CHAPTER 5

Evaluation of Functions

5.0 Introduction

The purpose of this chapter is to acquaint you with a selection of the techniques that are frequently used in evaluating functions. In Chapter 6, we will apply and illustrate these techniques by giving routines for a variety of specific functions. The purposes of this chapter and the next are thus mostly congruent. Occasionally, however, the method of choice for a particular special function in Chapter 6 is peculiar to that function. By comparing this chapter to the next one, you should get some idea of the balance between "general" and "special" methods that occurs in practice.

Insofar as that balance favors general methods, this chapter should give you ideas about how to write your own routine for the evaluation of a function that, while "special" to you, is not so special as to be included in Chapter 6 or the standard function libraries.

CITED REFERENCES AND FURTHER READING:

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

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover), Chapter 7.

5.1 Polynomials and Rational Functions

A polynomial of degree N is represented numerically as a stored array of coefficients, `c[j]` with `j`$= 0, \ldots, N$. We will always take `c[0]` to be the constant term in the polynomial and `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-1]` is large, one writes

```
p=c[n-1];
for(j=n-2;j>=0;j--) p=p*x+c[j];
```

or

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

We can formalize this by defining a function object (or functor) that binds a reference to an array of coefficients and endows them with a polynomial evaluation function,

poly.h

```
struct Poly {
Polynomial function object that binds a reference to a vector of coefficients.
    VecDoub &c;
    Poly(VecDoub &cc) : c(cc) {}
    Doub operator() (Doub x) {
    Returns the value of the polynomial at x.
        Int j;
        Doub p = c[j=c.size()-1];
        while (j>0) p = p*x + c[--j];
        return p;
    }
};
```

which allows you to write things like

```
y = Poly(c)(x);
```

where `c` is a coefficient vector.

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

```
p=c[n-1];
dp=0.;
for(j=n-2;j>=0;j--) {dp=dp*x+p; p=p*x+c[j];}
```

or

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

which yields the polynomial as `p` and its derivative as `dp` using coefficients `c[0..n-1]`.

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(VecDoub_I &c, const Doub x, VecDoub_O &pd)
```
poly.h

Given the coefficients of a polynomial of degree nc as an array c[0..nc] of size nc+1 (with c[0] being the constant term), and given a value x, this routine fills an output array pd of size nd+1 with the value of the polynomial evaluated at x in pd[0], and the first nd derivatives at x in pd[1..nd].

```
{
    Int nnd,j,i,nc=c.size()-1,nd=pd.size()-1;
    Doub cnst=1.0;
    pd[0]=c[nc];
    for (j=1;j<nd+1;j++) pd[j]=0.0;
    for (i=nc-1;i>=0;i--) {
        nnd=(nd < (nc-i) ? nd : nc-i);
        for (j=nnd;j>0;j--) pd[j]=pd[j]*x+pd[j-1];
        pd[0]=pd[0]*x+c[i];
    }
    for (i=2;i<nd+1;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 some extra addition. For example, the polynomial

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 \tag{5.1.1}$$

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

$$P(x) = [(Ax + B)^2 + Ax + C][(Ax + B)^2 + D] + E \tag{5.1.2}$$

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

$$\begin{aligned} A &= (a_4)^{1/4} \\ B &= \frac{a_3 - A^3}{4A^3} \\ D &= 3B^2 + 8B^3 + \frac{a_1 A - 2a_2 B}{A^2} \\ C &= \frac{a_2}{A^2} - 2B - 6B^2 - D \\ E &= a_0 - B^4 - B^2(C + D) - CD \end{aligned} \tag{5.1.3}$$

Fifth-degree polynomials can be evaluated in four multiplies and five adds; sixth-degree polynomials can be evaluated in four multiplies and seven adds; if any of this strikes you as interesting, consult references [3-5]. The subject has something of the same 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.;
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].

poly.h

```
void poldiv(VecDoub_I &u, VecDoub_I &v, VecDoub_O &q, VecDoub_O &r)
```

Divide a polynomial `u` by a polynomial `v`, and return the quotient and remainder polynomials in `q` and `r`, respectively. The four polynomials are represented as vectors of coefficients, each starting with the constant term. There is no restriction on the relative lengths of `u` and `v`, and either may have trailing zeros (represent a lower degree polynomial than its length allows). `q` and `r` are returned with the size of `u`, but will usually have trailing zeros.

```
{
    Int k,j,n=u.size()-1,nv=v.size()-1;
    while (nv >= 0 && v[nv] == 0.) nv--;
    if (nv < 0) throw("poldiv divide by zero polynomial");
    r = u;                                  May do a resize.
    q.assign(u.size(),0.);                  May do a resize.
    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;
}
```

5.1.1 Rational Functions

You evaluate a rational function like

$$R(x) = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{5.1.4}$$

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 the numerator and denominator by any other q_0. In that case, it is often convenient to have both sets of coefficients, omitting q_0, stored in a single array, in the order

$$(p_0, p_1, \ldots, p_\mu, q_1, \ldots, q_\nu) \tag{5.1.5}$$

The following object encapsulates a rational function. It provides constructors from either separate numerator and denominator polynomials, or a single array like (5.1.5) with explicit values for `n` $= \mu + 1$ and `d` $= \nu + 1$. The evaluation function makes `Ratfn` a functor, like `Poly`. We'll make use of this object in §5.12 and §5.13.

poly.h

```
struct Ratfn {
Function object for a rational function.
    VecDoub cofs;
    Int nn,dd;                          Number of numerator, denominator coefficients.

    Ratfn(VecDoub_I &num, VecDoub_I &den) : cofs(num.size()+den.size()-1),
    nn(num.size()), dd(den.size()) {
    Constructor from numerator, denominator polyomials (as coefficient vectors).
        Int j;
        for (j=0;j<nn;j++) cofs[j] = num[j]/den[0];
        for (j=1;j<dd;j++) cofs[j+nn-1] = den[j]/den[0];
    }

    Ratfn(VecDoub_I &coffs, const Int n, const Int d) : cofs(coffs), nn(n),
    dd(d) {}
    Constructor from coefficients already normalized and in a single array.

    Doub operator() (Doub x) const {
    Evaluate the rational function at x and return result.
        Int j;
        Doub sumn = 0., sumd = 0.;
        for (j=nn-1;j>=0;j--) sumn = sumn*x + cofs[j];
        for (j=nn+dd-2;j>=nn;j--) sumd = sumd*x + cofs[j];
        return sumn/(1.0+x*sumd);
    }

};
```

5.1.2 Parallel Evaluation of a Polynomial

A polynomial of degree N can be evaluated in about $\log_2 N$ parallel steps [6]. This is best illustrated by an example, for example with $N = 5$. Start with the vector of coefficients, imagining appended zeros:

$$c_0, \quad c_1, \quad c_2, \quad c_3, \quad c_4, \quad c_5, \quad 0, \quad \ldots \tag{5.1.6}$$

Now add the elements by pairs, multiplying the second of each pair by x:

$$c_0 + c_1 x, \quad c_2 + c_3 x, \quad c_4 + c_5 x, \quad 0, \quad \ldots \tag{5.1.7}$$

Now the same operation, but with the multiplier x^2:

$$(c_0 + c_1 x) + (c_2 + c_3 x)x^2, \quad (c_4 + c_5 x) + (0)x^2, \quad 0 \quad \ldots \tag{5.1.8}$$

And a final time with multiplier x^4:

$$[(c_0 + c_1 x) + (c_2 + c_3 x)x^2] + [(c_4 + c_5 x) + (0)x^2]x^4, \quad 0 \quad \ldots \tag{5.1.9}$$

We are left with a vector of (active) length 1, whose value is the desired polynomial evaluation. You can see that the zeros are just a bookkeeping device for taking care of the case where the active subvector has an odd length; in an actual implementation you can avoid most operations on the zeros. This parallel method generally has better roundoff properties than the standard sequential coding.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: 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. 1997, *Seminumerical Algorithms*, 3rd 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, "On the number of multiplications necessary to compute certain functions," *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]

Estrin, G. 1960, quoted in Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6.4.[6]

5.2 Evaluation of Continued Fractions

Continued fractions are often powerful ways of evaluating functions that occur in scientific applications. A continued fraction looks like this:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cdots}}}}} \tag{5.2.1}$$

Printers prefer to write this as

$$f(x) = b_0 + \frac{a_1}{b_1 +} \frac{a_2}{b_2 +} \frac{a_3}{b_3 +} \frac{a_4}{b_4 +} \frac{a_5}{b_5 +} \cdots \tag{5.2.2}$$

In either (5.2.1) or (5.2.2), the a's and b's can themselves be functions of x, usually linear or quadratic monomials at worst (i.e., constants times x or times x^2). For example, the continued fraction representation of the tangent function is

$$\tan x = \frac{x}{1 -} \frac{x^2}{3 -} \frac{x^2}{5 -} \frac{x^2}{7 -} \cdots \tag{5.2.3}$$

Continued fractions frequently converge much more rapidly than power series expansions, and in a much larger domain in the complex plane (not necessarily including the domain of convergence of the series, however). Sometimes the continued fraction converges best where the series does worst, although this is not a general rule. Blanch [1] gives a good review of the most useful convergence tests for continued fractions.

There are standard techniques, including the important *quotient-difference algorithm*, for going back and forth between continued fraction approximations, power series approximations, and rational function approximations. Consult Acton [2] for an introduction to this subject, and Fike [3] for further details and references.

How do you tell how far to go when evaluating a continued fraction? Unlike a series, you can't just evaluate equation (5.2.1) from left to right, stopping when the change is small. Written in the form of (5.2.1), the only way to evaluate the continued fraction is from right to left, first (blindly!) guessing how far out to start. This is not the right way.

The right way is to use a result that relates continued fractions to rational approximations, and that gives a means of evaluating (5.2.1) or (5.2.2) from left to right. Let f_n denote the result of evaluating (5.2.2) with coefficients through a_n and b_n. Then

$$f_n = \frac{A_n}{B_n} \tag{5.2.4}$$

where A_n and B_n are given by the following recurrence:

$$\begin{aligned} A_{-1} &\equiv 1 \qquad & B_{-1} &\equiv 0 \\ A_0 &\equiv b_0 \qquad & B_0 &\equiv 1 \\ A_j &= b_j A_{j-1} + a_j A_{j-2} \qquad & B_j &= b_j B_{j-1} + a_j B_{j-2} \qquad j = 1, 2, \ldots, n \end{aligned} \tag{5.2.5}$$

This method was invented by J. Wallis in 1655 (!) and is discussed in his *Arithmetica Infinitorum* [4]. You can easily prove it by induction.

In practice, this algorithm has some unattractive features: The recurrence (5.2.5) frequently generates very large or very small values for the partial numerators and denominators A_j and B_j. There is thus the danger of overflow or underflow of the floating-point representation. However, the recurrence (5.2.5) is linear in the A's and B's. At any point you can rescale the currently saved two levels of the recurrence, e.g., divide A_j, B_j, A_{j-1}, and B_{j-1} all by B_j. This incidentally makes $A_j = f_j$ and is convenient for testing whether you have gone far enough: See if f_j and f_{j-1} from the last iteration are as close as you would like them to be. If B_j happens to be zero, which can happen, just skip the renormalization for this cycle. A fancier level of optimization is to renormalize only when an overflow is imminent, saving the unnecessary divides. In fact, the C library function `ldexp` can be used to avoid division entirely. (See the end of §6.5 for an example.)

Two newer algorithms have been proposed for evaluating continued fractions. *Steed's method* does not use A_j and B_j explicitly, but only the *ratio* $D_j = B_{j-1}/B_j$. One calculates D_j and $\Delta f_j = f_j - f_{j-1}$ recursively using

$$D_j = 1/(b_j + a_j D_{j-1}) \tag{5.2.6}$$

$$\Delta f_j = (b_j D_j - 1)\Delta f_{j-1} \tag{5.2.7}$$

Steed's method (see, e.g., [5]) avoids the need for rescaling of intermediate results. However, for certain continued fractions you can occasionally run into a situation where the denominator in (5.2.6) approaches zero, so that D_j and Δf_j are very large. The next Δf_{j+1} will typically cancel this large change, but with loss of accuracy in the numerical running sum of the f_j's. It is awkward to program around this, so Steed's method can be recommended only for cases where you know in advance that no denominator can vanish. We will use it for a special purpose in the routine `besselik` (§6.6).

The best general method for evaluating continued fractions seems to be the *modified Lentz's method* [6]. The need for rescaling intermediate results is avoided by using *both* the ratios

$$C_j = A_j/A_{j-1}, \qquad D_j = B_{j-1}/B_j \tag{5.2.8}$$

and calculating f_j by

$$f_j = f_{j-1} C_j D_j \tag{5.2.9}$$

From equation (5.2.5), one easily shows that the ratios satisfy the recurrence relations

$$D_j = 1/(b_j + a_j D_{j-1}), \qquad C_j = b_j + a_j/C_{j-1} \tag{5.2.10}$$

In this algorithm there is the danger that the denominator in the expression for D_j, or the quantity C_j itself, might approach zero. Either of these conditions invalidates (5.2.10). However, Thompson and Barnett [5] show how to modify Lentz's algorithm to fix this: Just shift the offending term by a small amount, e.g., 10^{-30}. If you work through a cycle of the algorithm with this prescription, you will see that f_{j+1} is accurately calculated.

In detail, the modified Lentz's algorithm is this:

- Set $f_0 = b_0$; if $b_0 = 0$, set $f_0 = \texttt{tiny}$.
- Set $C_0 = f_0$.
- Set $D_0 = 0$.
- For $j = 1, 2, \ldots$

  Set $D_j = b_j + a_j D_{j-1}$.
  If $D_j = 0$, set $D_j = \texttt{tiny}$.
  Set $C_j = b_j + a_j/C_{j-1}$.
  If $C_j = 0$, set $C_j = \texttt{tiny}$.
  Set $D_j = 1/D_j$.
  Set $\Delta_j = C_j D_j$.
  Set $f_j = f_{j-1}\Delta_j$.
  If $|\Delta_j - 1| < \texttt{eps}$, then exit.

Here `eps` is your floating-point precision, say 10^{-7} or 10^{-15}. The parameter `tiny` should be less than typical values of $|b_j|$, say 10^{-30}.

The above algorithm assumes that you can terminate the evaluation of the continued fraction when $|f_j - f_{j-1}|$ is sufficiently small. This is usually the case, but by no means guaranteed. Jones [7] gives a list of theorems that can be used to justify this termination criterion for various kinds of continued fractions.

There is at present no rigorous analysis of error propagation in Lentz's algorithm. However, empirical tests suggest that it is at least as good as other methods.

5.2.1 Manipulating Continued Fractions

Several important properties of continued fractions can be used to rewrite them in forms that can speed up numerical computation. An *equivalence transformation*

$$a_n \to \lambda a_n, \quad b_n \to \lambda b_n, \quad a_{n+1} \to \lambda a_{n+1} \tag{5.2.11}$$

leaves the value of a continued fraction unchanged. By a suitable choice of the scale factor λ you can often simplify the form of the a's and the b's. Of course, you can carry out successive equivalence transformations, possibly with different λ's, on successive terms of the continued fraction.

The *even* and *odd* parts of a continued fraction are continued fractions whose successive convergents are f_{2n} and f_{2n+1}, respectively. Their main use is that they converge twice as fast as the original continued fraction, and so if their terms are not much more complicated than the terms in the original, there can be a big savings in computation. The formula for the even part of (5.2.12) is

$$f_{\text{even}} = d_0 + \frac{c_1}{d_1 +} \; \frac{c_2}{d_2 +} \; \cdots \tag{5.2.12}$$

where in terms of intermediate variables

$$\alpha_1 = \frac{a_1}{b_1}$$
$$\alpha_n = \frac{a_n}{b_n b_{n-1}}, \qquad n \geq 2 \tag{5.2.13}$$

we have

$$d_0 = b_0, \quad c_1 = \alpha_1, \quad d_1 = 1 + \alpha_2$$
$$c_n = -\alpha_{2n-1}\alpha_{2n-2}, \quad d_n = 1 + \alpha_{2n-1} + \alpha_{2n}, \qquad n \geq 2 \tag{5.2.14}$$

You can find the similar formula for the odd part in the review by Blanch [1]. Often a combination of the transformations (5.2.14) and (5.2.11) is used to get the best form for numerical work.

We will make frequent use of continued fractions in the next chapter.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, §3.10.

Blanch, G. 1964, "Numerical Evaluation of Continued Fractions," *SIAM Review*, vol. 6, pp. 383–421.[1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 11.[2]

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 1.

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), §8.2, §10.4, and §10.5.[3]

Wallis, J. 1695, in *Opera Mathematica*, vol. 1, p. 355, Oxoniae e Theatro Shedoniano. Reprinted by Georg Olms Verlag, Hildeshein, New York (1972).[4]

Thompson, I.J., and Barnett, A.R. 1986, "Coulomb and Bessel Functions of Complex Arguments and Order," *Journal of Computational Physics*, vol. 64, pp. 490–509.[5]

Lentz, W.J. 1976, "Generating Bessel Functions in Mie Scattering Calculations Using Continued Fractions," *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 Series and Their Convergence

Everybody knows that an analytic function can be expanded in the neighborhood of a point x_0 in a power series,

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k \tag{5.3.1}$$

Such series are straightforward to evaluate. You don't, of course, evaluate the kth power of $x - x_0$ *ab initio* for each term; rather, you keep the $k-1$st power and update

it with a multiply. Similarly, the form of the coefficients a_k is often such as to make use of previous work: Terms like $k!$ or $(2k)!$ can be updated in a multiply or two.

How do you know when you have summed enough terms? In practice, the terms had better be getting small fast, otherwise the series is not a good technique to use in the first place. While not mathematically rigorous in all cases, standard practice is to quit when the term you have just added is smaller in magnitude than some small ϵ times the magnitude of the sum thus far accumulated. (But watch out if isolated instances of $a_k = 0$ are possible!)

Sometimes you will want to compute a function from a series representation even when the computation is *not* efficient. For example, you may be using the values obtained to fit the function to an approximating form that you will use subsequently (cf. §5.8). If you are summing very large numbers of slowly convergent terms, pay attention to roundoff errors! In floating-point representation it is more accurate to sum a list of numbers in the order starting with the smallest one, rather than starting with the largest one. It is even better to group terms pairwise, then in pairs of pairs, etc., so that all additions involve operands of comparable magnitude.

A weakness of a power series representation is that it is guaranteed *not* to converge farther than that distance from x_0 at which a singularity is encountered *in the complex plane*. This catastrophe is not usually unexpected: When you find a power series in a book (or when you work one out yourself), you will generally also know the radius of convergence. An insidious problem occurs with series that converge everywhere (in the mathematical sense), but almost nowhere fast enough to be useful in a numerical method. Two familiar examples are the sine function and the Bessel function of the first kind,

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \tag{5.3.2}$$

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!(k+n)!} \tag{5.3.3}$$

Both of these series converge for all x. But both don't even start to converge until $k \gg |x|$; before this, their terms are increasing. Even worse, the terms alternate in sign, leading to large cancellation errors with finite precision arithmetic. This makes these series useless for large x.

5.3.1 Divergent Series

Divergent series are often very useful. One class consists of power series outside their radius of convergence, which can often be summed by the acceleration techniques we will describe below. Another class is asymptotic series, such as the Euler series that comes from Euler's integral (related to the exponential integral E_1):

$$E(x) = \int_0^{\infty} \frac{e^{-t}}{1+xt}\,dt \simeq \sum_{k=0}^{\infty} (-1)^k k!\, x^k \tag{5.3.4}$$

Here the series is derived by expanding $(1+xt)^{-1}$ in powers of x and integrating term by term. The series diverges for all $x \neq 0$. For $x = 0.1$, the series gives only three significant digits before diverging. Nevertheless, convergence acceleration

techniques allow effortless evaluation of the function $E(x)$, even for $x \sim 2$, when the series is wildly divergent!

5.3.2 Accelerating the Convergence of Series

There are several tricks for accelerating the rate of convergence of a series or, equivalently, of a sequence of partial sums

$$s_n = \sum_{k=0}^{n} a_k \tag{5.3.5}$$

(We'll use the terms sequence and series interchangeably in this section.) An excellent review has been given by Weniger [1]. Before we can describe the tricks and when to use them, we need to classify some of the ways in which a sequence can converge. Suppose s_n converges to s, say, and that

$$\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = \rho \tag{5.3.6}$$

If $0 < |\rho| < 1$ we say the convergence is *linear*; if $\rho = 1$ it is *logarithmic*; and if $\rho = 0$ it is *hyperlinear*. Of course, if $|\rho| > 1$ the sequence diverges. (More rigorously, this classification should be given in terms of the so-called remainders $s_n - s$ [1]. However, our definition is more practical and is equivalent if we restrict the logarithmic case to terms of the same sign.)

The prototype of linear convergence is a geometric series,

$$s_n = \sum_{k=0}^{n} x^k = \frac{1 - x^{n+1}}{1 - x} \tag{5.3.7}$$

It is easy to see that $\rho = x$, and so we have linear convergence for $0 < |x| < 1$. The prototype of logarithmic convergence is the series for the Riemann zeta function,

$$\zeta(x) = \sum_{k=1}^{\infty} \frac{1}{k^x}, \qquad x > 1 \tag{5.3.8}$$

which is notoriously slowly convergent, especially as $x \to 1$. The series (5.3.2) and (5.3.3), or the series for e^x, exemplify hyperlinear convergence. We see that hyperlinear convergence doesn't necessarily imply that the series is easy to evaluate for all values of x. Sometimes convergence acceleration is helpful only once the terms start decreasing.

Probably the most famous series transformation for accelerating convergence is the Euler transformation (see, e.g., [2,3]), which dates from 1755. Euler's transformation works on *alternating series* (where the terms in the sum alternate in sign). Generally it is advisable to do a small number of terms directly, through term $n-1$, say, and then apply the transformation to the rest of the series beginning with term n. The formula (for n even) is

$$\sum_{s=0}^{\infty} (-1)^s a_s = a_0 - a_1 + a_2 \ldots - a_{n-1} + \sum_{s=0}^{\infty} \frac{(-1)^s}{2^{s+1}} [\Delta^s a_n] \tag{5.3.9}$$

Here Δ is the *forward difference operator*, i.e.,

$$\begin{aligned} \Delta a_n &\equiv a_{n+1} - a_n \\ \Delta^2 a_n &\equiv a_{n+2} - 2a_{n+1} + a_n \\ \Delta^3 a_n &\equiv a_{n+3} - 3a_{n+2} + 3a_{n+1} - a_n \end{aligned} \qquad \text{etc.} \tag{5.3.10}$$

Of course you don't actually do the infinite sum on the right-hand side of (5.3.9), but only the first, say, p terms, thus requiring the first p differences (5.3.10) obtained from the terms starting at a_n. There is an elegant and subtle implementation of Euler's transformation due to van Wijngaarden [6], discussed in full in a Webnote [7].

Euler's transformation is an example of a *linear* transformation: The partial sums of the transformed series are linear combinations of the partial sums of the original series. Euler's transformation and other linear transformations, while still important theoretically, have generally been superseded by newer *nonlinear* transformations that are considerably more powerful. As usual in numerical work, there is no free lunch: While the nonlinear transformations are more powerful, they are somewhat riskier than linear transformations in that they can occasionally fail spectacularly. But if you follow the guidance below, we think that you will never again resort to puny linear transformations.

The oldest example of a nonlinear sequence transformation is *Aitken's* Δ^2 *process*. If s_n, s_{n+1}, s_{n+2} are three successive partial sums, then an improved estimate is

$$s'_n \equiv s_n - \frac{(s_{n+1} - s_n)^2}{s_{n+2} - 2s_{n+1} + s_n} = s_n - \frac{(\Delta s_n)^2}{\Delta^2 s_n} \tag{5.3.11}$$

The formula (5.3.11) is exact for a geometric series, which is one way of deriving it. If you form the sequence of s'_i's, you can apply (5.3.11) a second time to *that* sequence, and so on. (In practice, this iteration will only rarely do much for you after the first stage.) Note that equation (5.3.11) should be computed as written; there exist algebraically equivalent forms that are much more susceptible to roundoff error.

Aitken's Δ^2-process works only on linearly convergent sequences. Like Euler's transformation, it has also been superseded by algorithms such as the two we will now describe. After giving routines for these algorithms, we will supply some rules of thumb on when to use them.

The first "modern" nonlinear transformation was proposed by Shanks. An efficient recursive implementation was given by Wynn, called the ϵ algorithm. Aitken's Δ^2-process is a special case of the ϵ algorithm, corresponding to using just three terms at a time. Although we will not give a derivation here, it is easy to state exactly what the ϵ algorithm does: If you input the partial sums of a power series, the ϵ algorithm returns the "diagonal" Padé approximants (§5.12) evaluated at the value of x used in the power series. (The coefficients in the approximant itself are not calculated.) That is, if $[M/N]$ denotes the Padé approximant with a polynomial of degree M in the numerator and degree N in the denominator, the algorithm returns the numerical values of the approximants

$$[0,0], \quad [1/0], \quad [1/1], \quad [2/1], \quad [2/2], \quad [3,2], \quad [3,3] \quad \ldots \tag{5.3.12}$$

(The object `Epsalg` below is roughly equivalent to `pade` in §5.12 followed by an evaluation of the resulting rational function.)

In the object `Epsalg`, which is based on a routine in [1], you supply the sequence term by term and monitor the output for convergence in the calling program. Internally, the routine contains a check for division by zero and substitutes a large number

for the result. There are three conditions under which this check can be triggered: (i) Most likely, the algorithm has already converged, and should have been stopped earlier; (ii) there is an "accidental" zero term, and the program will recover; (iii) hardly ever in practice, the algorithm can actually fail because of a perverse combination of terms. Because (i) and (ii) are vastly more common than (iii), `Epsalg` hides the check condition and instead returns the last-known good estimate.

series.h

```
struct Epsalg {
```

Convergence acceleration of a sequence by the ϵ algorithm. Initialize by calling the constructor with arguments `nmax`, an upper bound on the number of terms to be summed, and `epss`, the desired accuracy. Then make successive calls to the function `next`, with argument the next partial sum of the sequence. The current estimate of the limit of the sequence is returned by `next`. The flag `cnvgd` is set when convergence is detected.

```
    VecDoub e;                                  Workspace.
    Int n,ncv;
    Bool cnvgd;
    Doub eps,small,big,lastval,lasteps;         Numbers near machine underflow and
                                                    overflow limits.
    Epsalg(Int nmax, Doub epss) : e(nmax), n(0), ncv(0),
    cnvgd(0), eps(epss), lastval(0.) {
        small = numeric_limits<Doub>::min()*10.0;
        big = numeric_limits<Doub>::max();
    }

    Doub next(Doub sum) {
        Doub diff,temp1,temp2,val;
        e[n]=sum;
        temp2=0.0;
        for (Int j=n; j>0; j--) {
            temp1=temp2;
            temp2=e[j-1];
            diff=e[j]-temp2;
            if (abs(diff) <= small)
                e[j-1]=big;
            else
                e[j-1]=temp1+1.0/diff;
        }
        n++;
        val = (n & 1) ? e[0] : e[1];            Cases of n even or odd.
        if (abs(val) > 0.01*big) val = lastval;
        lasteps = abs(val-lastval);
        if (lasteps > eps) ncv = 0;
        else ncv++;
        if (ncv >= 3) cnvgd = 1;
        return (lastval = val);
    }

};
```

The last few lines above implement a simple criterion for deciding whether the sequence has converged. For problems whose convergence is robust, you can simply put your calls to `next` inside a `while` loop like this:

```
Doub val, partialsum, eps=...;
Epsalg mysum(1000,eps);
while (! mysum.cnvgd) {
    partialsum = ...
    val = mysum.next(partialsum);
}
```

For more delicate cases, you can ignore the `cnvgd` flag and just keep calling `next` until you are satisfied with the convergence.

A large class of modern nonlinear transformations can be derived by using the concept of a *model sequence*. The idea is to choose a "simple" sequence that approximates the asymptotic form of the given sequence and construct a transformation that sums the model sequence exactly. Presumably the transformation will work well for other sequences with similar asymptotic properties. For example, a geometric series provides the model sequence for Aitken's Δ^2-process.

The *Levin transformation* is probably the best single sequence acceleration method currently known. It is based on approximating a sequence asymptotically by an expression of the form

$$s_n = s + \omega_n \sum_{j=0}^{k-1} \frac{c_j}{(n+\beta)^j} \tag{5.3.13}$$

Here ω_n is the dominant term in the remainder of the sequence:

$$s_n - s = \omega_n [c + O(n^{-1})], \qquad n \to \infty \tag{5.3.14}$$

The constants c_j are arbitrary, and β is a parameter that is restricted to be positive. Levin showed that for a model sequence of the form (5.3.13), the following transformation gives the exact value of the series:

$$s = \frac{\displaystyle\sum_{j=0}^{k} (-1)^j \binom{k}{j} \frac{(\beta+n+j)^{k-1}}{(\beta+n+k)^{k-1}} \frac{s_{n+j}}{\omega_{n+j}}}{\displaystyle\sum_{j=0}^{k} (-1)^j \binom{k}{j} \frac{(\beta+n+j)^{k-1}}{(\beta+n+k)^{k-1}} \frac{1}{\omega_{n+j}}} \tag{5.3.15}$$

(The common factor $(\beta+n+k)^{k-1}$ in the numerator and denominator reduces the chances of overflow for large k.) A derivation of equation (5.3.15) is given in a Webnote [4].

The numerator and denominator in (5.3.15) are not computed as written. Instead, they can be computed efficiently from a single recurrence relation with different starting values (see [1] for a derivation):

$$D_{k+1}^n(\beta) = D_k^{n+1}(\beta) - \frac{(\beta+n)(\beta+n+k)^{k-1}}{(\beta+n+k+1)^k} D_k^n(\beta) \tag{5.3.16}$$

The starting values are

$$D_0^n(\beta) = \begin{cases} s_n/\omega_n, & \text{numerator} \\ 1/\omega_n, & \text{denominator} \end{cases} \tag{5.3.17}$$

Although D_k^n is a two-dimensional object, the recurrence can be coded in a one-dimensional array proceeding up the counterdiagonal $n+k =$ constant.

The choice (5.3.14) doesn't determine ω_n uniquely, but if you have analytic information about your series, this is where you can make use of it. Usually you won't

be so lucky, in which case you can make a choice based on heuristics. For example, the remainder in an alternating series is approximately half the first neglected term, which suggests setting ω_n equal to a_n or a_{n+1}. These are called the Levin t and d transformations, respectively. Similarly, the remainder for a geometric series is the difference between the partial sum (5.3.7) and its limit $1/(1-x)$. This can be written as $a_n a_{n+1}/(a_n - a_{n+1})$, which defines the Levin v transformation. The most popular choice comes from approximating the remainder in the ζ function (5.3.8) by an integral:

$$\sum_{k=n+1}^{\infty} \frac{1}{k^x} \approx \int_{n+1}^{\infty} \frac{dk}{k^x} = \frac{(n+1)^{1-x}}{x-1} = \frac{(n+1)a_{n+1}}{x-1} \tag{5.3.18}$$

This motivates the choice $(n+\beta)a_n$ (Levin u transformation), where β is usually chosen to be 1. To summarize:

$$\omega_n = \begin{cases} (\beta+n)a_n, & u \text{ transformation} \\ a_n, & t \text{ transformation} \\ a_{n+1}, & d \text{ transformation (modified } t \text{ transformation)} \\ \dfrac{a_n a_{n+1}}{a_n - a_{n+1}}, & v \text{ transformation} \end{cases} \tag{5.3.19}$$

For sequences that are not partial sums, so that the individual a_n's are not defined, replace a_n by Δs_{n-1} in (5.3.19).

Here is the routine for Levin's transformation, also based on the routine in [1]:

series.h

```
struct Levin {
```

Convergence acceleration of a sequence by the Levin transformation. Initialize by calling the constructor with arguments nmax, an upper bound on the number of terms to be summed, and epss, the desired accuracy. Then make successive calls to the function next, which returns the current estimate of the limit of the sequence. The flag cnvgd is set when convergence is detected.

```
    VecDoub numer,denom;
    Int n,ncv;
    Bool cnvgd;
    Doub small,big;
    Doub eps,lastval,lasteps;
```

Numerator and denominator computed via (5.3.16).

Numbers near machine underflow and overflow limits.

```
    Levin(Int nmax, Doub epss) : numer(nmax), denom(nmax), n(0), ncv(0),
    cnvgd(0), eps(epss), lastval(0.) {
        small=numeric_limits<Doub>::min()*10.0;
        big=numeric_limits<Doub>::max();
    }

    Doub next(Doub sum, Doub omega, Doub beta=1.) {
```

Arguments: sum, the nth partial sum of the sequence; omega, the nth remainder estimate ω_n, usually from (5.3.19); and the parameter beta, which should usually be set to 1, but sometimes 0.5 works better. The current estimate of the limit of the sequence is returned.

```
        Int j;
        Doub fact,ratio,term,val;
        term=1.0/(beta+n);
        denom[n]=term/omega;
        numer[n]=sum*denom[n];
        if (n > 0) {
            ratio=(beta+n-1)*term;
            for (j=1;j<=n;j++) {
```

```
                fact=(n-j+beta)*term;
                numer[n-j]=numer[n-j+1]-fact*numer[n-j];
                denom[n-j]=denom[n-j+1]-fact*denom[n-j];
                term=term*ratio;
            }
        }
        n++;
        val = abs(denom[0]) < small ? lastval : numer[0]/denom[0];
        lasteps = abs(val-lastval);
        if (lasteps <= eps) ncv++;
        if (ncv >= 2) cnvgd = 1;
        return (lastval = val);
    }
};
```

You can use, or not use, the `cnvgd` flag exactly as previously discussed for `Epsalg`.

An alternative to the model sequence method of deriving sequence transformations is to use extrapolation of a polynomial or rational function approximation to a series, e.g., as in Wynn's ρ algorithm [1]. Since none of these methods generally beats the two we have given, we won't say any more about them.

5.3.3 Practical Hints and an Example

There is no general theoretical understanding of nonlinear sequence transformations. Accordingly, most of the practical advice is based on numerical experiments [5]. You might have thought that summing a wildly divergent series is the hardest problem for a sequence transformation. However, the difficulty of a problem depends more on whether the terms are all of the same sign or whether the signs alternate, rather than whether the sequence actually converges or not. In particular, logarithmically convergent series with terms all of the same sign are generally the most difficult to sum. Even the best acceleration methods are corrupted by rounding errors when accelerating logarithmic convergence. You should always use double precision and be prepared for some loss of significant digits. Typically one observes convergence up to some optimum number of terms, and then a loss of significant digits if one tries to go further. Moreover, there is no single algorithm that can accelerate every logarithmically convergent sequence. Nevertheless, there are some good rules of thumb.

First, note that among divergent series it is useful to separate out asymptotic series, where the terms first decrease before increasing, as a separate class from other divergent series, e.g., power series outside their radius of convergence. For alternating series, whether convergent, asymptotic, or divergent power series, Levin's u transformation is almost always the best choice. For monotonic linearly convergent or monotonic divergent power series, the ϵ algorithm typically is the first choice, but the u transformation often does a reasonable job. For logarithmic convergence, the u transformation is clearly the best. (The ϵ algorithm fails completely.) For series with irregular signs or other nonstandard features, typically the ϵ algorithm is relatively robust, often succeeding where other algorithms fail. Finally, for monotonic asymptotic series, such as (6.3.11) for $\mathrm{Ei}(x)$, there is nothing better than direct summation without acceleration.

The v and t transformations are almost as good as the u transformation, except that the t transformation typically fails for logarithmic convergence.

If you have only a few numerical terms of some sequence and no theoretical insight, blindly applying a convergence accelerator can be dangerous. The algorithm

can sometimes display "convergence" that is only apparent, not real. The remedy is to try two different transformations as a check.

Since convergence acceleration is so much more difficult for a series of positive terms than for an alternating series, occasionally it is useful to convert a series of positive terms into an alternating series. Van Wijngaarden has given a transformation for accomplishing this [6]:

$$\sum_{r=1}^{\infty} v_r = \sum_{r=1}^{\infty} (-1)^{r-1} w_r \tag{5.3.20}$$

where

$$w_r \equiv v_r + 2v_{2r} + 4v_{4r} + 8v_{8r} + \cdots \tag{5.3.21}$$

Equations (5.3.20) and (5.3.21) replace a simple sum by a two-dimensional sum, each term in (5.3.20) being itself an infinite sum (5.3.21). This may seem a strange way to save on work! Since, however, the indices in (5.3.21) increase tremendously rapidly, as powers of 2, it often requires only a few terms to converge (5.3.21) to extraordinary accuracy. You do, however, need to be able to compute the v_r's efficiently for "random" values r. The standard "updating" tricks for sequential r's, mentioned above following equation (5.3.1), can't be used.

Once you've generated the alternating series by Van Wijngaarden's transformation, the Levin d transformation is particularly effective at summing the series [8]. This strategy is most useful for linearly convergent series with ρ close to 1. For logarithmically convergent series, even the transformed series (5.3.21) is often too slowly convergent to be useful numerically.

As an example of how to call the routines `Epsalg` or `Levin`, consider the problem of evaluating the integral

$$I = \int_0^{\infty} \frac{x}{1+x^2} J_0(x)\, dx = K_0(1) = 0.4210244382\ldots \tag{5.3.22}$$

Standard quadrature methods such as `qromo` fail because the integrand has a long oscillatory tail, giving alternating positive and negative contributions that tend to cancel. A good way of evaluating such an integral is to split it into a sum of integrals between successive zeros of $J_0(x)$:

$$I = \int_0^{\infty} f(x)\, dx = \sum_{j=0}^{\infty} I_j \tag{5.3.23}$$

where

$$I_j = \int_{x_{j-1}}^{x_j} f(x)\, dx, \qquad f(x_j) = 0, \quad j = 0, 1, \ldots \tag{5.3.24}$$

We take x_{-1} equal to the lower limit of the integral, zero in this example. The idea is to evaluate the relatively simple integrals I_j by `qromb` or Gaussian quadrature, and then accelerate the convergence of the series (5.3.23), since we expect the contributions to alternate in sign. For the example (5.3.22), we don't even need accurate values of the zeros of $J_0(x)$. It is good enough to take $x_j = (j + 1)\pi$ which is asymptotically correct. Here is the code:

levex.h

```
Doub func(const Doub x)
Integrand for (5.3.22).
{
    if (x == 0.0)
        return 0.0;
    else {
        Bessel bess;
        return x*bess.jnu(0.0,x)/(1.0+x*x);
    }
}

Int main_levex(void)
This sample program shows how to use the Levin u transformation to evaluate an oscillatory
integral, equation (5.3.22).
{
    const Doub PI=3.141592653589793;
    Int nterm=12;
    Doub beta=1.0,a=0.0,b=0.0,sum=0.0;
    Levin series(100,0.0);
    cout << setw(5) << "N" << setw(19) << "Sum (direct)" << setw(21)
        << "Sum (Levin)" << endl;
    for (Int n=0; n<=nterm; n++) {
        b+=PI;
        Doub s=qromb(func,a,b,1.e-8);
        a=b;
        sum+=s;
        Doub omega=(beta+n)*s;        Use u transformation.
        Doub ans=series.next(sum,omega,beta);
        cout << setw(5) << n << fixed << setprecision(14) << setw(21)
            << sum << setw(21) << ans << endl;
    }
    return 0;
}
```

Setting eps to 1×10^{-8} in qromb, we get 9 significant digits with about 200 function evaluations by $n = 8$. Replacing qromb with a Gaussian quadrature routine cuts the number of function evaluations in half. Note that $n = 8$ corresponds to an upper limit in the integral of 9π, where the amplitude of the integrand is still of order 10^{-2}. This shows the remarkable power of convergence acceleration. (For more on oscillatory integrals, see §13.9.)

CITED REFERENCES AND FURTHER READING:

Weniger, E.J. 1989, "Nonlinear Sequence Transformations for the Acceleration of Convergence and the Summation of Divergent Series," *Computer Physics Reports*, vol. 10, pp. 189–371.[1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at http://www.nr.com/aands, §3.6.[2]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), §2.3.[3]

Numerical Recipes Software 2007, "Derivation of the Levin Transformation," *Numerical Recipes Webnote No. 6*, at http://www.nr.com/webnotes?6 [4]

Smith, D.A., and Ford, W.F. 1982, "Numerical Comparisons of Nonlinear Convergence Accelerators," *Mathematics of Computation*, vol. 38, pp. 481–499.[5]

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 13 [van Wijngaarden's transformations].[6]

Numerical Recipes Software 2007, "Implementation of the Euler Transformation," *Numerical Recipes Webnote No. 5*, at `http://www.nr.com/webnotes?5` [7]

Jentschura, U.D., Mohr, P.J., Soff, G., and Weniger, E.J. 1999, "Convergence Acceleration via Combined Nonlinear-Condensation Transformations," *Computer Physics Communications*, vol. 116, pp. 28–54.[8]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), Chapter 3.

5.4 Recurrence Relations and Clenshaw's Recurrence Formula

Many useful functions satisfy recurrence relations, e.g.,

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \tag{5.4.1}$$

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{5.4.2}$$

$$nE_{n+1}(x) = e^{-x} - xE_n(x) \tag{5.4.3}$$

$$\cos n\theta = 2\cos\theta\cos(n-1)\theta - \cos(n-2)\theta \tag{5.4.4}$$

$$\sin n\theta = 2\cos\theta\sin(n-1)\theta - \sin(n-2)\theta \tag{5.4.5}$$

where the first three functions are Legendre polynomials, Bessel functions of the first kind, and exponential integrals, respectively. (For notation see [1].) These relations are useful for extending computational methods from two successive values of n to other values, either larger or smaller.

Equations (5.4.4) and (5.4.5) motivate us to say a few words about trigonometric functions. If your program's running time is dominated by evaluating trigonometric functions, you are probably doing something wrong. Trig functions whose arguments form a linear sequence $\theta = \theta_0 + n\delta$, $n = 0, 1, 2, \ldots$ are efficiently calculated by the recurrence

$$\begin{aligned}\cos(\theta+\delta) &= \cos\theta - [\alpha\cos\theta + \beta\sin\theta] \\ \sin(\theta+\delta) &= \sin\theta - [\alpha\sin\theta - \beta\cos\theta]\end{aligned} \tag{5.4.6}$$

where α and β are the precomputed coefficients

$$\alpha \equiv 2\sin^2\left(\frac{\delta}{2}\right) \qquad \beta \equiv \sin\delta \tag{5.4.7}$$

The reason for doing things this way, rather than with the standard (and equivalent) identities for sums of angles, is that here α and β do not lose significance if the incremental δ is small. Likewise, the adds in equation (5.4.6) should be done in the order indicated by the square brackets. We will use (5.4.6) repeatedly in Chapter 12, when we deal with Fourier transforms.

Another trick, occasionally useful, is to note that both $\sin\theta$ and $\cos\theta$ can be calculated via a single call to tan:

$$t \equiv \tan\left(\frac{\theta}{2}\right) \qquad \cos\theta = \frac{1-t^2}{1+t^2} \qquad \sin\theta = \frac{2t}{1+t^2} \tag{5.4.8}$$

The cost of getting both sin and cos, if you need them, is thus the cost of tan plus 2 multiplies, 2 divides, and 2 adds. On machines with slow trig functions, this can be a savings. *However*, note that special treatment is required if $\theta \to \pm\pi$. And also note that many modern machines have *very fast* trig functions; so you should not assume that equation (5.4.8) is faster without testing.

5.4.1 Stability of Recurrences

You need to be aware that recurrence relations are not necessarily *stable* against roundoff error in the direction that you propose to go (either increasing n or decreasing n). A three-term linear recurrence relation

$$y_{n+1} + a_n y_n + b_n y_{n-1} = 0, \qquad n = 1, 2, \ldots \tag{5.4.9}$$

has two linearly independent solutions, f_n and g_n, say. Only one of these corresponds to the sequence of functions f_n that you are trying to generate. The other one, g_n, *may* be exponentially growing in the direction that you want to go, or exponentially damped, or exponentially neutral (growing or dying as some power law, for example). If it is exponentially growing, then the recurrence relation is of little or no practical use in that direction. This is the case, e.g., for (5.4.2) in the direction of increasing n, when $x < n$. You cannot generate Bessel functions of high n by forward recurrence on (5.4.2).

To state things a bit more formally, if

$$f_n / g_n \to 0 \quad \text{as} \quad n \to \infty \tag{5.4.10}$$

then f_n is called the *minimal* solution of the recurrence relation (5.4.9). Nonminimal solutions like g_n are called *dominant* solutions. The minimal solution is unique, if it exists, but dominant solutions are not — you can add an arbitrary multiple of f_n to a given g_n. You can evaluate any dominant solution by forward recurrence, *but not the minimal solution*. (Unfortunately it is sometimes the one you want.)

Abramowitz and Stegun (in their Introduction!) [1] give a list of recurrences that are stable in the increasing or decreasing direction. That list does not contain all possible formulas, of course. Given a recurrence relation for some function $f_n(x)$, you can test it yourself with about five minutes of (human) labor: For a fixed x in your range of interest, start the recurrence not with true values of $f_j(x)$ and $f_{j+1}(x)$, but (first) with the values 1 and 0, respectively, and then (second) with 0 and 1, respectively. Generate 10 or 20 terms of the recursive sequences in the direction that you want to go (increasing or decreasing from j), for each of the two starting conditions. Look at the differences between the corresponding members of the two sequences. If the differences stay of order unity (absolute value less than 10, say), then the recurrence is stable. If they increase slowly, then the recurrence may be mildly unstable but quite tolerably so. If they increase catastrophically, then there is an exponentially growing solution of the recurrence. If you know that the function that you want actually corresponds to the growing solution, then you can keep the recurrence formula anyway (e.g., the case of the Bessel function $Y_n(x)$ for increasing n; see §6.5). If you don't know which solution your function corresponds to, you must at this point reject the recurrence formula. Notice that you can do this test *before* you go to the trouble of finding a numerical method for computing the two

starting functions $f_j(x)$ and $f_{j+1}(x)$. Stability is a property of the recurrence, not of the starting values.

An alternative heuristic procedure for testing stability is to replace the recurrence relation by a similar one that is linear with constant coefficients. For example, the relation (5.4.2) becomes

$$y_{n+1} - 2\gamma y_n + y_{n-1} = 0 \tag{5.4.11}$$

where $\gamma \equiv n/x$ is treated as a constant. You solve such recurrence relations by trying solutions of the form $y_n = a^n$. Substituting into the above recurrence gives

$$a^2 - 2\gamma a + 1 = 0 \qquad \text{or} \qquad a = \gamma \pm \sqrt{\gamma^2 - 1} \tag{5.4.12}$$

The recurrence is stable if $|a| \leq 1$ for all solutions a. This holds (as you can verify) if $|\gamma| \leq 1$ or $n \leq x$. The recurrence (5.4.2) thus cannot be used, starting with $J_0(x)$ and $J_1(x)$, to compute $J_n(x)$ for large n.

Possibly you would at this point like the security of some real theorems on this subject (although we ourselves always follow one of the heuristic procedures). Here are two theorems, due to Perron [2]:

Theorem A. If in (5.4.9) $a_n \sim an^\alpha$, $b_n \sim bn^\beta$, as $n \to \infty$, and $\beta < 2\alpha$, then

$$g_{n+1}/g_n \sim -an^\alpha, \qquad f_{n+1}/f_n \sim -(b/a)n^{\beta-\alpha} \tag{5.4.13}$$

and f_n is the minimal solution to (5.4.9).

Theorem B. Under the same conditions as Theorem A, but with $\beta = 2\alpha$, consider the *characteristic polynomial*

$$t^2 + at + b = 0 \tag{5.4.14}$$

If the roots t_1 and t_2 of (5.4.14) have distinct moduli, $|t_1| > |t_2|$, say, then

$$g_{n+1}/g_n \sim t_1 n^\alpha, \qquad f_{n+1}/f_n \sim t_2 n^\alpha \tag{5.4.15}$$

and f_n is again the minimal solution to (5.4.9). Cases other than those in these two theorems are inconclusive for the existence of minimal solutions. (For more on the stability of recurrences, see [3].)

How do you proceed if the solution that you desire *is* the minimal solution? The answer lies in that old aphorism, that every cloud has a silver lining: If a recurrence relation is catastrophically unstable in one direction, then that (undesired) solution will decrease very rapidly in the reverse direction. This means that you can start with *any* seed values for the consecutive f_j and f_{j+1} and (when you have gone enough steps in the stable direction) you will converge to the sequence of functions that you want, times an unknown normalization factor. If there is some other way to normalize the sequence (e.g., by a formula for the sum of the f_n's), then this can be a practical means of function evaluation. The method is called *Miller's algorithm*. An example often given [1,4] uses equation (5.4.2) in just this way, along with the normalization formula

$$1 = J_0(x) + 2J_2(x) + 2J_4(x) + 2J_6(x) + \cdots \tag{5.4.16}$$

Incidentally, there is an important relation between three-term recurrence relations and *continued fractions*. Rewrite the recurrence relation (5.4.9) as

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n + y_{n+1}/y_n} \tag{5.4.17}$$

Iterating this equation, starting with n, gives

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n -} \frac{b_{n+1}}{a_{n+1} -} \cdots \tag{5.4.18}$$

Pincherle's theorem [2] tells us that (5.4.18) converges if and only if (5.4.9) has a minimal solution f_n, in which case it converges to f_n/f_{n-1}. This result, usually for the case $n = 1$, and combined with some way to determine f_0, underlies many of the practical methods for computing special functions that we give in the next chapter.

5.4.2 Clenshaw's Recurrence Formula

Clenshaw's recurrence formula [5] is an elegant and efficient way to evaluate a sum of coefficients times functions that obey a recurrence formula, e.g.,

$$f(\theta) = \sum_{k=0}^{N} c_k \cos k\theta \qquad \text{or} \qquad f(x) = \sum_{k=0}^{N} c_k P_k(x)$$

Here is how it works: Suppose that the desired sum is

$$f(x) = \sum_{k=0}^{N} c_k F_k(x) \tag{5.4.19}$$

and that F_k obeys the recurrence relation

$$F_{n+1}(x) = \alpha(n,x)F_n(x) + \beta(n,x)F_{n-1}(x) \tag{5.4.20}$$

for some functions $\alpha(n,x)$ and $\beta(n,x)$. Now define the quantities y_k $(k = N, N-1, \ldots, 1)$ by the recurrence

$$\begin{aligned} y_{N+2} &= y_{N+1} = 0 \\ y_k &= \alpha(k,x)y_{k+1} + \beta(k+1,x)y_{k+2} + c_k \quad (k = N, N-1, \ldots, 1) \end{aligned} \tag{5.4.21}$$

If you solve equation (5.4.21) for c_k on the left, and then write out explicitly the sum (5.4.19), it will look (in part) like this:

$$\begin{aligned} f(x) = & \cdots \\ & + [y_8 - \alpha(8,x)y_9 - \beta(9,x)y_{10}]F_8(x) \\ & + [y_7 - \alpha(7,x)y_8 - \beta(8,x)y_9]F_7(x) \\ & + [y_6 - \alpha(6,x)y_7 - \beta(7,x)y_8]F_6(x) \\ & + [y_5 - \alpha(5,x)y_6 - \beta(6,x)y_7]F_5(x) \\ & + \cdots \\ & + [y_2 - \alpha(2,x)y_3 - \beta(3,x)y_4]F_2(x) \\ & + [y_1 - \alpha(1,x)y_2 - \beta(2,x)y_3]F_1(x) \\ & + [c_0 + \beta(1,x)y_2 - \beta(1,x)y_2]F_0(x) \end{aligned} \tag{5.4.22}$$

Notice that we have added and subtracted $\beta(1,x)y_2$ in the last line. If you examine the terms containing a factor of y_8 in (5.4.22), you will find that they sum to zero as a consequence of the recurrence relation (5.4.20); similarly for all the other y_k's down through y_2. The only surviving terms in (5.4.22) are

$$f(x) = \beta(1,x)F_0(x)y_2 + F_1(x)y_1 + F_0(x)c_0 \tag{5.4.23}$$

Equations (5.4.21) and (5.4.23) are *Clenshaw's recurrence formula* for doing the sum (5.4.19): You make one pass down through the y_k's using (5.4.21); when you have reached y_2 and y_1, you apply (5.4.23) to get the desired answer.

Clenshaw's recurrence as written above incorporates the coefficients c_k in a downward order, with k decreasing. At each stage, the effect of all previous c_k's is "remembered" as two coefficients that multiply the functions F_{k+1} and F_k (ultimately F_0 and F_1). If the functions F_k are small when k is large, *and* if the coefficients c_k are small when k is *small*, then the sum can be dominated by small F_k's. In this case, the remembered coefficients will involve a delicate cancellation and there can be a catastrophic loss of significance. An example would be to sum the trivial series

$$J_{15}(1) = 0 \times J_0(1) + 0 \times J_1(1) + \ldots + 0 \times J_{14}(1) + 1 \times J_{15}(1) \tag{5.4.24}$$

Here J_{15}, which is tiny, ends up represented as a canceling linear combination of J_0 and J_1, which are of order unity.

The solution in such cases is to use an alternative Clenshaw recurrence that incorporates the c_k's in an upward direction. The relevant equations are

$$y_{-2} = y_{-1} = 0 \tag{5.4.25}$$

$$y_k = \frac{1}{\beta(k+1,x)}[y_{k-2} - \alpha(k,x)y_{k-1} - c_k], \quad k = 0, 1, \ldots, N-1 \tag{5.4.26}$$

$$f(x) = c_N F_N(x) - \beta(N,x)F_{N-1}(x)y_{N-1} - F_N(x)y_{N-2} \tag{5.4.27}$$

The rare case where equations (5.4.25) – (5.4.27) should be used instead of equations (5.4.21) and (5.4.23) can be detected automatically by testing whether the operands in the first sum in (5.4.23) are opposite in sign and nearly equal in magnitude. Other than in this special case, Clenshaw's recurrence is always stable, independent of whether the recurrence for the functions F_k is stable in the upward or downward direction.

5.4.3 Parallel Evaluation of Linear Recurrence Relations

When desirable, linear recurrence relations can be evaluated with a lot of parallelism. Consider the general first-order linear recurrence relation

$$u_j = a_j + b_{j-1}u_{j-1}, \qquad j = 2, 3, \ldots, n \tag{5.4.28}$$

with initial value $u_1 = a_1$. To parallelize the recurrence, we can employ the powerful general strategy of *recursive doubling*. Write down equation (5.4.28) for $2j$ and for $2j-1$:

$$\begin{aligned} u_{2j} &= a_{2j} + b_{2j-1}u_{2j-1} \\ u_{2j-1} &= a_{2j-1} + b_{2j-2}u_{2j-2} \end{aligned} \tag{5.4.29}$$

Substitute the second of these equations into the first to eliminate u_{2j-1} and get

$$u_{2j} = (a_{2j} + a_{2j-1}b_{2j-1}) + (b_{2j-2}b_{2j-1})u_{2j-2} \tag{5.4.30}$$

This is a new recurrence of the same form as (5.4.28) but over only the even u_j's, and hence involving only $n/2$ terms. Clearly we can continue this process recursively, halving the number of terms in the recurrence at each stage, until we are left with a recurrence of length 1 or 2 that we can do explicitly. Each time we finish a subpart of the recursion, we fill in the odd terms in the recurrence, using the second equation in (5.4.29). In practice, it's even easier than it sounds. The total number of operations is the same as for serial evaluation, but they are done in about $\log_2 n$ parallel steps.

There is a variant of recursive doubling, called *cyclic reduction*, that can be implemented with a straightforward iteration loop instead of a recursive procedure [6]. Here we start by writing down the recurrence (5.4.28) for all adjacent terms u_j and u_{j-1} (not just the even ones, as before). Eliminating u_{j-1}, just as in equation (5.4.30), gives

$$u_j = (a_j + a_{j-1}b_{j-1}) + (b_{j-2}b_{j-1})u_{j-2} \tag{5.4.31}$$

which is a first-order recurrence with new coefficients a'_j and b'_j. Repeating this process gives successive formulas for u_j in terms of $u_{j-2}, u_{j-4}, u_{j-8}, \ldots$. The procedure terminates when we reach u_{j-n} (for n a power of 2), which is zero for all j. Thus the last step gives u_j equal to the last set of a'_j's.

In cyclic reduction, the length of the vector u_j that is updated at each stage does not decrease by a factor of 2 at each stage, but rather only decreases from $\sim n$ to $\sim n/2$ during all $\log_2 n$ stages. Thus the total number of operations carried out is $O(n \log n)$, as opposed to $O(n)$ for recursive doubling. Whether this is important depends on the details of the computer's architecture.

Second-order recurrence relations can also be parallelized. Consider the second-order recurrence relation

$$y_j = a_j + b_{j-2}y_{j-1} + c_{j-2}y_{j-2}, \qquad j = 3, 4, \ldots, n \tag{5.4.32}$$

with initial values

$$y_1 = a_1, \qquad y_2 = a_2 \tag{5.4.33}$$

With this numbering scheme, you supply coefficients $a_1, \ldots, a_n$, $b_1, \ldots, b_{n-2}$, and $c_1, \ldots, c_{n-2}$. Rewrite the recurrence relation in the form [6]

$$\begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix} = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix} \begin{pmatrix} y_{j-1} \\ y_j \end{pmatrix}, \qquad j = 2, \ldots, n-1 \tag{5.4.34}$$

that is,

$$\mathbf{u}_j = \mathbf{a}_j + \mathbf{b}_{j-1} \cdot \mathbf{u}_{j-1}, \qquad j = 2, \ldots, n-1 \tag{5.4.35}$$

where

$$\mathbf{u}_j = \begin{pmatrix} y_j \\ y_{j+1} \end{pmatrix}, \qquad \mathbf{a}_j = \begin{pmatrix} 0 \\ a_{j+1} \end{pmatrix}, \qquad \mathbf{b}_{j-1} = \begin{pmatrix} 0 & 1 \\ c_{j-1} & b_{j-1} \end{pmatrix} \tag{5.4.36}$$

and

$$\mathbf{u}_1 = \mathbf{a}_1 = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \tag{5.4.37}$$

This is a first-order recurrence relation for the vectors $\mathbf{u}_j$ and can be solved by either of the algorithms described above. The only difference is that the multiplications are matrix multiplications with the 2×2 matrices $\mathbf{b}_j$. After the first recursive call, the zeros in $\mathbf{a}$ and $\mathbf{b}$ are lost, so we have to write the routine for general two-dimensional vectors and matrices. Note that this algorithm does not avoid the potential instability problems associated with second-order recurrences that were discussed in §5.4.1. Also note that the algorithm generalizes in the obvious way to higher-order recurrences: An nth-order recurrence can be written as a first-order recurrence involving vectors and matrices of dimension n.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, pp. xiii, 697.[1]

Gautschi, W. 1967, "Computational Aspects of Three-Term Recurrence Relations," *SIAM Review*, vol. 9, pp. 24–82.[2]

Lakshmikantham, V., and Trigiante, D. 1988, *Theory of Difference Equations: Numerical Methods and Applications* (San Diego: Academic Press).[3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), pp. 20ff.[4]

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H.M. Stationery Office).[5]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §4.4.3, p. 111.

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), p. 76.

Hockney, R.W., and Jesshope, C.R. 1988, *Parallel Computers 2: Architecture, Programming, and Algorithms* (Bristol and Philadelphia: Adam Hilger), §5.2.4 and §5.4.2.[6]

5.5 Complex Arithmetic

Since C++ has a built-in class `complex`, you can generally let the compiler and the class 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 mistake the implementation of complex arithmetic for a completely trivial task, not requiring any particular finesse.

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 four multiplications, one addition, and one subtraction:

$$(a+ib)(c+id) = (ac-bd)+i(bc+ad) \tag{5.5.1}$$

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

$$(a+ib)(c+id) = (ac-bd)+i[(a+b)(c+d)-ac-bd] \tag{5.5.2}$$

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 (5.5.1) and (5.5.2) 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 is misguided enough to compute it as

$$|a+ib| = \sqrt{a^2+b^2} \qquad \text{(bad!)} \tag{5.5.3}$$

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

$$|a+ib| = \begin{cases} |a|\sqrt{1+(b/a)^2} & |a| \geq |b| \\ |b|\sqrt{1+(a/b)^2} & |a| < |b| \end{cases} \tag{5.5.4}$$

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

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c| \geq |d| \\ \dfrac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c| < |d| \end{cases} \tag{5.5.5}$$

Of course you should calculate repeated subexpressions, like c/d or d/c, only once.

Complex square root is even more complicated, since we must both guard intermediate results and also enforce a chosen branch cut (here taken to be the negative real axis). To take the square root of $c+id$ first compute

$$w \equiv \begin{cases} 0 & c = d = 0 \\ \sqrt{|c|}\sqrt{\dfrac{1+\sqrt{1+(d/c)^2}}{2}} & |c| \geq |d| \\ \sqrt{|d|}\sqrt{\dfrac{|c/d|+\sqrt{1+(c/d)^2}}{2}} & |c| < |d| \end{cases} \tag{5.5.6}$$

Then the answer is

$$\sqrt{c+id} = \begin{cases} 0 & w = 0 \\ w + i\left(\dfrac{d}{2w}\right) & w \neq 0, c \geq 0 \\ \dfrac{|d|}{2w} + iw & w \neq 0, c < 0, d \geq 0 \\ \dfrac{|d|}{2w} - iw & w \neq 0, c < 0, d < 0 \end{cases} \tag{5.5.7}$$

CITED REFERENCES AND FURTHER READING:

Midy, P., and Yakovlev, Y. 1991, "Computing Some Elementary Functions of a Complex Variable," *Mathematics and Computers in Simulation*, vol. 33, pp. 33–49.

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley) [see solutions to exercises 4.2.1.16 and 4.6.4.41].

5.6 Quadratic and Cubic Equations

The roots of simple algebraic equations can be viewed as being functions of the equations' coefficients. We are taught these functions in elementary algebra. Yet, surprisingly many people don't know the right way to solve a quadratic equation with two real roots, or to obtain the roots of a cubic equation.

There are two ways to write the solution of the *quadratic equation*

$$ax^2 + bx + c = 0 \tag{5.6.1}$$

with real coefficients a, b, c, namely

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.6.2}$$

and

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \tag{5.6.3}$$

If you use *either* (5.6.2) *or* (5.6.3) to get the two roots, you are asking for trouble: If either a or c (or both) is small, then one of the roots will involve the subtraction of b from a very nearly equal quantity (the discriminant); you will get that root very inaccurately. The correct way to compute the roots is

$$q \equiv -\frac{1}{2}\left[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right] \tag{5.6.4}$$

Then the two roots are

$$x_1 = \frac{q}{a} \qquad \text{and} \qquad x_2 = \frac{c}{q} \tag{5.6.5}$$

If the coefficients a, b, c, are complex rather than real, then the above formulas still hold, except that in equation (5.6.4) the sign of the square root should be chosen so as to make

$$\mathrm{Re}(b^* \sqrt{b^2 - 4ac}) \geq 0 \tag{5.6.6}$$

where Re denotes the real part and asterisk denotes complex conjugation.

Apropos of quadratic equations, this seems a convenient place to recall that the inverse hyperbolic functions $\sinh^{-1}$ and $\cosh^{-1}$ are in fact just logarithms of solutions to such equations

$$\sinh^{-1}(x) = \ln\left(x + \sqrt{x^2 + 1}\right) \tag{5.6.7}$$

$$\cosh^{-1}(x) = \pm \ln\left(x + \sqrt{x^2 - 1}\right) \tag{5.6.8}$$

Equation (5.6.7) is numerically robust for $x \geq 0$. For negative x, use the symmetry $\sinh^{-1}(-x) = -\sinh^{-1}(x)$. Equation (5.6.8) is of course valid only for $x \geq 1$.

For the *cubic equation*

$$x^3 + ax^2 + bx + c = 0 \tag{5.6.9}$$

with real or complex coefficients a, b, c, first compute

$$Q \equiv \frac{a^2 - 3b}{9} \qquad \text{and} \qquad R \equiv \frac{2a^3 - 9ab + 27c}{54} \tag{5.6.10}$$

If Q and R are real (always true when a, b, c are real) *and* $R^2 < Q^3$, then the cubic equation has three real roots. Find them by computing

$$\theta = \arccos(R/\sqrt{Q^3}) \tag{5.6.11}$$

in terms of which the three roots are

$$\begin{aligned} x_1 &= -2\sqrt{Q}\cos\left(\frac{\theta}{3}\right) - \frac{a}{3} \\ x_2 &= -2\sqrt{Q}\cos\left(\frac{\theta + 2\pi}{3}\right) - \frac{a}{3} \\ x_3 &= -2\sqrt{Q}\cos\left(\frac{\theta - 2\pi}{3}\right) - \frac{a}{3} \end{aligned} \tag{5.6.12}$$

(This equation first appears in Chapter VI of François Viète's treatise "De emendatione," published in 1615!)

Otherwise, compute

$$A = -\left[R + \sqrt{R^2 - Q^3}\right]^{1/3} \tag{5.6.13}$$

where the sign of the square root is chosen to make

$$\mathrm{Re}(R^* \sqrt{R^2 - Q^3}) \geq 0 \tag{5.6.14}$$

(asterisk again denoting complex conjugation). If Q and R are both real, equations (5.6.13) – (5.6.14) are equivalent to

$$A = -\mathrm{sgn}(R)\left[|R| + \sqrt{R^2 - Q^3}\right]^{1/3} \tag{5.6.15}$$

where the positive square root is assumed. Next compute

$$B = \begin{cases} Q/A & (A \neq 0) \\ 0 & (A = 0) \end{cases} \tag{5.6.16}$$

in terms of which the three roots are

$$x_1 = (A + B) - \frac{a}{3} \tag{5.6.17}$$

(the single real root when a, b, c are real) and

$$\begin{aligned} x_2 &= -\frac{1}{2}(A + B) - \frac{a}{3} + i\frac{\sqrt{3}}{2}(A - B) \\ x_3 &= -\frac{1}{2}(A + B) - \frac{a}{3} - i\frac{\sqrt{3}}{2}(A - B) \end{aligned} \tag{5.6.18}$$

(in that same case, a complex-conjugate pair). Equations (5.6.13) – (5.6.16) are arranged both to minimize roundoff error and also (as pointed out by A.J. Glassman) to ensure that no choice of branch for the complex cube root can result in the spurious loss of a distinct root.

If you need to solve many cubic equations with only slightly different coefficients, it is more efficient to use Newton's method (§9.4).

CITED REFERENCES AND FURTHER READING:

Weast, R.C. (ed.) 1967, *Handbook of Tables for Mathematics*, 3rd ed. (Cleveland: The Chemical Rubber Co.), pp. 130–133.

Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.

McKelvey, J.P. 1984, "Simple Transcendental Expressions for the Roots of Cubic Equations," *American Journal of Physics*, vol. 52, pp. 269–270; see also vol. 53, p. 775, and vol. 55, pp. 374–375.

5.7 Numerical Derivatives

Imagine that you have a procedure that computes a function $f(x)$, and now you want to compute its derivative $f'(x)$. Easy, right? The definition of the derivative, the limit as $h \to 0$ of

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{5.7.1}$$

practically suggests the program: Pick a small value h; evaluate $f(x+h)$; you probably have $f(x)$ already evaluated, but if not, do it too; finally, apply equation (5.7.1). What more needs to be said?

Quite a lot, actually. Applied uncritically, the above procedure is almost guaranteed to produce inaccurate results. Applied properly, it can be the right way to compute a derivative only when the function f is *fiercely* expensive to compute; when you already have invested in computing $f(x)$; and when, therefore, you want to get the derivative in no more than a single additional function evaluation. In such a situation, the remaining issue is to choose h properly, an issue we now discuss.

There are two sources of error in equation (5.7.1), truncation error and roundoff error. The truncation error comes from higher terms in the Taylor series expansion,

$$f(x+h) = f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + \tfrac{1}{6}h^3 f'''(x) + \cdots \tag{5.7.2}$$

whence

$$\frac{f(x+h) - f(x)}{h} = f' + \frac{1}{2}hf'' + \cdots \tag{5.7.3}$$

The roundoff error has various contributions. First there is roundoff error in h: Suppose, by way of an example, that you are at a point $x = 10.3$ and you blindly choose $h = 0.0001$. Neither $x = 10.3$ nor $x + h = 10.3000 1$ is a number with an exact representation in binary; each is therefore represented with some fractional error characteristic of the machine's floating-point format, ϵ_m, whose value in single precision may be $\sim 10^{-7}$. The error in the *effective* value of h, namely the difference between $x + h$ and x as represented in the machine, is therefore on the order of $\epsilon_m x$,

which implies a fractional error in h of order $\sim \epsilon_m x/h \sim 10^{-2}$! By equation (5.7.1), this immediately implies at least the same large fractional error in the derivative.

We arrive at Lesson 1: Always choose h so that $x+h$ and x differ by an exactly representable number. This can usually be accomplished by the program steps

$$\begin{aligned} \texttt{temp} &= x + h \\ h &= \texttt{temp} - x \end{aligned} \tag{5.7.4}$$

Some optimizing compilers, and some computers whose floating-point chips have higher internal accuracy than is stored externally, can foil this trick; if so, it is usually enough to declare `temp` as `volatile`, or else to call a dummy function `donothing(temp)` between the two equations (5.7.4). This forces `temp` into and out of addressable memory.

With h an "exact" number, the roundoff error in equation (5.7.1) is approximately $e_r \sim \epsilon_f |f(x)/h|$. Here ϵ_f is the fractional accuracy with which f is computed; for a simple function this may be comparable to the machine accuracy, $\epsilon_f \approx \epsilon_m$, but for a complicated calculation with additional sources of inaccuracy it may be larger. The truncation error in equation (5.7.3) is on the order of $e_t \sim |hf''(x)|$. Varying h to minimize the sum $e_r + e_t$ gives the optimal choice of h,

$$h \sim \sqrt{\frac{\epsilon_f f}{f''}} \approx \sqrt{\epsilon_f}\, x_c \tag{5.7.5}$$

where $x_c \equiv (f/f'')^{1/2}$ is the "curvature scale" of the function f or the "characteristic scale" over which it changes. In the absence of any other information, one often assumes $x_c = x$ (except near $x = 0$, where some other estimate of the typical x scale should be used).

With the choice of equation (5.7.5), the fractional accuracy of the computed derivative is

$$(e_r + e_t)/|f'| \sim \sqrt{\epsilon_f}(ff''/f'^2)^{1/2} \sim \sqrt{\epsilon_f} \tag{5.7.6}$$

Here the last order-of-magnitude equality assumes that f, f', and f'' all share the same characteristic length scale, which is usually the case. One sees that the simple finite difference equation (5.7.1) gives *at best* only the square root of the machine accuracy ϵ_m.

If you can afford two function evaluations for each derivative calculation, then it is significantly better to use the symmetrized form

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{5.7.7}$$

In this case, by equation (5.7.2), the truncation error is $e_t \sim h^2 f'''$. The roundoff error e_r is about the same as before. The optimal choice of h, by a short calculation analogous to the one above, is now

$$h \sim \left(\frac{\epsilon_f f}{f'''}\right)^{1/3} \sim (\epsilon_f)^{1/3} x_c \tag{5.7.8}$$

and the fractional error is

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3}/f' \sim (\epsilon_f)^{2/3} \tag{5.7.9}$$

which will typically be an order of magnitude (single precision) or two orders of magnitude (double precision) *better* than equation (5.7.6). We have arrived at Lesson 2: Choose h to be *the correct* power of ϵ_f or ϵ_m times a characteristic scale x_c.

You can easily derive the correct powers for other cases [1]. For a function of two dimensions, for example, and the mixed derivative formula

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{[f(x+h, y+h) - f(x+h, y-h)] - [f(x-h, y+h) - f(x-h, y-h)]}{4h^2} \tag{5.7.10}$$

the correct scaling is $h \sim \epsilon_f^{1/4} x_c$.

It is disappointing, certainly, that no simple finite difference formula like equation (5.7.1) or (5.7.7) gives an accuracy comparable to the machine accuracy ϵ_m, or even the lower accuracy to which f is evaluated, ϵ_f. Are there no better methods?

Yes, there are. All, however, involve exploration of the function's behavior over scales comparable to x_c, plus some assumption of smoothness, or analyticity, so that the high-order terms in a Taylor expansion like equation (5.7.2) have some meaning. Such methods also involve multiple evaluations of the function f, so their increased accuracy must be weighed against increased cost.

The general idea of "Richardson's deferred approach to the limit" is particularly attractive. For numerical integrals, that idea leads to so-called Romberg integration (for review, see §4.3). For derivatives, one seeks to extrapolate, to $h \to 0$, the result of finite difference calculations with smaller and smaller finite values of h. By the use of Neville's algorithm (§3.2), one uses each new finite difference calculation to produce both an extrapolation of higher order and also extrapolations of previous, lower, orders but with smaller scales h. Ridders [2] has given a nice implementation of this idea; the following program, `dfridr`, is based on his algorithm, modified by an improved termination criterion. Input to the routine is a function f (called `func`), a position x, and a *largest* stepsize `h` (more analogous to what we have called x_c above than to what we have called h). Output is the returned value of the derivative and an estimate of its error, `err`.

dfridr.h

```
template<class T>
Doub dfridr(T &func, const Doub x, const Doub h, Doub &err)
```

Returns the derivative of a function `func` at a point `x` by Ridders' method of polynomial extrapolation. The value `h` is input as an estimated initial stepsize; it need not be small, but rather should be an increment in `x` over which `func` changes substantially. An estimate of the error in the derivative is returned as `err`.

```
{
    const Int ntab=10;                      Sets maximum size of tableau.
    const Doub con=1.4, con2=(con*con);     Stepsize decreased by CON at each iteration.
    const Doub big=numeric_limits<Doub>::max();
    const Doub safe=2.0;                    Return when error is SAFE worse than the
                                                best so far.
    Int i,j;
    Doub errt,fac,hh,ans;
    MatDoub a(ntab,ntab);
    if (h == 0.0) throw("h must be nonzero in dfridr.");
    hh=h;
    a[0][0]=(func(x+hh)-func(x-hh))/(2.0*hh);
    err=big;
    for (i=1;i<ntab;i++) {
        Successive columns in the Neville tableau will go to smaller stepsizes and higher orders of extrapolation.
        hh /= con;
        a[0][i]=(func(x+hh)-func(x-hh))/(2.0*hh);     Try new, smaller stepsize.
        fac=con2;
```

```
        for (j=1;j<=i;j++) {
            a[j][i]=(a[j-1][i]*fac-a[j-1][i-1])/(fac-1.0);
            fac=con2*fac;
            errt=MAX(abs(a[j][i]-a[j-1][i]),abs(a[j][i]-a[j-1][i-1]));
            if (errt <= err) {
                err=errt;
                ans=a[j][i];
            }
        }
        if (abs(a[i][i]-a[i-1][i-1]) >= safe*err) break;
    }
    return ans;
}
```

Compute extrapolations of various orders, requiring no new function evaluations.

The error strategy is to compare each new extrapolation to one order lower, both at the present stepsize and the previous one.

If error is decreased, save the improved answer.

If higher order is worse by a significant factor SAFE, then quit early.

In `dfridr`, the number of evaluations of `func` is typically 6 to 12, but is allowed to be as great as 2×`NTAB`. As a function of input `h`, it is typical for the accuracy to get *better* as `h` is made larger, until a sudden point is reached where nonsensical extrapolation produces an early return with a large error. You should therefore choose a fairly large value for `h` but monitor the returned value `err`, decreasing `h` if it is not small. For functions whose characteristic x scale is of order unity, we typically take `h` to be a few tenths.

Besides Ridders' method, there are other possible techniques. If your function is fairly smooth, and you know that you will want to evaluate its derivative many times at arbitrary points in some interval, then it makes sense to construct a Chebyshev polynomial approximation to the function in that interval, and to evaluate the derivative directly from the resulting Chebyshev coefficients. This method is described in §5.8 – §5.9, following.

Another technique applies when the function consists of data that is tabulated at equally spaced intervals, and perhaps also noisy. One might then want, at each point, to least-squares *fit* a polynomial of some degree M, using an additional number n_L of points to the left and some number n_R of points to the right of each desired x value. The estimated derivative is then the derivative of the resulting fitted polynomial. A very efficient way to do this construction is via Savitzky-Golay smoothing filters, which will be discussed later, in §14.9. There we will give a routine for getting filter coefficients that not only construct the fitting polynomial but, in the accumulation of a single sum of data points times filter coefficients, evaluate it as well. In fact, the routine given, `savgol`, has an argument `ld` that determines which derivative of the fitted polynomial is evaluated. For the first derivative, the appropriate setting is `ld=1`, and the value of the derivative is the accumulated sum divided by the sampling interval h.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.), §5.4 – §5.6.[1]

Ridders, C.J.F. 1982, "Accurate computation of $F'(x)$ and $F'(x)F''(x)$," *Advances in Engineering Software*, vol. 4, no. 2, pp. 75–76.[2]

5.8 Chebyshev Approximation

The Chebyshev polynomial of degree n is denoted $T_n(x)$ and is given by the explicit formula

$$T_n(x) = \cos(n \arccos x) \tag{5.8.1}$$

This may look trigonometric at first glance (and there is in fact a close relation between the Chebyshev polynomials and the discrete Fourier transform); however, (5.8.1) can be combined with trigonometric identities to yield explicit expressions for $T_n(x)$ (see Figure 5.8.1):

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned} \tag{5.8.2}$$

(There also exist inverse formulas for the powers of x in terms of the T_n's — see, e.g., [1].)

The Chebyshev polynomials are orthogonal in the interval $[-1, 1]$ over a weight $(1 - x^2)^{-1/2}$. In particular,

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}\,dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases} \tag{5.8.3}$$

The polynomial $T_n(x)$ has n zeros in the interval $[-1, 1]$, and they are located at the points

$$x = \cos\left(\frac{\pi(k + \frac{1}{2})}{n}\right) \qquad k = 0, 1, \ldots, n-1 \tag{5.8.4}$$

In this same interval there are $n + 1$ extrema (maxima and minima), located at

$$x = \cos\left(\frac{\pi k}{n}\right) \qquad k = 0, 1, \ldots, n \tag{5.8.5}$$

At all of the maxima $T_n(x) = 1$, while at all of the minima $T_n(x) = -1$; it is precisely this property that makes the Chebyshev polynomials so useful in polynomial approximation of functions.

The Chebyshev polynomials satisfy a discrete orthogonality relation as well as the continuous one (5.8.3): If $x_k (k = 0, \ldots, m-1)$ are the m zeros of $T_m(x)$ given by (5.8.4), and if $i, j < m$, then

$$\sum_{k=0}^{m-1} T_i(x_k)T_j(x_k) = \begin{cases} 0 & i \neq j \\ m/2 & i = j \neq 0 \\ m & i = j = 0 \end{cases} \tag{5.8.6}$$

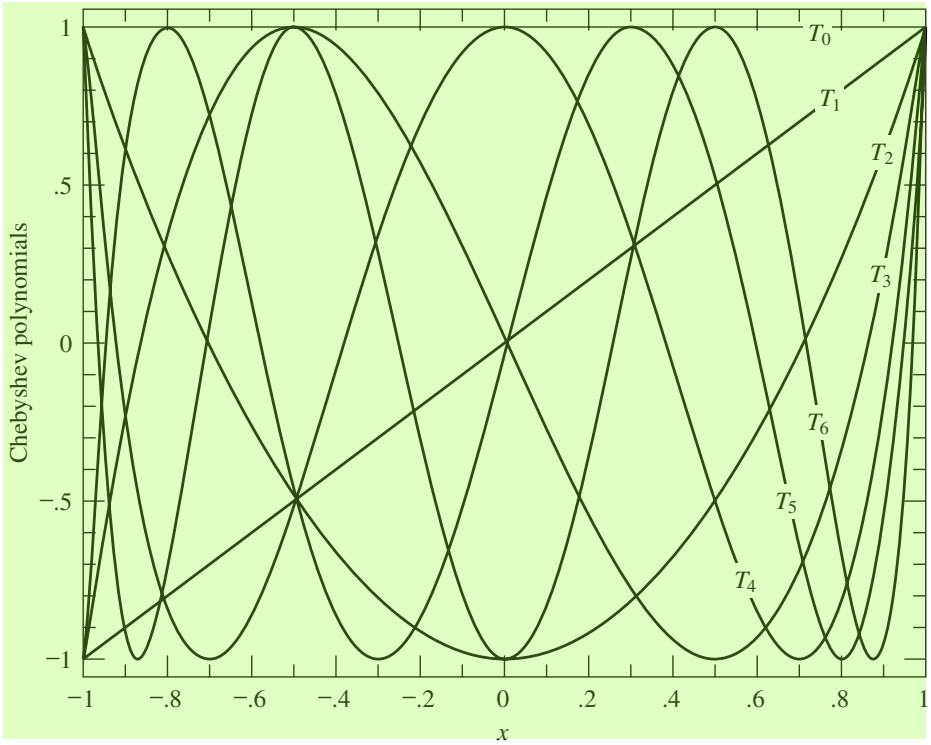

Figure 5.8.1. Chebyshev polynomials $T_0(x)$ through $T_6(x)$. Note that T_j has j roots in the interval $(-1, 1)$ and that all the polynomials are bounded between ± 1.

It is not too difficult to combine equations (5.8.1), (5.8.4), and (5.8.6) to prove the following theorem: If $f(x)$ is an arbitrary function in the interval $[-1, 1]$, and if N coefficients c_j, $j = 0, \ldots, N-1$, are defined by

$$\begin{aligned} c_j &= \frac{2}{N} \sum_{k=0}^{N-1} f(x_k) T_j(x_k) \\ &= \frac{2}{N} \sum_{k=0}^{N-1} f\left[\cos\left(\frac{\pi(k+\frac{1}{2})}{N}\right)\right] \cos\left(\frac{\pi j(k+\frac{1}{2})}{N}\right) \end{aligned} \tag{5.8.7}$$

then the approximation formula

$$f(x) \approx \left[\sum_{k=0}^{N-1} c_k T_k(x)\right] - \frac{1}{2} c_0 \tag{5.8.8}$$

is *exact* for x equal to all of the N zeros of $T_N(x)$.

For a fixed N, equation (5.8.8) is a polynomial in x that approximates the function $f(x)$ in the interval $[-1, 1]$ (where all the zeros of $T_N(x)$ are located). Why is this particular approximating polynomial better than any other one, exact on some other set of N points? The answer is *not* that (5.8.8) is necessarily more accurate

than some other approximating polynomial of the same order N (for some specified definition of "accurate"), but rather that (5.8.8) can be truncated to a polynomial of *lower* degree $m \ll N$ in a very graceful way, one that *does* yield the "most accurate" approximation of degree m (in a sense that can be made precise). Suppose N is so large that (5.8.8) is virtually a perfect approximation of $f(x)$. Now consider the truncated approximation

$$f(x) \approx \left[\sum_{k=0}^{m-1} c_k T_k(x) \right] - \frac{1}{2} c_0 \tag{5.8.9}$$

with the same c_j's, computed from (5.8.7). Since the $T_k(x)$'s are all bounded between ± 1, the difference between (5.8.9) and (5.8.8) can be no larger than the sum of the neglected c_k's ($k = m, \ldots, N-1$). In fact, if the c_k's are rapidly decreasing (which is the typical case), then the error is dominated by $c_m T_m(x)$, an oscillatory function with $m+1$ equal extrema distributed smoothly over the interval $[-1, 1]$. This smooth spreading out of the error is a very important property: The Chebyshev approximation (5.8.9) is very nearly the same polynomial as that holy grail of approximating polynomials the *minimax polynomial*, which (among all polynomials of the same degree) has the smallest maximum deviation from the true function $f(x)$. The minimax polynomial is very difficult to find; the Chebyshev approximating polynomial is almost identical and is very easy to compute!

So, given some (perhaps difficult) means of computing the function $f(x)$, we now need algorithms for implementing (5.8.7) and (after inspection of the resulting c_k's and choice of a truncating value m) evaluating (5.8.9). The latter equation then becomes an easy way of computing $f(x)$ for all subsequent time.

The first of these tasks is straightforward. A generalization of equation (5.8.7) that is here implemented is to allow the range of approximation to be between two arbitrary limits a and b, instead of just -1 to 1. This is effected by a change of variable

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \tag{5.8.10}$$

and by the approximation of $f(x)$ by a Chebyshev polynomial in y.

It will be convenient for us to group a number of functions related to Chebyshev polynomials into a single object, even though discussion of their specifics is spread out over §5.8 – §5.11:

chebyshev.h

```
struct Chebyshev {
Object for Chebyshev approximation and related methods.
    Int n,m;                        Number of total, and truncated, coefficients.
    VecDoub c;
    Doub a,b;                       Approximation interval.

    Chebyshev(Doub func(Doub), Doub aa, Doub bb, Int nn);
    Constructor. Approximate the function func in the interval [aa,bb] with nn terms.
    Chebyshev(VecDoub &cc, Doub aa, Doub bb)
        : n(cc.size()), m(n), c(cc), a(aa), b(bb) {}
    Constructor from previously computed coefficients.
    Int setm(Doub thresh) {while (m>1 && abs(c[m-1])<thresh) m--; return m;}
    Set m, the number of coefficients after truncating to an error level thresh, and return the
    value set.
```

```
    Doub eval(Doub x, Int m);
    inline Doub operator() (Doub x) {return eval(x,m);}
    Return a value for the Chebyshev fit, either using the stored m or else overriding it.

    Chebyshev derivative();          See §5.9.
    Chebyshev integral();

    VecDoub polycofs(Int m);         See §5.10.
    inline VecDoub polycofs() {return polycofs(m);}
    Chebyshev(VecDoub &pc);          See §5.11.

};
```

The first constructor, the one with an arbitrary function `func` as its first argument, calculates and saves `nn` Chebyshev coefficients that approximate `func` in the range `aa` to `bb`. (You can ignore for now the second constructor, which simply makes a `Chebyshev` object from already-calculated data.) Let us also note the method `setm`, which provides a quick way to truncate the Chebyshev series by (in effect) deleting, from the right, all coefficients smaller in magnitude than some threshold `thresh`.

chebyshev.h

```
Chebyshev::Chebyshev(Doub func(Doub), Doub aa, Doub bb, Int nn=50)
    : n(nn), m(nn), c(n), a(aa), b(bb)
```

Chebyshev fit: Given a function `func`, lower and upper limits of the interval [a,b], compute and save `nn` coefficients of the Chebyshev approximation such that $\texttt{func}(x) \approx [\sum_{k=0}^{\texttt{nn}-1} c_k T_k(y)] - c_0/2$, where y and x are related by (5.8.10). This routine is intended to be called with moderately large `n` (e.g., 30 or 50), the array of `c`'s subsequently to be truncated at the smaller value m such that c_m and subsequent elements are negligible.

```
{
    const Doub pi=3.141592653589793;
    Int k,j;
    Doub fac,bpa,bma,y,sum;
    VecDoub f(n);
    bma=0.5*(b-a);
    bpa=0.5*(b+a);
    for (k=0;k<n;k++) {                  We evaluate the function at the n points required
        y=cos(pi*(k+0.5)/n);                 by (5.8.7).
        f[k]=func(y*bma+bpa);
    }
    fac=2.0/n;
    for (j=0;j<n;j++) {                  Now evaluate (5.8.7).
        sum=0.0;
        for (k=0;k<n;k++)
            sum += f[k]*cos(pi*j*(k+0.5)/n);
        c[j]=fac*sum;
    }
}
```

If you find that the constructor's execution time is dominated by the calculation of N^2 cosines, rather than by the N evaluations of your function, then you should look ahead to §12.3, especially equation (12.4.16), which shows how fast cosine transform methods can be used to evaluate equation (5.8.7).

Now that we have the Chebyshev coefficients, how do we evaluate the approximation? One could use the recurrence relation of equation (5.8.2) to generate values for $T_k(x)$ from $T_0 = 1, T_1 = x$, while also accumulating the sum of (5.8.9). It is better to use Clenshaw's recurrence formula (§5.4), effecting the two processes simultaneously. Applied to the Chebyshev series (5.8.9), the recurrence is

$$\begin{aligned} d_{m+1} &\equiv d_m \equiv 0 \\ d_j &= 2x d_{j+1} - d_{j+2} + c_j \qquad j = m-1, m-2, \ldots, 1 \\ f(x) &\equiv d_0 = x d_1 - d_2 + \tfrac{1}{2} c_0 \end{aligned} \tag{5.8.11}$$

chebyshev.h

```
Doub Chebyshev::eval(Doub x, Int m)
```

Chebyshev evaluation: The Chebyshev polynomial $\sum_{k=0}^{m-1} c_k T_k(y) - c_0/2$ is evaluated at a point $y = [x - (b+a)/2]/[(b-a)/2]$, and the result is returned as the function value.

```
{
    Doub d=0.0,dd=0.0,sv,y,y2;
    Int j;
    if ((x-a)*(x-b) > 0.0) throw("x not in range in Chebyshev::eval");
    y2=2.0*(y=(2.0*x-a-b)/(b-a));          Change of variable.
    for (j=m-1;j>0;j--) {                  Clenshaw's recurrence.
        sv=d;
        d=y2*d-dd+c[j];
        dd=sv;
    }
    return y*d-dd+0.5*c[0];                Last step is different.
}
```

The method `eval` has an argument for specifying how many leading coefficients `m` should be used in the evaluation. If you simply want to use a stored value of `m` that was set by a previous call to `setm` (or, by hand, by you), then you can use the `Chebyshev` object as a functor. For example,

```
Chebyshev approxfunc(func,0.,1.,50);
approxfunc.setm(1.e-8);
...
y = approxfunc(x);
```

If we are approximating an *even* function on the interval $[-1, 1]$, its expansion will involve only even Chebyshev polynomials. It is wasteful to construct a `Chebyshev` object with all the odd coefficients zero [2]. Instead, using the half-angle identity for the cosine in equation (5.8.1), we get the relation

$$T_{2n}(x) = T_n(2x^2 - 1) \tag{5.8.12}$$

Thus we can construct a more efficient `Chebyshev` object for even functions simply by replacing the function's argument x by $2x^2 - 1$, and likewise when we evaluate the Chebyshev approximation.

An odd function will have an expansion involving only odd Chebyshev polynomials. It is best to rewrite it as an expansion for the function $f(x)/x$, which involves only even Chebyshev polynomials. This has the added benefit of giving accurate values for $f(x)/x$ near $x = 0$. Don't try to construct the series by evaluating $f(x)/x$ numerically, however. Rather, the coefficients c'_n for $f(x)/x$ can be found from those for $f(x)$ by recurrence:

$$\begin{aligned} c'_{N+1} &= 0 \\ c'_{n-1} &= 2c_n - c'_{n+1}, \qquad n = N-1, N-3, \ldots \end{aligned} \tag{5.8.13}$$

Equation (5.8.13) follows from the recurrence relation in equation (5.8.2).

If you insist on evaluating an odd Chebyshev series, the efficient way is to once again to replace x by $y = 2x^2 - 1$ as the argument of your function. Now, however,

you must also change the last formula in equation (5.8.11) to be

$$f(x) = x[(2y-1)d_1 - d_2 + c_0] \tag{5.8.14}$$

and change the corresponding line in `eval`.

5.8.1 Chebyshev and Exponential Convergence

Since first mentioning *truncation error* in §1.1, we have seen many examples of algorithms with an adjustable order, say M, such that the truncation error decreases as the Mth power of something. Examples include most of the interpolation methods in Chapter 3 and most of the quadrature methods in Chapter 4. In these examples there is also another parameter, N, which is the number of points at which a function will be evaluated.

We have many times warned that "higher order does not necessarily give higher accuracy." That remains good advice when N is held fixed while M is increased. However, a recently emerging theme in many areas of scientific computation is the use of methods that allow, in very special cases, M and N to be increased *together*, with the result that errors not only do decrease with higher order, but decrease exponentially!

The common thread in almost all of these relatively new methods is the remarkable fact that *infinitely smooth* functions become *exponentially* well determined by N sample points as N is increased. Thus, mere power-law convergence may be just a consequence of either (i) functions that are not smooth enough, or (ii) endpoint effects.

We already saw several examples of this in Chapter 4. In §4.1 we pointed out that high-order quadrature rules can have interior weights of unity, just like the trapezoidal rule; all of the "high-orderness" is obtained by a proper treatment near the boundaries. In §4.5 we further saw that variable transformations that push the boundaries off to infinity produce rapidly converging quadrature algorithms. In §4.5.1 we in fact proved exponential convergence, as a consequence of the Euler-Maclaurin formula. Then in §4.6 we remarked on the fact that the convergence of Gaussian quadratures could be exponentially rapid (an example, in the language above, of increasing M and N simultaneously).

Chebyshev approximation can be exponentially convergent for a different (though related) reason: Smooth *periodic* functions avoid endpoint effects by not having endpoints at all! Chebyshev approximation can be viewed as mapping the x interval $[-1, 1]$ onto the angular interval $[0, \pi]$ (cf. equations 5.8.4 and 5.8.5) in such a way that any infinitely smooth function on the interval $[-1, 1]$ becomes an infinitely smooth, even, periodic function on $[0, 2\pi]$. Figure 5.8.2 shows the idea geometrically. By projecting the abscissas onto a semicircle, a half-period is produced. The other half-period is obtained by reflection, or could be imagined as the result of projecting the function onto an identical lower semicircle. The zeros of the Chebyshev polynomial, or nodes of a Chebyshev approximation, are equally spaced on the circle, where the Chebyshev polynomial itself is a cosine function (cf. equation 5.8.1). This illustrates the close connection between Chebyshev approximation and periodic functions on the circle; in Chapter 12, we will apply the discrete Fourier transform to such functions in an almost equivalent way (§12.4.2).

The reason that Chebyshev works so well (and also why Gaussian quadratures work so well) is thus seen to be intimately related to the special way that the the

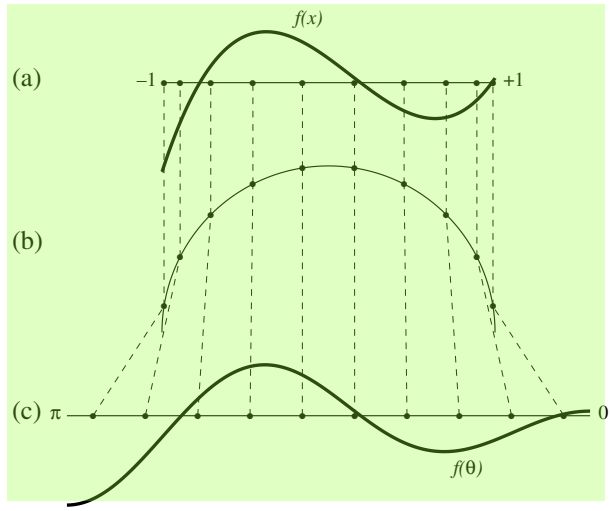

Figure 5.8.2. Geometrical construction showing how Chebyshev approximation is related to periodic functions. A smooth function on the interval is plotted in (a). In (b), the abscissas are mapped to a semicircle. In (c), the semicircle is unrolled. Because of the semicircle's vertical tangents, the function is now nearly constant at the endpoints. In fact, if reflected into the interval $[\pi, 2\pi]$, it is a smooth, even, periodic function on $[0, 2\pi]$.

sample points are bunched up near the endpoints of the interval. Any function that is bounded on the interval will have a convergent Chebyshev approximation as $N \to \infty$, even if there are nearby poles in the complex plane. For functions that are not infinitely smooth, the actual rate of convergence depends on the smoothness of the function: the more deriviatives that are bounded, the greater the convergence rate. For the special case of a C^∞ function, the convergence is exponential. In §3.0, in connection with polynomial interpolation, we mentioned the other side of the coin: equally spaced samples on the interval are about the *worst* possible geometry and often lead to ill-conditioned problems.

Use of the sampling theorem (§4.5, §6.9, §12.1, §13.11) is often closely associated with exponentially convergent methods. We will return to many of the concepts of exponentially convergent methods when we discuss spectral methods for partial differential equations in §20.7.

CITED REFERENCES AND FURTHER READING:

Arfken, G. 1970, *Mathematical Methods for Physicists*, 2nd ed. (New York: Academic Press), p. 631.[1]

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H.M. Stationery Office).[2]

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 8.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §4.4.1, p. 104.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.2, p. 334.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §1.10, p. 39.

5.9 Derivatives or Integrals of a Chebyshev-Approximated Function

If you have obtained the Chebyshev coefficients that approximate a function in a certain range (e.g., from `chebft` in §5.8), then it is a simple matter to transform them to Chebyshev coefficients corresponding to the derivative or integral of the function. Having done this, you can evaluate the derivative or integral just as if it were a function that you had Chebyshev-fitted *ab initio*.

The relevant formulas are these: If c_i, $i = 0, \ldots, m-1$ are the coefficients that approximate a function f in equation (5.8.9), C_i are the coefficients that approximate the indefinite integral of f, and c'_i are the coefficients that approximate the derivative of f, then

$$C_i = \frac{c_{i-1} - c_{i+1}}{2i} \qquad (i > 0) \tag{5.9.1}$$

$$c'_{i-1} = c'_{i+1} + 2i\,c_i \qquad (i = m-1, m-2, \ldots, 1) \tag{5.9.2}$$

Equation (5.9.1) is augmented by an arbitrary choice of C_0, corresponding to an arbitrary constant of integration. Equation (5.9.2), which is a recurrence, is started with the values $c'_m = c'_{m-1} = 0$, corresponding to no information about the $m+1$st Chebyshev coefficient of the original function f.

Here are routines for implementing equations (5.9.1) and (5.9.2). Each returns a new `Chebyshev` object on which you can `setm`, call `eval`, or use directly as a functor.

chebyshev.h

`Chebyshev Chebyshev::derivative()`

Return a new `Chebyshev` object that approximates the derivative of the existing function over the same range [a,b].

```
{
    Int j;
    Doub con;
    VecDoub cder(n);
    cder[n-1]=0.0;                          n-1 and n-2 are special cases.
    cder[n-2]=2*(n-1)*c[n-1];
    for (j=n-2;j>0;j--)                     Equation (5.9.2).
        cder[j-1]=cder[j+1]+2*j*c[j];
    con=2.0/(b-a);
    for (j=0;j<n;j++) cder[j] *= con;       Normalize to the interval b-a.
    return Chebyshev(cder,a,b);
}
```

chebyshev.h

`Chebyshev Chebyshev::integral()`

Return a new `Chebyshev` object that approximates the indefinite integral of the existing function over the same range [a,b]. The constant of integration is set so that the integral vanishes at a.

```
{
    Int j;
    Doub sum=0.0,fac=1.0,con;
    VecDoub cint(n);
    con=0.25*(b-a);                         Factor that normalizes to the interval b-a.
    for (j=1;j<n-1;j++) {
        cint[j]=con*(c[j-1]-c[j+1])/j;      Equation (5.9.1).
        sum += fac*cint[j];                 Accumulates the constant of integration.
        fac = -fac;                         Will equal ±1.
    }
    cint[n-1]=con*c[n-2]/(n-1);             Special case of (5.9.1) for n-1.
```

```
    sum += fac*cint[n-1];
    cint[0]=2.0*sum;
    return Chebyshev(cint,a,b);
}
```

Set the constant of integration.

5.9.1 Clenshaw-Curtis Quadrature

Since a smooth function's Chebyshev coefficients c_i decrease rapidly, generally exponentially, equation (5.9.1) is often quite efficient as the basis for a quadrature scheme. As described above, the `Chebyshev` object can be used to compute the integral $\int_a^x f(x)dx$ when many different values of x in the range $a \le x \le b$ are needed. If only the single definite integral $\int_a^b f(x)dx$ is required, then instead use the simpler formula, derived from equation (5.9.1),

$$\int_a^b f(x)dx = (b-a)\left[\frac{1}{2}c_0 - \frac{1}{3}c_2 - \frac{1}{15}c_4 - \cdots - \frac{1}{(2k+1)(2k-1)}c_{2k} - \cdots\right] \tag{5.9.3}$$

where the c_i's are as returned by `chebft`. The series can be truncated when c_{2k} becomes negligible, and the first neglected term gives an error estimate.

This scheme is known as *Clenshaw-Curtis quadrature* [1]. It is often combined with an adaptive choice of N, the number of Chebyshev coefficients calculated via equation (5.8.7), which is also the number of function evaluations of $f(x)$. If a modest choice of N does not give a sufficiently small c_{2k} in equation (5.9.3), then a larger value is tried. In this adaptive case, it is even better to replace equation (5.8.7) by the so-called "trapezoidal" or Gauss-Lobatto (§4.6) variant,

$$c_j = \frac{2}{N}\sum_{k=0}^{N}{}'' f\left[\cos\left(\frac{\pi k}{N}\right)\right]\cos\left(\frac{\pi jk}{N}\right) \qquad j = 0,\ldots,N-1 \tag{5.9.4}$$

where (N.B.!) the two primes signify that the first and last terms in the sum are to be multiplied by 1/2. If N is doubled in equation (5.9.4), then half of the new function evaluation points are identical to the old ones, allowing the previous function evaluations to be reused. This feature, plus the analytic weights and abscissas (cosine functions in 5.9.4), often give Clenshaw-Curtis quadrature an edge over high-order adaptive Gaussian quadrature (cf. §4.6.4), which the method otherwise resembles.

If your problem forces you to large values of N, you should be aware that equation (5.9.4) can be evaluated rapidly, and simultaneously for all the values of j, by a fast cosine transform. (See §12.3, especially equation 12.4.11. We already remarked that the nontrapezoidal form (5.8.7) can also be done by fast cosine methods, cf. equation 12.4.16.)

CITED REFERENCES AND FURTHER READING:

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), pp. 78–79.

Clenshaw, C.W., and Curtis, A.R. 1960, "A Method for Numerical Integration on an Automatic Computer," *Numerische Mathematik*, vol. 2, pp. 197–205.[1]

5.10 Polynomial Approximation from Chebyshev Coefficients

You may well ask after reading the preceding two sections: Must I store and evaluate my Chebyshev approximation as an array of Chebyshev coefficients for a transformed variable y?

Can't I convert the c_k's into actual polynomial coefficients in the original variable x and have an approximation of the following form?

$$f(x) \approx \sum_{k=0}^{m-1} g_k x^k, \qquad a \le x \le b \tag{5.10.1}$$

Yes, you can do this (and we will give you the algorithm to do it), but we caution you against it: Evaluating equation (5.10.1), where the coefficient g's reflect an underlying Chebyshev approximation, usually requires more significant figures than evaluation of the Chebyshev sum directly (as by `eval`). This is because the Chebyshev polynomials themselves exhibit a rather delicate cancellation: The leading coefficient of $T_n(x)$ for example, is 2^{n-1}; other coefficients of $T_n(x)$ are even bigger; yet they all manage to combine into a polynomial that lies between ± 1. *Only* when m is no larger than 7 or 8 should you contemplate writing a Chebyshev fit as a direct polynomial, and even in those cases you should be willing to tolerate two or so significant figures less accuracy than the roundoff limit of your machine.

You get the g's in equation (5.10.1) in two steps. First, use the member function `polycofs` in `Chebyshev` to output a set of polynomial coefficients equivalent to the stored c_k's (that is, with the range $[a, b]$ scaled to $[-1, 1]$). Second, use the routine `pcshft` to transform the coefficients so as to map the range back to $[a, b]$. The two required routines are listed here:

chebyshev.h

```
VecDoub Chebyshev::polycofs(Int m)
Polynomial coefficients from a Chebyshev fit. Given a coefficient array c[0..n-1], this routine
returns a coefficient array d[0..n-1] such that \sum_{k=0}^{n-1} d_k y^k = \sum_{k=0}^{n-1} c_k T_k(y) - c_0/2. The
method is Clenshaw's recurrence (5.8.11), but now applied algebraically rather than arithmeti-
cally.
{
    Int k,j;
    Doub sv;
    VecDoub d(m),dd(m);
    for (j=0;j<m;j++) d[j]=dd[j]=0.0;
    d[0]=c[m-1];
    for (j=m-2;j>0;j--) {
        for (k=m-j;k>0;k--) {
            sv=d[k];
            d[k]=2.0*d[k-1]-dd[k];
            dd[k]=sv;
        }
        sv=d[0];
        d[0] = -dd[0]+c[j];
        dd[0]=sv;
    }
    for (j=m-1;j>0;j--) d[j]=d[j-1]-dd[j];
    d[0] = -dd[0]+0.5*c[0];
    return d;
}
```

pcshft.h

```
void pcshft(Doub a, Doub b, VecDoub_IO &d)
Polynomial coefficient shift. Given a coefficient array d[0..n-1], this routine generates a co-
efficient array g[0..n-1] such that \sum_{k=0}^{n-1} d_k y^k = \sum_{k=0}^{n-1} g_k x^k where x and y are related
by (5.8.10), i.e., the interval -1 < y < 1 is mapped to the interval a < x < b. The array g is
returned in d.
{
    Int k,j,n=d.size();
    Doub cnst=2.0/(b-a), fac=cnst;
    for (j=1;j<n;j++) {                    First we rescale by the factor const...
        d[j] *= fac;
        fac *= cnst;
    }
    cnst=0.5*(a+b);                        ...which is then redefined as the desired shift.
```

```
    for (j=0;j<=n-2;j++)
        for (k=n-2;k>=j;k--)
            d[k] -= cnst*d[k+1];
}
```

We accomplish the shift by synthetic division, a miracle of high-school algebra.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), pp. 59, 182–183 [synthetic division].

5.11 Economization of Power Series

One particular application of Chebyshev methods, the *economization of power series*, is an occasionally useful technique, with a flavor of getting something for nothing.

Suppose that you know how to compute a function by the use of a convergent power series, for example,

$$f(x) \equiv \frac{1}{2} - \frac{x}{4} + \frac{x^2}{8} - \frac{x^3}{16} + \cdots \tag{5.11.1}$$

(This function is actually just $1/(x+2)$, but pretend you don't know that.) You might be doing a problem that requires evaluating the series many times in some particular interval, say $[0, 1]$. Everything is fine, except that the series requires a large number of terms before its error (approximated by the first neglected term, say) is tolerable. In our example, with $x = 1$, it takes about 30 terms before the first neglected term is $< 10^{-9}$.

Notice that because of the large exponent in x^{30}, the error is *much smaller* than 10^{-9} everywhere in the interval except at the very largest values of x. This is the feature that allows "economization": If we are willing to let the error elsewhere in the interval rise to about the same value that the first neglected term has at the extreme end of the interval, then we can replace the 30-term series by one that is significantly shorter.

Here are the steps for doing this:

1. Compute enough coefficients of the power series to get accurate function values everywhere in the range of interest.
2. Change variables from x to y, as in equation (5.8.10), to map the x interval into $-1 \le y \le 1$.
3. Find the Chebyshev series (like equation 5.8.8) that exactly equals your truncated power series.
4. Truncate this Chebyshev series to a smaller number of terms, using the coefficient of the first neglected Chebyshev polynomial as an estimate of the error.
5. Convert back to a polynomial in y.
6. Change variables back to x.

We already have tools for all of the steps, except for steps 2 and 3. Step 2 is exactly the inverse of the routine `pcshft` (§5.10), which mapped a polynomial from y (in the interval $[-1, 1]$) to x (in the interval $[a, b]$). But since equation (5.8.10) is a linear relation between x and y, one can also use `pcshft` for the inverse. The inverse of

`pcshft(a,b,d,n)`

turns out to be (you can check this)

```
void ipcshft(Doub a, Doub b, VecDoub_IO &d) {
    pcshft(-(2.+b+a)/(b-a),(2.-b-a)/(b-a),d);
}
```

pcshft.h

Step 3 requires a new `Chebyshev` constructor, one that computes Chebyshev coefficients from a vector of polynomial coefficients. The following code accomplishes this. The algorithm is based on constructing the polynomial by the technique of §5.3 starting with the highest coefficient `d[n-1]` and using the recurrence of equation (5.8.2) written in the form

$$\begin{aligned} xT_0 &= T_1 \\ xT_n &= \tfrac{1}{2}(T_{n+1} + T_{n-1}), \quad n \geq 1. \end{aligned} \tag{5.11.2}$$

The only subtlety is to multiply the coefficient of T_0 by 2 since it gets used with a factor 1/2 in equation (5.8.8).

chebyshev.h

```
Chebyshev::Chebyshev(VecDoub &d)
    : n(d.size()), m(n), c(n), a(-1.), b(1.)
```

Inverse of routine `polycofs` in `Chebyshev`: Given an array of polynomial coefficients `d[0..n-1]`, construct an equivalent `Chebyshev` object.

```
{
    c[n-1]=d[n-1];
    c[n-2]=2.0*d[n-2];
    for (Int j=n-3;j>=0;j--) {
        c[j]=2.0*d[j]+c[j+2];
        for (Int i=j+1;i<n-2;i++) {
            c[i] = (c[i]+c[i+2])/2;
        }
        c[n-2] /= 2;
        c[n-1] /= 2;
    }
}
```

Putting them all together, steps 2 through 6 will look something like this (starting with a vector `powser` of power series coefficients):

```
ipcshft(a,b,powser);
Chebyshev cpowser(powser);
cpowser.setm(1.e-9);
VecDoub d=cpowser.polycofs();
pcshft(a,b,d);
```

In our example, by the way, the number of terms required for 10^{-9} accuracy is reduced from 30 to 9. Replacing a 30-term polynomial with a 9-term polynomial without any loss of accuracy — that does seem to be getting something for nothing. Is there some magic in this technique? Not really. The 30-term polynomial defined a function $f(x)$. Equivalent to economizing the series, we could instead have evaluated $f(x)$ at enough points to construct its Chebyshev approximation in the interval of interest, by the methods of §5.8. We would have obtained just the same lower-order polynomial. The principal lesson is that the rate of convergence of Chebyshev coefficients has nothing to do with the rate of convergence of power series coefficients; and it is the *former* that dictates the number of terms needed in a polynomial approximation. A function might have a *divergent* power series in some region of interest, but if the function itself is well-behaved, it will have perfectly good polynomial approximations. These can be found by the methods of §5.8, but *not* by economization of series. There is slightly less to economization of series than meets the eye.

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 12.

5.12 Padé Approximants

A *Padé approximant*, so called, is that rational function (of a specified order) whose power series expansion agrees with a given power series to the highest possible order. If the rational function is

$$R(x) \equiv \frac{\sum_{k=0}^{M} a_k x^k}{1 + \sum_{k=1}^{N} b_k x^k} \tag{5.12.1}$$

then $R(x)$ is said to be a Padé approximant to the series

$$f(x) \equiv \sum_{k=0}^{\infty} c_k x^k \tag{5.12.2}$$

if

$$R(0) = f(0) \tag{5.12.3}$$

and also

$$\left.\frac{d^k}{dx^k} R(x)\right|_{x=0} = \left.\frac{d^k}{dx^k} f(x)\right|_{x=0}, \qquad k = 1, 2, \ldots, M+N \tag{5.12.4}$$

Equations (5.12.3) and (5.12.4) furnish $M + N + 1$ equations for the unknowns $a_0, \ldots, a_M$ and $b_1, \ldots, b_N$. The easiest way to see what these equations are is to equate (5.12.1) and (5.12.2), multiply both by the denominator of equation (5.12.1), and equate all powers of x that have either a's or b's in their coefficients. If we consider only the special case of a diagonal rational approximation, $M = N$ (cf. §3.4), then we have $a_0 = c_0$, with the remaining a's and b's satisfying

$$\sum_{m=1}^{N} b_m c_{N-m+k} = -c_{N+k}, \qquad k = 1, \ldots, N \tag{5.12.5}$$

$$\sum_{m=0}^{k} b_m c_{k-m} = a_k, \qquad k = 1, \ldots, N \tag{5.12.6}$$

(note, in equation 5.12.1, that $b_0 = 1$). To solve these, start with equations (5.12.5), which are a set of linear equations for all the unknown b's. Although the set is in the form of a Toeplitz matrix (compare equation 2.8.8), experience shows that the equations are frequently close to singular, so that one should not solve them by the methods of §2.8, but rather by full LU decomposition. Additionally, it is a good idea to refine the solution by iterative improvement (method `mprove` in §2.5) [1].

Once the b's are known, then equation (5.12.6) gives an explicit formula for the unknown a's, completing the solution.

Padé approximants are typically used when there is some unknown underlying function $f(x)$. We suppose that you are able somehow to compute, perhaps by laborious analytic expansions, the values of $f(x)$ and a few of its derivatives at $x = 0$: $f(0)$, $f'(0)$, $f''(0)$, and so on. These are of course the first few coefficients in the power series expansion of $f(x)$; but they are not necessarily getting small, and you have no idea where (or whether) the power series is convergent.

By contrast with techniques like Chebyshev approximation (§5.8) or economization of power series (§5.11) that only condense the information that you already know about a function, Padé approximants can give you genuinely new information about your function's values.

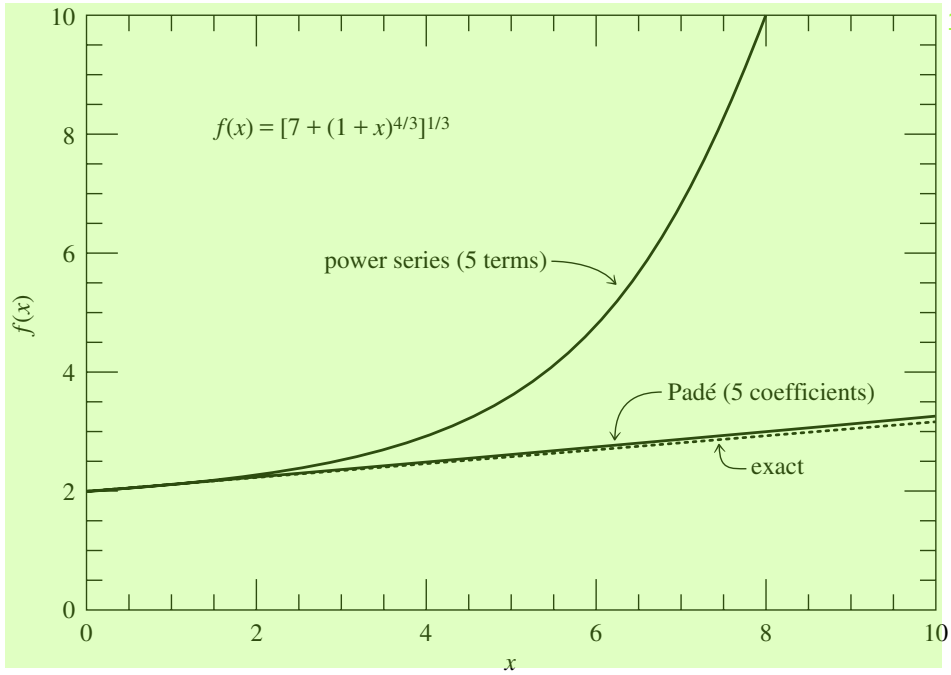

Figure 5.12.1. The five-term power series expansion and the derived five-coefficient Padé approximant for a sample function $f(x)$. The full power series converges only for $x < 1$. Note that the Padé approximant maintains accuracy far outside the radius of convergence of the series.

It is sometimes quite mysterious how well this can work. (Like other mysteries in mathematics, it relates to *analyticity*.) An example will illustrate.

Imagine that, by extraordinary labors, you have ground out the first five terms in the power series expansion of an unknown function $f(x)$.

$$f(x) \approx 2 + \frac{1}{9}x + \frac{1}{81}x^2 - \frac{49}{8748}x^3 + \frac{175}{78732}x^4 + \cdots \tag{5.12.7}$$

(It is not really necessary that you know the coefficients in exact rational form — numerical values are just as good. We here write them as rationals to give you the impression that they derive from some side analytic calculation.) Equation (5.12.7) is plotted as the curve labeled "power series" in Figure 5.12.1. One sees that for $x \gtrsim 4$ it is dominated by its largest, quartic, term.

We now take the five coefficients in equation (5.12.7) and run them through the routine `pade` listed below. It returns five rational coefficients, three a's and two b's, for use in equation (5.12.1) with $M = N = 2$. The curve in the figure labeled "Padé" plots the resulting rational function. Note that both solid curves derive from the *same* five original coefficient values.

To evaluate the results, we need *Deus ex machina* (a useful fellow, when he is available) to tell us that equation (5.12.7) is in fact the power series expansion of the function

$$f(x) = [7 + (1 + x)^{4/3}]^{1/3} \tag{5.12.8}$$

which is plotted as the dotted curve in the figure. This function has a branch point at $x = -1$, so its power series is convergent only in the range $-1 < x < 1$. In most of the range shown in the figure, the series is divergent, and the value of its truncation to five terms is rather meaningless. Nevertheless, those five terms, converted to a Padé approximant, give a remarkably good representation of the function up to at least $x \sim 10$.

Why does this work? Are there not other functions with the same first five terms in their power series but completely different behavior in the range (say) $2 < x < 10$? Indeed there are. Padé approximation has the uncanny knack of picking the function *you had in mind* from among all the possibilities. *Except when it doesn't!* That is the downside of Padé approximation: It is uncontrolled. There is, in general, no way to tell how accurate it is, or how far out in x it can usefully be extended. It is a powerful, but in the end still mysterious, technique.

Here is the routine that returns a `Ratfn` rational function object that is the Padé approximant to a set of power series coefficients that you provide. Note that the routine is specialized to the case $M = N$. You can then use the `Ratfn` object directly as a functor, or else read out its coefficients by hand (§5.1).

`Ratfn pade(VecDoub_I &cof)` pade.h

Given `cof[0..2*n]`, the leading terms in the power series expansion of a function, solve the linear Padé equations to return a `Ratfn` object that embodies a diagonal rational function approximation to the same function.

```
{
    const Doub BIG=1.0e99;
    Int j,k,n=(cof.size()-1)/2;
    Doub sum;
    MatDoub q(n,n),qlu(n,n);
    VecInt indx(n);
    VecDoub x(n),y(n),num(n+1),denom(n+1);
    for (j=0;j<n;j++) {                    Set up matrix for solving.
        y[j]=cof[n+j+1];
        for (k=0;k<n;k++) q[j][k]=cof[j-k+n];
    }
    LUdcmp lu(q);                          Solve by LU decomposition and backsubstitu-
    lu.solve(y,x);                             tion, with iterative improvement.
    for (j=0;j<4;j++) lu.mprove(y,x);
    for (k=0;k<n;k++) {                    Calculate the remaining coefficients.
        for (sum=cof[k+1],j=0;j<=k;j++) sum -= x[j]*cof[k-j];
        y[k]=sum;
    }
    num[0] = cof[0];
    denom[0] = 1.;
    for (j=0;j<n;j++) {                    Copy answers to output.
        num[j+1]=y[j];
        denom[j+1] = -x[j];
    }
    return Ratfn(num,denom);
}
```

CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), p. 14.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 2.

Graves-Morris, P.R. 1979, in *Padé Approximation and Its Applications*, Lecture Notes in Mathematics, vol. 765, L. Wuytack, ed. (Berlin: Springer).[1]

5.13 Rational Chebyshev Approximation

In §5.8 and §5.10 we learned how to find good polynomial approximations to a given function $f(x)$ in a given interval $a \le x \le b$. Here, we want to generalize the task to find

good approximations that are rational functions (see §5.1). The reason for doing so is that, for some functions and some intervals, the optimal rational function approximation is able to achieve substantially higher accuracy than the optimal polynomial approximation with the same number of coefficients. This must be weighed against the fact that finding a rational function approximation is not as straightforward as finding a polynomial approximation, which, as we saw, could be done elegantly via Chebyshev polynomials.

Let the desired rational function $R(x)$ have a numerator of degree m and denominator of degree k. Then we have

$$R(x) \equiv \frac{p_0 + p_1 x + \cdots + p_m x^m}{1 + q_1 x + \cdots + q_k x^k} \approx f(x) \qquad \text{for } a \le x \le b \tag{5.13.1}$$

The unknown quantities that we need to find are $p_0, \ldots, p_m$ and $q_1, \ldots, q_k$, that is, $m + k + 1$ quantities in all. Let $r(x)$ denote the deviation of $R(x)$ from $f(x)$, and let r denote its maximum absolute value,

$$r(x) \equiv R(x) - f(x) \qquad r \equiv \max_{a \le x \le b} |r(x)| \tag{5.13.2}$$

The ideal *minimax* solution would be that choice of p's and q's that minimizes r. Obviously there is *some* minimax solution, since r is bounded below by zero. How can we find it, or a reasonable approximation to it?

A first hint is furnished by the following fundamental theorem: If $R(x)$ is nondegenerate (has no common polynomial factors in numerator and denominator), then there is a unique choice of p's and q's that minimizes r, and for this choice, $r(x)$ has $m + k + 2$ extrema in $a \le x \le b$, *all of magnitude r and with alternating sign*. (We have omitted some technical assumptions in this theorem. See Ralston [1] for a precise statement.) We thus learn that the situation with rational functions is quite analogous to that for minimax polynomials: In §5.8 we saw that the error term of an nth-order approximation, with $n + 1$ Chebyshev coefficients, was generally dominated by the first neglected Chebyshev term, namely T_{n+1}, which itself has $n + 2$ extrema of equal magnitude and alternating sign. So, here, the number of rational coefficients, $m + k + 1$, plays the same role as the number of polynomial coefficients, $n + 1$.

A different way to see why $r(x)$ should have $m + k + 2$ extrema is to note that $R(x)$ can be made exactly equal to $f(x)$ at any $m + k + 1$ points x_i. Multiplying equation (5.13.1) by its denominator gives the equations

$$p_0 + p_1 x_i + \cdots + p_m x_i^m = f(x_i)(1 + q_1 x_i + \cdots + q_k x_i^k) \qquad i = 0, 1, \ldots, m + k \tag{5.13.3}$$

This is a set of $m + k + 1$ linear equations for the unknown p's and q's, which can be solved by standard methods (e.g., LU decomposition). If we choose the x_i's to all be in the interval (a, b), then there will generically be an extremum between each chosen x_i and x_{i+1}, plus also extrema where the function goes out of the interval at a and b, for a total of $m + k + 2$ extrema. For arbitrary x_i's, the extrema will not have the same magnitude. The theorem says that, for one particular choice of x_i's, the magnitudes can be beaten down to the identical, minimal, value of r.

Instead of making $f(x_i)$ and $R(x_i)$ equal at the points x_i, one can instead force the residual $r(x_i)$ to any desired values y_i by solving the linear equations

$$p_0 + p_1 x_i + \cdots + p_m x_i^m = [f(x_i) - y_i](1 + q_1 x_i + \cdots + q_k x_i^k) \qquad i = 0, 1, \ldots, m + k \tag{5.13.4}$$

In fact, if the x_i's are chosen to be the extrema (not the zeros) of the minimax solution, then the equations satisfied will be

$$p_0 + p_1 x_i + \cdots + p_m x_i^m = [f(x_i) \pm r](1 + q_1 x_i + \cdots + q_k x_i^k) \qquad i = 0, 1, \ldots, m + k + 1 \tag{5.13.5}$$

where the $\pm$ alternates for the alternating extrema. Notice that equation (5.13.5) is satisfied at $m + k + 2$ extrema, while equation (5.13.4) was satisfied only at $m + k + 1$ arbitrary points. How can this be? The answer is that r in equation (5.13.5) is an additional unknown, so that

the number of both equations and unknowns is $m + k + 2$. True, the set is mildly nonlinear (in r), but in general it is still perfectly soluble by methods that we will develop in Chapter 9.

We thus see that, given only the *locations* of the extrema of the minimax rational function, we can solve for its coefficients and maximum deviation. Additional theorems, leading up to the so-called *Remes algorithms* [1], tell how to converge to these locations by an iterative process. For example, here is a (slightly simplified) statement of *Remes' Second Algorithm*: (1) Find an initial rational function with $m + k + 2$ extrema x_i (not having equal deviation). (2) Solve equation (5.13.5) for new rational coefficients and r. (3) Evaluate the resulting $R(x)$ to find its actual extrema (which will not be the same as the guessed values). (4) Replace each guessed value with the nearest actual extremum of the same sign. (5) Go back to step 2 and iterate to convergence. Under a broad set of assumptions, this method will converge. Ralston [1] fills in the necessary details, including how to find the initial set of x_i's.

Up to this point, our discussion has been textbook standard. We now reveal ourselves as heretics. We don't much like the elegant Remes algorithm. Its two nested iterations (on r in the nonlinear set 5.13.5, and on the new sets of x_i's) are finicky and require a lot of special logic for degenerate cases. Even more heretical, we doubt that compulsive searching for the *exactly best*, equal deviation approximation is worth the effort — except perhaps for those few people in the world whose business it is to find optimal approximations that get built into compilers and microcode.

When we use rational function approximation, the goal is usually much more pragmatic: Inside some inner loop we are evaluating some function a zillion times, and we want to speed up its evaluation. Almost never do we need this function to the last bit of machine accuracy. Suppose (heresy!) we use an approximation whose error has $m + k + 2$ extrema whose deviations differ by a factor of 2. The theorems on which the Remes algorithms are based guarantee that the perfect minimax solution will have extrema somewhere within this factor of 2 range — forcing down the higher extrema will cause the lower ones to rise, until all are equal. So our "sloppy" approximation is in fact within a fraction of a least significant bit of the minimax one.

That is good enough for us, especially when we have available a very robust method for finding the so-called "sloppy" approximation. Such a method is the least-squares solution of overdetermined linear equations by singular value decomposition (§2.6 and §15.4). We proceed as follows: First, solve (in the least-squares sense) equation (5.13.3), not just for $m + k + 1$ values of x_i, but for a significantly larger number of x_i's, spaced approximately like the zeros of a high-order Chebyshev polynomial. This gives an initial guess for $R(x)$. Second, tabulate the resulting deviations, find the mean absolute deviation, call it r, and then solve (again in the least-squares sense) equation (5.13.5) with r fixed and the $\pm$ chosen to be the sign of the observed deviation at each point x_i. Third, repeat the second step a few times.

You can spot some Remes orthodoxy lurking in our algorithm: The equations we solve are trying to bring the deviations not to zero, but rather to plus-or-minus some consistent value. However, we dispense with keeping track of actual extrema, and we solve only linear equations at each stage. One additional trick is to solve a *weighted* least-squares problem, where the weights are chosen to beat down the largest deviations fastest.

Here is a function implementing these ideas. Notice that the only calls to the function `fn` occur in the initial filling of the table `fs`. You could easily modify the code to do this filling outside of the routine. It is not even necessary that your abscissas `xs` be exactly the ones that we use, though the quality of the fit will deteriorate if you do not have several abscissas between each extremum of the (underlying) minimax solution. The function returns a `Ratfn` object that you can subsequently use as a functor, or from which you can extract the stored coefficients.

ratlsq.h

```
Ratfn ratlsq(Doub fn(const Doub), const Doub a, const Doub b, const Int mm,
    const Int kk, Doub &dev)
```

Returns a rational function approximation to the function `fn` in the interval `(a, b)`. Input quantities `mm` and `kk` specify the order of the numerator and denominator, respectively. The maximum absolute deviation of the approximation (insofar as is known) is returned as `dev`.

```
{
    const Int NPFAC=8,MAXIT=5;
    const Doub BIG=1.0e99,PIO2=1.570796326794896619;
```

```
    Int i,it,j,ncof=mm+kk+1,npt=NPFAC*ncof;
    Number of points where function is evaluated, i.e., fineness of the mesh.
    Doub devmax,e,hth,power,sum;
    VecDoub bb(npt),coff(ncof),ee(npt),fs(npt),wt(npt),xs(npt);
    MatDoub u(npt,ncof);
    Ratfn ratbest(coff,mm+1,kk+1);
    dev=BIG;
    for (i=0;i<npt;i++) {                  Fill arrays with mesh abscissas and function values.
        if (i < (npt/2)-1) {               At each end, use formula that minimizes round-
            hth=PIO2*i/(npt-1.0);          off sensitivity.
            xs[i]=a+(b-a)*SQR(sin(hth));
        } else {
            hth=PIO2*(npt-i)/(npt-1.0);
            xs[i]=b-(b-a)*SQR(sin(hth));
        }
        fs[i]=fn(xs[i]);
        wt[i]=1.0;                         In later iterations we will adjust these weights to
        ee[i]=1.0;                             combat the largest deviations.
    }
    e=0.0;
    for (it=0;it<MAXIT;it++) {             Loop over iterations.
        for (i=0;i<npt;i++) {              Set up the "design matrix" for the least-squares
            power=wt[i];                       fit.
            bb[i]=power*(fs[i]+SIGN(e,ee[i]));
            Key idea here: Fit to fn(x)+e where the deviation is positive, to fn(x)−e where
            it is negative. Then e is supposed to become an approximation to the equal-ripple
            deviation.
            for (j=0;j<mm+1;j++) {
                u[i][j]=power;
                power *= xs[i];
            }
            power = -bb[i];
            for (j=mm+1;j<ncof;j++) {
                power *= xs[i];
                u[i][j]=power;
            }
        }
        SVD svd(u);                        Singular value decomposition.
        svd.solve(bb,coff);
        In especially singular or difficult cases, one might here edit the singular values, replacing
        small values by zero in w[0..ncof-1].
        devmax=sum=0.0;
        Ratfn rat(coff,mm+1,kk+1);
        for (j=0;j<npt;j++) {              Tabulate the deviations and revise the weights.
            ee[j]=rat(xs[j])-fs[j];
            wt[j]=abs(ee[j]);              Use weighting to emphasize most deviant points.
            sum += wt[j];
            if (wt[j] > devmax) devmax=wt[j];
        }
        e=sum/npt;                         Update e to be the mean absolute deviation.
        if (devmax <= dev) {               Save only the best coefficient set found.
            ratbest = rat;
            dev=devmax;
        }
        cout << " ratlsq iteration= " << it;
        cout << "  max error= " << setw(10) << devmax << endl;
    }
    return ratbest;
}
```

Figure 5.13.1 shows the discrepancies for the first five iterations of `ratlsq` when it is applied to find the $m = k = 4$ rational fit to the function $f(x) = \cos x/(1 + e^x)$ in the interval $(0, \pi)$. One sees that after the first iteration, the results are virtually as good as the

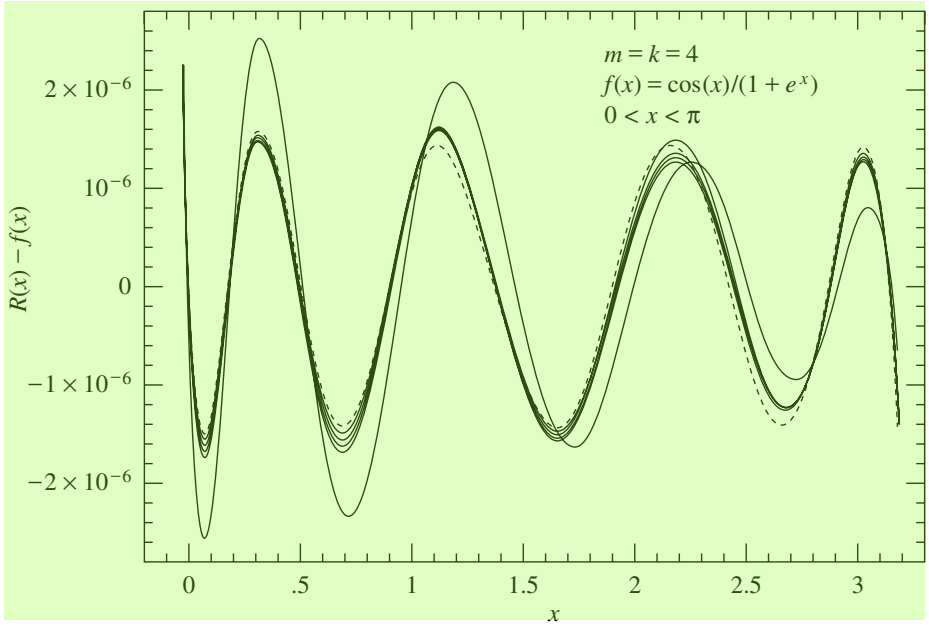

Figure 5.13.1. Solid curves show deviations $r(x)$ for five successive iterations of the routine `ratlsq` for an arbitrary test problem. The algorithm does not converge to exactly the minimax solution (shown as the dotted curve). But, after one iteration, the discrepancy is a small fraction of the last significant bit of accuracy.

minimax solution. The iterations do not converge in the order that the figure suggests. In fact, it is the second iteration that is best (has smallest maximum deviation). The routine `ratlsq` accordingly returns the best of its iterations, not necessarily the last one; there is no advantage in doing more than five iterations.

CITED REFERENCES AND FURTHER READING:

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), Chapter 13.[1]

5.14 Evaluation of Functions by Path Integration

In computer programming, the technique of choice is not necessarily the most efficient, or elegant, or fastest executing one. Instead, it may be the one that is quick to implement, general, and easy to check.

One sometimes needs only a few, or a few thousand, evaluations of a special function, perhaps a complex-valued function of a complex variable, that has many different parameters, or asymptotic regimes, or both. Use of the usual tricks (series, continued fractions, rational function approximations, recurrence relations, and so forth) may result in a patchwork program with tests and branches to different formulas. While such a program may be highly efficient in execution, it is often not the shortest way to the answer from a standing start.

A different technique of considerable generality is direct integration of a function's defining differential equation — an *ab initio* integration for each desired function value — along a path in the complex plane if necessary. While this may at first seem like swatting a fly with a golden brick, it turns out that when you already have the brick, and the fly is asleep right under it, all you have to do is let it fall!

As a specific example, let us consider the complex hypergeometric function ${}_2F_1(a,b,c;z)$, which is defined as the analytic continuation of the so-called hypergeometric series,

$$
\begin{aligned}
{}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \\
+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots
\end{aligned}
\tag{5.14.1}
$$

The series converges only within the unit circle $|z| < 1$ (see [1]), but one's interest in the function is often not confined to this region.

The hypergeometric function ${}_2F_1$ is a solution (in fact *the* solution that is regular at the origin) of the hypergeometric differential equation, which we can write as

$$
z(1-z)F'' = abF - [c-(a+b+1)z]F' \tag{5.14.2}
$$

Here prime denotes d/dz. One can see that the equation has regular singular points at $z = 0, 1$, and ∞. Since the desired solution is regular at $z = 0$, the values 1 and ∞ will in general be branch points. If we want ${}_2F_1$ to be a single-valued function, we must have a branch cut connecting these two points. A conventional position for this cut is along the positive real axis from 1 to ∞, though we may wish to keep open the possibility of altering this choice for some applications.

Our golden brick consists of a collection of routines for the integration of sets of ordinary differential equations, which we will develop in detail later, in Chapter 17. For now, we need only a high-level, "black-box" routine that integrates such a set from initial conditions at one value of a (real) independent variable to final conditions at some other value of the independent variable, while automatically adjusting its internal stepsize to maintain some specified accuracy. That routine is called `Odeint` and, in one particular invocation, it calculates its individual steps with a sophisticated Bulirsch-Stoer technique.

Suppose that we know values for F and its derivative F' at some value z_0, and that we want to find F at some other point z_1 in the complex plane. The straight-line path connecting these two points is parametrized by

$$
z(s) = z_0 + s(z_1 - z_0) \tag{5.14.3}
$$

with s a real parameter. The differential equation (5.14.2) can now be written as a set of two first-order equations,

$$
\begin{aligned}
\frac{dF}{ds} &= (z_1 - z_0)F' \\
\frac{dF'}{ds} &= (z_1 - z_0)\left(\frac{abF - [c-(a+b+1)z]F'}{z(1-z)}\right)
\end{aligned}
\tag{5.14.4}
$$

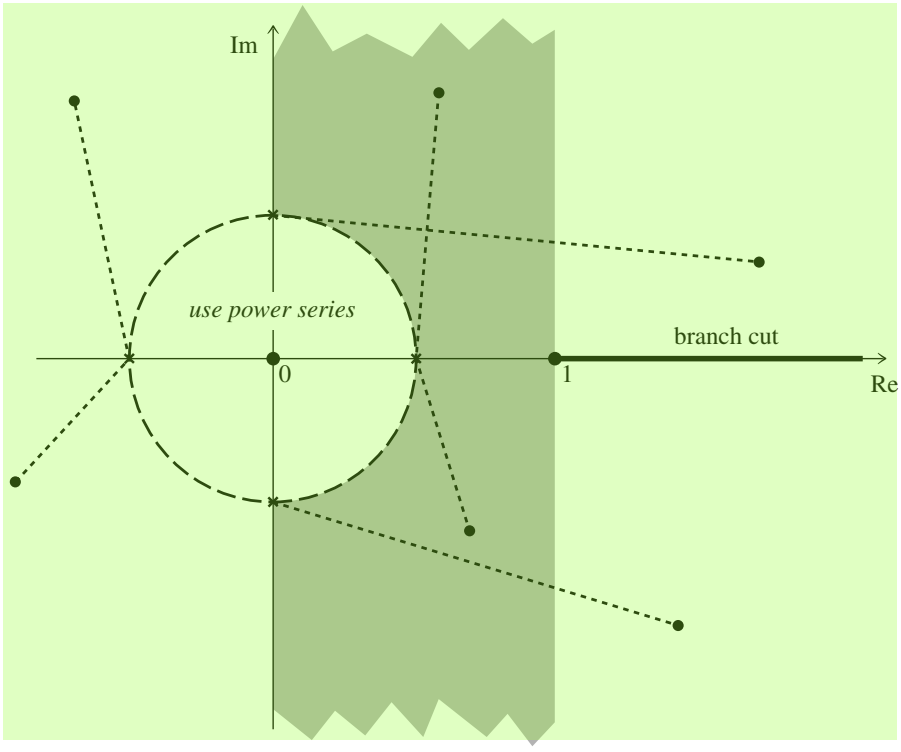

Figure 5.14.1. Complex plane showing the singular points of the hypergeometric function, its branch cut, and some integration paths from the circle $|z| = 1/2$ (where the power series converges rapidly) to other points in the plane.

to be integrated from $s = 0$ to $s = 1$. Here F and F' are to be viewed as two independent complex variables. The fact that prime means d/dz can be ignored; it will emerge as a consequence of the first equation in (5.14.4). Moreover, the real and imaginary parts of equation (5.14.4) define a set of four *real* differential equations, with independent variable s. The complex arithmetic on the right-hand side can be viewed as mere shorthand for how the four components are to be coupled. It is precisely this point of view that gets passed to the routine `Odeint`, since it knows nothing of either complex functions or complex independent variables.

It remains only to decide where to start, and what path to take in the complex plane, to get to an arbitrary point z. This is where consideration of the function's singularities, and the adopted branch cut, enter. Figure 5.14.1 shows the strategy that we adopt. For $|z| \le 1/2$, the series in equation (5.14.1) will in general converge rapidly, and it makes sense to use it directly. Otherwise, we integrate along a straight-line path from one of the starting points $(\pm 1/2, 0)$ or $(0, \pm 1/2)$. The former choices are natural for $0 < \mathrm{Re}(z) < 1$ and $\mathrm{Re}(z) < 0$, respectively. The latter choices are used for $\mathrm{Re}(z) > 1$ above and below the branch cut; the purpose of starting away from the real axis in these cases is to avoid passing too close to the singularity at $z = 1$ (see Figure 5.14.1). The location of the branch cut is *defined* by the fact that our adopted strategy never integrates across the real axis for $\mathrm{Re}(z) > 1$.

An implementation of this algorithm is given in §6.13 as the routine `hypgeo`.

A number of variants on the procedure described thus far are possible and easy to program. If successively called values of z are close together (with identical values of a, b, and c), then you can save the state vector (F, F') and the corresponding value of z on each call, and use these as starting values for the next call. The incremental integration may then take only one or two steps. Avoid integrating across the branch cut unintentionally: The function value will be "correct," but not the one you want.

Alternatively, you may wish to integrate to some position z by a dog-leg path that *does* cross the real axis $\mathrm{Re}(z) > 1$, as a means of *moving* the branch cut. For example, in some cases you might want to integrate from $(0, 1/2)$ to $(3/2, 1/2)$, and go from there to any point with $\mathrm{Re}(z) > 1$ — with either sign of $\mathrm{Im}z$. (If you are, for example, finding roots of a function by an iterative method, you do not want the integration for nearby values to take different paths around a branch point. If it does, your root-finder will see discontinuous function values and will likely not converge correctly!)

In any case, be aware that a loss of numerical accuracy can result if you integrate through a region of large function value on your way to a final answer where the function value is small. (For the hypergeometric function, a particular case of this is when a and b are both large and positive, with c and $x \gtrsim 1$.) In such cases, you'll need to find a better dog-leg path.

The general technique of evaluating a function by integrating its differential equation in the complex plane can also be applied to other special functions. For example, the complex Bessel function, Airy function, Coulomb wave function, and Weber function are all special cases of the *confluent hypergeometric function*, with a differential equation similar to the one used above (see, e.g., [1] §13.6, for a table of special cases). The confluent hypergeometric function has no singularities at finite z: That makes it easy to integrate. However, its essential singularity at infinity means that it can have, along some paths and for some parameters, highly oscillatory or exponentially decreasing behavior: That makes it hard to integrate. Some case-by-case judgment (or experimentation) is therefore required.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`.[1]