Thompson, I.J., and Barnett, A.R. 1986, *Journal of Computational Physics*, vol. 64, pp. 490–509. [5]

Lentz, W.J. 1976, *Applied Optics*, vol. 15, pp. 668–671. [6]

Jones, W.B. 1973, in *Padé Approximants and Their Applications*, P.R. Graves-Morris, ed. (London: Academic Press), p. 125. [7]

5.3 Polynomials and Rational Functions

A polynomial of degree $N-1$ is represented numerically as a stored array of coefficients, c(j) with j= $1,\ldots,N$. We will always take c(1) to be the constant term in the polynomial, c(N) the coefficient of x^{N-1}; but of course other conventions are possible. There are two kinds of manipulations that you can do with a polynomial: *numerical* manipulations (such as evaluation), where you are given the numerical value of its argument, or *algebraic* manipulations, where you want to transform the coefficient array in some way without choosing any particular argument. Let's start with the numerical.

We assume that you know enough *never* to evaluate a polynomial this way:

```
p=c(1)+c(2)*x+c(3)*x**2+c(4)*x**3+c(5)*x**4
```

Come the (computer) revolution, all persons found guilty of such criminal behavior will be summarily executed, and their programs won't be! It is a matter of taste, however, whether to write

```
p=c(1)+x*(c(2)+x*(c(3)+x*(c(4)+x*c(5))))
```

or

```
p=(((c(5)*x+c(4))*x+c(3))*x+c(2))*x+c(1)
```

If the number of coefficients is a large number n, one writes

```
p=c(n)
do 11 j=n-1,1,-1
   p=p*x+c(j)
enddo 11
```

Another useful trick is for evaluating a polynomial $P(x)$ and its derivative $dP(x)/dx$ simultaneously:

```
p=c(n)
dp=0.
do 11 j=n-1,1,-1
   dp=dp*x+p
   p=p*x+c(j)
enddo 11
```

which returns the polynomial as p and its derivative as dp.

The above trick, which is basically *synthetic division* [1,2], generalizes to the evaluation of the polynomial and nd-1 of its derivatives simultaneously:

```
SUBROUTINE ddpoly(c,nc,x,pd,nd)
INTEGER nc,nd
REAL x,c(nc),pd(nd)
    Given the coefficients of a polynomial of degree nc-1 as an array c(1:nc) with c(1) being
    the constant term, and given a value x, and given a value nd>1, this routine returns the
    polynomial evaluated at x as pd(1) and nd-1 derivatives as pd(2:nd).
INTEGER i,j,nnd
REAL const
pd(1)=c(nc)
do 11 j=2,nd
    pd(j)=0.
enddo 11
do 13 i=nc-1,1,-1
    nnd=min(nd,nc+1-i)
    do 12 j=nnd,2,-1
        pd(j)=pd(j)*x+pd(j-1)
    enddo 12
    pd(1)=pd(1)*x+c(i)
enddo 13
const=2.                           After the first derivative, factorial constants come in.
do 14 i=3,nd
    pd(i)=const*pd(i)
    const=const*i
enddo 14
return
END
```

As a curiosity, you might be interested to know that polynomials of degree $n > 3$ can be evaluated in *fewer* than n multiplications, at least if you are willing to precompute some auxiliary coefficients and, in some cases, do an extra addition. For example, the polynomial

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 \tag{5.3.1}$$

where $a_4 > 0$, can be evaluated with 3 multiplications and 5 additions as follows:

$$P(x) = [(Ax + B)^2 + Ax + C][(Ax + B)^2 + D] + E \tag{5.3.2}$$

where $A, B, C, D,$ and E are to be precomputed by

$$\begin{aligned}
A &= (a_4)^{1/4} \\
B &= \frac{a_3 - A^3}{4A^3} \\
D &= 3B^2 + 8B^3 + \frac{a_1A - 2a_2B}{A^2} \\
C &= \frac{a_2}{A^2} - 2B - 6B^2 - D \\
E &= a_0 - B^4 - B^2(C + D) - CD
\end{aligned} \tag{5.3.3}$$

Fifth degree polynomials can be evaluated in 4 multiplies and 5 adds; sixth degree polynomials can be evaluated in 4 multiplies and 7 adds; if any of this strikes you as interesting, consult references [3-5]. The subject has something of the same entertaining, if impractical, flavor as that of fast matrix multiplication, discussed in §2.11.

Turn now to algebraic manipulations. You multiply a polynomial of degree $n-1$ (array of length n) by a monomial factor $x-a$ by a bit of code like the following,

```
c(n+1)=c(n)
do 11 j=n,2,-1
    c(j)=c(j-1)-c(j)*a
enddo 11
c(1)=-c(1)*a
```

Likewise, you divide a polynomial of degree $n-1$ by a monomial factor $x-a$ (synthetic division again) using

```
rem=c(n)
c(n)=0.
do 11 i=n-1,1,-1
    swap=c(i)
    c(i)=rem
    rem=swap+rem*a
enddo 11
```

which leaves you with a new polynomial array and a numerical remainder `rem`.

Multiplication of two general polynomials involves straightforward summing of the products, each involving one coefficient from each polynomial. Division of two general polynomials, while it can be done awkwardly in the fashion taught using pencil and paper, is susceptible to a good deal of streamlining. Witness the following routine based on the algorithm in [3].

```
SUBROUTINE poldiv(u,n,v,nv,q,r)
INTEGER n,nv
REAL q(n),r(n),u(n),v(nv)
    Given the n coefficients of a polynomial in u(1:n), and the nv coefficients of another
    polynomial in v(1:nv), divide the polynomial u by the polynomial v ("u"/"v") giving
    a quotient polynomial whose coefficients are returned in q(1:n-nv+1), and a remainder
    polynomial whose coefficients are returned in r(1:nv-1). The arrays q and r are dimen-
    sioned with lengths n, but the elements r(nv)...r(n) and q(n-nv+2)...q(n) will be
    returned as zero.
INTEGER j,k
do 11 j=1,n
    r(j)=u(j)
    q(j)=0.
enddo 11
do 13 k=n-nv,0,-1
    q(k+1)=r(nv+k)/v(nv)
    do 12 j=nv+k-1,k+1,-1
        r(j)=r(j)-q(k+1)*v(j-k)
    enddo 12
enddo 13
do 14 j=nv,n
    r(j)=0.
enddo 14
return
END
```

Rational Functions

You evaluate a rational function like

$$R(x) = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{5.3.4}$$

in the obvious way, namely as two separate polynomials followed by a divide. As a matter of convention one usually chooses $q_0 = 1$, obtained by dividing numerator and denominator by any other q_0. It is often convenient to have both sets of coefficients stored in a single array, and to have a standard subroutine available for doing the evaluation:

```
FUNCTION ratval(x,cof,mm,kk)
INTEGER kk,mm
DOUBLE PRECISION ratval,x,cof(mm+kk+1)     Note precision! Change to REAL if desired.
    Given mm, kk, and cof(1:mm+kk+1), evaluate and return the rational function (cof(1)+
    cof(2)x+···+cof(mm+1)x^mm)/(1+cof(mm+2)x+···+cof(mm+kk+1)x^kk).
INTEGER j
DOUBLE PRECISION sumd,sumn
sumn=cof(mm+1)
do 11 j=mm,1,-1
    sumn=sumn*x+cof(j)
enddo 11
sumd=0.d0
do 12 j=mm+kk+1,mm+2,-1
    sumd=(sumd+cof(j))*x
enddo 12
ratval=sumn/(1.d0+sumd)
return
END
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 183, 190. [1]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 361–363. [2]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6. [3]

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 4.

Winograd, S. 1970, *Communications on Pure and Applied Mathematics*, vol. 23, pp. 165–179. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley). [5]

5.4 Complex Arithmetic

Since FORTRAN has the built-in data type COMPLEX, you can generally let the compiler and intrinsic function library take care of complex arithmetic for you. Generally, but not always. For a program with only a small number of complex operations, you may want to code these yourself, in-line. Or, you may find that your compiler is not up to snuff: It is disconcertingly common to encounter complex operations that produce overflows or underflows when both the complex operands and the complex result are perfectly representable. This occurs, we think, because software companies assign inexperienced programmers to what they believe to be the perfectly trivial task of implementing complex arithmetic.

Actually, complex arithmetic is not *quite* trivial. Addition and subtraction are done in the obvious way, performing the operation separately on the real and imaginary parts of the operands. Multiplication can also be done in the obvious way, with 4 multiplications, one addition, and one subtraction,

$$(a+ib)(c+id)=(ac-bd)+i(bc+ad) \tag{5.4.1}$$

(the addition before the i doesn't count; it just separates the real and imaginary parts notationally). But it is sometimes faster to multiply via

$$(a+ib)(c+id)=(ac-bd)+i[(a+b)(c+d)-ac-bd] \tag{5.4.2}$$

which has only three multiplications (ac, bd, $(a+b)(c+d)$), plus two additions and three subtractions. The total operations count is higher by two, but multiplication is a slow operation on some machines.

While it is true that intermediate results in equations (5.4.1) and (5.4.2) can overflow even when the final result is representable, this happens only when the final answer is on the edge of representability. Not so for the complex modulus, if you or your compiler are misguided enough to compute it as

$$|a+ib|=\sqrt{a^2+b^2} \qquad \text{(bad!)} \tag{5.4.3}$$

whose intermediate result will overflow if either a or b is as large as the square root of the largest representable number (e.g., 10^{19} as compared to 10^{38}). The right way to do the calculation is

$$|a+ib|=\begin{cases}|a|\sqrt{1+(b/a)^2} & |a|\geq|b|\\ |b|\sqrt{1+(a/b)^2} & |a|<|b|\end{cases} \tag{5.4.4}$$

Complex division should use a similar trick to prevent avoidable overflows, underflow, or loss of precision,

$$\frac{a+ib}{c+id}=\begin{cases}\dfrac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c|\geq|d|\\[2ex] \dfrac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c|<|d|\end{cases} \tag{5.4.5}$$