Feeling No Pain in the Argand Plane

Julian V. Noble

August 4, 2003

Abstract

It's no fun to program complex arithmetic in a language that doesn't support it. Worse, the resulting code is cumbersome, opaque, and hard to maintain. In this column I illustrate how complex arithmetic simplifies algorithms in two-dimensional Cartesian vector space. I also show how straying into the complex plane can make difficult numerical integrals tractable. In other words, I am arguing that languages for scientific computation should support complex arithmetic.

Introduction

One of the questions I am often asked is "Why do you program in an obscure language like Forth?" The most honest answer, as I am sure it would be for most of us, is "History." That is, we use our favorite programming languages because we have learned to think in them, just as with natural languages.

However, I also have a specific reason that goes beyond history, and that provides the theme of today's column. In the early 1980's, when I moved from mainframes to personal computers, I was using primarily Fortran 77. Many of my programs incorporated complex arithmetic, which Fortran has supported since Fortran II. (And this feature may well be one of the reasons for Fortran's longevity, despite that language's manifold deficiencies.) BASIC, C and Pascal, of course, never pretended to support complex arithmetic.

But it was a shock when the ostensibly compliant Fortran 77 I bought for my PC did not support complex arithmetic either. Complex arithmetic *can*, of course, be expressed as real-number arithmetic, but the result is messy. As the *Numerical Recipes in C* file `complex.c` [1] makes clear, each complex operation—even addition and multiplication—requires a separate subroutine.

This left a choice between the two extensible languages, Lisp and Forth, both of which let me add complex types and arithmetic operations as needed. But further research revealed that commercial Lisps of that era were memory hogs, whereas Forth fit in a mere 32Kb. And finally, I already understood Forth's postfix notation because of my familiarity with Hewlett-Packard calculators, whereas Lisp's prefix notation—and the accompanying shoals of scope-defining parentheses—intimidated me.

Of course nowadays there are Python, C++ and other languages beside Fortran (including an upcoming version of C) that include complex types and operators. Nor is memory a major issue anymore. So today I probably would not bother with a language—however extensible—for which I would have to write complex extensions from scratch. And that would be a pity, because I would thereby miss out on the other features that endear Forth to its fans.

Since I contemplate future forays into the complex plane in upcoming columns, it seemed reasonable to introduce the subject with two applications where complex arithmetic simplifies the programming. These are vector analysis in two dimensions, and evaluating certain integrals numerically.

Operations with complex numbers

The complex numbers

$$z = x + iy, \quad i^2 = -1\,,$$

defined in terms of real number pairs (x, y), were introduced to provide solutions to such polynomial equations as $x^2 - 2x + 5 = 0$, which obviously

cannot be satisfied by any real number substituted for x, but is satisfied by the two complex roots $z = 1 \pm 2i$.

In $z = x + iy$, x is called the *real* part of z, denoted by $\mathrm{Re}(z)$; the number y is called, for historical reasons, the *imaginary* part, denoted by $\mathrm{Im}(z)$. Operations like addition, multiplication and division can be defined by the usual laws of algebra. For example,

$$\begin{aligned} (x + iy)\,(u + iv) &= xu + iyu + ixv + i^2 yv \\ &\equiv (xu - yv) + i\,(yu + xv)\ . \end{aligned}$$

That is, when we multiply two complex numbers the result is another complex number, and similarly with addition[1]. One important new operation we shall need is *complex conjugation*, denoted by a $*$ superscript and defined as "reverse the sign of the imaginary part":

$$z^* = (x + iy)^* \stackrel{df}{=} x - iy\,.$$

Adding complex numbers and complex operations to an extensible programming language is an instructive exercise[2]. A quick-and-dirty definition of, say, the absolute value of a complex number,

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

is easy[3]:

```
real function zabs (z)
    return  sqrt( (Re(z))^2 + (Im(z))^2 )
```

[1] A mathematician would say the complex numbers are *closed* under addition and multiplication.

[2] See the file **complex.f** under the link "Complex Number Lexicon for Forth" on my web page.

[3] As usual I use a Fortran-ish *cum* C-ish pseudocode for code fragments. Working code in C and Forth can be found on my web site:

http://www.phys.virginia.edu/~jvn/

under the link to my course "Computational Methods of Physics".

but it would never do in a library: if, say, $\mathrm{Im}(z)$ were larger than the square root of the maximum floating point number, then $(\mathrm{Im}(z))^2$ would overflow [2]. So the correct way to write the code is

```
real function zabs (z)
    complex z
    if  z = cmplx(0,0)  return  0    \ check for z = 0
    else
        x = max( abs ( Re(z) ), abs ( Im(z) ) )
        y = min( abs ( Re(z) ), abs ( Im(z) ) )
        return  x * sqrt( 1 + ( y/x )^2 )
```

Many modern machines have floating point divide operations that are substantially slower than adds or multiplies. Since the division takes about the same time as the square root, on such machines the second version will take about twice as long to execute as the naive version (the square root is the speed bottleneck).

Two dimensional vectors

Once we have a library of complex functions and operators we can use them to simplify certain problems. For example, the complex number $z = x + iy$ can be regarded as a vector $\vec{r} = x\hat{\mathbf{x}} + y\hat{\mathbf{y}}$ in 2-dimensional Cartesian coordinates (here $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are unit vectors in the horizontal, or x, direction and the vertical, or y, direction). This 2-dimensional vector space is called the *Argand plane*. The magnitude of a complex number is just the length of its corresponding Argand vector:

$$|z| = |\vec{r}| = \sqrt{x^2 + y^2}\,. \tag{1}$$

Figure 1 below illustrates these definitions.

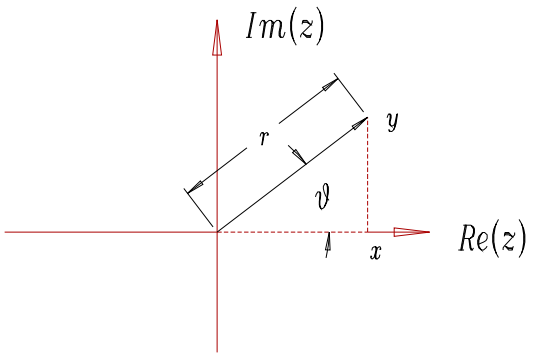


Figure 1 also exhibits the polar representation, $z = r\,e^{i\vartheta}$, which exploits Euler's relation,

$$e^{i\theta} = \cos\theta + i\sin\theta\,,$$

so that

$$\begin{aligned} x &= r\cos\vartheta \\ y &= r\sin\vartheta\,. \end{aligned}$$

This one-to-one correspondence between complex numbers and Cartesian 2-vectors lets us replace vector operations, usually expressed as matrix multiplication, by simple complex arithmetic. For example, we can rotate a vector by an angle φ counter-clockwise about the origin *via*

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}\,.$$

Complex arithmetic simplifies this to

$$z' = e^{i\varphi} z\,.$$

Again, suppose we want the dot product of two vectors

$$\vec{r}\cdot\vec{s} = xu + yv$$

where

$$\vec{r} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad \vec{s} = \begin{pmatrix} u \\ v \end{pmatrix};$$

In most languages the dot product must be written as a function or subroutine. But with complex arithmetic we may write it as a simple multiplication:

$$\vec{r} \cdot \vec{s} = xu + yv = \mathrm{Re}\left[(x - iy)(u + iv)\right] \equiv \mathrm{Re}\left(z^* w\right) .$$

Similarly, we can compute the vector product of two 2-dimensional vectors (the usual vector product points perpendicular to the $x - y$ plane; I label that direction $\hat{\zeta}$ to distinguish it from the complex number z)

$$(\vec{r} \times \vec{s}) \cdot \hat{\zeta} \stackrel{df}{=} xv - yu \equiv \mathrm{Im}\left(z^* w\right) .$$

That is, one complex multiplication, $z^* w$, evaluates simultaneously both the dot and vector products of the vectors represented by z and w.

Geometric relations

Hidden line removal algorithms need to determine whether two line-segments in a plane intersect, and if so, where. Although I am not planning to write a graphics program any time soon, a future column *will* require discovering whether a specific point in a plane lies within a given triangle. I will now show how both these problems can be solved more simply with complex arithmetic than with vectors.

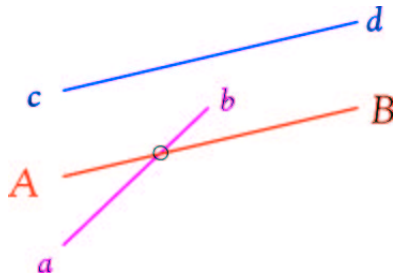
Two line-segments can be represented parametrically by

$$z(t) = a(1 - t) + bt \equiv a + (b - a)t$$
$$w(u) = A(1 - u) + Bu \equiv A + (B - A)u$$

where a, b and A, B are the endpoints of the segments. The parameters t and u are real numbers lying in the interval $[0, 1]$. We must first decide whether the lines are parallel, since parallel lines do not intersect. The cross-product of parallel vectors vanishes, so we compute

$$[\vec{r}(t) - \vec{r}(0)] \times [\vec{s}(u) - \vec{s}(0)] \Leftrightarrow \mathrm{Im}\left[(z(t) - z(0))^* \, (w(u) - w(0))\right] . \qquad (2)$$

If the cross-product is $\neq 0$ the (infinitely extended) lines will intersect *somewhere*. We therefore locate the point of intersection and determine whether it corresponds to values of t and u both lying between 0 and 1, *i.e.* the intersection point is common to both segments. The segments cross if and only if this condition is met. The possible cases are illustrated by the line segments $\widehat{ab}$ and $\widehat{AB}$ (crossing segments); the segments $\widehat{ab}$ and $\widehat{cd}$ (non-parallel, non-crossing); and $\widehat{AB}$ and $\widehat{cd}$ (parallel, non-crossing) shown in Figure 2 below.



The point of intersection of the two lines is given by

$$a + (b - a)t = A + (B - A)u \, ;$$

multiplying through by $(b - a)^*$ we have

$$(b - a)^*(b - a)t = |b - a|^2 \, t = (b - a)^* \, (A - a) + (b - a)^*(B - A)u \, .$$

Taking the imaginary parts of both sides, and recalling that t, u and $|b - a|^2$ are all real, we find

$$u = \frac{\mathrm{Im}\left[(b - a)^* \, (A - a)\right]}{\mathrm{Im}\left[(B - A)^*(b - a)\right]} \qquad (3a)$$

and, *mutatis mutandis*,

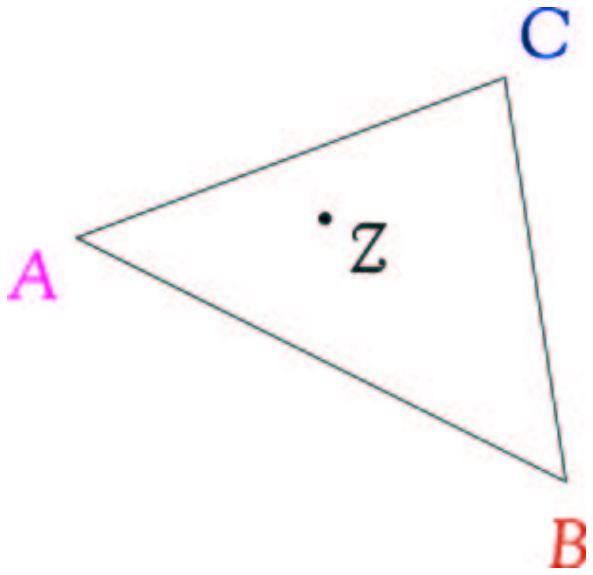
$$t = \frac{\mathrm{Im}\left[(B-A)^*\,(A-a)\right]}{\mathrm{Im}\left[(B-A)^*(b-a)\right]}\,. \tag{3b}$$

The denominators are the same in both cases. This is hardly surprising since, had we worked this out by solving coupled linear equations for t and u, we would have seen that the denominators are the determinant of the 2×2 matrix. Moreover, we see from Eq. 2 and Eq. 3a,b that if the lines had been parallel, the point of intersection would be infinitely far from the origin (because we would be dividing by zero).

This algorithm is easily translated to pseudocode:

```
logical function intersect? (zi, zf, wi, wf)
    /* returns TRUE if segments intersect */
    complex zi, zf, wi, wf, b, bb
    real t, u, Det
    b  = zf - zi
    bb = wf - wi
    Det = Im( conjg(bb) * b )
    if  abs( Det ) < 1.0e-10   \ check for parallelism
        return false
    else
        t = Im( conjg(bb) * (wi - zi) ) / Det
        u = Im( conjg(b)  * (wi - zi) ) / Det
        return ( ( 0 <= t <= 1) && ( 0 <= u <= 1) )
```

Next we shall determine whether a given point lies within a given triangle. Label the corners of the triangle with complex numbers A, B and C as shown in Figure 3:



Then it is easy to see that if the line $\widehat{AB}$, for example, were rotated about the point A until it were horizontal (taking the sense of rotation to leave the rotated point C' above $\widehat{A'B'}$) then the rotated point Z' lies above the line $\widehat{A'B'}$ if it lies within the triangle. Of course, a point Z' outside the triangle might lie above one (rotated) side, say $\widehat{A'B'}$. But an outside point cannot simultaneously lie "above" all three (rotated) sides. Thus if we repeat the rotation for the sides $\widehat{BC}$ and $\widehat{CA}$, an exterior Z' will end up below at least one rotated side. We therefore transform Z once per side, each time inquiring whether the new Z' lies above the corresponding side. If Z is "above" all three sides, it must be an interior point of Δ_{ABC}.

To transform Z relative to a given line $\widehat{AB}$, by translating the origin to A and then rotating about A until $\widehat{A'B'}$ is horizontal, we write

$$B - A = |B - A|\, e^{i\theta}\,,$$

and thus

$$Z' = e^{-i\theta}\,(Z - A) = \frac{(B - A)^*}{|B - A|}\,(Z - A)\ .$$

Since we need only the algebraic sign of $\mathrm{Im}(Z')$, we can eliminate the division by the (positive) length $|B - A|$. We are left with the criterion

$$\mathrm{Im}\left[(B - A)^* (Z - A)\right] > 0\,.$$

However, we might have rotated the wrong way, so that the point C' now lies *below* $\widehat{A'B'}$ rather than above. We could do more geometry to get the correct sense of rotation, but it seems simpler to apply the transformation once to Z and once to C and to multiply the criteria. We also have to check that the area of the triangle is greater than some minimum, since we are not interested in degenerate triangles.

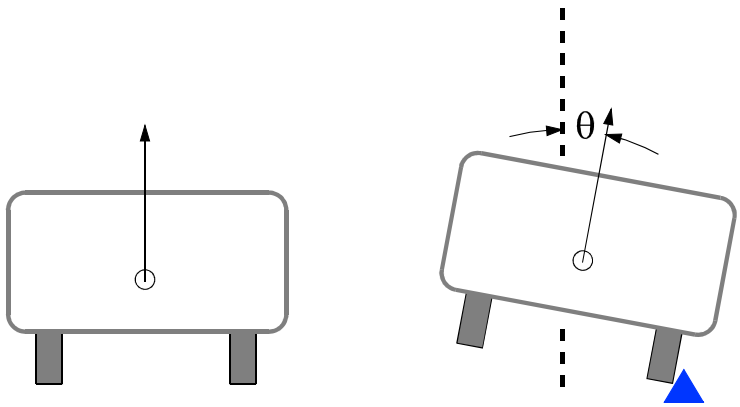
The pseudocode for this algorithm is then

```
integer function isign (x, k)       \ needed function
    real x
    integer k
    if x > 0
        return k
    else
        if x = 0
            return 0
        else
            return -k

integer function above? (z, a, b)   \ a and b are the ends
    complex z, a, b
    return isign( Im( (z - a) * conjg( b - a) ) , 1 )

logical function inside? (z, a, b, c)   \ true if z is inside
    complex z, a, b, c
    integer k, l, m
    if abs( Im( conjg(c - a) * (b - a) ) ) < 1.0e-10
        abort" Triangle is degenerate!"
    else
        k = above? (z, a, b) * above? (c, a, b)
        l = above? (z, b, c) * above? (a, b, c)
```

```
m = above? (z, c, a) * above? (b, c, a)
return ( (k > 0) && (l > 0) ) && (m > 0)
```

Motion in two dimensions

Let us next apply complex arithmetic to the description of an out-of-control vehicle sliding sideways, that encounters a low obstacle and flips over as a result ("wheels-pinned rollover accident"). This is shown schematically in Figure 4:



We model the vehicle by a rectangle with principal moment of inertia (about its center of mass) I, mass M and dimensions h (height) and w (width). If we represent its center of mass position by z, then Newton's Second Law of Motion is

$$M\ddot{z} = F - iMg$$

where the (complex) force $F = F_x + iF_y$ represents all external forces that act on the vehicle with the exception of gravity. For example, when the vehicle

slides across horizontal pavement,

$$F = -\mu N \operatorname{sgn}\left(\operatorname{Re}\left(\dot{z}\right)\right) + \beta + iN$$

where μ is the coefficient of sliding friction, N is the sum of the vertical forces the pavement exerts on the tires, and β is the force exerted by the curb. When the near tire collides with the barrier it experiences an impulsive force f in the horizontal direction. If the vehicle takes to the air, $N \to 0$ and the only force left is gravity.

The rate of change of angular momentum equals the applied torque (we oversimplify this for succinctness—the pavement forces on a real car act at separated points):

$$I\ddot{\theta} = \operatorname{Im}\left[e^{i\theta}\left(z - z_r\right)^* \left(F - iMg\right)\right]$$

where we take the axis of rotation to be the point of collision between wheel and curb. Forth and C programs (with animated graphics and more detailed explanations) can be found on my Web page.

Numerical integration in the Argand plane

In an earlier column [3] I gave an example of how a numerically difficult real-valued Cauchy principal value integral

$$I = \mathcal{P}\int_0^\infty dx \frac{1}{1 - x^3}$$

could be simplified by rotating to the contour $z = re^{-i\pi/3}$, and integrating in the complex plane.

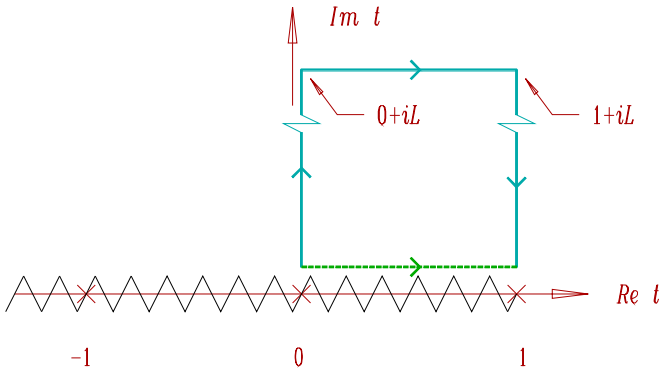
More recently I recently came across the integral

$$I_B\left(A\right) = \operatorname{Re}\int_0^{\pi/2} d\theta\, e^{iA\cos\theta}\left(\sin\theta\right)^B$$

on one of the news groups (perhaps **sci.math.num-analysis**). Clearly, $I_B(A)$ is (up to an uninteresting factor) a Bessel function. However if A is large, the integral is difficult to evaluate precisely. Evaluation of oscillatory integrals seems to be a topic of some current interest [4].

Suppose we rewrite the integral, using the transformation $t = cos\theta$, in the form

$$I_B(A) = \mathrm{Re}\int_0^1 dt\, e^{iAt}\left(1-t^2\right)^{(B-1)/2} \equiv \mathrm{Re}\int_0^1 dt\, e^{iAt}\left(1-t^2\right)^{\lambda}. \qquad (4)$$



The function $(1-t^2)^{\lambda}$ has (for non-integer λ) branch points at $t = \pm 1$; the corresponding branch lines can be arranged to run along the real $t-$axis from $-\infty$ to $+1$. The original contour is the line $t = x + i\varepsilon$, $0 \le x \le 1$ (shown in green in Figure 5). By Cauchy's Theorem [5, 6],

$$\oint_\Gamma dz\, f(z) = 0$$

we may distort the original contour to the tall, thin rectangle of base 1 and height L (shown in blue) since we can do so without crossing singularities of

the function $f(z)$. Thus we may write

$$\begin{aligned} I_B(A) &= \lim_{L\to\infty} \int_{i0^+}^{iL} dt\, e^{iAt} \left(\left(1-t^2\right)^\lambda - e^{iA} \left(1-(1+t)^2\right)^\lambda \right) \\ &\quad + \lim_{L\to\infty} e^{-AL} \int_0^1 dx\, e^{iAx} \left(1-(x+iL)^2\right)^\lambda . \end{aligned}$$

The second integral can be neglected in the limit $L \to \infty$ since it is bounded in magnitude by

$$\begin{aligned} \left| e^{-AL} \int_0^1 dx\, e^{iAx} \left(1-(x+iL)^2\right)^\lambda \right| &\leq e^{-AL} \int_0^1 dx \left| e^{iAx} \right| \left| \left(1-(x+iL)^2\right)^\lambda \right| \\ &\leq e^{-AL} \int_0^1 dx \left(3+L^2\right)^\lambda \\ &= \left(3+L^2\right)^\lambda e^{-AL} \underset{L\to\infty}{\to} 0 . \end{aligned}$$

The change of variable $t \to iy$ allows us to express $I_B(A)$ as

$$\begin{aligned} I_B(A) &= \mathrm{Re}\, i \int_0^\infty dy\, e^{-Ay} \left[\left(1+y^2\right)^\lambda - e^{iA} \left(y\,(y-2i)\right)^\lambda \right] \qquad (5) \\ &= \mathrm{Im}\, e^{iA} \int_0^\infty dy\, e^{-Ay} \left(y\,(y-2i)\right)^\lambda , \end{aligned}$$

where the second line of Eq. 5 follows from the assumption that λ is real (for complex λ we must keep both terms from the first line). For moderate values of λ the integral is rapidly convergent.

We have thereby replaced an integrand that definitely converges (it oscillates in sign and decreases in magnitude) with one that decays exponentially, for which a Gauss-Laguerre quadrature formula would be appropriate. Table 1 below lists values of the integral Eq. 4, for several large values of A and $\lambda = 7$, computed directly using adaptive quadrature. Table 1 also lists values of the transformed integral Eq. 5 computed using adaptive (Simpson's-rule)

Table 1: **Quadrature by Real- and Complex Arithmetic**

	Real		**Complex**	
A	N_{calls}	$I(A)$	N_{calls}	$I(A)$
10	2445	7.31477251207E-3	1545	7.31477251218E-3
20	3601	-2.19385695893E-5	373	-2.19385695736E-5
30	4917	8.87060992799E-7	177	8.87060992040E-7

Table 2: **Adaptive *vs.* Gauss-Laguerre Quadrature**

	Adaptive		**10-pt Gauss-Laguerre**
A	N_{calls}	$I(A)$	$I(A)$
10	1545	7.31477251218E-3	7.31477251203E-3
20	373	-2.19385695736E-5	-2.19385695900E-5
30	177	8.87060992040E-7	8.87060992893E-7

quadrature. The number of evaluations of the integrand is also given. Table 2 compares the evaluation of the transformed integral by adaptive quadrature with the results of a 10-point Gauss-Laguerre rule. The absolute error criterion for the adaptive quadrature was 10^{-12}; as is clear, both the real and complex adaptive methods agree with each other to this precision; the 10-point Gauss-Laguerre method agrees with both to this precision and requires by far the fewest evaluations, so it is the method of choice.

Rotating the integration variable of an oscillatory integral into the complex plane is a trick I have long found useful. Even when the integrand involves the solution of a differential equation the method still works, since most such equations are sufficiently analytic that they can be continued (numerically) into the complex plane. The key issue is to make sure that in deforming the contour of integration one crosses no singularities.

References

[1] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes in C, 2nd ed.* (Cambridge University Press, New York, 1992), Appendix C.

[2] This sort of thing is discussed by David Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* **23** (1991) pp. 5-48.

[3] J.V. Noble, "Gauss-Legendre Principal Value Integration", *CiSE* **2**(1)(2000) 92-95.

[4] Ulf T. Ehrenmark, *A note on a recent study of oscillatory integration rules* (`http://www.lgu.ac.uk/cismres/Papers/jcam.pdf`).

[5] E.T. Copson, *An introduction to the theory of functions of a complex variable* (Clarendon Press, Oxford, 1955).

[6] E.T. Whittaker and G.N. Watson, *A course of modern analysis, 4th ed.* (Cambridge University Press, New York, 1963).

Julian Noble is Professor Emeritus of Physics at the
Department of Physics
University of Virginia
P.O. Box 400714
Charlottesville, VA 22904-4714

He may be contacted at jvn@virginia.edu .

His interests are eclectic, both in and out of physics. His philosophy of teaching computational methods is "no black boxes".

Figure captions

1. The Argand plane, illustrating both the vector representation of a complex number, and its polar representation.

2. Line segments that cross and others that do not.

3. A point within a triangle.

4. Cross-sectional view of a wheels-pinned rollover accident.

5. The original (green) and distorted (cyan) contours of a certain numerical integral.