zeta := c(z)$ maps the slit z plane onto the ζ plane outside the circle $|\zeta| = 1$; vertical lines in the z plane map to stream lines in the vertical flow of a fluid around the circle. Implementing $c(z)$, the programmer notices that he can reduce two expensive square roots to one by rewriting

$$c(z) := z + \sqrt{(z^2+1)}\ \mathrm{copysign}(1, \mathrm{Re}(z)) .$$

The two expressions for $c(z)$ match everywhere in IEEE style arithmetic; but when zero has only one sign, say $+$, the second expression maps the lower part of the imaginary axis, where $z/i < -1$, into the inside instead of the outside of the circle, although $c(z)$ should be continuous there.

The ease with which IEEE style arithmetic handled the important singularities near $z = \pm i$ in the examples above should not be allowed to persuade the reader that all singularities can be dispatched so easily. The singularities $f(0)$ and $f(\infty)$ and the overflows near $z = \infty$ would have to be handled in the usual ways if they did not lie so far off the left-hand side of the picture that nobody cares. Another kind of singularity that did not matter here, but might matter elsewhere, insinuated weasel words like "not usually", "tends to be" and "normally" into the earlier discussion of sums and differences that normally vanish by cancellation. Sums and differences can vanish without cancellation if they combine terms that have already vanished; an example is $h(x) := (x+1)^2 - 1$ when $x = 0$. Evaluating $h(\pm 0)$ in IEEE style real arithmetic yields $+0$ instead of ± 0 respectively, losing the sign of zero. $h(x)$ has other troubles; it signals Underflow when x is very tiny, suffers inaccuracy when x is very near -1, and becomes Invalid at $x = -\infty$. Simply rewriting $h(x) := x(1+x)$ dispels all these troubles, but is slightly less accurate for very tiny x than is $h(x) := x + x^2$, which preserves accuracy

and the sign of zero for all tiny real x. Complex arithmetic complicates this situation. Both expressions $z+z^2$ and $z(1+z)$ produce zeros with the wrong sign for $Im(h(z))$ on various segments of the real z-axis; to get the correct sign and better accuracy requires an expression like

$$h(x + iy) := x(1+x)-y^2 + 2iy(x+0.5)$$

regardless of arithmetic style. For similar reasons, the expression for $f(z)$ used above for the conformal map would have to be rewritten if the interesting part of its domain were the left instead of right half-plane.

IEEE style complex arithmetic appears to burden the implementers of compilers and run-time libraries with a host of complicated details that need rarely bother the user if they are dispatched properly; and then familiar identities will persist, despite roundoff, more often than in other styles of arithmetic. This thought would comfort us more if the aberrations were easier to uncover. Locating potential aberrations remains an onerous task for an application programmer, regardless of the style of arithmetic; however that style can affect the locus of aberration fundamentally. In IEEE style arithmetic, a programmed implementation of a complex analytic function can take aberrant boundary values, different from what would be produced by continuation from the interior, because of roundoff or similar phenomena. In arithmetic without a signed zero, such an aberration can be caused as well by an unfortunate choice of analytic expression, though the programmer has implemented it faithfully. The fact that an analytic expression determines the values of an analytic function correctly inside its domain is no reason to expect the boundary values to be determined correctly too when zero is unsigned.

Principal values on the slits, hp-15C style.

Of course, the hp-15C is not the only machine with an unsigned zero; the DEC VAX 11/xxx is similar but lacks so far a careful software implementation of some of the functions under discussion;-- in time that lack will be remedied. Many other machines, the IBM 370 series among them, have a signed zero in their hardware but no provision for propagating its sign in a coherent and useful way, so they are customarily programmed as if zero were unsigned. All these machines discourage attempts to distinguish one side of a slit from the other on the slit itself.

What we have to do is attach each slit to one of its sides in accordance with some reasonable rule, thereby obtaining a principal value which is continuous up to the slit from that side but not from the other. In other words, we have to assign a sign to zero on each slit and then compute the same principal value as would have been computed using IEEE-style arithmetic. The assignment cannot be arbitrary; for instance we cannot change sides in the middle of the slit lest a gratuitous singularity be introduced by the change. On the other hand, some degree of arbitrariness is obligatory. For instance, the two functions

[illegible]

are indistinguishable everywhere except on the slit [illegible] across which they are discontinuous, but in hp-15C style arithmetic one function must be continuous onto the top of the slit and the other onto the bottom. [illegible]

solely upon the slit's shape nor solely upon the function's values off the slit. And yet, paradoxically, the mp-15C appears to follow just such a rule, namely

> Counter-Clockwise Continuity (CCC) :
> Attach each slit to whichever side is approached when the finite branch-point at its end is circled counter-clockwise.

For instance, when $z < 0$ this rule implies that $\sqrt{z} = +i\sqrt{(-z)}$ and $\ln(z) = \ln(-z) + i\pi$. Actually CCC is merely a mnemonic summary of the implications, for the nine functions that are the subject of this note, of the following more general convention applicable also to $b(z)$ above, as CCC is not.

> The Principal Expression:
> Assign to each elementary function in question not merely a Principal Value but also a *Principal Expression* in terms of $\ln(z)$, using the simplest formula that manifests its behavior at finite branch-points without gratuitous singularities elsewhere.

What makes this convention effective is a canonical association between the archetypal branch-points of $\ln(z)$ and $\sqrt{z} = \exp(\ln(z)/2)$ on the one hand, and on the other any isolated branch-point at the end of a slit belonging to any other elementary function. For example,

$\arcsin(z) = \pi/2 - ($ power series in $1-z)\sqrt{(1-z)}$ for z near 1 ,
$\operatorname{arccosh}(z) = \ln(z) - ($ power series in $1/z)$ when $|z|$ is huge,
$\operatorname{arctanh}(z) = -0.5\ln(1-z) + ($ power series in $1-z)$ for z near 1.

In each case the power series is determined uniquely. In general, if β is a finite branch-point at the end of a slit belonging to one of our nine functions $f(z)$, and if $f(z)$ is analytic inside some circular disk $|z-\beta| < \rho$ except on the slit, then $f(z)$ can be represented inside that slit disk by one of the formulas

$f(z) = P(z-\beta) + p(z-\beta)\sqrt{(z-\beta)/c)}$, or
$f(z) = P(z-\beta) + p(z-\beta)\ln((z-\beta)/c)$, or
$f(z) = P($ some non-integral power of $\sqrt{((z-\beta)/c)})$,

where $c = \lim (\beta-z)/|\beta-z|$ as $z \to \beta$ along the slit, so $|c| = 1$ and $(z-\beta)/c < 0$ in the slit, and $P(t)$ and $p(t)$ are representable by power series around $t = 0$. Given β and f and its slit, c and P and p are *canonical* (determined uniquely). Formulas slightly more general than these, but still essentially unique, cope with more general elementary functions or with isolated branch-points at ∞ .

The dominant terms of these canonical formulas provide approximations useful near branch-points, and are therefore precious to analysts and programmers who have to exploit or compensate for singularities, so these formulas should not be violated unnecessarily on the slits. Programs that handle singularities are complicated enough without the additional burden of treating specially those slits that need no special care so long as programs remain as valid on the slits as off them near their ends. Then programmers can predict from Principal Expressions how their programs will take to slits. The Principal Expressions for all nine of our elementary functions are determined by convention and tabulated nearby. For other functions the choice of Principal Expression is forced by the choice of slits except when a slit contains just two singularities, both finite branch points at

its ends. In the exceptional case the Principal Expression tells which side of that slit is attached to it. For instance, the Fortran programmer can define the

COMPLEX FUNCTION B(Z) = CSQRT(Z-1.0)*CSQRT(Z+1.0)

when he wishes to attach its slit to its upper side, and invoke -B(-Z) when he wishes to attach the slit to its lower side. Another example has two definitions

arccot(z) := arctan(1/z) and arccot(z) := π/2 - arctan(z)

that are both widely used though they differ by π in the left half-plane. The first has one slit on the imaginary axis $-1 < z^2 < 0$ between $z = \pm i$. The second has two slits on the imaginary axis $z^2 < -1$ beyond $z = \pm i$. But arctan(1/z) is not a Principal Expression for arccot(z) because it has a gratuitous singularity at z = 0 where its slit changes sides. A correct Principal Expression for the first definition of arccot(z) is either $i\ln((z-i)/(z+i))/2$ or $\ln((z+i)/(z-i))/(2i)$ according to whether its slit be attached respectively to the left half-plane or to the right; except on the slit, these Principal Expressions are equal and satisfy arccot(-z) = -arccot(z). Whichever one be chosen, the other is -arccot(-z). Similarly for $\pm$arccoth($\pm$z) := ln((z+1)/(z-1))/2 .

TABLE =

Conventional Principal Expressions for Elementary Functions:

$-\pi < \arg(z) \le \pi$; and $-\pi < \arg(z)$ if 0 has just one sign.

$\ln(z)$	:=	$\ln(\lvert z\rvert) + i\arg(z)$	
z^w	:=	$\exp(w \ln(z))$ (and $z^0 = 1$, $0^w = 0$ if $Re(w) > 0$)	
$\sqrt{z}$	:=	$z^{1/2}$	
arctanh(z)	:=	$(\ln(1+z) - \ln(1-z))/2$	= -arctanh(-z)
arctan(z)	:=	arctanh(iz)/i	= -arctan(-z)
arcsinh(z)	:=	$\ln(z + \sqrt{1 + z^2})$	= -arcsinh(-z)
arcsin(z)	:=	arcsinh(iz)/i	= -arcsin(-z)
arccos(z)	:=	$2\ln(\sqrt{(1+z)/2} + i\sqrt{(1-z)/2})/i$	= π/2 - arcsin(z)
arccosh(z)	:=	$2\ln(\sqrt{(z+1)/2} + \sqrt{(z-1)/2})$	

In general the definitions of Principal Expressions can and should be honored in all styles of arithmetic, though they must be implemented carefully if they are to survive roundoff. Careful implementations of our nine elementary functions will be presented later in this paper. But some familiar identities satisfied in IEEE style arithmetic must be violated

when 0 is unsigned no matter how the slits be attached. For instance, no elementary function f in the table except arctan and arcsinh can satisfy $f(z^*) = f(z)^*$ when z lies in a slit in the real axis. Similarly

$$\ln(1/z) = -\ln(z) \quad \text{and} \quad \sqrt{}(1/z) = 1/\sqrt{}(z)$$

must be violated at $z = -1$ and therefore everywhere in the slit $z < 0$. Other familiar identities violated only in a slit include

$\text{arctanh}(z) = \ln((1+z)/(1-z))/2$, violated when $z > 1$,
$\arctan(z) = \imath\ln((\imath+z)/(\imath-z))/2$, violated when $\imath z < -1$, and
$\arccos(z) = 2\arctan(\sqrt{}((1-z)/(1+z)))$, violated when $z < -1$.

Other writers have put forward different formulas as definitions for our nine elementary functions. Comparing various definitions, and choosing among them, is a tedious business prone to error. Some ostensibly different definitions, like

$$\text{arccosh}(z) = \ln(z + \sqrt{}(z-1)\sqrt{}(z+1)) ,$$

give the same results as ours. Some are quite wrong, as are

$$\text{arccosh}(z) = \ln(z + \sqrt{}(z^2-1)) \quad \text{and} \quad \arccos(z) = \ln(z + \sqrt{}(z^2-1))/\imath ,$$

because their slits are in the wrong places. Some are different on only part of a slit, as is

$$\text{arccosh}(z) = -\ln(z - \sqrt{}(z-1)\sqrt{}(z+1))$$

which is continuous from below that part of the slit where $z < -1$ and therefore violates the canonical formula around infinity. Some are very close to ours; for instance, a proposal to introduce complex functions into APL recommended at first (but not any more) the formula

$$\text{arccosh}(z) = \ln(z + (z+1)\sqrt{}((z-1)/(z+1)))$$

which yields the same principal value as our formula except for a gratuitous removable singularity at $z = -1$. The same proposal advocated initially

$$\arctan(z) = -\imath\ln((1 + \imath z)\sqrt{}(1/(z^2+1)))$$

because its range matches that of $\arcsin(z)$, though no reason was given why the ranges should match (but see below), and because it was alleged that the CCC rule should be reversed around a branch point at which the function is infinite, though doing so would introduce anomalies in the relation between $\ln$ and $\sqrt{}$, thereby vitiating the formula being advocated. Another well-known formula

$$\arctan(z) = \imath\ln(\sqrt{}((\imath+z)/(\imath-z)))$$

is continuous one way around one branch-point and the opposite way around the other, thereby violating $\arctan(-z) = -\arctan(z)$ on the slits. Our formula given earlier, which is equivalent to

$$\arctan(z) = \imath(\ln(1-\imath z) - \ln(1+\imath z))/2 ,$$

follows the CCC rule and seems simplest, but it does violate two cherished formulas

$$\arcsin(z) = \arctan(z/\sqrt{}(1-z^2)) \quad \text{and}$$
$$\arccos(z) = 2\arctan(\sqrt{}((1-z)/(1+z)))$$

on the slit. These formulas are satisfied almost everywhere by the APL proposal's definition of arctan mentioned above, the exceptions $\arccos(-1)$ and $\arcsin(\pm 1)$ arising because, like zero, $1/0$ has no sign and therefore $\arctan(1/0)$ has to be either undefined or chosen arbitrarily from $\{\pm\pi/2\}$. Rather than debate the merits of cherished formulas satisfied everywhere except at some finite branch-points versus canonical formulas satisfied around every finite branch-point, we choose what seem to be the more perspicuous definitions. For similar reasons, our formula above for arctanh seems preferable to the APL proposal's

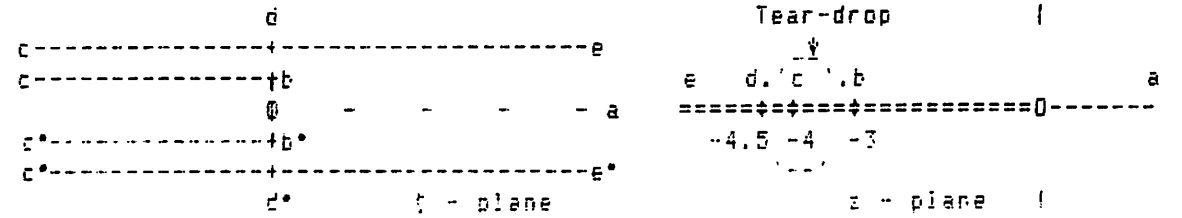
$$\text{arctanh}(z) = \ln((1+z)\sqrt{}(1/(1-z^2))) ,$$

Regardless of whether our *Principal Expressions* really are preferable to someone else's, and regardless of the style of arithmetic, good reasons exist to seek universal agreement upon a set of *Principal Expressions* to define *Principal Values* for familiar elementary functions. The first to benefit from such an agreement would be analysts, who would suffer less confusion when reading each other's results. More importantly, programmers would make fewer mistakes, and find them sooner, when implementing conformal maps from complex analytic expressions. Although those benefits might follow from any kind of agreement, Principal Expressions offer the further advantage that they introduce no unnecesary singularities. That advantage goes beyond mere parsimony, because control of singularities is the essence of the subject.

Programs that involve singularities are especially difficult to debug because so many programmers tend to think more like algebraists than like analysts or geometers. Unaccustomed to manipulating inequalities, they have trouble locating the slits that are implicit in complex expressions that contain any of our nine elementary functions. Instead, too many programmers are inclined to test complex expressions in the same way as they often test real expressions, by evaluating them at a handful of trial arguments to see whether the results agree with prior expectations. Because this test strategy usually works for real analytic expressions, programmers mostly ignore warnings that it is unreliable; what else should we expect in a society where drunk driving is still regarded widely as a mere *peccadillo*? But this strategy is truly a dangerous way to test complex analytic expressions of conformal maps with corners because those maps are notorious for mapping tiny regions into huge ones. When a tiny region like that is missed by a scattering of trial arguments, the test can be quite deceptive. The next example illustrates the point.

Let $g(z) := 2\,\mathrm{arccosh}(1 + 2z/3) - \mathrm{arccosh}(^8/_3 - (^8/_3)/(z+4))$, and construe the equation $\zeta := g(z)$ as a conformal map of the z-plane, slit along the negative real axis $z < 0$, onto a slotted strip in the plane of $\zeta = \xi + i\eta$. The strip lies where $|\eta| \le 2\pi$, and the slot within it lies where $\xi < 0$ and $|\eta| < \pi$. The boundary of the slotted strip is the image of both sides of the slit in IEEE style arithmetic; with an unsigned zero the slit maps onto only that part of the boundary in the upper half-plane.

PICTURE of Conformal Map $\zeta = g(z)$:



Conformal Map $\zeta = g(z)$ of Slit Plane to Slotted Strip

The cost of computing $g(z)$ comes mostly from two logarithms entailed by

two cells upon arccosh. Two logarithms can be reduced to one by means of a page or so of algebraic manipulation starting from the Principal Expression tabulated for arccosh above; the result is a *proof* that

$$q(z) = 2\ln(\sqrt{(z+4)/3}\,(\sqrt{(z+3)} + \sqrt{z/(2\sqrt{z+3})} + \sqrt{z})) .$$

Without Principal Expressions, one might resort instead to formulas like

$$\mathrm{Arccosh}(z) = \ln(z \pm \sqrt{(z^2-1)}) + 2\pi k\mathrm{i} \quad \text{for } k = \ldots, -2, -1, 0, 1, 2, \ldots$$

or to identities like

$$\mathrm{Arccosh}(z) \pm \mathrm{Arccosh}(\zeta) = \mathrm{Arccosh}(z\zeta \pm \sqrt{((z^2-1)(\zeta^2-1))}) ,$$

with results that are hard to predict. A possible outcome is the expression

$$q(z) := 2\,\mathrm{arccosh}(2(z+3)\sqrt{((z+3)/(27(z+4)))}) ,$$

which matches the desired $q(z)$ everywhere in the z-plane except in a small tear-drop shaped region situated symmetrically about the segment $-4.5 < z < -3$ on the real axis. The tear-drop's boundary is the locus in the plane of $z = x+\mathrm{i}y$ whereon the argument of arccosh in $q(z)$ takes values on the slit between 0 and -1 ; the boundary's equation is

$$y^2 + (x+3)^2(2x+9)/(2x+5) = 0 \quad \text{for} \quad -4.5 \le x \le -3 .$$

Whereas $\zeta = g(z)$ maps the tear-drop onto two half-strips in the left-half of the ζ-plane, $\zeta = o(z)$ maps the tear-drop into two half-strips in the right half-plane. Indeed, $q(z) = -g(z)$ in the tear-drop except, if zero is unsigned, $q(z) = -g(z)^*$ for $-4.5 < z < -4$. Is it likely that a few trial evaluations will reveal the difference between $q(z)$ and $g(z)$?

The examples presented in this paper may give the impression that an analyst will benefit far less than a programmer from Principal Expressions because their benefits seem meagre unless slits run along straight lines. Moreover a signed zero seems useless except when slits lie in the real and imaginary axes. True; but not the whole truth. Despite that applications of elementary functions frequently relocate their slits to nonstandard places, the functions so constructed have to be communicated to humans and to computers in terms of combinations of the standard elementary functions with which we are all acquainted. For instance, let $e(z)$ be an analytic extension of $\arcsin(z)$ from the upper half-plane across its slits $z^2 > 1$ into the lower half-plane, where we relocate the slits to run down from ± 1 along some paths to $-\mathrm{i}\infty$. Can $e(z)$ be expressed in terms of $\arcsin(z)$? Yes. In the upper half-plane or between the new slits, $e(z) := \arcsin(z)$. Elsewhere we define $s := \mathrm{copysign}(1, \mathrm{Im}(z))$ and calculate

$$e(z) := s\arcsin\cdot z + \mathrm{copysign}((1-s)\pi/2, \mathrm{Re}(z)) ,$$

which is continuous across the old slits in IEEE style arithmetic. If 0 is unsigned, the last expression must be replaced by something somewhat more complicated.

Readers who recoil from tedious labor may rather acquiesce to all the foregoing assertions than verify any of them personally, despite that such assertions are notoriously rife with mistakes. Yet, lest the pleasures of analysis be eschewn altogether, the writer tenders some simple exercises for the reader's amusement; in each line the object is to discover the whole domain, including boundary, wherein one expression equals another.

Exercises: Where are two expressions on the same line Equal?

$\sqrt{(z^2-1)}$, $\sqrt{(z-1)}\sqrt{(z+1)}$, $-\sqrt{(1-z)}\sqrt{(-1-z)}$, $i\sqrt{(1-z)}\sqrt{(1+z)}$

$\sqrt{(z-1)}/\sqrt{z}$, $\sqrt{(1 - 1/z)}$, $\sqrt{(z^2-z)}/z$, $\sqrt{(y(y-1) - y^2 + 2iy(x-1/2))}/z$

$\sqrt{(z)}/\sqrt{(z-1)}$, $\sqrt{(z/(z-1))}$

$2\,\mathrm{arctanh}(z)$, $\ln((1+z)/(1-z))$, $\mathrm{arcsinh}(2z/(1-z^2))$

$\cos(n\,\arccos(z))$, $\cosh(n\,\mathrm{arccosh}(z))$, for nonintegers n

$\arctan(z) + \arctan(1/z)$, $\pi/2$, $-\pi/2$

$\mathrm{arccosh}(z)$, $\mathrm{arccosh}(2z^2-1)/2$, $2\,\mathrm{arcsinh}(\sqrt{((z-1)/2)})$, $i\,\arccos(z)$

$\mathrm{arccosh}(z) - \mathrm{arccosh}(-z)$, $i\pi$, $-i\pi$

The answers may depend upon whether arithmetic is performed in hp-15C style or in IEEE style, the difference appearing only when a slit lies in the real or the imaginary axis.

Summary

Two styles of computer arithmetic induce two different mental attitudes towards the connection between analytic expressions and analytic functions.

IEEE style arithmetic encourages the extension by continuity of every complex analytic function from the interior of its domain to the boundary, including both sides of slits. Either side can be selected with the aid of a signed ± 0 . Consequently, two expressions that represent the same function everywhere inside its domain are likely to match everywhere on the boundary too; most exceptions are correlated with roundoff problems.

Arithmetic with an unsigned 0 permits continuous extension to one side of a slit but not to both. Consequently, two expressions that represent the same function everywhere inside its domain often take different values on the boundary. Choosing among such expressions is tantamount to choosing among boundary values for what is otherwise the same function. Our nine elementary functions are among those defined by Principal Expressions determined along with their Principal Values by convention. Other complex functions have to be defined on and inside boundaries by apt compositions of Principal Expressions, or else by *ad hoc* assignments on boundaries.

Regardless of the style of arithmetic, analytic expressions provide at best a statement of intent, at worst wishful thinking about complex analytic functions. Implementations faithful to the expressions despite roundoff and over/underflow must overcome nontrivial technical challenges.

Implementation Notes

Six inverse trigonometric and hyperbolic functions are defined in terms of ln and √ by Principal Expressions tabulated above in such a way as might appear to provide one-line programs to compute those functions in, say, Fortran. Unfortunately, roundoff can cause such programs to lose their relative accuracy near their zeros or poles; and overflow can occur for large arguments even though the desired function has an unexceptionable value. Programs to compute complex elementary functions robustly and fairly accurately are surprisingly complicated, so much so as to justify supplying them in this paper. Actually, we supply algorithms that can be converted into programs on various machines by being adapted to the peculiarities of diverse programming languages and computing environments.

All our pseudo-programs written to look like Function Subroutines pass their arguments *by value*, and hence may write over them, contrary to common practice in Fortran.

Certain details, particularly those that pertain to ∞ and NaN, are peculiar to IEEE style arithmetic. Otherwise the algorithms presented here for various complex elementary transcendental functions, though designed for IEEE style arithmetic, can be used with other reasonably rounded *binary* floating-point arithmetics to get comparable results. Our algorithms assume either that zero always has a + sign, in which case copysign behaves just like Fortran's SIGN function, or else that ±0's sign obeys the rules specified by IEEE 754 and 854. Those standards also specify rules for +∞ and -∞ and for something called "NaN", which stands for "Not a Number" and is produced by invalid operations like 0/0, 0∞, ∞/∞ and ∞-∞. Predicates like x=y, x≤y and x≤y are all *false*; but x≠y and x≱y are *true* when either or both of x and y are NaN. Algebraic operations upon a NaN reproduce it. Both infinities and NaNs can be produced by our algorithms, and both will be accepted as inputs to them. There are machines that do not conform to IEEE 754 or 854 but do possess things like NaNs; examples are the DEC VAX, CDC Cybers and Crays. The latter two have ∞ too. Whether our algorithms will accept their kind of NaN and ∞ is not easy to determine; besides, those machines are most often configured to stop rather than continue execution with a NaN or ∞.

Certain *Environmental Constants* that characterize important attributes of computer arithmetic may be specified precisely when that arithmetic conforms to IEEE 754 or 854; otherwise they might be slightly vague:

Ω := Overflow threshold = Nextafter(+∞, 0)
ε := Roundoff threshold = 1.0 - Nextafter(1.0, 0)
λ := Underflow threshold = 4(1-ε)/Ω in IEEE 754
Smallest positive no. = Nextafter(0.0, 1) = 2ελ in IEEE 754

Here *Nextafter* is a function specified in the appendix to IEEE 754; it perturbs its first argument by one *ulp* (one *U*nit in its *L*ast *P*lace) towards the second. That appendix also includes *copysign*, which was described early in this paper, and two functions *scalb* and *logb* that will appear later. Let β be the arithmetic's *radix*, 2 for IEEE 754, or 2 or 10 for 854. For any floating-point x and integer N, scalb(x, N) := $\beta^N x$ computed without first computing β^N, so Over/Underflow is signaled only if the final value deserves it. logb(NaN) is NaN; logb(±∞) := +∞; logb(0) := -∞ with Divide-by-Zero signaled; and if λ ≤ |x| < ∞ then logb(x) is an integer such that 1 ≤ scalb(x, -logb(x)) < β. The same

may be true when $0 < |x| < \lambda$, but early implementations may instead yield $\log(x) := \log(\lambda)$ in that case. Like the procedures *ldexp* and *frexp* in the *C* library, scalb and logb are practically indispensible for scaling and for computing logarithms and exponentials. Our algorithms are intended for radix = 2, but many of them work for other radices too.

The IEEE standards prescribe responses to five kinds of *exceptions*: *Invalid Operation*, *Overflow*, *Divide-by-Zero*, *Underflow*, *Inexact*. Each kind has its *flag*, to be raised to signal that its kind of exception has occurred; each kind produces a *default result*, respectively

NaN, $+\infty$, $+\infty$, gradual underflow, rounded result.

Gradual underflow approximates any value between $\pm\lambda$ with an error smaller than $\varepsilon\lambda$ instead of flushing it to zero. Neither this feature nor flags figure as much as they could and should in our algorithms. In environments that conform fully to IEEE 754, as does the Standard Apple Numerical Environment (SANE) on Apple computers, robust exception-handling complicates programs much less than ours have been complicated by our desire to provide algorithms adaptable also to machines that do *not* conform to the standards. Most of our algorithms can be adapted to such machines by merely excising references to features that those machines do not support. For instance, a statement like "If $x = \infty$ then ..." will be deleted for machines that have no infinity; however, some obvious precaution against division by zero may have to be inserted elsewhere instead. Machines that flush underflows to zero instead of underflowing gradually may produce less accurate results when they approach the underflow thresholds $\pm\lambda$.

Our algorithms would be simpler, some much simpler, if every arithmetic operation accepted and produced intermediate results of wider range and precision than our algorithms are normally expected to accept or produce. Such a situation arises when the transcendental functions are intended for a higher-level language like Fortran that supports only Single- and Double-precision variables, but the implementer has access to another wider format like IEEE 754's *Extended* format. That is implemented in floating-point coprocessor chips such as the Intel i8087 and i80287 used in the IBM PC, PC/XT and PC/AT, the Motorola 68881 used in a host of 68000-based work-stations, the Western Electric 32106, and also in Apple's SANE. But no such Extended format is provided by the National Semiconductor 32081 used in the IBM RT/PC, nor by the Weitek 1164/1165 chips used in the Sun III among others, nor by the NCUBE multiprocessor array, nor by Fairchild's *Clipper* nor AMD's 29027 nor INMOS' IMS T800-30 ; for their benighted sakes we have to use devious formulas to preserve accuracy and avoid spurious overflows.

In the programs below, φ, ρ, θ, s, t, u, v, x, y, ξ and η denote real variables: $w := u + \imath v$, $z := x + \imath y$ and $\zeta := \xi + \imath\eta$ denote complex variables; and a star denotes not multiplication but complex conjugation: $z^* = x - \imath y$. Mixed-mode arithmetic upon one real and one complex variable is presumed *NOT* to be performed by coercing the real to complex, but rather in a way that avoids unnecessary hazards like 0∞ or $\infty-\infty$ by avoiding unnecessary real operations:

$\varphi + z := (\varphi+x) + \imath y$, $\varphi z := \varphi x + \imath\varphi y$, $z/\varphi := x/\varphi + \imath v/\varphi$; but

$\varphi/z := \varphi/(x+\imath y/x\,y) - \imath(y/x)(\varphi/(x+\imath y/x\,y))$ if $|y| \le |x|$,

$:= (x/y)(\varphi/(y+\imath(x/y)x)) - \imath\varphi/(y+(x/y)x)$ if $|x| \le |y|$,

with due attention to set right over/underflows and zeros and infinities.

Ideally, the operators Re and Im , that select the Real and Imaginary part respectively, should be interpreted in a way that avoids unnecessary computation of the unwanted part whenever possible. For instance, Re(wz) should be evaluated by computing only ux-vy , without evaluating Im(wz) too. Besides saving time, this policy avoids spurious exceptions like over/underflow that might afflict only the unwanted part.

Note too, to conserve ± 0 , that -z is not 0-z though they be equal arithmetically; and similarly w-z is the same as -z+w but not -(z-w) . Multiplication or division by $\imath = \sqrt{-1}$ should be accomplished not by actual multiplication but rather by swaps and sign reversal; $\imath z := -y + \imath x$. In a similar way, an expression that is syntactically pure imaginary with an unsigned zero for its real part should be handled in a way that avoids both unnecessary arithmetic and unnecessary hazards. For instance, $\imath\beta + z := x + \imath(\beta+y)$, $(\imath\beta)z := \imath(\beta z)$, $z/(\imath\beta) := -\imath(z/\beta)$, $(\imath\beta)/z := \imath(\beta/z)$. In languages where a construction like CMPLX(x, y) is used to create the complex value $z := x + \imath y$, the expression CMPLX(0, β) should be treated as $\imath\beta$, whereas CMPLX(+0, β) and CMPLX(-0, β) should be treated as intentional attempts by the programmer to introduce an appropriately signed zero into the calculation. Of course, both attempts will produce the same CMPLX(+0, β) on a machine whose only zero is +0 .

Complex Zeros and Infinities

All four zeros $\pm 0 \pm \imath 0$ are arithmetically equal. Whether all complex infinities should be arithmetically equal is a topological question. When dealing with complex algebraic (not transcendental) functions, the most convenient topology is that of the Riemann sphere with its unique point at infinity. A *metric* (distance function) that induces that topology is the *Chordal Metric* :

$$\begin{aligned} \mathrm{Chord}(z, \zeta) &:= |z-\zeta| / \sqrt{(1+|z|^2)(1+|\zeta|^2)} \quad \text{if } |z| < \infty \text{ and } |\zeta| < \infty , \\ &:= \mathrm{Chord}(1/z, 1/\zeta) \quad \text{if } z \neq 0 \text{ and } \zeta \neq 0 ; \\ &\leq \mathrm{Chord}(0, \infty) := \mathrm{Chord}(\infty, 0) := 1 . \end{aligned}$$

In this topology, every algebraic function is a continuous (though perhaps multi-valued) map of the sphere to itself. So are our nine elementary functions f(z) . Only a function discontinuous at infinity can be affected by its multiplicity of representations there; an important instance is the equality predicate (z = ζ ?). To combat ambiguity at infinity a programmer can map all its representations upon one of them, namely real $+\infty$, by invoking the function

$$\begin{aligned} \mathrm{PROJ}(x + \imath y) &:= x + \imath y \quad \text{if } |x| \neq \infty \text{ and } |y| \neq \infty , \\ &:= +\infty + \imath\, \mathrm{copysign}(0, y) \quad \text{otherwise,} \end{aligned}$$

before performing any operation discontinuous at infinity. Note that PROJ lies on the same side of the real axis as its argument except for machines that lack a signed zero. And of course PROJ reduces to the identity function on machines that lack a way to represent ∞ . Otherwise

$$\mathrm{PROJ}(\infty+\imath\beta) = \mathrm{PROJ}(\beta\pm\imath\infty) = |\infty+\imath\beta| = |\beta\pm\imath\infty| = \infty \quad \text{for all real } \beta ,$$

finite or infinite, and hence also when β is NaN .

*** WARNING: Some implementors have overlooked that last inference. ***

Digression: Some programmers are understandably uneasy about allowing a

NaN to disappear from a computation leaving no trace that it ever existed. Since a NaN is a reserved operand associated with words like *indefinite*, *undefined* and *invalid*, one might naturally surmise that no computation that generated a NaN could produce a valid result too. That surmise is wrong. NaNs were designed to permit computation to continue in the face of incorrect or invalid intermediate results that turn out to be irrelevant later; that is why ways must exist for NaNs to disappear quietly. In particular, if a function $f(x, y)$ is so defined that $f(x_o, y)$ is a constant, the same for all finite and infinite y, then $f(x_o, y)$ must be independent of y; therefore $f(x_o, NaN) = f(x_o, y)$ too. Consequently $PROJ(\infty+i\,NaN) = |\infty+i\,NaN| = \infty$. Also $1/(\infty+i\,NaN) = 1/(NaN+i\,\infty) = 0$, which makes complex division more interesting to implement as well as annihilating a NaN. That NaN does not vanish without leaving a trace. IEEE standards require that the *Invalid Operation Flag* be raised whenever an arithmetic operation creates a NaN; and even if the program subsequently lowers that flag, a record that a NaN was created will be preserved by well-designed schemes for *retrospective diagnostics*. ... End of digression.

The topology of the Riemann sphere is inappropriate for functions like e^z that have an essential singularity at infinity. Instead, different representations of infinity are customarily associated with different paths that tend to infinity in some asymptotic way, justifying assertions like

$$\exp(-\infty+iy) = 0 \quad \text{and} \quad |\exp(+\infty+iy)| = \infty \quad \text{for all finite } y .$$

For example, " $\infty+i2$ " could represent a path asymptotically parallel to the positive real axis and 2 units above it; " $\infty+i\infty$ " would have to represent a path parallel to that traced by $\exp(\xi+i\theta)$ as $\xi \to +\infty$ for some fixed but unknown θ strictly between 0 and $\pi/2$. Unfortunately, programming languages like Fortran represent complex variables by pairs of reals in such a way as allows at most nine asymptotic directions (θ) to be represented by two real variables of which at least one is $\pm\infty$. Those directions are

θ:	$\pm\pi$	$-3\pi/4$	$-\pi/2$	$-\pi/4$	± 0	$\pi/4$	$\pi/2$	$3\pi/4$	NaN
z:	$-\infty \pm i\xi$	$-\infty-i\infty$	$\xi-i\infty$	$+\infty-i\infty$	$+\infty \pm i\xi$	$+\infty+i\infty$	$\xi+i\infty$	$-\infty+i\infty$	$NaN \pm i\infty$ or $\pm\infty \pm i\,NaN$

(Here ξ stands for any finite real number.)

These complex infinities z are the only ones available. By default, in the absence of some contrivance programmed explicitly to cope with other asymptotic directions, every infinite complex result, especially of multiplication and division, has to be approximated by something chosen from the available complex infinities z in a fashion resembling the way real numbers are rounded to the ones representable in floating-point. That default rounding, while fully satisfactory in the topology of the Riemann sphere, can approximate arbitrary asymptotic directions at best crudely.

Crudely, but not quite arbitrarily. The approximations should be predictable and consistent with reasonable expectations; in particular, it seems reasonable to expect fundamental relations like

$$w\,z = \exp(\ln(w) + \ln(z)) \quad \text{and} \quad w/z = \exp(\ln(w) - \ln(z))$$

to hold within an allowance for roundoff even for infinite or zero products and quotients. Those relations imply $|w\,z| = |w|\,|z|$ and $|w/z| = |w|/|z|$ at 0 and ∞; these equations can be satisfied exactly. But the relations $\arg(w\,z) = \arg(w) + \arg(z) \bmod 2\pi$ and $\arg(w/z) = \arg(w) - \arg(z) \bmod 2\pi$ have to be approximated within the set of ten values available for $\arg(\zeta)$ when ζ is zero or infinite. Those values turn out to be ...

$$\arg(+0 \pm i0) = \arg(+\infty \pm i\beta) = \pm 0 \text{ for all finite } \beta ,$$
$$\arg(+\infty \pm i\infty) = \pm\pi/4 ,$$
$$\arg(\beta \pm i\infty) = \pm\pi/2 \text{ for all finite } \beta ,$$
$$\arg(-\infty \pm i\infty) = \pm 3\pi/4 ,$$
$$\arg(-0 \pm i0) = \arg(-\infty \pm i\beta) = \pm\pi \text{ for all finite } \beta ;$$

$\arg(NaN + i\langle Anything\rangle)$ and $\arg(\langle Anything\rangle + iNaN)$ are both NaN . Thus, any coherent scheme for computing complex products, quotients and logarithms at zero and infinity can be regarded as a scheme that rounds $\arg(\zeta)$ into one of the ten values above when ζ is zero or infinite. To be acceptable, such a scheme should not add much to the cost of complex multiplication and division. The schemes that follow seems tolerable though far from ideal.

Digression: The fastest computers get that way by executing arithmetic operations concurrently, either in parallel (in multiple arithmetic units) or overlapped (in pipelines). Consequently, users of these machines tend to run roughshod over singularities and special cases that would insinuate into computer programs the kinds of tests and branches that, by waiting for the outcomes of prior operations before choosing which subsequent operations to perform, inhibit concurrency and retard computation. One way to divorce tests and branches from singularities is to remove them by *presubstitution*; this assigns in advance an exceptional value to be delivered by exceptional operations that need it. For example, the removable singularity at $x = 0$ in the expression $\sin(\lambda x)/\sinh(x)$, which approaches λ as $x \to 0$, could be removed for the purposes of a loop as follows:

```
presubstitute λ for 0/0 ;
for k = 1 to n do v_k := sin(λx_k)/sinh(x_k) ;  ... in parallel .
repeal presubstitution .
```

The presubstitution statement would send λ either to memory where a trap-handler could find it, or to a register inside the divider hardware, to be used in place of NaN when 0/0 occurs. The compiler for a vectorized machine without such a register in its divider would treat presubstitution as a request to compile the loop as if it had been written thus:

```
for k = 1 to n do v_k := if x_k=0 then λ else sin(λx_k)/sinh(x_k) ;
```

This would replace the values v_k that had become NaN after 0/0 by λ with the aid of vectorized boolean operations. Presubstitution's advantage over policies that react to exceptions after they occur is that implementing it without inhibiting concurrency is inexpensive provided the hardware is designed with that possibility in mind. ... End of digression.

Normally the definition of complex multiplication has to be free from tests and branches that would slow down a machine that executes arithmetic operations concurrently. The obvious definition appears to go that way:

Multiplication computes $\xi + i\eta := \zeta := w\,z = (u + iv)(x + iy)$ given w and z thus: $\xi := ux - vy$; $\eta := uy + vx$.

But closer scrutiny reveals three defects in these formulas. Minor defects are due to roundoff and underflow. Although roundoff can obliterate ξ or η it cannot damage both seriously because the magnitude of the rounding error in ζ cannot exceed a few rounding errors in $|\zeta|$. If underflow is abrupt to zero, it can corrupt ξ and η relatively badly; but if it is gradual, as it is when IEEE 754 or 854 holds sway, then it is ignorable unless $|\zeta|$ would underflow too. The third defect, overflow, is serious. Unless

$|\xi| + \eta^2/(4\Omega) < \Omega$ and $|\eta| + \xi^2/(4\Omega) < \Omega$,
where Ω is the overflow threshold, overflow could obliterate ξ or η (but not both) undesevedly. These inequalities describe a safe region that encloses the circle $|\xi+i\eta| < \Omega$ but lies inside the square $|\xi| < \Omega$ and $|\eta| < \Omega$, of which it occupies over 7/8 of the area. Outside the square overflow has to affect at least one of ξ and η . Between that safe region and the square neither ξ nor η should overflow but either could be ruined by overflow in at least one of the products ux, vy, uy, vx. What then?

Reluctantly, I have come to the conclusion that nothing can be done, to preclude spurious overflows outside the safe region, without incurring an overhead cost due to tests and branches that is intolerable on machines that execute arithmetic operations concurrently. Consequently, overflows during multiplication will generate complex infinities like $\infty + i\infty$ or $\infty + i$NaN , and subsequent multiplications or additions of complex infinities will too often produce NaN + i NaN unless the programmer inserts tests for overflows or invalid operations to undo them. Such tests must be used sparingly, not in the middle of loops in matrix multiplications. And for many situations a way exists to avoid tests almost entirely: use presubstitution instead. Here is how presubstitution works when it is used correctly. The statement

presubstitute 0 for ∞ 0 and $\infty-\infty$

causes what would have been invalid multiplications and subtractions to produce zero instead of NaN . The sign of zero is determined as usual, as if ∞ were a finite number. Afterwards, overflowed sums and products of finite complex numbers raise the overflow flag and produce a complex infinity without a NaN in it, and subsequent products of nonzero complex numbers by such a complex infinity produce another complex infinity. These infinities are produced, instead of NaNs, because one of the expressions

ux - vy , uy + vx

has to be an addition of magnitudes, the other a subtraction, so if $\infty-\infty$ occurs in one of them then $\infty+\infty$ or $\infty \times$ (finite) must occur in the other. Thus, presubstitution conserves complex infinities that contain no NaN . But subtractions of complex infinities and multiplications of them by zero can produce plausible but wrong finite numbers while that presubstitution is in force, so this expedient should not be used indiscriminately; ...

repeal presubstitution

after presubstitution has outlived its usefulness. If implemented properly, neither invoking presubstitution nor repealing it will inhibit concurrency.

Alternatives to presubstitution are functions that filter out unwanted NaNs after they have been created. An example is the function PROJ mentioned before, and others of a similar kind could be created as needed, for instance ...

CORNER(x+iy) := x + i copysign(∞,y) if |x| = ∞ and y is NaN ,
:= copysign(∞,x) + iy if |y| = ∞ and x is NaN ,
:= x + iy otherwise.

Because such filters contain tests and branches, they must be slower than presubstitution unless (as is most unlikely) they be built into hardware to run as fast as any other floating-point operation.

Reasoning similar to the justification for presubstitution leads to the inference that if w and z contain no NaN but should produce an infinite product because of overflow or otherwise, then their crudely computed product w*z will have either a real or an imaginary part that is infinite,

so filtering will produce a complex infinity with no NaN , but possibly not the same infinity as produced by presubstitution.

On strictly sequential computers that tolerate tests and branches well, the definition of complex multiplication should include automatically the tests and appropriate corrective measures for exceptional multiplications, as has been done on the HP-71B, although the relentless urge for speed may tempt compiler writers to include them only when a programmer asks for them explicitly by calling upon a complex-valued function CMULT(w, z) defined perhaps thus:

```
CMULT(u+iv, x+iy):  ...  =  (u+iv)(x+iy)  allowing for infinities and NaNs.
     Save and Lower the Overflow and Invalid flags ;
     ξ := ux - vy ;    η := uy + vx ;
     if  (neither  ξ nor η  is  NaN)
        then { Restore the Invalid flag's saved value ;
               if the saved  Overflow  flag was raised
                    then  Restore its saved value ;
               Return (ξ + iη)  } ...  the normal case.

        else { ... now take care of unusual cases ...
               if  (Invalid flag was raised)
                  then { if (u=0 and v=0 or x=0 and y=0)
                              then { ... invalid  0*∞  case
                                     if the saved Invalid flag was raised
                                        then restore its saved value ;
                                     restore the Overflow flag's saved value ;
                                     return (ξ + iη)  } ... NaNs exit.
                  else { ... filter out unwanted  NaNs ...
                         (u+iv) := CORNER(u+iv) ;
                         (x+iv) := CORNER(x+iy) } ;
               Restore both flags' saved values ;
               Presubstitute  ξ  for  0*∞  and  ∞-∞ :
               ξ := ux - vy ;   η := uy + vx ;
               Repeat presubstitution ;
               Return (ξ + iη)  } ;  end CMULT .
```

The code that precedes the first *else* may be emitted inline, leaving the rest for an exception-handling subroutine. That subroutine defines the way infinite complex products are rounded into the set of numbers available for them. It is not so nice a rounding as has been implemented on the HP-71B but it is the only one I have found compatible with what will have to be done on the fastest computers, pipelined, vectorized and concurrent.

Complex division ξ+iη := ζ := w/z = (u+iv)/(x+iy) cannot be freed from tests and branches, so it might as well be a subroutine. In the absence of special operands w and z , the nastest algorithm would be one, traceable to Robert L. Smith, that does something like this:

```
        (u+iv) := CORNER(u+iv) ;  (x+iy) := CORNER(x+iy) ;
        if  |x| < |y|  then replace  "(u+iv)/(x+iy)"  by  "(v-iu)/(y-ix)" ;
        ...  now  |x| ≥ |y| ,  or at least one is NaN and signaled invalid
        if  |y| = ∞  then  y := copysign(1.,y) ;
        if  x=0 and y=0  then  Return  CMULT(u+iv,  1/x - iy/y ) ; ...  exits.
        r := y/x ;  d := ry + x ;  ...  |r| ≤ 1 ≤ d/x ≤ 2  normally
        Return ( (rv+u)/d + i(v-ru)/d ) ;  end .
```

The main trouble with this code is that it can suffer from spurious overflow in case either 2w or 2z would overflow, and it can lose accuracy if w or z has underflowed but remains nonzero. These troubles are so rare they may go unnoticed by all but the most conscientious programmer; and even she might overlook the underflow signals that can be generated when one of ry, rv or ru underflows, though those signals are ignorable only if underflow is gradual (IEEE 754) and neither |z| nor |w| underflows. A better but slower (unless multiplication is enormously faster than division) way is this:

```
CDIV(u+iv, x+iy):  ... = (u+iv)/(x+iy)  carefully.
     u+iv := CORNER(u+iv) ;  x+iy := CORNER(x+iy) ;
     if any of  u, v, x, y  is  NaN
        then  Return  ( 0*u+v*x + i(0*u+v*y) ) ;  ...  NaN exit.
     Save all flags ;
     L := integer nearest Logb(Ω)/2 - 1 ; ...  Ω is the overflow threshold.
     n := Logb(max(|x|,|y|)) ;  if  |n| = ∞  then  n := copysign(8L, n) ;
     K := L - (integer nearest n ) ;
     x := Scalb(x, K) ;  y := Scalb(y, K) ;
     h := Logb(max(|u|,|v|)) ;  if  |h| = ∞  then  n := copysign(8L, h) ;
     J := L - (integer nearest h ) ;
     u := Scalb(u, J) ;  v := Scalb(v, J) ;
     d := x² + y² ; ...  cannot over/underflow.
     if  d=∞ or d=0  then  {  if |x|=d then  x := copysign(1,x) ;
                              if y≠0 then  y := copysign(1,y)  } ;
     ξ+iη := CMULT(u+iv, x-iy) ;
     restore all flags ;
     Return  ( Scalb(ξ/d, K-J) + iScalb(η/d, K-J) ;  end CDIV .
```

This program appears to produce neither spurious signals nor spurious over/underflows , and it preserves the identity |w/z| = |w|/|z| at 0 and ∞ , and it produces correctly rounded results for small Gaussian integer inputs.

```
To compute   ρ  :=  |z|  =  |x+iy|  =  √(x²+y²)  ,  ...
ABS(x+iy):   ...   =  Fortran's  CABS(Z)   =  C's  hypot(x,y)   ...
     ...  The obvious formula can produce errors bigger than one ulp .
     ...  and could over/underflow spuriously.  Not so for what follows: ...
     Constants  r2 := √2 ,  r2p1 := 1+√2 ,  t2p1 := 1+√2 - r2p1 ;
        ...  These constants must be correctly rounded to working precision;
        ...  consequently   r2p1 + t2p1 = 1+√2  to double that precision.
     Save  Invalid flag ;  ...   This suppresses spurious  Invalid Operation
                   ...  signals from  NaN  comparison or  ∞-∞ ;  but spurious
                   ...  Inexact  signals can be generated by this program.
     x := |x| ;   y := |y| ;   s := 0.0 ;
     If  x < y  then  swap  x and y ;   ...  so  x ≥ y ≥ 0  if not  NaN .
     If  y = ∞  then  x := y ;
     t := x - y ;
     If  x ≠ ∞  and  t ≠ x  then
          {  ...  executed if  x ≠ ∞  and  y ≠ ∞  and  y  is not negligible.
          Save Underflow flag;
          If  t > y  then  ...  when  2 < x/y < 2/ε ,  ...
                              {  s := x/y ;   s := s + √(1+s²) }
                         else  ...  when  1 ≤ x/y ≤ 2 ,  ...
                              {  s := t/y ;   t := (2+s)s ;
                                 s := ((t2p1 + t/(r2 + √(2+t))) + s) + r2p1  };
          s := y/s   ...  Harmless  Gradual Underflow  can occur here.
          Restore  Underflow flag ;
          };
     Restore  Invalid flag ;  ...  Only if deserved can Overflow happen now.
     Return  x + s ;   end  ABS .
Another version of  ABS  would use  CBSQS  below in an obvious way.


To compute   θ  :=  arg(z)  =  arg(x + iy)  ,  ...
ARG(x + iy):   ...   =  Fortran's  ATAN2(y, x)  ...
     If  x = 0  and  y = 0   then  x := copysign(1, x) ;
     If  |x| = ∞  or  |y| = ∞   then  z := CBQX(z) ;
                                       ... leaves signs unchanged.
     If  |y| > |x|    then  θ := copysign(π/2, y) - arctan(x/y)
     else if  x < 0   then  θ := copysign(π, y) + arctan(y/x)
                      else  θ := arctan(y/x) ;
     Suppress any  Underflow signal  unless  |θ| < 0.125 ,  say.
     ...  Better accuracy may be obtained by further case reduction and use
     ...  of identities like   arctan(y/x)  =  π/4 + arctan((y-x)/(y+x)) .
     Return  θ ;   end  ARG .
```

```
To compute   x + iy  =  z  :=  ζ²  =  (ξ + iη)²  ,  ...
CSQUARE(ξ + iη):
     x := (ξ-η)(ξ+η) ;   ...  not  ξ²-η² ,
     y := ξη + ξη ;      ...  ONE multiply, one add.
     ... If a spurious  NaN  is created by overflow it gets removed thus:
     If  x≠x  then
                 (  if  |y|=∞  then  x := copysign(0, ξ)
                    else if  |η| = ∞  then  y := -∞
                    else if  |ξ| = ∞  then  x := ∞  )
     else if  y≠y  and  |x|=∞  then  y := copysign(0, y) ;
     Return  ( x + iy ) ;  end  CSQUARE .
```

The principal use for CSQUARE is for raising a complex number to an integer power by repeated squaring.

```
To compute   ρ := |(x+iy)/2^k|²  scaled to avoid  Over/Underflow ...
CSSQS(x + iy):   ...  =  ρ + iκ ,   with an  integer  k  ...
     Integer  k ;
     k := 0 ;
     Save and lower the  Over/Underflow flags ;
     ρ := x² + y² ;  ...  Multiply twice and add.
     If  ( ρ is not finite ) and ( |x| = ∞  or  |y| = ∞ )  then  ρ := ∞
     else if   (  the Overflow flag was just raised, or
                  the Underflow flag was just raised and  ρ < λ/ε  )
          then (  k := logb( max( |x|, |y| ) ) ;
                  ρ := scalb(x, -k)² + scalb(y, -k)²   ) ;
     Restore the  Over/Underflow flags ;
     Return  ( ρ + iκ ) ;   end  CSSQS.
```

```
To compute   ξ + ιη  =  ζ  :=  √z  =  √(x + ιy)  ,  ...
CS@RT(x + ιy):
     Real  ρ ;  Integer  k ;
     ρ + ιk  :=  CSS@S(x + ιy) ;  ...  Sum-of-Squares Scaled;  see above.
     If  x = x  then  ρ := scalb( |x|, -k) + √ρ ;
     If  k is odd  then     k := (k-1)/2
                   else  {  k := k/2 - 1 ;  ρ := ρ + ρ  }  ;
     ρ := scalb( √ρ, k) ;  ... = √( (|x+ιy| + |x|)/2 ) without over/underflow
     ξ := ρ ;  η := y ;
     if  ρ ≠ 0  then
               {  if  |η| ≠ ∞  then  {   η := (η/ρ)/2 ;
                                         if  η  underflowed,  signal it } ;
                  if   x < 0   then  {   ξ := |η| ;
                                         η := copysign(ρ, y) }
               }
     Return  ( ξ + ιη ) ;
     ...
     ...  This program appears to handle all special cases correctly:
     ...  √(-ρ ± ι0) = +0 ± ι√(ρ)   for all  ρ ≥ 0 ,
     ...  √(x ± ι∞) = +∞ ± ι∞  for  all  x ,  finite, infinite or  NaN ,
     ...    and if  x  is  NaN  then  "Invalid Comparison"  is signaled too.
     ...  For all  finite  ρ ,
     ...     √(NaN+ιρ),  √(ρ+ιNaN)  and  √(NaN+ιNaN)  are all  NaN + ιNaN ;
     ...     √(+∞ ± ιρ) = +∞ ± ι0 ;     √(+∞ ± ιNaN) = +∞ + ιNaN ;
     ...     √(-∞ ± ιρ) = +0 ± ι∞ ;     √(-∞ ± ιNaN) = NaN ± ι∞ .
     End  CS@RT .


To compute   ξ + ιη  =  ζ  :=  ln(2^J z)  =  ln(2^J (x+ιy))  with  integer J, ...
CLOGS(x + ιy , J):  ...  for use with  J ≠ 0  only when  |x+ιy|  is huge.
     ...  This program is particularly helpful for inverse trigonometric and
     ...  hyperbolic functions that behave like  ln(2z)  for huge  |z| .
     ...  This program uses a subprogram  lnlp(x) := ln(1+x)  presumed to be
     ...  available with full relative accuracy for all tiny real  x .  Such
     ...  a program exists in various math. libraries,  including that for
     ...  4.3 BSD Unix,  Intel's CEL and  Apple's SANE.  The accuracy of
     ...  lnlp  influences the choice of thresholds  T0, T1 and T2 .
     Constants  T0 := 1/√2 ;  T1 := 5/4 ;  T2 := 3 ;  ln2 := ln(2) ;
     Real  ρ ;  Integer  k ;
     ρ + ιk := CSS@S(x + ιy) ;   ... = |(x+ιy)/2^k|² + ιk ;  see above.
     ϱ := max( |x|, |y|) ;  θ := min( |x|, |y|) ;
     If  k = 0  and  T0 < ϱ  and ( ϱ ≤ T1  or  ϱ < T2 )
               then  ρ := lnlp( (ϱ-1)(ϱ+1) + θ² )/2
               else  ρ := ln(ρ)/2  + (k+J) ln2 ;
     θ := ARG(x + ιy) ;
     Return   ( ρ + ιθ ) ;   end  CLOGS.


To compute   ξ + ιη  =  ζ  :=  ln(z)  =  ln(x + ιy) , ...
CLOG(z) := CLOGS(z, 0) .
```

```
To compute    ξ + ιη  =  ζ  :=  arccos(z)  =  arccos(x + ιy)  ,  ...
CACOS(z):      ... based upon formulas
     ...       ξ := 2 arctan( Re(√(1-z))/Re(√(1+z)) ) ;
               ...  suppress any  Divide-by-Zero signal  when  z ≤ -1 .
     ...       η := arcsinh( Im( √(1+z)·√(1-z) ) ) ;
     Return ( ξ + ιη ) ;  end  CACOS .


To compute    ξ + ιη  =  ζ  :=  arccosh(z)  =  arccosh(x + ιy)  ,  ...
CACOSH(z):     ... based upon formulas
     ...       ξ := arcsinh( Re( √(z-1)·√(z+1) ) ) ;
     ...       η := 2 arctan( Im(√(z-1))/Re(√(z+1)) ) ;
               ...  suppress any  Divide-by-Zero signal  when  z ≤ -1 .
     Return ( ξ + ιη ) ;  end  CACOSH .


To compute    ξ + ιη  =  ζ  :=  arcsin(z)  =  arcsin(x + ιy)  ,  ...
CASIN(x + ιy): ... based upon formulas
     ...       ξ := arctan( x/Re( √(1-z)·√(1+z) ) ) ;
               ...  suppress any  Divide-by-Zero signal  when  z ≤ -1 .
     ...       η := arcsinh( Im( √(1-z)·√(1+z) ) ) ;
     Return ( ξ + ιη ) ;  end  CASIN .


To compute    ξ + ιη  =  ζ  :=  arcsinh(z)  =  arcsinh(x + ιy)  ,  ...
CASINH(z) := -ι CASIN(ιz) .


To compute    ξ + ιη  =  ζ  :=  arctanh(z)  =  arctanh(x + ιy)  ,  ...
CATANH(x + ιy):
     Constants  θ := √(Ω)/4 ,   ρ := 1/θ ;
     ρ := copysign(1, x) ;  z := ρ z· ;  ... copes with unsigned 0
     if  x > θ  or  |y| > θ  ...  to avoid overflow ,
          then   {   η := copysign(π/2, y) ;   ξ := Re(1/(x+ιy))  }
     else if  x = 1
          then   {   ξ := ln(√(|y|)/√(√(4+(|y|+ρ)²))) ;
                     η := copysign( π - arctan((|y|+ρ)/2) , y )/2   }
          else   ... Normal case ...
           {...  using  lnlp(u) := ln(1+u)  accurately even if  u  is tiny,
              ξ := lnlp( 4x/((1-x)² + (|y|+ρ)²) )/4 ;
              η := aro( (1-x)(1+x)-(|y|+ρ)² + 2ιy )/2
            } ;  ...  all exceptional cases appear to be handled correctly.
     Return  ( ρ ζ· ) ;  end  CATANH .


To compute  ξ + ιη  =  ζ  :=  arctan(z)  =  arctan(x + ιy)  ,  ...
CATAN(z) := -ιCATANH(ιz) .
```

```
To compute  x + iy  =  z  :=  tanh(ζ)  =  tanh(ξ + iη)  ,  ...
CTANH(ξ + iη):
     If  |ξ| > arcsinh(Ω)/4  ... avoid overflow ...
          then  z := copysign(1, ξ) + i copysign(0, η)
          else  {
                  save Divide-by-Zero flag ;
                  t := tan(η) ; ... suppress any Div-by-Zero signal here.
                  restore  Divide-by-Zero flag ;
                  ϱ := 1 + t² ;  ...  1/cos²η  ...
                  s := sinh(ξ) ;
                  ρ := √(1 + s²) ;  ...  cosh ξ  ...
                  if  |t| = ∞  then  z := ρ/s + i/t  ...  may signal if s=0
                               else  z := ( ϱρs + it )/( 1 + ϱs² )
                } ;
     return  z  ;   end  CTANH .


To compute  x + iy  =  z  := tan(ζ)  =  tan(ξ + iη)  ,  ...
CTAN(ζ) := -i CTANH(iζ) .


>>>>>>>>>>>>>     Still to come are details about     <<<<<<<<<<<
                CLOGIP , CEXP , CPOWER , Ctrigs
          though most of these details are now predictable.
```

The exponential function z^w , and 0^0 .

The function z^w has two very different definitions. One is recursive and applicable only when w is an integer:

$z^0 = 1$ and $z^{(w+1)} = z^w z$ whenever z^w exists.

The second definition is analytic:

$z^w := \lim_{\zeta \to z} \exp(w \ln(\zeta))$ provided the limit exists using the principal value and domain of $\ln(\zeta)$.

The limit process is necessary to cope smoothly with $z = 0$. Since the recursive definition makes sense when z is a number or a square matrix or a nonlinear map of some domain into itself, regardless of whether $\ln(z)$ exists, the fact that both definitions coincide when w is an integer and $\ln(z)$ exists must be a nontrivial theorem. The fact that both definitions agree that $z^0 = 1$ for every z is doubly significant because programmers who have implemented z^w on computers have so often decreed 0^0 to be a capital offense.

I can only speculate on why 0^0 might be feared. Perhaps fear is induced by the singularity that z^w possesses at $z = w = 0$; if both z and w are compelled to approach 0 but allowed to do so independently along any paths, then paths may be chosen on which z^w holds fast to any preassigned values whatsoever. Assuming for the sake of argument (because it is generally not so) that neither z nor w could be exactly zero but must instead be approximately zero because of roundoff or underflow, the

expression 0^0 would have to be treated as if it really ought to have been $(roundoff)^{roundoff}$, which generally defies estimation.

To draw conclusions based upon something better than fear or speculation, we need estimates for certain costs and benefits. Setting $z^0 := 1$ without exception confers the benefit of adherence to simply stated rules; but it introduces some risk that we might unwittingly accept 1 for 0^0 instead of an unknown but preferred value ζ^ν with tiny ζ and ν. That added risk should be judged in the light of the greater and unavoidable risk that z^w might unwittingly be accepted when z and w are both nonzero but tiny and quite wrong because of roundoff. In other words, only on those extremely rare occasions when a program of unknown reliability betrays its inaccuracy by a chance encounter with 0^0 will we benefit from outlawing 0^0. But outlawing 0^0 incurs the cost of departing from a simple rule; it imposes upon those programmers who prefer to take $z^0 = 1$ for granted, regardless of whether $z = 0$, the extra burden of remembering to insert extra code to cope with a rare eventuality.

There are two situations in which programmers are fully entitled to take $0^0 = 1$ for granted. The first arises in languages like Fortran and Pascal that distinguish variables of type INTEGER from floating-point variables of type REAL and COMPLEX. Suppose that M is of type INTEGER but w has a floating-point type; then z^M can be distinguished from z^w, and particularly z^0 from $z^{0.0}$, because they call upon different subroutines from a library of intrinsic functions. Since roundoff cannot possibly obscure the value of an exponent M of type INTEGER in the way it might obscure the value of a floating-point variable w that happens to vanish, there is no reason to doubt that $z^0 = 1$ for every z regardless of one's fears about $0.0^{0.0}$. Therefore, in every language in which M can be declared of INTEGER type, the exponential function z^M must be consistent with its recursive definition even if computed, at least when $|M|$ is huge, with the aid of logarithms; in short,

when $M = 0$ then $z^M = 1$ regardless of z.

A second situation in which programmers might presume that $0.0^{0.0} = 1$ arises frequently. Consider two expressions $z := z(\xi)$ and $w := w(\xi)$ that depend upon some variable ξ, and suppose that $z(\xi) = w(\xi) = 0$ and that z and w are analytic functions of ξ in some open neighborhood of $\xi = \xi$. This means that $z(\xi)$ and $w(\xi)$ can be expanded in Taylor series in powers of $\xi-\xi$ valid near $\xi = \xi$, and both series begin with positive powers of $\xi-\xi$. Then we find that $z \to 0$ and $w \to 0$ and $z^w = \exp(w \ln(z)) \to 1$ as $\xi \to \xi$ regardless of the branch chosen for $\ln$. Since this phenomenon occurs for *all* pairs of analytic expressions z and w, it is very common.

In the light of the foregoing considerations, $0.0^{0.0} = 0^0 = 1$ seems to be the only reasonable choice; similar considerations imply $\infty^{0.0} = \infty^0 = 1$ too, and consequently $z0.0 = 1$ foe every z including NaN.

Some other exponential expressions involving infinite operands require further thought. For instance, 1.0^∞ is clearly an invalid operation, but $1^\infty = 1$ might be acceptable. Somewhat less clear are the signs of results like

$$(\pm 0.5)^\infty = 0^\infty = (\pm 2)^{-\infty} = (\pm\infty)^{-\infty} = 0 \quad , \quad \text{and}$$
$$(\pm 0.5)^{-\infty} \ , \quad 0^{-\infty} \ , \quad (-2)^{-\infty} \ , \quad (-\infty)^\infty \ , \quad \text{all } +0 \ .$$

It is possible to argue that all these results should be assigned + signs in real arithmetic on any North American computer; the argument goes thus:

> Since all sufficiently big floating-point numbers on such machines are even integers, taking the limit makes ∞ an even integer too.

Whether equally fulgent reasoning can be applied to complex arithmetic remains to be seen. And whether $0^{-\infty} = \infty$ should signal *Division by Zero*, as 0^{-1} and $1/0$ must, is no matter of taste; no signal is needed for $0^{-\infty}$ because "Division by Zero" is a misnomer imposed for historical reasons in place of the more appropriate phrase

> "an infinite result produced exactly from finite operands."

When z is neither zero nor infinite, and when w is not an integer, the complex function z^w could be assigned a multiplicity of values; they are arranged around a circle if w is real, or otherwise along an Archimedean spiral in the complex plane. What distinguishes the Principal Value defined above from all others is that its logarithm has minimum magnitude; this definition is conventional. Respectable accuracy can be difficult to achieve when either $|w|$ or $|w \ln(z)|$ is big, requiring extraordinarily careful calculation of $\ln(z)$, but that is a story to unfold elsewhere.

Acknowledgments

I am indebted to Prof. Paul Penfield Jr. of M.I.T. for a conversation that illuminated some of the reasons behind the differences between his APL proposal and the complex arithmetic implemented on the hp-15C by Dr. J. Tanzini, then at Hewlett-Packard. The author's own work has been supported in part also by grants from the U.S. Department of Energy, the Office of Naval Research, and the Air Force Office of Scientific Research.

This manuscript is an extension of, and completely supersedes, an earlier version that appeared in Sept. 1982 as report PAM-405 of the Center for Pure and Applied Mathematics at the University of California at Berkeley. That version was prepared as an exercise on an APPLE /// using the PASCAL editor and Colin McMaster's SCRIPT /// text formatter slightly modified. The author thanks his friends at APPLE for that opportunity.

Another version of this paper appears in the proceedings of a joint IMA/SIAM conference on "The State of the Art in Numerical Analysis" held at the University of Birmingham on April 14-18, 1986, edited by A. Iserles and M. J. D. Powell for the Oxford University Press, 1987. This later version supersedes that one.

I am grateful to Prof. B. N. Parlett and to Prof. M. J. D. Powell for several suggestions that helped to clarify muddy spots in the paper.

BIBLIOGRAPHIC NOTES

Penfield's proposal "Principal Values and Branch Cuts in Complex APL" appeared in *APL Quote Quad* vol. 12, no. 1, Sept. 1981.

The complex functions implemented in the hp-15C are described in

Section 3 of the *Hewlett-Packard* HP-15C *Advanced Functions Handbook*, Aug. 1982, part no. 00015-90011. The formulas that tell where that calculator puts the branch cuts were first published in an article "Scientific Pocket Calculator Extends Range of Built-In Functions" by Eric Evett, Paul McClellan and Joe Tanzini in the *Hewlett-Packard Journal* of May 1983, vol. 34 no. 5, pp. 25 - 35. More about that calculator, plus a formula for computing arccosh accurately, may be found in my paper "Mathematics Written in Sand," pp. 12 - 26 in the Statistical Computing Section of the *Proceedings* of the Joint Statistical Meetings of the American Statistical Association etc. held in Toronto in August 1983. The conformal map onto a slotted strip is adapted from that paper.

The ANSI/IEEE standard 754-1985 for Binary Floating-Point Arithmetic has numerous other features not mentioned herein. Its specifications are available as stock number SH10116 from the IEEE Service Center, 445 Hoes Lane, Piscataway NJ 08854 ; telephone (201) 981-0060 . A more readable exposition of 754 and the newly approved Binary and Decimal Standard 854 has been published in pp.86-100 of the Aug. 1984 issue of the IEEE magazine *MICRO* ; to obtain a reprint from the IEEE , cite document number 0272-1732/84/0800-0086 . Early versions of 754, now superseded, plus some supporting materials have appeared in the March 1981 and January 1980 issues of the IEEE magazine *Computer*, and in a special issue, October 1979, of the ACM *SIGNUM Newsletter*. Implementations of IEEE 754 abound, ranging in size and speed from the ELXSI 6400 to the Apple II and Macintosh. The Standard Apple Numerical Environment (SANE) is now the most thorough implementation, and is documented in the *Apple Numerics Manual* published in 1986 by Addison-Wesley, Reading, Mass.

The Intel 18087 and 180287 floating-point coprocessor chips were designed to conform to an early draft of IEEE 754; they very nearly conform to the present standard. Though widely used in the IBM PC, PC/XT and PC/AT, they are not yet well supported by software in that realm. A fine library of elementary functions for them, real ones coded by Steve Baumel, complex by Dr. Phil Fallace, comes with Intel's Fortran for its 286/310 and 286/330 computers running under both Xenix and RMX86 operating systems. That library's algorithms are much like ours above. The real functions are documented in Intel's *80287 Support Library Reference Manual* (1983), order no. 122129. Real functions similar to those, and almost as accurate, are implemented on the Motorola 68881 and documented in the *MC68881 Floating-Point Coprocessor User's Manual* (1985, preliminary edition), order no. MC68881UM/AD. I do not yet have public documentation for analogous libraries running on the ELXSI 6400 (programmed by Peter Tang), on the National Semiconductor 32081 floating-point slave processor chip, and on the IBM RT/PC. The latter two machines' libraries are very much like the *C* Math Library for IEEE 754 - conforming machines programmed mostly by Dr. Kwok-Choi Ng and now distributed with 4.3 BSD UNIX by the University of California at Berkeley; that library is intended ultimately to be distributed independently of Berkeley UNIX.

The hp-71B is currently the only implementation in Decimal arithmetic of 854 ; that hand-held computer is the subject of the July 1984 issue of the *Hewlett-Packard Journal*, vol.35 no. 17. Many of the complex elementary functions, plus PROJ , have been implemented in the hp-71B 's Math Pac , HP 82480A; but its implementors were compelled by limitations

upon time and space to acquiesce to a few compromises that I wish they could have avoided. For instance, users of that machine have to write Z*Z instead of Z^2 to compute z^2, and (-IMPT(Z), REPT(Z)) instead of (0, 1)*Z to compute iz, if they wish to conserve the sign of zero.

Some of the ideas that lead to canonical formulas around branch-points are explained in pp.276-286 of volume III of A. I. Markushevich's *Theory of Functions of a Complex Variable* translated by R. I. Silverman, 1967, Prentice-Hall, N.J. The conformal map from the right half-plane to a liquid jet was adapted, with corrections, from pp.122-5 of *Theory of Functions as applied to Engineering Problems* edited by Rothe, Ollendorf and Pohlhausen, translated by Herzenberg in 1933, reprinted in 1961 by Dover, N.Y. Another Dover reprint is the *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* edited by M. Abramowitz and Irene Stegun, issued originally in 1964 as no. 55 in the U. S. National Bureau of Standards Applied Math. Series. Its Chapter 4 locates the slits for all nine elementary functions considered here, but its formulas 4.4.37-9 for complex Arcsin, Arccos and Arctan are non-committal on the slits and generally vulnerable to roundoff; and it lacks a formula for complex Arccosh. During the Handbook's ninth reprinting its definition of *arccot(z)* changed from $\pi/2$-arctan(z) to arctan(1/z). Finally, H. Kober's *Dictionary of Conformal Representations* contains pictures of many useful conformal maps; this too was reprinted by Dover, in 1957.