

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+iy) := CORNER(x+iy) ;
                    Restore both flag's saved values ;
                    Presubstitute 0 for 0∞ and ∞-∞ ;
                    ξ := ux - vy ; η := uy + vx ;
                    Repeal 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 $\xi+i\eta := \zeta := w/z = (u+iv)/(x+iy)$ 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 neatest algorithm would be one, traceable to Robert L. Smith, that goes something like this:

```

(u+iv) := CORNER(u+iv) ; (x+iy) := CORNER(x+iy) ;
if |x| < |y| then replace "(u+iv)/(x+iy)" by "(v-iv)/(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) ; ... 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 ( 0uvxy + i0uvxy ) } ; ... NaN exit.
Save all flags ;
L := integer nearest Logb(Ω)/2 - 1 ; ... Ω is the overflow threshold.
h := Logb(max{|x|,|y|}) ; if |h| = ∞ then h := copysign(8L, h) ;
K := L - (integer nearest h) ;
x := Scalb(x, K) ; y := Scalb(y, K) ;
h := Logb(max{|u|,|v|}) ; if |h| = ∞ then h := 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 $\rho := |z| = |x+iy| = \sqrt{(x^2+y^2)}$, ...
 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 $r2 := \sqrt{2}$, $r2p1 := 1+\sqrt{2}$, $t2p1 := 1+\sqrt{2} - r2p1$;
 ... These constants must be correctly rounded to working precision;
 ... consequently $r2p1 + t2p1 = 1+\sqrt{2}$ to double that precision.
 Save Invalid flag ; ... This suppresses spurious Invalid Operation
 ... signals from NaN comparison or $\infty-\infty$; 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 $x \geq y \geq 0$ if not NaN .
 If $y = \infty$ then $x := y$;
 $t := x - y$;
 If $x \neq \infty$ and $t \neq x$ then
 { ... executed if $x \neq \infty$ and $y \neq \infty$ and y is not negligible.
 Save Underflow flag ;
 If $t > y$ then ... when $2 < x/y < 2/\epsilon$, ...
 { $s := x/y$; $s := s + \sqrt{1+s^2}$ }
 else ... when $1 \leq x/y \leq 2$, ...
 { $s := t/y$; $t := (2+s)s$;
 $s := ((t2p1 + t/(r2 + \sqrt{2+t})) + s) + r2p1$ } ;
 $s := y/s$... 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 CSSQS below in an obvious way.

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

To compute $x + iy = z := \zeta^2 = (\xi + i\eta)^2$, ...
 CSQUARE($\xi + i\eta$):
 $x := (\xi - \eta)(\xi + \eta)$; ... not $\xi^2 - \eta^2$.
 $y := \xi\eta + \xi\eta$; ... ONE multiply, one add.
 ... If a spurious NaN is created by overflow it gets removed thus:
 If $x \neq x$ then
 { if $|y| = \infty$ then $x := \text{copysign}(0, \xi)$
 else if $|\eta| = \infty$ then $y := -\infty$
 else if $|\xi| = \infty$ then $x := \infty$ }
 else if $y \neq y$ and $|x| = \infty$ then $y := \text{copysign}(0, y)$;
 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 $\rho := |(x+iy)/2^k|^2$ scaled to avoid Over/Underflow ...
 CSSQS($x + iy$): ... = $\rho + ik$, with an integer k ...
 Integer k ;
 $k := 0$;
 Save and lower the Over/Underflow flags;
 $\rho := x^2 + y^2$; ... Multiply twice and add.
 If (ρ is not finite) and ($|x| = \infty$ or $|y| = \infty$) then $\rho := \infty$
 else if (the Overflow flag was just raised, or
 the Underflow flag was just raised and $\rho < \lambda/\epsilon$)
 then { $k := \text{logb}(\max(|x|, |y|))$;
 $\rho := \text{scalb}(x, -k)^2 + \text{scalb}(y, -k)^2$ };
 Restore the Over/Underflow flags;
 Return ($\rho + ik$); end CSSQS.

To compute $\xi + \eta = \zeta := \sqrt{z} = \sqrt{x + iy}$, ...
 CSQRT(x + iy):

```

Real  $\rho$  ; Integer k ;
 $\rho + ik := \text{CSSQS}(x + iy)$  ; ... Sum-of-Squares Scaled; see above.
If x = x then  $\rho := \text{scalb}(|x|, -k) + \sqrt{\rho}$  ;
If k is odd then k := (k-1)/2
                else { k := k/2 - 1 ;  $\rho := \rho + \rho$  } ;
 $\rho := \text{scalb}(\sqrt{\rho}, k)$  ; ... =  $\sqrt{(|x+iy| + |x|)/2}$  without over/underflow
 $\xi := \rho$  ;  $\eta := y$  ;
if  $\rho \neq 0$  then
    { if  $|\eta| \neq \infty$  then {  $\eta := (\eta/\rho)/2$  ;
                            if  $\eta$  underflowed, signal it } ;
      if x < 0 then {  $\xi := |\eta|$  ;
                      $\eta := \text{copysign}(\rho, y)$  }
    }
Return (  $\xi + i\eta$  ) ;
...
... This program appears to handle all special cases correctly:
...  $\sqrt{-\beta \pm i0} = +0 \pm i\sqrt{\beta}$  for all  $\beta \geq 0$  .
...  $\sqrt{x \pm i\infty} = +\infty \pm i\infty$  for all x , finite, infinite or NaN ,
... and if x is NaN then "Invalid Comparison" is signaled too.
... For all finite  $\beta$  ,
...  $\sqrt{\text{NaN} + i\beta}$  ,  $\sqrt{\beta + i\text{NaN}}$  and  $\sqrt{\text{NaN} + i\text{NaN}}$  are all NaN + iNaN ;
...  $\sqrt{+\infty \pm i\beta} = +\infty \pm i0$  ;  $\sqrt{+\infty \pm i\text{NaN}} = +\infty + i\text{NaN}$  ;
...  $\sqrt{-\infty \pm i\beta} = +0 \pm i\infty$  ;  $\sqrt{-\infty \pm i\text{NaN}} = \text{NaN} \pm i\infty$  .
End CSQRT .

```

To compute $\xi + i\eta = \zeta := \ln(2^J z) = \ln(2^J(x+iy))$ with integer J, ...
 CLOGS(x + iy , J): ... for use with J $\neq 0$ only when $|x+iy|$ 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  $\text{lnlp}(x) := \ln(1+x)$  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
...  $\text{lnlp}$  influences the choice of thresholds T0, T1 and T2 .
Constants T0 := 1/√2 ; T1 := 5/4 ; T2 := 3 ;  $\ln2 := \ln(2)$  ;
Real  $\rho$  ; Integer k ;
 $\rho + ik := \text{CSSQS}(x + iy)$  ; ... =  $|(x+iy)/2^k|^2 + ik$  ; see above.
 $\beta := \max(|x|, |y|)$  ;  $\theta := \min(|x|, |y|)$  ;
If k = 0 and T0 <  $\beta$  and ( $\beta \leq T1$  or  $\rho < T2$  )
    then  $\rho := \text{lnlp}((\beta-1)(\beta+1) + \theta^2)/2$ 
    else  $\rho := \ln(\rho)/2 + (k+J) \ln2$  ;
 $\theta := \text{ARG}(x + iy)$  ;
Return (  $\rho + i\theta$  ) ; end CLOGS.

```

To compute $\xi + i\eta = \zeta := \ln(z) = \ln(x + iy)$, ...
 CLOG(z) := CLOGS(z, 0) .

```

To compute  $\xi + \eta = \zeta := \arccos(z) = \arccos(x + iy)$  , ...
CACOS(z):  ... based upon formulas
...       $\xi := 2 \arctan( \operatorname{Re}(\sqrt{1-z})/\operatorname{Re}(\sqrt{1+z}) )$  ;
...      ... suppress any Divide-by-Zero signal when  $z \leq -1$  .
...       $\eta := \operatorname{arcsinh}( \operatorname{Im}(\sqrt{1+z}) \sqrt{1-z} )$  ;
Return (  $\xi + \eta$  ) ; end CACOS .

```

```

To compute  $\xi + \eta = \zeta := \operatorname{arccosh}(z) = \operatorname{arccosh}(x + iy)$  , ...
CACOSH(z):  ... based upon formulas
...       $\xi := \operatorname{arcsinh}( \operatorname{Re}(\sqrt{z-1}) \sqrt{z+1} )$  ;
...       $\eta := 2 \arctan( \operatorname{Im}(\sqrt{z-1})/\operatorname{Re}(\sqrt{z+1}) )$  ;
...      ... suppress any Divide-by-Zero signal when  $z \leq -1$  .
Return (  $\xi + \eta$  ) ; end CACOSH .

```

```

To compute  $\xi + \eta = \zeta := \arcsin(z) = \arcsin(x + iy)$  , ...
CASIN(x + iy):  ... based upon formulas
...       $\xi := \arctan( x/\operatorname{Re}(\sqrt{1-z}) \sqrt{1+z} )$  ;
...      ... suppress any Divide-by-Zero signal when  $z \leq -1$  .
...       $\eta := \operatorname{arcsinh}( \operatorname{Im}(\sqrt{1-z}) \sqrt{1+z} )$  ;
Return (  $\xi + \eta$  ) ; end CASIN .

```

```

To compute  $\xi + \eta = \zeta := \operatorname{arcsinh}(z) = \operatorname{arcsinh}(x + iy)$  , ...
CASINH(z) := -i CASIN(iz) .

```

```

To compute  $\xi + \eta = \zeta := \operatorname{arctanh}(z) = \operatorname{arctanh}(x + iy)$  , ...
CATANH(x + iy):
  Constants  $\theta := \sqrt{\Omega}/4$  ,  $\rho := 1/\theta$  ;
   $\beta := \operatorname{copysign}(1, x)$  ;  $z := \beta z^*$  ; ... copes with unsigned 0
  If  $x > \theta$  or  $|y| > \theta$  ... to avoid overflow ,
  then {  $\eta := \operatorname{copysign}(\pi/2, y)$  ;  $\xi := \operatorname{Re}(1/(x+iy))$  }
  else if  $x = 1$ 
  then {  $\xi := \ln(\sqrt{|y|}/\sqrt{4+(|y|+\rho)^2})$  ;
         $\eta := \operatorname{copysign}(\pi - \arctan((|y|+\rho)/2), y)/2$  }
  else ... Normal case ...
  { ... using  $\operatorname{lnlp}(u) := \ln(1+u)$  accurately even if  $u$  is tiny,
     $\xi := \operatorname{lnlp}(4x/((1-x)^2 + (|y|+\rho)^2))/4$  ;
     $\eta := \operatorname{arg}((1-x)(1+x) - (|y|+\rho)^2 + 2iy)/2$ 
  } ; ... all exceptional cases appear to be handled correctly.
Return (  $\beta \zeta^*$  ) ; end CATANH .

```

```

To compute  $\xi + \eta = \zeta := \arctan(z) = \arctan(x + iy)$  , ...
CATAN(z) := -iCATANH(iz) .

```

```

To compute  $x + iy = z := \tanh(\zeta) = \tanh(\xi + i\eta)$  , ...
CTANH( $\xi + i\eta$ ):
  If  $|\xi| > \operatorname{arcsinh}(\Omega)/4$  ... avoid overflow ...
  then  $z := \operatorname{copysign}(1, \xi) + i \operatorname{copysign}(0, \eta)$ 
  else {
    save Divide-by-Zero flag ;
     $t := \tan(\eta)$  ; ... suppress any Div-by-Zero signal here.
    restore Divide-by-Zero flag ;
     $\beta := 1 + t^2$  ; ...  $1/\cos^2\eta$  ...
     $s := \sinh(\xi)$  ;
     $\rho := \sqrt{1 + s^2}$  ; ...  $\cosh \xi$  ...
    if  $|t| = \infty$  then  $z := \rho/s + i/t$  ... may signal if  $s=0$ 
      else  $z := (\beta\rho s + it)/(1 + \beta s^2)$ 
    } ;
  return  $z$  ; end CTANH .

```

```

To compute  $x + iy = z := \tan(\zeta) = \tan(\xi + i\eta)$  , ...
CTAN( $\zeta$ ) := -i CTANH(i $\zeta$ ) .

```

```

>>>>>>>>>> Still to come are details about <<<<<<<<<<<<
CLOG1P , 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:

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

The second definition is analytic:

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

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 $(\text{roundoff})^{\text{roundoff}}$, 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 ζv with tiny ζ and v . 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 $z(\beta) = w(\beta) = 0$ and that z and w are analytic functions of ξ in some open neighborhood of $\xi = \beta$. This means that $z(\xi)$ and $w(\xi)$ can be expanded in Taylor series in powers of $\xi - \beta$ valid near $\xi = \beta$, and both series begin with positive powers of $\xi - \beta$. Then we find that $z \rightarrow 0$ and $w \rightarrow 0$ and $z^w = \exp(w \ln(z)) \rightarrow 1$ as $\xi \rightarrow \beta$ 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 $z0.0 = 1$ for 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

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

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-105 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 i8087 and i80287 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 Faillace, 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 $\operatorname{arccot}(z)$ changed from $\pi/2 - \operatorname{arctan}(z)$ to $\operatorname{arctan}(1/z)$. Finally, H. Kober's *Dictionary of Conformal Representations* contains pictures of many useful conformal maps; this too was reprinted by Dover, in 1957.