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 is represented numerically as a stored array of coefficients, `c[j]` with `j`$= 0, \dots, N$. We will always take `c[0]` to be the constant term in the polynomial, `c[`N`]` the coefficient of x^N; 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[0]+c[1]*x+c[2]*x*x+c[3]*x*x*x+c[4]*x*x*x*x;
```

or (even worse!),

```
p=c[0]+c[1]*x+c[2]*pow(x,2.0)+c[3]*pow(x,3.0)+c[4]*pow(x,4.0);
```

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[0]+x*(c[1]+x*(c[2]+x*(c[3]+x*c[4])));
```

or

```
p=(((c[4]*x+c[3])*x+c[2])*x+c[1])*x+c[0];
```

If the number of coefficients `c[0..n]` is large, one writes

```
p=c[n];
for(j=n-1;j>=0;j--) p=p*x+c[j];
```

or

```
p=c[j=n];
while (j>0) p=p*x+c[--j];
```

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

```
p=c[n];
dp=0.0;
for(j=n-1;j>=0;j--) {dp=dp*x+p; p=p*x+c[j];}
```

or

```
p=c[j=n];
dp=0.0;
while (j>0) {dp=dp*x+p; p=p*x+c[--j];}
```

which yields 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 of its derivatives simultaneously:

```
void ddpoly(float c[], int nc, float x, float pd[], int nd)
Given the nc+1 coefficients of a polynomial of degree nc as an array c[0..nc] with c[0]
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[0] and nd derivatives as pd[1..nd].
{
    int nnd,j,i;
    float cnst=1.0;

    pd[0]=c[nc];
    for (j=1;j<=nd;j++) pd[j]=0.0;
    for (i=nc-1;i>=0;i--) {
        nnd=(nd < (nc-i) ? nd : nc-i);
        for (j=nnd;j>=1;j--)
            pd[j]=pd[j]*x+pd[j-1];
        pd[0]=pd[0]*x+c[i];
    }
    for (i=2;i<=nd;i++) {          After the first derivative, factorial constants come in.
        cnst *= i;
        pd[i] *= cnst;
    }
}
```

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 range `[0..n-1]`) by a monomial factor $x-a$ by a bit of code like the following,

```
c[n]=c[n-1];
for (j=n-1;j>=1;j--) c[j]=c[j-1]-c[j]*a;
c[0] *= (-a);
```

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

```
rem=c[n];
c[n]=0.0;
for(i=n-1;i>=0;i--) {
    swap=c[i];
    c[i]=rem;
    rem=swap+rem*a;
}
```

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].

```
void poldiv(float u[], int n, float v[], int nv, float q[], float r[])
```
Given the `n+1` coefficients of a polynomial of degree `n` in `u[0..n]`, and the `nv+1` coefficients of another polynomial of degree `nv` in `v[0..nv]`, divide the polynomial `u` by the polynomial `v` ("`u`"/"`v`") giving a quotient polynomial whose coefficients are returned in `q[0..n]`, and a remainder polynomial whose coefficients are returned in `r[0..n]`. The elements `r[nv..n]` and `q[n-nv+1..n]` are returned as zero.
```
{
    int k,j;

    for (j=0;j<=n;j++) {
        r[j]=u[j];
        q[j]=0.0;
    }
    for (k=n-nv;k>=0;k--) {
        q[k]=r[nv+k]/v[nv];
        for (j=nv+k-1;j>=k;j--) r[j] -= q[k]*v[j-k];
    }
    for (j=nv;j<=n;j++) r[j]=0.0;
}
```

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 function available for doing the evaluation:

```
double ratval(double x, double cof[], int mm, int kk)
Given mm, kk, and cof[0..mm+kk], evaluate and return the rational function (cof[0] +
cof[1]x + ··· + cof[mm]x^mm)/(1 + cof[mm+1]x + ··· + cof[mm+kk]x^kk).
{
    int j;
    double sumd,sumn;               Note precision! Change to float if desired.

    for (sumn=cof[mm],j=mm-1;j>=0;j--) sumn=sumn*x+cof[j];
    for (sumd=0.0,j=mm+kk;j>=mm+1;j--) sumd=(sumd+cof[j])*x;
    return sumn/(1.0+sumd);
}
```

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

As we mentioned in §1.2, the lack of built-in complex arithmetic in C is a nuisance for numerical work. Even in languages like FORTRAN that have complex data types, 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.