Reliable Computing 1 (3) (1995), pp. 239–249

# An automatic and guaranteed determination of the number of roots of an analytic function interior to a simple closed curve in the complex plane

JONATHAN HERLOCKER and JEFFREY ELY

A well known result from complex analysis allows us, under suitable circumstances, to compute the number of roots of an analytic function, $f(z)$, that lie inside a counterclockwise, simple closed curve, $C$, by computing the integral,

$$\frac{1}{2\pi i}\int_C \frac{f'(z)}{f(z)}dz.$$

We employ interval arithmetic and automatic differentiation to give an automatic and guaranteed bound on the integral. Furthermore, we explore the interplay of the choice of curve $C$, the location of the roots relative to $C$, the number of subdivisions, and the arithmetic precision used, upon the time necessary to obtain satisfactory bounds.

# Автоматическое гарантированное определение числа корней аналитической функции, лежащих внутри простой замкнутой кривой в комплексной плоскости

Дж. ХЕРЛОКЕР, Дж. ИЛИ

Из комплексного анализа известно, что при определенных условиях число корней аналитической функции $f(z)$, лежащих внутри простой замкнутой кривой $C$, направленной против часовой стрелки, можно вычислить, взяв интеграл

$$\frac{1}{2\pi i}\int_C \frac{f'(z)}{f(z)}dz.$$

Применив интервальную арифметику и автоматическое дифференцирование, мы получаем гарантированные границы этого интеграла. Кроме того, исследуется влияние выбора кривой $C$, расположения корней по отношению к $C$, количества разбиений и разрядности используемой арифметики на время, требуемое для получения удовлетворительных границ.

© J. Herlocker, J. Ely, 1995

# 1. Introduction

The argument principle, a well known result from complex analysis, asserts that if $C$ is a simple, closed, counterclockwise contour and if $f(z)$ is a function which is analytic inside and on $C$, except possibly for a finite number of poles interior to $C$, and if $f(z)$ has at most a finite number of zeros inside $C$ and none on $C$, then

$$\frac{1}{2\pi i}\int_C \frac{f'(z)}{f(z)}dz = N_{\text{zero}}(f) - N_{\text{pole}}(f)$$

where $N_{\text{zero}}$ is the number of zeros and $N_{\text{pole}}$ is the number of poles of $f$ inside $C$ (including multiplicity) [1].

Henrici [3] observes that the argument principle can be used to find first approximations to zeros of functions (which other methods may then refine) and further mentions an application to the theory of automatic control.

The goal of this paper is to explore the tools necessary to automatically compute the above integral, bounding all errors, both round-off and discretization, and to experiment with various parameters that affect the time necessary to obtain a satisfactory bound.

# 2. Choice of contour

The choice of a contour, $C$, determines $z$ as a funtion of $t$, that is,

$$z = z(t), \quad \text{with } t \in [t_0, t_n]$$

and the integral above becomes

$$\int_{t_0}^{t_n} g(t)dt, \quad \text{where } g(t) = \frac{f'\big(z(t)\big)}{f\big(z(t)\big)}\frac{dz(t)}{dt}.$$

The only types of contours considered were circles and squares.

# 3. Approximation technique

In order to approximate this numerically, we use Simpson's rule,

$$\int_{t_0}^{t_n} g(t)dt \approx \frac{\Delta t}{3}[g(t_0) + 4g(t_1) + 2g(t_2) + \cdots + 2g(t_{n-2}) + 4g(t_{n-1}) + g(t_n)].$$

Complex interval arithmetic [5] is used to bound any round-off error that occurs in computing this sum.

# 4. Control of discretization error

The Simpson sum is, of course, not exact even if there were no round-off error. For a *real* valued function, $g(t)$, the error inherent in the method is well known to be bounded by

$$\frac{M_4(t_n - t_0)^5}{180n^4}$$

where $M_4$ is an upper bound on $|g^{(4)}(t)|$, as $t$ ranges over the *interval* $[t_0, t_n]$. The stress here on the word, *interval*, emphasizes that interval arithmetic is ideal for computing this discretization error as well as the round-off error of the sum, *provided* that a program is available to calculate the fourth derivative, $g^{(4)}(t)$, of $g(t)$. Since $g(t)$ is complex valued, we use the real and imaginary parts of $g^{(4)}(t)$ to separately bound the discretization errors associated with the real and the imaginary parts of the Simpson sum. We further observe that since the final result is an integer (hence real) we really only need the imaginary part of the integral (which when divided by $2\pi i$ gives us the real value).

# 5. Computing derivatives

If the function, $f(z)$, is a polynomial, then it is easy to code $f'(z)$. Likewise, given either the circular or square contours mentioned above, $z(t)$ and $\frac{dz(t)}{dt}$ are also easy to code, hence $g(t)$ is easy to code as

$$g(t) = \frac{f'\big(z(t)\big)}{f\big(z(t)\big)} \frac{dz(t)}{dt}$$

but control of the discretization error inherent in Simpson's rule requires us to compute $g^{(4)}(t)$. Automatic differentiation [4] is clearly the tool of choice here. When coupled with interval arithmetic, automatic differentiation will automatically bound $|g^{(4)}(t)|$ as $t$ ranges over the interval $[t_0, t_n]$.

# 6. Software tools

We used the VPI software package [2]. This is a collection of classes, written in C++ [7], built around a variable precision, real data type called `afloat`. Directed rounding makes it possible to build intervals (class `interval`) on top of the afloats, complex intervals (`compivl`) on top of intervals, and variable degree taylor series (`ctaylor`) on top of complex intervals. This last class is VPI's implementation of automatic differentiation. While VPI supports other data types as well (e.g. `matrix`), the `afloat`, `interval`, `compivl`, and `ctaylor` classes suffice for this problem.

# 7. Circles vs. squares

The use of a square with sides parallel to the axes proved to be ten times faster than the corresponding calculation using the inscribed circle. We presume this to be largely an artifact of very slow sine and cosine routines in the VPI interval math package. After this observation, only squares were used as contours. The choice of squares, while limiting in some situations, is perfectly adequate in others such as Henrici's algorithm for using the argument principle to locate zeros [3].

# 8. How much precision?

We hypothesized that the discretization error would greately overshadow round-off error, hence we expected that the main mechanism for "tightening up" a result would be to increase the number of subdivisions rather than to increase the precision of the arithmetic. We were concerned, however, that if the number of subdivisions were extremely large, this might lead to significant round-off (more terms in the sum, each with a round-off error to contribute), but this proved not to be a major factor in our experiments which sometimes required several thousands of subdivisions.

# 9. Frequency of calculating discretization error

Suppose we wish to approximate

$$\int_0^1 g(t)dt$$

as we have described, using $N = 100$ subdivisions of the interval $[0, 1]$. At one extreme, we can compute this with one application of Simpson's rule, computing the Simpson sum and then the discretization error based on $g^{(4)}([0, 1])$. At the other extreme, we could use the same number of subdivisions, 100, but apply Simpson's rule over $[0, .02]$ with 2 subdivisions, then apply it over $[.02, .04]$ with 2 subdivisions, then $[.04, .06]$, and so on until, finally, we apply Simpson's rule over $[.98, 1.00]$ with 2 subdivisions. The desired result is the sum of the fifty values. The difference is that in the first case we compute $g^{(4)}(t)$ only once as $t$ ranges over the entire interval $[0, 1]$, while in the second approach, we calculate $g^{(4)}(t)$ fifty different times *but* each of these bounds is taken as $t$ ranges over only a small interval. We expect the former to be faster but the latter to be tighter.

(It should be noted that, if $t$ ranges over too wide an interval, we may not be able to bound $g^{(4)}(t)$ at all. The reason is that since divisions are involved, $\left(\text{recall that } g(t) = \frac{f'(z(t))}{f(z(t))}\frac{dz(t)}{dt}\right)$, if t ranges over too wide an interval, we may wind up, in the course of the calculation, trying to divide by an interval that contains zero.)

Let $N =$ the total number of subdivisions.

Let $n =$ the number of subintegrals to cover the range of $t$.

Let $m =$ the number of subdivisions per subintegral.

Obviously, $N = n * m$.

In our example above, $N = 100$. The first strategy calls for $n = 1$ and $m = 100$. The second strategy calls for $n = 50$ and $m = 2$. This problem only requires the final result to bracket a unique integer, hence we do not need a very tight bound on the computation (width $< 1$), still, are we better off, in an effort to achieve such "tightness" as we do require, to use the former or later approach?

In a typical test case, we compared $m = 16$ against $m = 2$. We found the minimal $n$ in each instance that would trap the integral to a unique integer (i.e. the number of roots in the square). When $m = 16$, the minimal $n$ required was 208, hence $N = 16 * 208 = 3328$. For $m = 2$, the minimal $n$ was 226 and $N = 2 * 226 = 452$. As this was also three times faster, we followed the $m = 2$ strategy in all subsequent experiments.

# 10. Errors vs. subdivisions: an example

Consider the experiment pictured in Figure 1 and its accompanying data in Table 1. Here $f(z) = z^3 - z^2 - z - 2$, with roots at 2, $\frac{(-1 \pm i\sqrt{3})}{2}$. The square contour has center $2.5 + 0i$ and radius $r = \frac{1}{2} + 2^{-11}$. We began with two subintegrals per side of the square ($n = 2$) and doubled this until we had a sufficiently high value to guarantee the trap of the number of roots to a unique integer. Then we used a bisection method to determine the minimum value, $n^*$, of $n$ necessary to compute the trap. Notice in Table 1 that the error in computing the Simpson sum (which is solely due to round-off) is insignificant with respect to the the error associated with discretization. While there is also a round-off component in computing the discretization error term, we believe this supports our previous hypothesis about the relative un-importance of round-off error in this problem.

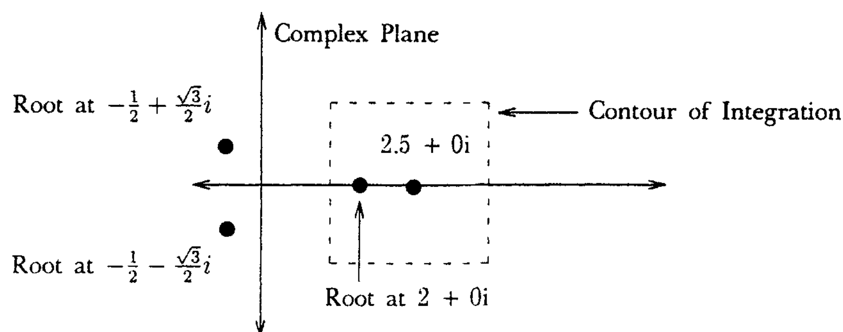


Figure 1.

Initially, this round-off error remains more or less constant, varying from 7.48512E−13 to 5.4623E−14. Once we have reached a power of 2 that will successfully locate the number of roots ($n = 2048$), and we begin bisecting to find the minimal value of $n$, our round-off error jumps by a factor of $10^3$. Up to now, since $r = \frac{1}{2} + 2^{-11}$ and $n$ has been a small power of 2, the end points of each subinterval have been exactly representable on the computer. When $n$ is no longer a power of 2, the end points are no longer exactly representable, we insert round-off error into the calculation, and this is magnified by further computations. Still, a round-off error of $10^{-10}$ presents no serious problem.

Next consider the progression of the discretization error as we increase $n$. Table 1 shows that in the early stages doubling $n$ improves the discretization error by a factor of $10^3$ to $10^4$! This is much better than the factor of 16 suggested by the $n^4$ term in Simpson's error formula. The explanation seems to be that $M_4$ is being calculated much more tightly as we double $n$, and this further explains our earlier observation that $m = 2$ leads to the fastest computation even though we are computing the fourth derivative more frequently.

# 11. Roots near the contour

If $f(z)$ has a root on the contour, then $g(t) = \frac{f'(z(t))}{f(z(t))} \frac{dz(t)}{dt}$ is singular and the argument principle is not applicable. What, we wondered, will happen if $f(z)$ has a root close to but not

| $n$ | Width of Simpson sum interval | Width of discretization interval |
|---|---|---|
| 2 | 5.04485E−13 | 1.28869E+29 |
| 4 | 2.66454E−13 | 2.0944E+25 |
| 8 | 1.5099E−13 | 6.78944E+21 |
| 16 | 9.50351E−14 | 2.90748E+18 |
| 32 | 6.75016E−14 | 1.30382E+15 |
| 64 | 5.4623E−14 | 1.28571E+12 |
| 128 | 7.28306E−14 | 2.0112E+09 |
| 256 | 9.03722E−14 | 3.68387E+06 |
| 512 | 1.70863E−13 | 7065.2 |
| 1024 | 3.66596E−13 | 13.5919 |
| 2048 | 7.48512E−13 | 0.0639715 |
| 1536 | 1.67348E−10 | 0.427248 |
| 1280 | 1.20802E−10 | 1.83239 |
| 1408 | 1.43207E−10 | 0.788859 |
| 1344 | 1.32E−10 | 1.18829 |
| 1376 | 1.37153E−10 | 0.965446 |
| 1360 | 1.34938E−10 | 1.07032 |
| 1368 | 1.35862E−10 | 1.01635 |
| 1372* | 1.36706E−10 | 0.990526 |
| 1370 | 1.36362E−10 | 1.00334 |
| 1371 | 1.26903E−10 | 0.699263 |

$n^*$ = minimum # of subintegrals per side = 1372

Table 1.

on the contour [3]? Consider, again, the function $f(z) = z^3 - z^2 - z - 2$. We first construct a square with center $d = 2^{-10}$ units from the root at 2 (center $= 2 + 2^{-10} + 0i$). The radius of the square is initially taken as $r = d + E$, $E = 2^{-1}$, which well encompasses the root at 2. We then find the minimal number of subdivisions, $n^*$, necessary to trap the number of roots to a unique integer. In this case $n^* = 4$ (see the top row of Table 2). We then repeat the experiment with a square having the same center but radius, $r = d + 2^{-2}$, which brings the border of the new square closer to the root ($E$ has been halved) and again we compute $n^*$. Repeatedly halving $E$ brings the boundary closer to the root and computing $n^*$ each time gives the data in Table 2.

A look at this data and that of other similar experiments, suggests that for simple roots (multiplicity 1), if $d \leq E$, $n^*(r) \approx$ small constant, but for $0 < E < d$, we have $n^*(d+E/2) \approx 2n^*(d+E)$, where $n^*$ is viewed as a function of the radius, $r$. This result holds even when $d = 0$, i.e. when the square is centered on the root, for then $0 = d < E$ always, and $n^*(r) \approx$ small constant. Figure 2 pictorially summarizes this discussion.

# 12. Meeting the goal of automatic computation

As an illustration, Code Segment 1 shows a typical program that a user might supply for polynomial functions. The well known synthetic division scheme is used to evaluate the

| $E$ | $n^*$ | $E$ | $n^*$ | $E$ | $n^*$ | $E$ | $n^*$ |
|---|---|---|---|---|---|---|---|
| $2^{-1}$ | 4 | $2^{-6}$ | 2 | $2^{-11}$ | 5 | $2^{-16}$ | 87 |
| $2^{-2}$ | 3 | $2^{-7}$ | 2 | $2^{-12}$ | 7 | $2^{-17}$ | 173 |
| $2^{-3}$ | 3 | $2^{-8}$ | 2 | $2^{-13}$ | 13 | $2^{-18}$ | 343 |
| $2^{-4}$ | 2 | $2^{-9}$ | 3 | $2^{-14}$ | 23 | $2^{-19}$ | 685 |
| $2^{-5}$ | 2 | $2^{-10}$ | 3 | $2^{-15}$ | 45 | $2^{-20}$ | 1369 |

$r = d + E$

$d = 2^{-10}$

Table 2.

polynomial, $f$, and its derivative, $f'$ [6] in order to compute, the value of $f'/f$ when $z$ is given. Since this segment must also supply information about the fourth derivative of $R = f'/f$, we use an extension of automatic differentiation which we have implemented in a C++ class called `ctaylor`. The data structure for this class is largely just an array of taylor series coefficients, $< R_0, R_1, R_2, R_3, R_4 >$, where $R_0 = (f'/f)$, $R_1 = d(f'/f)/dt$, etc. Because all of the variables in the code segment are declared as type `ctaylor`, C++ will invoke taylor series arithmetic routines (part of VPI) when performing the various addition, multiplication, and division operations that appear in the segment.

In this code segment, the user has been obliged to supply code that computes $f'(z)$ as well as $f(z)$. While it is easy to code $f'$ when $f$ is a polynomial and not too terribly difficult otherwise, still, this requirement is at odds with our goal of having an automatic computation, which should only require the user to code $f(z)$. Furthermore, it seems aesthetically disconcerting to use the machinery of automatic differentiation to compute derivatives for $f(z)$ while making the taylor series for $f$ and to compute derivatives for $f'(z)$ while making the taylor series for $f'$, but not to use automatic differentiation to compute $f'$ from $f$. If this could be done, then the automatic differentiation would *automatically* compute $f'$, leaving the user responsible for $f$ only.

At first glance, it might appear that if $f$ is represented by the taylor series coefficients $< f_0, f_1, f_2, f_3, f_4 >$, then we can merely extend the number of terms by one to get $< f_0, f_1, f_2, f_3, f_4, f_5 >$, extract the last five to get $< f_1, f_2, f_3, f_4, f_5 >$, and that these will be the taylor series coefficients for $f'$. This is not correct for two reasons. First, it is not the first, second, third, fourth, and fifth derivatives that are stored, but rather those values divided by the appropriate factorial, i.e. the taylor coeffcients. This can be easily fixed by mutliplying each extracted term by the appropriate integer (= the degree of the derivative associated with the term). The second flaw in this line of reasoning is that the automatic derivatives are with respect to the variable $t$, while $f'$ is with respect to the variable $z$. This is also easily fixed if we maintain square contours, for then $\dot{z}(t)$ is linear (on each side of the square), $dz/dt$ is constant and this constant can be factored out of successively higher derivatives as the automatic differentiation proceeds. The result of these corrections, is that for

$$g(t) = \frac{f'\big(z(t)\big)}{f\big(z(t)\big)} \frac{dz(t)}{dt}$$

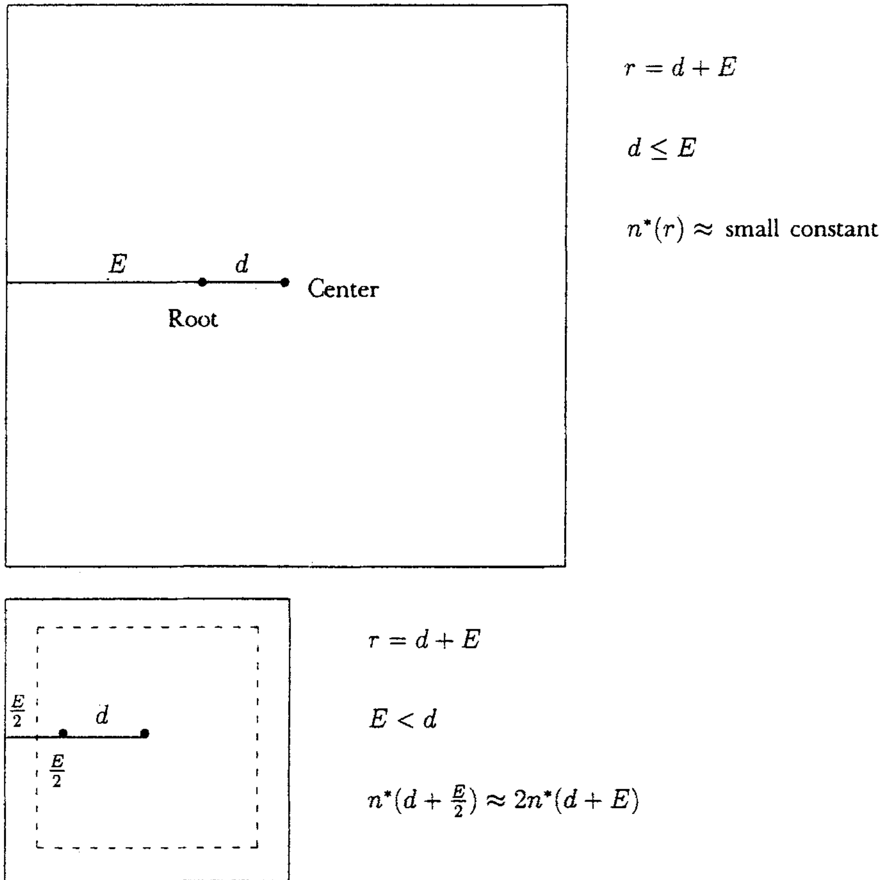


Figure 2.

we can compute the associated taylor coefficients for $g$ from those of $f$, via the taylor series division of

$$\frac{< f_1, 2f_2, 3f_3, 4f_4, 5f_5 >}{< f_0, f_1, f_2, f_3, f_4 >}$$

giving

$$< g_0, g_1, g_2, g_3, g_4 > .$$

Multiplying $g_4$ by 4! gives us the fourth derivative, $g^{(4)}(t)$, that we need for our discretization bound.

The code for $g$ can be found in Code Segment 2. Now the user need only supply code to compute $f$ and not bother with $f'$. The revised code when $f$ is a polynomial is also found in Code Segment 2. If a function other than a polynomial is called for, the user need only change the code that computes $f$ and not worry about $f'$ (or $g$).

Interestingly enough, while automation was our motivation here, we gain an efficiency also. The synthetic division approach required $d = \text{degree} f$ multiplications for $f$ and $d - 1$ for $f'$ for a total of $2d - 1$. Each of these multiplications is between two 5-term taylor series arrays. Such multiplications require $1 + 2 + 3 + 4 + 5 = 15$ multiplications of the base type, complex

numbers (recall the procedure for multiplying two power series). Hence a total of $(2d-1)*15$ complex number multiplies are required by the synthetic division strategy. The new strategy requires only $d$ taylor series multiplications, each of which will take $1+2+3+4+5+6=21$ complex number multiplications (recall that we need 1 more term in our taylor series), to compute $f$, and 4 more complex multiplications when the terms of $f$ are extracted to make the terms of $f'$ for a total of $d*21+4$. So this new approach requires about $21/30 = 70\%$ of the multiplications of synthetic division. In practice we found that the new version (Code Segment 2) took about 80% of the time of the synthetic one.

**Code Segment 1**

```
ctaylor F(ctaylor& z)
// the complex function to integrate
// F(z(t)) = ( f'(z(t)) / f(z(t))   )
// z(t) any function of t
// the user must code f and f'
// here, synthetic division is used for f,f', f a polynomial
{
 int k ;
 ctaylor f,fprime,R ;

 f = coeff[degree] ;
 if (degree > 0)  fprime = f ;  else fprime = 0 ;
 for (k = degree - 1 ; k >= 0 ; k--) {
     f = f*z + coeff[k] ;
     if (k > 0) fprime = fprime*z + f ;
 }

 R = fprime / f ;
 return R ;
}
```

**Code Segment 2**

```
ctaylor f(ctaylor& z)
// the user must code f only
{
 ctaylor Poly = coeff[degree] ; // starts poly as a constant
 for (k = \degree - 1 ; k >= 0 ; k--) {
     Poly = Poly*z + coeff[k] ;  // this will make Poly same deg at z
 }

 return Poly ;
}
```

```
ctaylor g(ctaylor& z)
// g(z(t)) = ( f'(z(t)) / f(z(t))    ) * (dz(t)/dt)
// z(t) linear
// ff computed to same num terms as z
// fprime to 1 less term
// Result to same num terms as fprime
// the user need NOT code this
{
 int k ;

 ctaylor ff = f(z) ;   // calling the user supplied code for f

 ctaylor fprime = ctaylor(ff.term[1], 0, ff.degree - 1) ;
 for (k = 2 ; k <= ff.degree ; k++) {
    fprime.term[k-1] = k*ff.term[k] ;
 }

 ff.degree-- ; // ensures result computed to same num terms as fprime
 ctaylor R = fprime / ff ;
 ff.degree++ ;  // ensures reclamation of all storage in ff

 return R ;
}
```

# 13. Summary

In order to compute the number of roots of an analytic function, $f$, that are inside a square, we have used Simpson's rule to approximate the winding number integral. Low precision, interval arithmetic generally proved adequate to bound the round-off errors and automatic differentiation permitted us to bound the discretization error. Not surprisingly, discretization error greatly overshadowed round-off error. The number of subdivisions to adequately bound the result proved quite small as long as the root was closer to the center of the square than to the boundary, but worsened dramatically when this was not the case. We have quantified this deterioration for roots of order 1, but have not pursued a complete study. Finally, considerable care was taken to adapt automatic differentiation so that the user need only supply code for the function $f$, and not for $f'$ also.

# References

[1] Churchill, R., Brown, J., and Verhey, R. *Complex variables and applications.* McGraw-Hill, New York, 1976.

[2] Ely, J. *The VPI software package for variable precision interval arithmetic.* Interval Computations 2 (1993), pp. 135–153.

[3] Henrici, P. *Applied and computational complex analysis. Vol. 1.* Wiley, New York, 1974.

[4] Kagiwada, H., Kalaba, R., Rasakhoo, N., and Spingarn, K. *Numerical derivatives and nonlinear analysis.* Plenum, New York, 1986.

[5] Moore, R. E. *Methods and applications of interval analysis.* SIAM, Philadelphia, 1979.

[6] Press, W., Flannery, B., Teukolsky, S., and Vetterling, W. *Numerical recipes in C.* Cambridge University Press, Cambridge, MA, 1988.

[7] Stroustrup, B. *The C++ programming language.* Addison-Wesley, Reading, Mass., 1986.

Received: March 1, 1994
Revised version: December 14, 1994

Department of Mathematical Sciences
Lewis and Clark College
Portland, OR 97219
USA
E-mail: herlock@lclark.edu
jeff@lclark.edu