previous one. In this process, the program practically "writes itself." For this project, we might begin the refinement process by determining what needs to be passed to the subroutine, and naming those variables appropriately. For example, the subroutine should be given the limits of the bracketed interval: let's name these variables `Left` and `Right`, for example, in the subroutine. (Note, however, that different names, such as `x_initial` and `x_final`, might be more appropriate in the main program.) In one sense, providing reasonable names for variables is simple stuff, easy to implement, and not very important. Not important, that is , until you try to remember (6 months from now) what a poorly named variable, like `x34b`, was supposed to mean!

Give your variables meaningful names, and declare them appropriately.

We also need to provide the subroutine with the name of the function. The function, or at least its name, could be defined within the subroutine, but then the subroutine would need to be changed in order to investigate a different function. One of the goals of writing modular code is to write it *once*, and *only once!* So we'll declare the function as `EXTERNAL` in the main program, and pass its name to the subroutine. Oh, and we want the root passed back to the calling program! Our first refinement of the code might then look like

```
      < Main Program identification>
*
*  Type Declarations
*
       DOUBLE PRECISION x_initial, x_final, Root, FofX
*
       External FofX     ! The function f(x) will be provided
                         !  in a separate subprogram unit.
*
*  Initialize variables
*
       x_initial = 0.d0
       x_final   = 1.57d0  !  A close approximation to pi/2.
*
*  call the root-finding subroutine
*
       call Bisect( x_initial, x_final, Root, FofX )
*
       < print result, and end>
```

```
*----------------------------------------------------------------
      Double Precision function FofX(x)
*
*   This is an example of a nonlinear function whose
*   root cannot be found analytically.
*
      Double Precision x
      FofX = cos(x) - x
      end
*----------------------------------------------------------------
      Subroutine Bisect( Left, Right, middle, F )
*
     <prepare for looping:  set initial values, etc.>
     <TOP of the loop>
       ...
     <Body of the loop:  divide the interval in half,
                  determine which half contains the root,
                  redefine the limits of the bracket>
       ...
     IF (bracket still too big) go to the TOP of the loop
       ...
     < loop finished:
       put on finishing touches (if needed), and end>
```

The main program is almost complete, although we've yet to begin the root-finding subroutine itself! This is a general characteristic of the "top-down" programing we've described — first, write an outline for the overall design of the program, and then refine it successively until all the details have been worked out. In passing, we note that the *generic* cosine function is used in the function definition rather than the double precision function dcos — the generic functions, which include abs, sin, and exp, will always match the data type of the argument.

Now, let's concentrate on the root-finding subroutine. The beginning of the code might look something like the following:

```
*----------------------------------------------------------------
      Subroutine Bisect( Left, Right, Middle, F )
*
* Paul L. DeVries, Department of Physics, Miami University
*
* Finding the root of f(x), known to be bracketed between
* Left and Right, by the Bisection method. The root is
* returned in the variable Middle.
```

The relative error in the first situation is $|\frac{0.1}{1178.4}| = 0.00008$, while in the second case it is $|\frac{0.1}{0.25}| = 0.4$. The higher accuracy of the first case is clearly associated with the much smaller relative error. Note, of course, that relative error is not defined if the true value is zero. And as a practical matter, the only quantity we can actually compute is an approximation to the relative error,

$$\text{Approximate Relative Error} \\ = \left| \frac{\text{Best Approximation} - \text{Previous Approximation}}{\text{Best Approximation}} \right|. \tag{2.4}$$

Thus, while there may be times when absolute accuracy is appropriate, most of the time we will want relative accuracy. For example, wanting to know x to within 1% is usually a more reasonable goal than simply wanting to know x to within 1, although this is not always true. (For example, in planning a trip to the Moon you might well want to know the distance to within 1 meter, not to within 1%!) Let's assume that in the present case our goal is to obtain results with relative error less than 5×10^{-8}. In this context, the "error" is simply our uncertainty in locating the root, which in the bisection method is just the width of the interval. (By the way, this accuracy in a trip to the Moon gives an absolute error of about 20 meters. Imagine being 20 meters above the surface, the descent rate of your lunar lander brought to zero, and out of gas. How hard would you hit the surface?) After declaring these additional variables, adding `write` statements, and cleaning up a few loose ends, the total code might look something like

```
*-----------------------------------------------------------
      Subroutine Bisect( Left, Right, Middle, F )
*
* Paul L. DeVries, Department of Physics, Miami University
*
*  Finding the root of the function "F" by BISECTION.  The
*  root is known(?) to be bracketed between LEFT and RIGHT.
*
*                          start date:    1/1/93
*
*
*  Type Declarations
*
      DOUBLE PRECISION    Left,  Right,  Middle
      DOUBLE PRECISION f, fLeft, fRight, fMiddle
      DOUBLE PRECISION TOL, Error
*
* Parameter declaration
```

```
*
      PARAMETER( TOL = 5.d-03)
*
* Initialization of variables
*
      FLeft   = f(Left)
      FRight  = f(Right)
*
* Top of the Bisection loop
*
100   Middle  = (Left+Right)/2.d0
      FMiddle = f(Middle)
*
* Determine which half of the interval contains the root
*
      IF( fLeft * fMiddle .le. 0 ) THEN
*
*        The root is in left subinterval:
*
             Right =  Middle
            fRight = fMiddle
      ELSE
*
*        The root is in right subinterval:
*
             Left =  Middle
            fLeft = fMiddle
      ENDIF
*
*  The root is now bracketed between (new) Left and Right !
*
*  Check for the relative error condition:  If too big,
*    bisect again; if small enough, print result and end.
*
      Error = ABS(  (Right-Left)/Middle  )
      IF( Error .gt. TOL ) GOTO 100
                                              !  Remove after
      write(*,*)' Root found at ',Middle      !  routine has
                                              !  been debugged.
      end
```

We've introduced, and declared as DOUBLE PRECISION, the parameter TOL to describe the desired tolerance — you might recall that the parameter statement allows us to treat TOL as a named variable in FORTRAN statements but

To see how this works, take a look at Figure 2.3. At $x = a$ the function and its derivative, which is tangent to the function, are known. Assuming that the function doesn't differ too much from a straight line, a good approximation to where the *function* crosses zero is where the *tangent line* crosses zero. This point, being the solution to a *linear* equation, is easily found — it's given by Equation (2.18)! Then this point can be taken as a new guess for the root, the function and its derivative evaluated, and so on. The idea of using one value to generate a better value is called *iteration*, and it is a very practical technique which we will use often. Changing the notation a little, we can calculate the $(i+1)$-th value x_{i+1} from the i-th value by the iterative expression

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{2.19}$$

For the function $f(x) = \cos x - x$, we have that $f'(x) = -\sin x - 1$. We know that $\cos x = 1$ at $x = 0$, and that $\cos x = 0$ at $x = \pi/2$, so we might guess that $x = \cos x$ somewhere around $x_0 = \pi/4$. We then calculate the zero to be at

$$x_1 = \frac{\pi}{4} - \frac{\cos\pi/4 - \pi/4}{-\sin\pi/4 - 1} = 0.739536134. \tag{2.20}$$

This is a pretty good result, much closer to the correct answer of 0.739085133 than was the initial guess of $\pi/4 = 0.785398163$. And as noted, this result can be used as a new guess to calculate another approximation to the location of the zero. Thus

$$x_2 = 0.739536134 - \frac{\cos(0.739536134) - 0.739536134}{-\sin(0.739536134) - 1} = 0.739085178, \tag{2.21}$$

a result accurate to 7 significant digits.

Beginning with an initial guess x_0, the expression is iterated to generate $x_1, x_2, \ldots$, until the result is deemed sufficiently accurate. Typically we want a result that is accurate to about eight significant digits, e.g., a relative error of 5×10^{-8}. That means that after each evaluation of x_{i+1} it should be compared to x_i; if the desired accuracy has been obtained, we should quit. On the other hand, if the accuracy has not been obtained, we should iterate again. Since we don't know how many iterations will be needed, a `DO` loop is not appropriate for this task. Rather, we need to implement a loop with a `GOTO` statement, being sure to include an exit out of the loop after the error condition has been met.

```
Program ROOTS
Double Precision x, FofX, DERofF
```

```
      External FofX, DERofF
      x = 0.8d0
      call Newton ( x, FofX, DERofF )
      write(*,*)' Root found at x =',x
      end
*-----------------------------------------------------------
      Double Precision Function FofX(x)
*
*  This is an example of a nonlinear function whose
*  root cannot be found analytically.
*
      Double Precision x
      FofX = cos(x) - x
      end
*-----------------------------------------------------------
      Double Precision Function DERofF(x)
*
*  This function is the derivative of "F of X."
*
      Double Precision x
      DERofF  = -sin(x) - 1.d0
      end
*-----------------------------------------------------------
      Subroutine Newton ( x, F, Fprime )
*
* Paul L. DeVries, Department of Physics, Miami University
*
*  Preliminary code for root finding with Newton-Raphson
*
*                         start date:    1/1/93
*
*  Type declarations
*
      DOUBLE PRECISION x, F, Fprime, delta, error, TOL
*
* Parameter declarations
*
      PARAMETER(  TOL = 5.d-03)
*
* Top of the  loop
*
100   delta = -f(x)/fprime(x)
      x = x + delta
*
```

```
*  Check for the relative error condition: If too big,
*  loop again; if small enough,  end.
*
      Error = ABS(  delta / x )
      IF ( Error .gt. TOL) GOTO 100
      end
```

EXERCISE 2.2

Verify that this code is functioning properly by finding (again) where $x = \cos x$. Compare the effort required to find the root with the Newton–Raphson and the bisection methods.

As we noted earlier, finding the roots of equations often occurs in a larger context. For example, in Chapter 4 we will find that the zeros of Legendre functions play a special role in certain integration schemes. So, let's consider the Legendre polynomial

$$P_8(x) = \frac{6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35}{128}, \quad -1 \le x \le 1, \quad (2.22)$$

and try to find its roots. Where to start? What would be a good initial guess? Since only the first derivative term was retained in developing the Newton–Raphson method, we suspect that we need to be close to a root before using it. For $|x| < 1$, we have $x^8 < x^6 < x^4 < x^2$. Let's (temporarily) ignore all the terms in the polynomial except the last two, and set this truncated function equal to zero. We thus have

$$P_8(x_0) \approx \frac{-1260x_0^2 + 35}{128} = 0, \qquad (2.23)$$

and hence

$$x_0 = \pm\sqrt{\frac{35}{1260}} = \pm\sqrt{\frac{1}{36}} = \pm\frac{1}{6}. \qquad (2.24)$$

Thus 0.167 should be an excellent guess to begin the iteration for the smallest non-negative root.

EXERCISE 2.3

Use the Newton–Raphson method to find the smallest non-negative root of $P_8(x)$.

Root-finding in general, and the Newton–Raphson method in particular, arise in various rather unexpected places. For example, how could we listed in Table 3.1.

TABLE 3.1 Bessel Functions

ρ	$J_0(\rho)$	$J_1(\rho)$	$J_2(\rho)$
0.0	1.00000 00000	0.00000 00000	0.00000 00000
1.0	0.76519 76866	0.44005 05857	0.11490 34849
2.0	0.22389 07791	0.57672 48078	0.35283 40286
3.0	−0.26005 19549	0.33905 89585	0.48609 12606
4.0	−0.39714 98099	−0.06604 33280	0.36412 81459
5.0	−0.17759 67713	−0.32757 91376	0.04656 51163
6.0	0.15064 52573	−0.27668 38581	−0.24287 32100
7.0	0.30007 92705	−0.00468 28235	−0.30141 72201
8.0	0.17165 08071	0.23463 63469	−0.11299 17204
9.0	−0.09033 36112	0.24531 17866	0.14484 73415
10.0	−0.24593 57645	0.04347 27462	0.25463 03137

EXERCISE 3.1

Using pencil and paper, estimate $J_1(5.5)$ by linear, quadratic, and cubic interpolation.

You probably found the computations in the Exercise to be tedious — certainly, they are for high order approximations. One of the advantages of the Lagrange approach is that its coding is very straightforward. The crucial fragment of the code, corresponding to Equations (3.3) and (3.5), might look like

```
*
*  Code fragment for Lagrange interpolation using N points.
*  The approximation P is required at XX, using the
*  tabulated values X(j) and F(j).
*
        P = 0.d0
        DO j = 1, N
*
*  Evaluate the j-th coefficient
*
           Lj = 1.d0
           DO  k = 1, N
              IF(j .ne. k) THEN
                 Lj = Lj * ( xx-x(k) )/( x(j)-x(k) )
              ENDIF
```

```
          END DO
*
*   Add contribution of j-th term to the polynomial
*
          P = P + Lj * F(j)
        END DO
        ...
```

We should point out that products need to be initialized to 1, just as sums need to be initialized to 0. This fragment has one major flaw — what happens if any two of the x_i are the same? While this doesn't happen in the present example, it's a possibility that should be addressed in a general interpolation program.

In the above exercise, which ρ values did you use in your linear interpolation? Nothing we've said so far would prevent you from having used $\rho = 0$ and 1, but you probably used $\rho = 5$ and 6, didn't you? You know that at $\rho = 5$ the approximating function is exact. As you move away from 5, you would expect there to be a difference to develop between the exact function and the approximation, but this difference (i.e., error) must become zero at $\rho = 6$. So if you use tabulated values at points that surround the point at which you are interested, the error will be kept to a minimum.

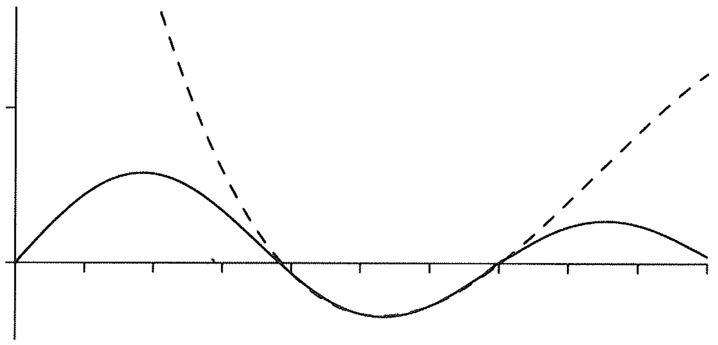

FIGURE 3.1 The Bessel function $J_1(\rho)$ and the cubic Lagrange polynomial approximation to it, using the tabular data at $\rho = 4, 5, 6$, and 7.

The converse is particularly illuminating. Using the interpolating polynomial to *extrapolate* to the region exterior to the points sampled can

```
* This subroutine is called with the first derivatives
* at x=x(1), FP1, and at x=x(n), FPN, specified, as
* well as the X's and the F's. (And n.)
*
* It returns the second derivatives in SECOND.
*
* The arrays X, F, and SECOND are dimensioned in the
* calling routine.
*
*  < Initialize A, B, C --- the subdiagonal, main diagonal,
*     and super-diagonal.>
*  < Initialize R --- the right-hand side.>
*
*  < Call TRISOLVE to solve for SECOND, the second
*     derivatives. This requires the additional array BETA.>
*
       end
```

This subroutine will certainly work. But... Are all those arrays *really* necessary? If we leave the solution of the tridiagonal system to a separate subroutine, they probably are. However, it would be just as useful to have a specialized version of TRISOLVE within the spline code. This would make the spline subroutine self-contained, which can be a valuable characteristic in itself. As we think about this, we realize that TRISOLVE will be the *major* portion of the spline routine — what we *really* need to do is to *start* with TRISOLVE, and add the particular features of spline interpolation to it. This new point of view is considerably different from what we began with, and not obvious from the beginning. The process exemplifies an important component of good programming and successful computational physics — don't get "locked in" to one way of doing things; always look to see if there is a better way. Sometimes there isn't. And sometimes, a new point of view can lead to dramatic progress.

So, let's start with TRISOLVE. To begin, we should immediately change the name of the array X to SECOND, to avoid confusion with the coordinates that we'll want to use. Then we should realize that we don't need both A and C, since the particular tridiagonal matrix used in the cubic spline problem is symmetric. In fact, we recognize that it's not necessary to have arrays for the diagonals of the matrix at all — we can evaluate them as we go! That is, instead of having a DO loop to evaluate the components of A, for example, we can evaluate the specific element we need within the Gauss elimination loop. The arrays A, B, C, and R are not needed! The revised code looks like this:

```
       Subroutine SplineInit(x,f,fp1,fpn,second,n)
```

```
*
* This subroutine performs the initialization of second
* derivatives necessary for CUBIC SPLINE interpolation.
* The arrays X and F contain N values of function and the
* position at which it was evaluated, and FP1 and FPn
* are the first derivatives at x = x(1) and x = x(n).
*
* The subroutine returns the second derivatives in SECOND.
*
* The arrays X, F, and SECOND are dimensioned in the
* calling routine --- BETA is dimensioned locally.
*
      Integer n
      Double Precision X(n), F(n), Second(n)
      Double Precision FP1,FPn
      Double Precision BETA(100)
*
*  In a cubic spline, the approximation to the function and
*  its first two derivatives are required to be continuous.
*  The primary function of this subroutine is to solve the
*  set of tridiagonal linear equations resulting from this
*  requirement for the (unknown) second derivatives and
*  knowledge of the first derivatives at the ends of the
*  interpolating region. The equations are solved by
*  Gaussian elimination, restricted to tridiagonal systems,
*  with A, B, and C being sub, main, and super diagonals,
*  and R the right hand side of the equations.
*
      If(n .gt. 100)
     +        STOP 'Array too large for SPLINE INIT'
      b1 = 2.d0*(x(2)-x(1))
      beta(1) = b1
      If (beta(1) .eq. 0)
     +        STOP 'Zero diagonal element in SPLINE INIT'
      r1 = 6.d0*( (f(2)-f(1))/(x(2)-x(1)) - FP1 )
      second(1) = r1
      DO j=2,n
         IF (j.eq.n) THEN
            bj= 2.d0 * (x(n)-x(n-1))
            rj= -6.d0*((f(n)-f(n-1))/(x(n)-x(n-1))-FPn)
*
*        For j=2,...,n-1, do the following
         ELSE
            bj=2.d0*( x(j+1) - x(j-1) )
```

```
                 rj=6.d0*( (f(j+1)-f( j ))/(x(j+1)-x( j ))
                             -(f( j )-f(j-1))/(x( j )-x(j-1)))
              ENDIF
*
*  Evaluate the off-diagonal elements. Since the
*  matrix is symmetric, A and C are equivalent.
*
              aj = x( j ) - x(j-1)
              c  = aj
              beta(j) = bj - aj * c / beta(j-1)
              second(j)  = rj - aj* second(j-1) / beta(j-1)
              IF(beta(j) .eq. 0)
     +             STOP 'Zero diagonal element in SPLINE INIT'
           END DO
*
* Now, for the back substitution...
*
           second(n) = second(n) / beta(n)
           DO j = 1, n-1
              c = x(n-j+1)-x(n-j)
              second(n-j) =
     +           ( second(n-j) - c * second(n-j+1) )/beta(n-j)
           END DO
           end
```

This code could be made more "compact" — that is, we don't really need to use intermediate variables like aj and c, and they could be eliminated — but the clarity might be diminished in the process. It's far better to have a clear, reliable code than one that is marginally more "efficient" but is difficult to comprehend.

Of course, we haven't interpolated anything yet! SplineInit yields the second derivatives, which we need for interpolation, but doesn't do the interpolation itself. SplineInit needs to be called, once, before any interpolation is done.

The interpolation itself, the embodiment of Equation (3.31), is done in the function Spline:

```
           Double Precision Function Spline(xvalue,x,f,second)
*
* This subroutine performs the CUBIC SPLINE interpolation,
* after SECOND has been initialized by SPLINE INIT.
* The arrays F, SECOND, and X contain N values of function,
* its second derivative,  and the positions at which they
* were evaluated.
*
* The subroutine returns the cubic spline approximation
* to the function at XVALUE.
*
* The arrays X, F, and SECOND are dimensioned in the
* calling routine.
*
       Integer n
       Double Precision Xvalue, X(n), F(n), Second(n)
*
*  Verify that XVALUE is between x(1) and x(n).
*
       IF (xvalue .lt. x(1))
     +    Stop 'XVALUE is less than X(1) in SPLINE'
       IF (xvalue .gt. x(n))
     +    Stop 'XVALUE is greater than X(n) in SPLINE'
*
*  Determine the interval containing XVALUE.
*
       j = 0
100    j = j + 1
       IF ( xvalue .gt. x(j+1) ) goto 100
*
* Xvalue is between x(j) and x(j+1).
*
* Now, for the interpolation...
*
         ...
       spline = ...
       end
```

Some of the code has been left for you to fill in.

EXERCISE 3.6

Use the cubic spline approximation to the Bessel function to find the location of its first zero.

In this spline approximation, we need to know the derivatives of the function at the endpoints. But how much do they matter? If we had simply "guessed" at the derivatives, how much would it affect the approximation? One of the advantages of a computational approach is that we answer these coefficient is made to be unity. (We note that this scaling is done for comparison purposes only, and need not be included as a step in the actual computation.) The swapping of rows and scaling makes the coding a little involved, and so we'll provide a working subroutine, LUSolve, that incorporates these complications.

```
      Subroutine LUsolve( A, x, b, det, ndim, n )
*----------------------------------------------------------
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine solves the linear set of equations
*
*          A x = b
*
* by the method of L U decomposition.
*
*  INPUT:    ndim    the size of the arrays, as dimensioned
*                       in the calling routine
*            n       the actual size of the arrays for
*                       this problem
*            A       an n by n array of coefficients,
*                       altered on output
*            b       a vector of length n
*
*  OUTPUT:   x       the 'solution' vector
*            det     the determinant of A.  If the
*                    determinant is zero, A is SINGULAR.
*
*                                 1/1/93
*
      integer ndim,n,order(100),i,j,k, imax,itemp
      double precision a(ndim,ndim),x(ndim),b(ndim),det
      double precision scale(100), max, sum, temporary
      if(n.gt.100)stop ' n too large in LUsolve'
      det = 1.d0
*
* First, determine a scaling factor for each row.   (We
* could "normalize" the equation by multiplying by this
* factor.   However, since we only want it for comparison
* purposes, we don't need to actually perform the
* multiplication.)
*
      DO i = 1, n
         order(i) = i
```

```
             max = 0.d0
             DO  j = 1, n
                if( abs(a(i,j)) .gt. max) max = abs(a(i,j))
             END DO
             scale(i) = 1.d0/max
          END DO
*
* Start the LU decomposition. The original matrix A
* will be overwritten by the elements of L and U as
* they are determined. The first row and column
* are specially treated, as is L(n,n).
*
          DO  k = 1, n-1
*
* Do a column of L
*
             IF( k .eq. 1 ) THEN
*                  No work is necessary.
             ELSE
*                Compute elements of L from Eq. (3.105).
*
                DO i = k, n   !
                   sum = a(i,k)
                   DO j = 1, k-1
                      sum = sum - a(i,j)*a(j,k)
                   END DO
                   a(i,k) = sum     !  Put L(i,k) into A.
                END DO
             ENDIF
*
* Do we need to interchange rows? We want the largest
* (scaled) element of the recently computed column of L
* moved to the diagonal (k,k) location.
*
             max = 0.d0
             DO i = k, n
                IF(scale(i)*a(i,k) .ge.  max)THEN
                   max = scale(i)*a(i,k)
                   imax=i
                ENDIF
             END DO
*
* Largest element is L(imax,k). If imax=k, the largest
* (scaled) element is already on the diagonal.
```

```
*
          IF(imax .eq. k)THEN
*             No need to exchange rows.
          ELSE
*             Exchange rows...
*
             det = -det
             DO  j = 1, n
                temporary = a(imax,j)
                a(imax,j) = a(k,j)
                a(k,j)    = temporary
             END DO
*
*      scale factors...
*
             temporary   = scale(imax)
             scale(imax) = scale(k)
             scale(k)    = temporary
*
*      and record changes in the ordering
*
             itemp       = order(imax)
             order(imax) = order(k)
             order(k)    = itemp
*
          ENDIF
          det = det * a(k,k)
*
* Now compute a row of U.
*
          IF(k.eq.1) THEN
*    The first row is treated special, see Eq. (3.102).
*
             DO j = 2, n
                a(1,j) = a(1,j) / a(1,1)
             END DO
          ELSE
*    Compute U(k,j) from Eq. (3.106).
*
             DO  j = k+1, n
                sum = a(k,j)
                DO i = 1, k-1
                   sum = sum - a(k,i)*a(i,j)
                END DO
```

```
*   Put the element U(k,j) into A.
*
                a(k,j) = sum / a(k,k)
             END DO
          ENDIF
1000    CONTINUE
*
* Now, for the last element of L
*
       sum = a(n,n)
       DO  j = 1, n-1
          sum = sum - a(n,j)*a(j,n)
       END DO
       a(n,n) = sum
       det = det * a(n,n)
*
* LU decomposition is now complete.
*
* We now start the solution phase.   Since the equations
* have been interchanged, we interchange the elements of
* B the same way, putting the result into X.
*
       DO i = 1, n
          x(i) = b( order(i) )
       END DO
*
* Forward substitution...
*
       x(1) = x(1) / a(1,1)
       DO i = 2, n
          sum = x(i)
          DO  k = 1, I-1
             sum = sum - a(i,k)*x(k)
          END DO
          x(i) = sum / a(i,i)
       END DO
*
* and backward substitution...
*
       DO  i = 1, n-1
         sum = x(n-i)
         DO k = n-i+1, n
            sum = sum - a(n-i,k)*x(k)
         END DO
```

```
            x(n-i) = sum
          END DO
*
* and we're done!
*
          end
```

A calculation of the determinant of the matrix has been included. The determinant of a product of matrices is just the product of the determinants of the matrices. For triangular matrices, the determinant is just the product of the diagonal elements, as you'll find if you try to expand it according to Cramer's rule. So after the matrix **A** has been written as **LU**, the evaluation of the determinant is straightforward. Except, of course, that permuting the rows introduces an overall sign change, which must be accounted for. If the determinant is zero, no unique, nontrivial solution exists; small determinants are an indication of an ill-conditioned set of equations whose solution might be very difficult.

EXERCISE 3.15

You might want to test the subroutine by solving the equations

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \tag{3.117}$$

which you've seen before.

EXERCISE 3.16

As an example of a system that might be difficult to solve, consider

$$\begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}. \tag{3.118}$$

The matrix with elements $H_{ij} = 1/(i+j-1)$ is called the Hilbert matrix, and is a classic example of an ill-conditioned matrix. With double precision arithmetic, you are equipped to solve this particular problem. However, as the dimension of the array increases it becomes tween a and b. To calculate the function average, we simply evaluate $f(x)$ at each of the randomly selected points, and divide by the number of points:

$$\langle f \rangle_N = \frac{1}{N} \sum_{i=1}^{N} f(x_i). \tag{4.117}$$

As the number of points used in calculating the average increases, $\langle f \rangle_N$ tends towards the "real" average, $\langle f \rangle$, and so we write the Monte Carlo estimate of the integral as

$$\int_a^b f(x)\,dx \approx (b-a)\,\frac{1}{N} \sum_{i=1}^{N} f(x_i). \tag{4.118}$$

Finding a "list" of random numbers could be a real problem, of course. Fortunately for us, it's a problem that has already been tackled by others, and as a result *random number generators* are fairly common. Unfortunately, there is a wide range in the quality of these generators, with some of them being quite unacceptable. For a number of years a rather poor random number generator was widely distributed in the scientific community, and more recently algorithms now described as "mediocre" were circulated. For our purposes, which are not particularly demanding, the subroutine RANDOM, supplied with the FORTRAN compiler, will suffice. If our needs became more stringent, however, verifying the quality of the generator, and replacing it if warranted, would take a high priority. In any event, we should note that these numbers are generated by a computer algorithm, and hence are not truly random — they are in fact *pseudo-random* — but they'll serve our purpose. Unfortunately, the argument of RANDOM must be a REAL variable; we also need the subroutine SEED, which initializes the random number generator. A suitable code for estimating the integral then would be

```
      Program MONTE_CARLO
*
*  Paul L. DeVries, Department of Physics, Miami University
*
*  This program computes a Monte Carlo style estimate of
*  the integral exp(x) between 0 and 1.  (= e-1)
*
      double precision sum,e,ran1, x, error, monte
      real xxx
      integer N,i
      integer*2 value
      parameter ( e = 2.718281828459045d0 )
*
```

```
* Initialize the "seed" used in the Random Number
* Generator, and set the accumulated SUM to zero.
*
      value = 1
      call seed( value )
      sum =0.d0
*
*  Calculate the function a total of 1,000 times, printing
*  an estimate of the integral after every 10 evaluations.
*
      DO i = 1, 100
*
*  Evaluate the function another 10 times.    SUM is the
*  accumulated total.
*
         DO j = 1, 10
            call random( xxx )
            x = xxx
            sum = sum + exp(x)
         END DO
*
*  The function has now been evaluated a total of
*  ( 10 * i ) times.
*
         N = i * 10
         MONTE = sum / N
*
*  Calculate the relative error, from the known value
*  of the integral.
*
         error = abs( monte - (e-1.d0) )/( e - 1.d0 )
         write(*,*) n, MONTE, error
      END DO

      end
```

Before using RANDOM, SEED is called to initialize the generation of a sequence of random numbers. The random number generator will provide the same sequence of random numbers every time it's called, after it's been initialized with a particular value of `value`. This is essential to obtaining reproducible results and in debugging complicated programs. To obtain a *different* sequence of random numbers, simply call SEED with a different initial `value`.

This code prints the estimate of the integral after every 10 function the general scientific community. Perhaps the clearest explanation of the algorithm, however, is due to Danielson and Lanczos. If N is an even number, then we can write the DFT as a sum over even-numbered points and a sum over odd-numbered points:

$$\begin{aligned} g(n\Delta\omega) &= \sum_{m=0}^{N-1} f(m\Delta t)e^{-i2\pi mn/N} \\ &= \sum_{m=0,even}^{N-1} f(m\Delta t)e^{-i2\pi mn/N} + \sum_{m=0,odd}^{N-1} f(m\Delta t)e^{-i2\pi mn/N} \\ &= \sum_{j=0}^{N/2-1} f(2j\Delta t)e^{-i2\pi 2jn/N} + \sum_{j=0}^{N/2-1} f((2j+1)\Delta t)e^{-i2\pi(2j+1)n/N}, \end{aligned} \tag{6.116}$$

where we've let $m = 2j$ in the first term (even points), and $m = 2j + 1$ in the second term (odd points). But this is simply

$$\begin{aligned} g(n\Delta\omega) = & \sum_{j=0}^{N/2-1} f(2j\Delta t)e^{-i2\pi jn/(N/2)} \\ & + e^{-i2\pi n/N} \sum_{j=0}^{N/2-1} f((2j+1)\Delta t)e^{-i2\pi jn/(N/2)} \\ = & \, g_{even}(n\Delta\omega) + e^{-i2\pi n/N} g_{odd}(n\Delta\omega), \end{aligned} \tag{6.117}$$

where we've recognized that the sums are themselves DFTs, with half as many points, over the original even- and odd-numbered points. The original calculation of the DFT was to take on the order of N^2 operations, but this decomposition shows that it actually only requires $2 \times (N/2)^2$ operations! But there's no reason to stop here — each of these DFTs can be decomposed into even and odd points, and so on, as long as they each contain an even number of points. Let's say that $N = 2^k$ — then after k steps, there will be N transforms to be evaluated, each containing only one point! The total operation count is thus *not* on the order of N^2, but rather on the order of $N \log_2 N$! The fast Fourier transform is *fast!*

The coding of the FFT can be a little complicated, due to the even/odd interweaving that must be done. Since there's little to be gained by such an exercise, we simply present some code for your use.

```
*---------------------------------------------------------------
* Paul L. DeVries, Department of Physics, Miami University
```

```
*
* Just a little program to check our FFT.
*
*                               start date:  March, 1965
*
* Type declarations
*
      Complex*16 a(8)
      Integer i, k, inv, N
      k = 3
      N = 2**k
       DO i = 1, N
         a(i) = 0
      END DO
      a(3) = 1

      inv = 0
      call FFT( a, k, inv )
      write(*,*)a
      end
*
      Subroutine FFT(A,m,INV)
*----------------------------------------------------------
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine performs the Fast Fourier Transform by
* the method of Cooley and Tukey --- the FORTRAN code was
* adapted from
*
*   Cooley, Lewis, and Welch, IEEE Transactions E-12
*       (March 1965).
*
* The array A contains the complex data to be transformed,
* 'm' is log2(N), and INV is an index = 1 if the inverse
* transform is to be computed. (The forward transform is
* evaluated if INV is not = 1.)
*
*                                  start:  1965
*                          last modified:  1993
*
      Complex*16 A(1), u, w, t
      Double precision ang, pi
      Integer N, Nd2, i, j, k, l, le, le1, ip
      Parameter (pi = 3.141592653589793d0)
```

```
*
*  This routine computes the Fast Fourier Transform of the
*  input data and returns it in the same array. Note that
*  the k's and x's are related in the following way:
*
*    IF    K = range of k's       and     X = range of x's
*
*    THEN  delta-k = 2 pi / X     and   delta-x = 2 pi / K
*
*  When the transform is evaluated, it is assumed that the
*  input data is periodic. The output is therefore periodic
*  (you have no choice in this). Thus, the transform is
*  periodic in k-space, with the first N/2 points being
*  'most significant'. The second N/2 points are the same
*  as the fourier transform at negative k!!! That is,
*
*              FFT(N+1-i) = FFT(-i)   ,i = 1,2,....,N/2
*
      N   = 2**m
      Nd2 = N/2
      j   = 1
      DO i = 1, N-1
         IF( i .lt. j ) THEN
            t    = A(j)
            A(j) = A(i)
            A(i) = t
         ENDIF
         k = Nd2
100      IF( k .lt. j ) THEN
            j = j-k
            k = k/2
            goto 100
         ENDIF
         j  = j+k
      END DO
      le = 1
      DO l = 1, m
         le1 = le
         le  = le + le

         u = ( 1.D0, 0.D0 )
         ang = pi / dble(le1)
         W = Dcmplx( cos(ang), -sin(ang) )
         IF(inv .eq. 1) W = Dconjg(W)
         DO j = 1, le1
            DO i = j, N, le
               ip   = i+le1
               t    = A(ip)*u
               A(ip)= A(i)-t
               A(i) = A(i)+t
            END DO
            u=u*w
         END DO

      END DO

      IF(inv .ne. 1) THEN
         DO i = 1, N
            A(i) = A(i) / dble(N)
         END DO
      ENDIF
      end
```

EXERCISE 6.12

Verify that this code performs correctly. What should the results be? Note that, as often happens when doing numerical work, the computer might give you a number like 10^{-17} where you had expected to find a zero.

Life in the Fast Lane

The development of the fast Fourier transform is one of the most significant achievements ever made in the field of numerical analysis. This is due to *i)* the fact that the Fourier transform is found in a large number of diverse areas, such as electronics, optics, and quantum mechanics, and hence is extremely important; and *ii)* the FFT is *fast*. For our present purposes, we will consider the sampling of some function at various time steps — the Fourier transform will then be in the frequency domain. Such an analysis is sometimes referred to as being an harmonic analysis, or a function is said to be synthesized (as in electronic music generation.)

Earlier, we noted that while it appeared that the highest frequency used in the DFT (or the FFT) was $(N-1)\Delta\omega$, the highest frequency actually used is only half that. The reason is quite simple: while $f(t)$ is periodic in time, $g(\omega)$ is periodic in frequency! (This is obvious, after the fact, since the