Prologue to
Branch Cuts for Complex Elementary Functions

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

This manuscript has been prepared on an IBM PC to be printed on an EPSON FX80 printer with a font of the author's making downloaded beforehand. As stored in the computer, the manuscript contains various control characters and other characters used in a nonstandard way to print mathematical symbols from that font. This page describes what those characters do.

Most of the manuscript is set in an Elite font, 12 characters per inch. Subscripts and superscripts are set in a Condensed font, about 17 or 18 characters per inch. Headings are set in a Pica font, 10 characters per inch, or less when "Proportional". Font changes are controlled thus:

Ctrl O Esc M sets default to Elite and prepares switch to Condensed.
Ctrl N switches to Double Width; Ctrl T switches back.
Esc G turns on **Double-Strike**; Esc H turns it off.
Esc P switches to Condensed; Esc M switches back to Elite.
Esc P Ctrl R switches to Pica; Ctrl O Esc M gets back to Elite etc.
Esc E switches Pica to **Bold Pica**; Esc F switches back.
Esc p 1 switches Pica to Proportional Bold; Esc p 0 switches back.

Esc 4 switches to *Italics* ; Esc 5 switches back.
Esc S 0 turns on Superscripts ; Esc S 1 turns on Subscripts ;
Esc T turns Superscripts and Subscripts off.

Ctrl H = nondestructive backspace, for overstriking ≠ , ≤ , etc.

Ctrl B = sqrt √	Ctrl D = iota ι	Ctrl W = Infinity ∞
Ctrl } = high : :	Ctrl 6 = Delta Δ	Ctrl - = pi π
Ctrl Q = Gamma Γ	Ctrl V = umlaut	Ctrl C = Norm bars ‖
ASCII 128 = beta β	ASCII 135 = rho ρ	ASCII 153 = eta η
ASCII 147 = xi ξ	ASCII 136 = chi χ	ASCII 154 = zeta ζ
ASCII 139 = Omega Ω	ASCII 130 = Esc 4 Ctrl B Esc 5 = epsilon ε	
	ASCII 133 = Esc 4 Ctrl E Esc 5 = lambda λ	

NOTE: During file transfers to diverse computer systems, take care NOT to lose each byte's most-sig. bit lest β become NULL, ρ become BELL, etc.

For several years this manuscript has been accreting refinements and improvements, some suggested by readers. The author welcomes all such suggestions.

WORK IN PROGRESS

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 √ , 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.

WORK IN PROGRESS

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

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

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

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

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 sign*(*y,x*)* mentioned above. However, a few extraordinary arithmetic operations *must be* affected by zero's sign; for example $1/(+0) = +\infty$ but $1/(-0) = -\infty$. 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 $0 \neq t \neq \infty$, 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 $z = \pm 0$. 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 $z^* = x-\iota y$ denote the complex conjugate of $z = x+\iota y$; 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-points*; 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 $-\imath$ and $+\imath$ 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 $z \le +1$. 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.

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

$\sqrt{(-1 + \iota 0)} = +0 + \iota = \lim \sqrt{(-1 + \iota y)}$ at $y = 0+$;
$\sqrt{(-1 - \iota 0)} = +0 - \iota = \lim \sqrt{(-1 + \iota y)}$ 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.

$\arccos(2 + \iota 0) = +0 - \iota\,\mathrm{arccosh}(2) = \lim \arccos(2 + \iota y)$ at $y = 0+$,
$\arccos(2 - \iota 0) = +0 + \iota\,\mathrm{arccosh}(2) = \lim \arccos(2 + \iota y)$ at $y = 0-$.
An implementation of arccos that preserves full accuracy in the imaginary part of $\arccos(2 + \iota y)$ when $|y|$ is very tiny can be expected to get its sign right when $y = \pm 0$ 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

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

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 $\ln(z) = \ln(\rho) + \iota\theta$, where $\rho = |z|$ and $\theta = \arg(z)$ are the polar coordinates of $z = x + \iota y$ 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

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

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

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

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 $z^2 < 0$ 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 $z - 1$ into $z - (1+\imath 0)$, 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 $z = x + \imath y$ 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

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

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; $F(x \pm \imath 0) = f(x \pm \imath 0)$ 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 $F(x + \iota y) = f(x + \iota y)$, 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:

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

~~~~~~

```
          ->  liquid flow  ->       ιπ                      |
--- wall -----------------------------.                air  |
               air                   ':       ↓             |ι
.....................................'/                     :    liquid
                                                            :
         <-   liquid jet    <-        <- liquid flow      - 0 - <- - <- -
.....................................                       :
               air                   ':       ↑             :
___ wall ____________________________/                 air  |-ι
          ->  liquid flow  ->      -ιπ   ζ - plane          |  z - plane
```

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 $z = x + \iota y$ to the plane of $\zeta = \xi + \iota\eta$, 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 $\pm\iota$. The free surfaces of the jet, curving forward from $\zeta = \pm\iota\pi$ and then back to $\zeta = -\infty \pm \iota\pi/2$, should be the image of that segment of the imaginary axis $-1 < z^2 < 0$ between $\pm\iota$.

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 $z/\iota < -1$, is mapped during the computation of $f(z)$ into the slit that belongs to $\sqrt{}$ and ln ; the upper part $z/\iota > 1$ 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 $\pm\iota$. 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 $c(z) := z - \iota\sqrt{(\iota z+1)}\sqrt{(\iota z-1)}$, whose slit lies in the imaginary axis $-1 < z^2 < 0$ between $\pm\iota$. Now $\zeta := c(z)$ maps the slit z plane onto the ζ plane outside the circle $|\zeta| \geq 1$; 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 $z/\iota < -1$, 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 $z = \pm\iota$ 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 $h(x) := x + x^2$ 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 $\mathrm{Im}(h(z))$ on various segments of the real z-axis; to get the correct sign and better accuracy requires an expression like

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

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 preduced 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.