CHAPTER 7

Introduction to Fortran

The Fortran Programming Language

The Fortran programming language was one of the first (if not the first) "high level" languages developed for computers. It is referred to as a high level language to contrast it with machine language or assembly language which communicate directly with the computer's processor with very primitive instructions. Since all that a computer can really understand are these primitive machine language instructions, a Fortran program must be translated into machine language by a special program called a Fortran compiler before it can be executed. Since the processors in various computers are not all the same, their machine languages are not all the same. For a variety of reasons, not all Fortran compilers are the same. For example, more recent Fortran compilers allow operations not allowed by earlier versions. In this chapter, we will only describe features that one can expect to have available with whatever compiler one may have available.

Fortran was initially developed almost exclusively for performing numeric computations (Fortran is an acronym for "Formula Translation"), and a host of other languages (Pascal, Ada, Cobol, C, etc.) have been developed that are more suited to nonnumerical operations such as searching databases for information. Fortran has managed to adapt itself to the changing nature of computing and has survived, despite repeated predictions of its death. It is still the major language of science and is heavily used in statistical computing.

The most standard version of Fortran is referred to as Fortran 77 since it is based on a standard established in 1977. A new standard was developed in 1990 that incorporates some of the useful ideas from other languages but we will restrict ourselves to Fortran 77.

The Basic Elements of Fortran

A Fortran program consists of a series of lines of code. These lines are executed in the order that they appear unless there are statements that cause execution to jump from one place in the series of lines to another. At the end of this chapter, we have listed the code of a complete program and we will refer to this program as we go along. Learning to write Fortran programs is usually done by imitating programs written by others. This chapter cannot possibly provide all of the details of Fortran, but rather to point out those features that are particularly important in statistical computing.

1. Statements begin in column 7 of a line and end before column 72 (if a statement will not fit in columns 7–72, it can be continued onto the next line (or lines) as long as a character is placed in column 6 of each continuation line). Columns 1–5 of a line can contain "a statement number" so that the statement on that line can be referenced by other statements. Lines that begin with the letter c in column one are interpreted as comments and are not executed. It is important to include enough comments in any program in any language so that you (or somebody else) can later understand what the lines of code are intended to do.

2. A file containing a Fortran program begins with a main program which may be followed by subprograms.

Subprograms come in two kinds; functions and subroutines. The main program and the subprograms it calls can be in different files which can be compiled separately and then linked together to form an executable file.

4. A main program begins with a series of statements declaring the names of the variables that are going to be used by the program and what type of variables they are (see "data types" below). Note that variable names must begin with letters of the alphabet, contain letters and the digits 0 through 9, and (to be safe as different compilers allow different numbers of characters) contain 6 characters or fewer. The program ends with the lines `stop` and `end`.

5. Between the declarations and the `stop` and `end` there are a wide variety of operations that can be performed, but usually a program consists of 1) reading some information from somewhere (for example, from a file or from the keyboard from the person using the program 2) performing some numerical or graphical task, and 3) placing the results of the task somewhere so that the user of the program can use them (perhaps on the user's screen or into a file).

6. Often there is some subtask that needs to be performed several times in a program. For example, it may be necessary to form the plot of one vector versus another. Rather than have the same set of lines of code in several different places in the main program, one can form them into what is called a subprogram. A subprogram begins and ends with special lines of code that tell the compiler that it is in fact a subprogram and not part of the main program. Another use of subprograms is in modular programming where one breaks a complicated task into a series of tasks that are more managable and writing a subprogram for each one.

Data Types

Fortran performs numerical and logical operations on variables of a few types, including:

1. **Integers**, that is, numbers that have no decimal part. There are two kinds of integers; long integers (also called four-byte integers or `integer*4`'s, they range from roughly -2^{31} to 2^{31} and use four bytes of memory) and short integers (also called two-byte integers or `integer*2`'s, they range from roughly 2^{-15} to 2^{15} and use two bytes of memory). One declares variables to be short or long integers by statements of the form
`integer*2 n,m,k`

```
integer*2 n,m,k
```

or

```
integer*4 n,m,k
```

respectively.

2. **Floating point numbers**, that is, numbers that can contain decimal parts. There are two kinds of floating point numbers; double precision (also called `real*8` reals, they range from roughly 10^{-300} to 10^{300}, take eight bytes of memory, and are accurate to roughly 13 or 14 decimal places) and single precision (also called `real*4` reals, they range from roughly 10^{-38} to 10^{38}, take four bytes of memory, and are accurate to roughly six or seven decimal places). Single and double precision variables are declared by lines of the form

```
real*4 x,y
```

or

```
real*8 x,y
```

respectively.

3. **Complex numbers**, that is variables that can take on complex values. Actually, complex variables can be thought of as a pair of `real*4` or `real*8` floating point numbers. Single and double precision complex variables are declared by lines of the form

```
complex*8 x,y
```

or

```
complex*16 x,y
```

respectively.

4. **Logical variables**, that is, variables that can only take on the "values" `.TRUE.` or `.FALSE.`.

5. **Character variables**, that is, variables that contain characters such as letters of the alphabet or numerical digits. A character variable can be as long as desired. To declare variables called `st1` and `st2` to be of length 10 and 20 respectively, one uses the statement

```
character st1*10,st2*20
```

6. **Arrays of variables**. In addition to scalar variables, one can declare a variable to be a vector (one dimensional array) or matrix (two dimensional array), or a higher dimensional array. To declare a single precision 100 by 3 matrix called `x`, one would have the statement

```
real*4 x(100,3)
```

while the statement

```
character st1(100)*10,st2(200)*20
```

would declare character arrays of length 100 and 200 called `st1` and `st2` where each of the 100 elements of `st1` can contain 10 characters, while each of the 200 elements of `st2` can contain as many as 20 characters. Note that Fortran can only perform operations on individual elements of arrays, that is, it has no built-in ability to perform operations (such as addition) on all the elements of arrays. Most computer systems have libraries of subprograms that perform operations on entire arrays.

Undeclared Names

It is very good programming pactice to declare every variable that a program uses (most compilers have a way to warn a programmer of any variables that have not been declared). However, one should be aware of the fact that any variable whose name begins with one of the letters i, j, k, l, m, or n, is considered to be an integer (usually an `integer*4`) unless otherwise declared explicitly, while any variable whose name begins with one of the other letters of the alphabet is taken to be a real (usually `real*4`). Often it is useful to follow these conventions in the naming of variables (although they should also be explicitly declared) so that someone reading the code will know what type the variables are without having to look at the declarations.

Assignments and Arithmetic Operations

An important part of a Fortran program is assigning values to variables and performing arithmetic operations on variables. There are five arithmetic operations, namely addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`). If more than one operation is contained in a statement, then exponentiation is done first, then multiplication and division (if there are more than one of these they are done from left to right), and finally addition and subtraction (again if there are more than one of these, they are done from left to right). Expressions within parentheses are done first.

When all operands of an arithmetic expression are of the same type, then the final type of the value of the expression will be of that type. When they are of different types, the value of the expression will be that of the last binary operation performed where the value of a binary operation of data of different types takes on the type of the higher of the two types, where types are ranked from lowest to highest by `integer*2`, `integer*4`, `real*4`, `real*8`, `complex*8`, and `complex*16`. One must be careful of expressions such as `x*(i/j)` or `x**(i/j)` when `i` and `j` are integers since the result of the division is truncated and not rounded.

Once an expression has been evaluated, it is converted to the type of the variable on the left hand side of the expression.

Intrinsic Functions

Fortran has a wide variety of functions built into it. These functions are called intrinsic functions. Examples include trigonometric and logarithm functions. The table below contains a list of the functions that a Fortran programmer can expect to have available when writing code. Note that these functions need not be declared before use. Also, starting with Fortran 77 the concept of generic functions was developed. For example, in previous versions of Fortran there were four functions for finding the absolute value of a number depending on its type. Now there is only the single function `abs` which will return a value having the same type as its input type. The old functions `iabs`, `abs`, `dabs`, and `cabs` are still available as well. The prefixes `i, a, d, c` are used in these functions for integer, real, double precision, and comples, respectively.

Intrinsic Functions[1]

`functions`	Definition
Numerical functions:	
`sdc=COS(sdc), sdc=SIN(sdc), sd=TAN(sd)`	Trigonometric functions
`sd=ACOS(sd), sd=ASIN(sd), sd=ATAN(sd)`	Inverse trig functions[2]
`sd=ATAN2(sd1,sd2)`	
`sd=COSH(sd), sd=SINH(sd), sd=TANH(sd)`	Hyperbolic trigonometric functions
`sdc=LOG(sdc), sd=LOG10(sd)`	Log functions
`sdc=SQRT(sdc)`	Square root
`isds=ABS(isdc)`	Absolute value
`sdc=EXP(sdc)`	Exponentiation
`isd=MOD(isd1,isd2)`	Remainder[3]
`isd=SIGN(isd1,isd2)`	Sign function[4]
`isd=MAX(isd1,isd2,...), isd=MIN(isd1,isd2,...)`	Maximum and minimum
`s=AMAX0(i1,i2,...), s=AMIN0(i1,i2,...)`	Integer input, single output
`i=MAX1(s1,s2,...), i=MIN1(s1,s2,...)`	Single input, integer output
`isd=DIM(isd1,isd2)`	Positive difference: `MAX(isd1-isd2,0)`
`d=DPROD(s1,s2)`	Double precision product of arguments
Type conversion:	
`i=INT(isdc), s=REAL(isdc), d=DBLE(isdc)`	
`c=CMPLX(isd)`	Input is real part, imaginary part is 0
`c=CMPLX(c)`	Identity

`c=CMPLX(isd1,isd2)`	First argument is real part, 2nd is imaginary part
Complex numbers:	
`s=REAL(c), s=AIMAG(c)`	Real and Imaginary parts
`c=CONJG(c), s=CABS(c)`	Complex conjugate and modulus
Rounding and truncating5:	
`sd=AINT(sd), sd=ANINT(sd)`	Truncation and rounding
`i=NINT(sd)`	Rounding to nearest integer
Operations with characters:	
`C=CHAR(i)`	Character having input ASCII code
`i=ICHAR(C)`	ASCII code of input character
`i=INDEX(str1,str2)`	Compare strings[6]
`i=LEN(str)`	Number of characters in the string
`L=LGE(str1,str2), L=LGT(str1,str2),` `L=LLE(str1,str2), L=LLT(str1,str2)`	Lexicographical ordering[7]

[1]The table lists the input and output type of the function, for example, the notation `isds=ABS(isdc)` means that for integer, single, double, and complex input, the output is integer, single, double, and single respectively (note that `CABS(c)` is single). For characters, `C` represents a single character while `str` represents a string of characters. Logical variables are denotd by `L`.
[2]The function `ATAN2` has two arguments x_1 and x_2 and is defined to be ?????
[3]The remaindering function applied to arguments x_1 and x_2 is defined to be $x_1 - \mathtt{INT}(x_1/x_2) * x_2$.
[4]The sign function having two arguments x_1 and x_2 is $|x_1|$ if $x_2 \geq 0$ and $-|x_1|$ if $x_2 < 0$.
[5]Truncation means, for example, `REAL(INT(s))`. Rounding means, for example, `REAL(INT(s+.5)` if `s`≥ 0 or `REAL(INT(s-0.5))` if `s`< 0.
[6]If `str2` is contained in `str1` then the result is the position of the first character of `str2` in `str1`; otherwise the result is 0.
[6]For example, `LGE(str1,str2)` is true if the two strings are the same or if `str1` follows `str2` in lexicographical order (comparing ASCII codes character by character).

Input and Output

Perhaps the most difficult part of Fortran programming are the operations of inputting information into the program and outputting information from the program to a place where it can be used later. These operations are performed using the `read` and `write` statements respectively. There are four basic parts of these statements:

1. A description of where the information is coming from or going to,
2. A description of the format that the information is in if it is being read or a description of the format that is desired for the information if it is being written,
3. The statement number of the statement to be executed next if something goes wrong during the reading or writing,
4. The names of the variables that are to receive the information or the names of the variables whose values are to be written.

It is almost impossible to describe all of the variations on these four basic ideas. Most Fortran programmers learn them by imitating what other programmers have done. An example which contains all four elements is provided by the statements

```
      read(2,10,err=20) ((x(i,j),j=1,10),i=1,5)
  10  format(10x,6f10.0)
```

The read statement says to read 50 numbers from the file having "logical unit number" 2 according to the format statement having statement number 10, and assign these numbers to the upper left hand corner of the matrix `x` (upper five rows and 10 columns). The `err=20` means that if something goes wrong during the reading process (such as if a nonnumeric character were mistakenly entered in the file) then the next statement to be executed by the program is the one having statement number 20. The `6f10.0` in the format statement says that the 50 numbers should be read six per line, with the first of the six in columns 11–20, the next in 21-30, and so on, and the the numbers can be anywhere in these columns as long as there is a decimal point (note that the `10x` says to skip the first 10 columns on each line—this makes it posible to have extraneous information in the first 10 columns (such as a date if one were reading information collected over time and wanted the date in the file but didn't want to input it into the program).

Opening a File for Reading or Writing

Before reading from or writing to a file, one must first alert the computer's operating system what the name of the file is using the `open` statement. One also uses the `open` statement to attach a logical unit number to the file (such as the `2` in the write statement above) and then uses this number in future `read` statements. Note that one can have several files open at the same time, each with its own logical unit number.

Looping and Branching

An important part of Fortran are the statements that allow one to perform loops, that is executing sets of lines of code repeatedly. The most famous example is a "do loop", which is of the form

```
      do 10 i=i1,i2,i3
         ....
         ....
   10 continue
```

where `i1`, `i2`, and `i3` are integers (or integer expressions). The statements between the `do` and the `continue` are all executed with the variable `i` having value `i1`, then they are done again with `i` having value `i1+i3`, and so on, until the value of `i` is outside of the range `i1` to `i2`. Note that do loops can be nested, and that the do loop counter (the variable `i` in the example above) can not be changed during the loop.

Another example of a loop is the so-called "implied loop" which is of the form

```
      i=1
    5 continue
      ....
      ....
      if(i.eq.n) go to 10
      i=i+1
      go to 5
   10 continue
```

There are a number of branching constructs available. The `go to` used above is an unconditional branch. The line

```
    if(i.eq.3) go to 10
```

is an example of a conditional branch. In addition to `.eq.` we have `.ne.`, `.le.`, `.lt.`, `.ge.`, and `.gt.`. The use of `go to`'s is discouraged as they make reading the code of a program difficult. Other, more readable branches are available such as

```
      if(i.eq.1) then
```

```
         ...
         ...
      elseif(i.eq.2) then
         ...
         ...
      else
         ...
         ...
      endif
```

Subroutines and Functions

There are two kinds of subprograms in Fortran; subroutines and functions. We have already seen examples of functions in the intrinsic functions above, for example the statement `x=sqrt(8.0*atan(1.0))` uses two intrinsic functions; `atan` and `sqrt`. Functions return single values "through their names" and any number of values (scalars or arrays) through their argument list. A subroutine on the other hand returns nothing through its name, can return any number of values through its argument list, and is invoked by a statement of the form

```
call subname(x,n,m,y,z)
```

To illustrate writing functions and subroutines, consider the following code:

```
      double precision function inprod(x,y,n)
      dimension x(n),y(n)

      inprod=0.0d0
      do 10 i=1,n
   10 inprod=inprod+dble(x(i))*dble(y(i))

      return
      end

      subroutine mtmult(a,b,n,ndim,c)
      dimension a(ndim,n),b(ndim,n),c(ndim,n)

      do 10 i=1,n
      do 10 j=1,n
         c1=0.0
         do 20 k=1,n
   20    c1=c1+a(i,k)*b(k,j)
   10 c(i,j)=c1

      return
      end
```

Thus functions and subroutines both end with the statements `return` and `end`, and begin with a line that gives their name (note how functions and subroutines name themselves differently and that functions declare their own type) and specifies a list of arguments, that is variables that are to be used in the function or subroutine.

There are a few simple rules that one must use in writing and calling subroutines:

1. Subprograms can call other subprograms but a subprogram can never call itself either directly or indirectly (by calling another subprogram which in turn calls the original subprogram). Some modern compilers allow this recursive calling, but most still do not so we will not discuss it.

2. A subroutine begins with a statement beginning with the word `subroutine` followed by a space, followed by the name of the subroutine, followed (optionally) by a list of variable names in parentheses and

separated by commas. These variables are referred to as the arguments of the subroutine. The variables in the calling program are not available to the subprogram unless they are passed to it through the argument list (one can also use a `common` statement but we ignore that for now). We consider arguments in more detail later in the section discussing arrays in Fortran.

3. One of the most common errors in Fortran occurs if there is a mismatch between the type of a variable in a main program and the type expected by the subroutine. For example, if a subroutine expects a single precision variable and is passed a double precision number, then the number it actually receives will not be what is expected and in many cases this will cause a problem (note that the compiler does not check for "type mismatches.")

Compiling and Linking

The first program considered later in this chapter (which is stored in the file `regswp.f`) is self-contained except that it uses a subroutine called `tcdf` which is in another file called `tcdf.f`. To compile and link `regswp.f` and `tcdf.f` and create an executable file called `regswp`, one does the following (the `-c` switch on the first line says to only compile `tcdf.f` while the `-o` switch on the second line says to name the executable file `regswp`).

```
f77 -c tcdf.f
f77 -o regswp regswp.f tcdf.o
```

Arrays in Fortran

Arrays are an often poorly understood part of Fortran, particularly in relation to how they are used in subprograms. Unfortunately, arrays and subprograms are often covered at the end of introductory Fortran courses, if at all. We will consider one dimensional arrays (vectors), two dimensional arrays (matrices having row and column indices), and three dimensional arrays (often visualized as a vector of matrices, the first two indices being the row and column numbers and the third index being the matrix number). The extension to higher dimensional arrays should be obvious.

Arrays are Statically Allocated

With a few notable exceptions, Fortran compilers force the programmer to specify the size of an array in a main program before it is compiled and linked. This is done with a `dimension` statement (such as `dimension a(10,10,4)`, which allocates a three dimensional array `a` of four 10 by 10 matrices). Thus the memory for an array is said to be "statically allocated." Many programming languages allow "dynamic memory allocation," that is, the program can ask the user to specify the size of a problem and then allocate exactly the right amount of space for the arrays that are involved. In contrast, the Fortran programmer must anticipate the maximum size that a user might want and do the `dimension` accordingly.

Arrays are Stored in Memory in "Column Order"

Most features of a language such as Fortran do not require the programmer to keep in mind how the program interacts with a computer's memory. However, the ordering of the elements of an array is a notable exception to this rule. The elements are stored in consecutive locations in memory in what is called "column order." For a vector this just means that consecutive elements are stored in consecutive locations in memory, but how the ordering is done for higher dimensional arrays is not obvious. For a matrix, the basic rule is that the elements of the first column are stored in order followed by the elements of the second column, and so on. Thus code such as

```
      dimension a(4,4)
      data a/11,21,31,41,12,22,32,42,13,23,33,43,14,24,34,44/
```

```
      do 10 i=1,4
   10 write(*,20) (a(i,j),j=1,4)
   20 format(1x,4f4.0)
```

leads to

```
11. 12. 13. 14.
21. 22. 23. 24.
31. 32. 33. 34.
41. 42. 43. 44.
```

that is, the first four elements in the `data` statement are placed into the first column of `a`, the next four in the second column, and so on.

For a three dimensional array, we have rows, columns, and matrix number. If we had dimensioned `a` above by `dimension a(2,2,4)`, then the four matrices would be

```
11  31      12  32      13  33      14  34
21  41      22  42      23  43      24  44,
```

that is, the first four elements go into the first matrix (with the first two in the first column and the next two in the second), the next four in the second matrix, and so on.

Perhaps the best way to visualize this is to think of all arrays as vectors, with the dimensions specifying how that vector gets reshaped into the array. The important point is that in the assignment of elements into the array so that "the first index varies most rapidly." Thus if we added the line

```
      write(*,20) a
```

to the code above, we would get

```
11. 21. 31. 41.
12. 22. 32. 42.
13. 23. 33. 43.
14. 24. 34. 44.
```

Dimensioning Arrays in a Subprogram

An important part of programming is to create subprograms (subroutines or functions) to perform often needed calculations (such as adding or multiplying matrices). In Fortran, variables defined in one "module" (main program or subprogram) are unknown to any other module unless the programmer does something special to "pass" the values from one module to the other. The standard method to do this in Fortran is to have an argument in the subprogram list and in the call to the subprogram for the variable of interest. This is called "passing a variable" to the subprogram. It is still necessary to dimension any passed array in the subroutine. Further, it is posible to use variables to dimension arrays in a subprogram (note that such a variable must also be in the argument list).

A simple example of this is to write a subroutine which writes out the elements of an n by n matrix

```
      subroutine matprt(b,n)
      dimension b(n,n)
      do 10 i=1,n
   10 write(*,20) (b(i,j),j=1,n)
   20 format(1x,6f12.5)
      return
      end
```

Note that the variables in such a subprogram are called "dummy variables" and need not have the same names as the corresponding variables in the calling subprogram.

In keeping with what we said above, our subroutine will take the first n^2 elements being passed from the array in the calling routine, form the matrix b by column and then write out the rows of the result. But what are the first n^2 elements of the array in the calling routine? In the following main program,

```
      dimension a(4,4)
      data a/11,21,31,41,12,22,32,42,13,23,33,43,14,24,34,44/
      call matprt(a,3)
      stop
      end
```

the first 9 elements of `a` get passed to `matprt` and thus the output of the program will be

```
  11.00000    41.00000   32.00000
  21.00000    12.00000   42.00000
  31.00000    22.00000   13.00000
```

which is probably not at all what the programmer intended (presumably they wanted the upper left hand 3 by 3 portion of the matrix displayed).

The "ndim problem"

This last example illustrates a problem which causes the vast majority of hard-to-find Fortran bugs. What's worse is that many times the programmer has no idea that there is a problem (in the example we knew what the "correct" values should have been). One can only hope that the incorrect passing leads to a big enough problem (such as division by zero) so that the subtle passing problem gets diagnosed.

I refer to this problem as the "ndim problem" since the solution is to include a third argument in the subroutine (often called `ndim`) corresponding to the actual row dimension of the array in the calling routine (in this case it would be 4). In our example, `ndim` is the natural name for this argument as it is the actual dimension corresponding to the argument `n` in the calling routine. Thus the subroutine would be modified to be

```
      subroutine matprt(b,n,ndim)
      dimension b(ndim,n)
      do 10 i=1,n
   10 write(*,20) (b(i,j),j=1,n)
   20 format(1x,6f12.5)
      return
      end
```

and the call would be

```
     call matprt(a,3,4)
```

which would give the intended output.

A Bonus

A very powerful (but often dangerous) feature of Fortran is that one can pass an array having one number of indices (vector, matrix, three dimensional, etc.) to a subroutine which is expecting an array having a different number. This allows us to do something such as

```
      dimension x(8,2)
      data x/11,21,31,41,12,22,32,42,13,23,33,43,14,24,34,44/
      call mean(x(1,2),8,xbar)
      write(*,10) xbar
   10 format(1x,'xbar = ',f12.5)
      stop
      end
      subroutine mean(x,n,xbar)
      dimension x(n)
      xbar=0.0
      do 10 i=1,n
   10 xbar=xbar+x(i)
      xbar=xbar/n
      return
      end
```

which would find the sample mean of the second column of the matrix `x`. This example also illustrates another important fact about array passing. We've seen that a subroutine takes consecutive elements being passed from the calling routine, shapes them into an array according to the dimension in the subprogram, and then operates on the array. In the line `call mean(x(1,2),8,xbar)`, we see that the argument in the call is actually only "pointing" to the first element in `x` to be passed (note that if the argument had just been `x`, the pointer would be at the first element in `x`, that is, `x(1,1)`). This is useful in something such as

```
      dimension x(100),xbar(100)
      n=100
      do 10 i=1,n
   10 x(i)=i
      do 20 i=1,n
   20 call mean(x(n-i+1),i,xbar(i))
```

which finds the mean of the last i x's for $i = 1, \ldots, 100$.

A final example of this is given by writing a subroutine to calculate $X^T X$ for an $n \times m$ matrix X:

```
      subroutine xprx(x,n,m,ndim,mdim,xtx)
      dimension x(ndim,m),xtx(mdim,mdim)
      double precision dbprod
c
      do 10 i=1,m
      do 10 j=1,i
      xtx(i,j)=dbprod(x(1,i),x(1,j),n)
   10 xtx(j,i)=xtx(i,j)
      return
      end
c
      double precision function dbprod(x,y,n)
      dimension x(1),y(1)
      dbprod=0.0d0
c
      do 10 i=1,n
   10 dbprod=dbprod+dble(x(i))*dble(y(i))
      return
      end
```

There are several things to note here:

1. We need both an `ndim` and an `mdim` in this case.

2. The $(i,j)th$ element of X^TX is the inner product of its ith column with its jth column so we are using the inner product function `dbprod` and just pointing to the beginning of the two columns of interest in the call to `dbprod`.
3. Since X^TX is symmetric, `xprx` only calculates the the elements in the lower triangle and puts each element into the corresponding place in the upper triangle as well.
4. The variables `x` and `y` have been "dimensioned with a 1" in the function `dbprod`. This is consistent with what we have said above about the role of dimensions. This sort of thing is sometimes very useful in avoiding extra arguments. For example, if we were writing a subroutine to find the zeros of the polynomial $g(z) = \sum_{j=0}^{p} \alpha_j z^j$ we would probably pass $\alpha_0, \alpha_1, \ldots, \alpha_p$ in the vector `alpha` of length $p+1$ (since there is no zero index in standard Fortran) and to avoid passing the value of $p+1$, we can just dimension `alpha` with a 1.

An Example

We consider a program to do basic multiple linear regression which illustrates reading and writing and how to use subroutines.

Review of Regression Analysis

A very common situation in statistics is to have a set of n "objects" with measurements on a set of $m+1$ "variables" for each object and we wish to explain the behavior of one of the variables y (the "dependent" variable) in terms of the other m variables $X_1, \ldots, X_m$ (the "independent" or "regressor" variables) according to

$$y_i = \beta_0 + \beta_1 X_{i1} + \cdots + \beta_m X_{im} + \epsilon_i, \qquad i = 1, \ldots, n,$$

where y_i and $X_{i1}, \ldots, X_{im}$ are the measurements for the ith object, $\beta_0, \beta_1, \ldots, \beta_m$ are a set of unknown parameters, and ϵ_i is called the error for the ith object. The ϵ's are assumed to be statistically independent for the different objects, have expectation 0, all have the same variance σ^2, and are included in the model to explain the fact that not all objects whose X values are the same will have the same y value.

The aims of the regression analysis are twofold: 1) to determine which of the X's are important in explaining the behavior of y, and 2) to develop a prediction formula that can be used to find reasonable values of y for a new object for which we only know the X's.

An example of such a situation is the Hald data set included at the end of this report, wherein $n = 13$, $m = 4$, y is the amount of heat generated during the hardening of cement, and the X's are the amount of four different chemicals used in the production of the cement. In the data file, y is listed first followed by the four X's.

If we rewrite the multiple linear regression model above in matrix form as

$$\mathrm{y} = \mathrm{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where y is the $n \times 1$ vector of the y values, X is the $n \times (m+1)$ matrix of the X values with a column of 1's appended at the beginning, $\boldsymbol{\beta}$ is the $(m+1) \times 1$ vector of coefficients, and $\boldsymbol{\epsilon}$ is the $n \times 1$ vector of ϵ's, then it is easy to write down the basic results of a regression analysis. First, the best linear unbiased estimators $\hat{\boldsymbol{\beta}}$ of the β's and an unbiased estimator s^2 of σ^2 are given by

$$\hat{\boldsymbol{\beta}} = (\mathrm{X}^T\mathrm{X})^{-1}\mathrm{X}^T\mathrm{y}, \quad s^2 = \frac{RSS}{n-m-1}, \quad RSS = \mathrm{e}^T\mathrm{e}, \quad \mathrm{e} = \mathrm{y} - \hat{\mathrm{y}}, \quad \hat{\mathrm{y}} = \mathrm{X}\hat{\boldsymbol{\beta}},$$

where the vectors $\hat{\mathrm{y}}$ and e are called the vectors of fitted values and residuals, and RSS is the residual sum of squares.

If we further assume that the ϵ's are normally distributed, then the hypothesis that β_j is zero while the other β's are not is rejected at the α significance level if $|t_j| > t_{\alpha/2,n-m-1}$ where $t_{\alpha/2,n-m-1}$ is the upper $\alpha/2$ critical point of a t distribution having $n-m-1$ degrees of freedom and

$$t_j = \frac{\hat{\beta}_j}{\sqrt{s^2(\mathrm{X}^T\mathrm{X})^{-1}_{j+1,j+1}}}.$$

Note that the denominator of t_j is called the standard error of $\hat{\beta}_j$. Typically, instead of just reporting whether the hypothesis is rejected or not, one determines the p-value of $\hat{\beta}_j$, that is, the probability that a t with $n-m-1$ degrees of freedom is outside the interval $\pm t_j$, in which case the null hypothesis is rejected if $p_j < \alpha$.

A measure of how much of the variability in y is explained by its linear dependence on the X's is

$$R^2 = 1 - \frac{s_e^2}{s_y^2},$$

where s_e^2 and s_y^2 are the sample variances of the e's and the y's, respectively.

A Multiple Linear Regression Program Using SWEEP

In this section we give as an example a regression program which

1. Asks the user for a file name containing a regression data set to analyze and a file name to receive the results of the analysis. The program assumes that the file starts with a line containing a label to be displayed on the output and a line with the number of observations, n, and the number of independent variables, m, in the regression (not counting an intercept term). Each of the remaining n lines in the input file is assumed to contain the value of the dependent variable followed by the values of the independent variables. An example input file is given below which contains the so-called Hald Data which has 13 observations and four independent variables:

```
Hald Data (From Draper and Smith, pg. 296, 304, 630)
13 4
 78.5   7 26  6 60
 74.3   1 29 15 52
104.3  11 56  8 20
 87.6  11 31  8 47
 95.9   7 52  6 33
109.2  11 55  9 22
102.7   3 71 17  6
 72.5   1 31 22 44
 93.1   2 54 18 22
115.9  21 47  4 26
 83.8   1 40 23 34
113.3  11 66  9 12
109.4  10 68  8 12
```

2. Reads the data into the vector y and the matrix X (putting a column of 1's into the beginning of X so that it is $n \times (m+1)$ after reading the data), forms the augmented cross-product matrix

$$A = \begin{bmatrix} X^T X & X^T y \\ y^T X & y^y \end{bmatrix},$$

and then gets the regression information by using the matrix SWEEP algorithm.

3. The least squares estimates, their standard errors, t-statistics, and corresponding p-values are then written out, as well as the value of R^2. The result for the Hald Data are:

```
Hald Data (From Draper and Smith, pg. 296, 304, 630)

Regression results, R-Squared = 0.98237562, d.f. =   8

         i     beta(i)       se(i)        t(i)     pval(i)
     -------------------------------------------------------
         0   62.405369   70.070959    0.890602    0.399134
         1    1.551103    0.744770    2.082660    0.070822
         2    0.510168    0.723788    0.704858    0.500901
```

3	0.101909	0.754709	0.135031	0.895923
4	-0.144061	0.709052	-0.203174	0.844071

The Program

```
c  -------------------------------------------
c   REGSWP.F: Regression program using SWEEP
c
c   Uses functions inprod and smean and subroutines
c   tcdf, tpdf, betai, and gama
c  -------------------------------------------

c  ---------------------------------------------
c   Declare variables and external functions:
c  ---------------------------------------------

      character*60 fnin,fnout,label

      real*8 x(1000,20),y(1000),a(21,21),beta(21),se(21),t(21),pval(21)

      real*8 cdf,pdf,inprod,rss,s2,ybar,ssy,rsq

      real*8 smean

      logical fnexist

      data ndim/21/

c  ------------
c   Get data:
c  ------------

      write(*,10)
 10   format(' Enter file name containing the regression data: '$)

      read(*,20) fnin
 20   format(a60)

      inquire(file=fnin,exist=fnexist)

      if(.not.fnexist) go to 100

      write(*,30)
 30   format(' Enter the file name to get the output: '$)
      read(*,20) fnout

      open(2,file=fnout,status='new')
      open(1,file=fnin,status='old')

c   Read first two lines of input file:

      read(1,20,err=110) label
      read(1,*,err=120) n,np

c   Read data, putting X's in columns 2,...,m+1 and 1's in 1st column:

      m=np+1
      do 40 i=1,n
      x(i,1)=1.0
 40   read(1,*,err=130) y(i),(x(i,j),j=2,m)

c  ----------------------------------------
c   Form the augmented crossproduct matrix:
c  ----------------------------------------

      mp1=m+1
      do 60 i=1,m
             a(i,mp1)=inprod(x(1,i),y,n)
             a(mp1,i)=a(i,mp1)
             do 50 j=1,i
                    a(i,j)=inprod(x(1,i),x(1,j),n)
 50                 a(j,i)=a(i,j)
 60   continue
      a(mp1,mp1)=inprod(y,y,n)

c  ---------------------------------------------------------
c   Sweep it on first m diagonals and check for singularity:
c  ---------------------------------------------------------
```

```

      call sweep(a,ndim,mp1,1,m,ier)

      if(ier.eq.1) go to 90


c ---------------------------------------------------------------
c  Now get LSE's, RSS, R squared, se's, t-statistics, and p-values:
c ---------------------------------------------------------------

      rss=a(mp1,mp1)
      s2=rss/(n-m)

      ybar=smean(y,n)

      do 70 i=1,n
 70   ssy=ssy+(y(i)-ybar)**2

      rsq=1.-(rss/ssy)

      do 80 i=1,m
             beta(i)=a(i,mp1)
             se(i)=sqrt(s2*a(i,i))
             t(i)=beta(i)/se(i)
             call tcdf(dabs(dble(t(i))),n-m,cdf,pdf)
 80          pval(i)=2.*(1.-cdf)

c --------------------
c  write out results:
c --------------------

      write(2,81) label
 81   format(1x,a60)

      write(2,82) rsq,n-m
 82   format(/,' Regression results, R-Squared = ',f10.8,
     1          ', d.f. = ',i3)

      write(2,83)
 83   format(/,10x,'i',5x,'beta(i)',7x,'se(i)',8x,'t(i)',5x,'pval(i)',/
     1        6x,53(1h-))

      do 84 i=1,m
 84   write(2,85) i-1,beta(i),se(i),t(i),pval(i)
 85   format(6x,i5,4f12.6)
      go to 99

c -----------------
c  Error handling:
c -----------------

 90   write(*,*) ' Regression matrix singular'
      go to 99

100   write(*,*) ' File doesn''t exist'
      go to 99

110   write(*,*) ' Error reading label'
      go to 99

120   write(*,*) ' Error reading n and m'
      go to 99

130   write(*,135) i
135   format(' Error reading row ',i5)

 99   continue
      stop
      end
      subroutine sweep(a,mdim,m,k1,k2,ier)
c ---------------------------------------------------------------
c
c  Subroutine to sweep the mxm matrix a on its k1 st thru k2 th
c  diagonals.  ier is 1 if a is singular. mdim is row dimension
c  of a in calling routine.
c
c ---------------------------------------------------------------

      double precision a(mdim,1)
      double precision d
```

```

      ier=1

      do 50 k=k1,k2

             if(abs(a(k,k)).lt.1.e-20) return

             d=1./a(k,k)
             a(k,k)=1.
             do 10 i=1,m
             a(k,i)=d*a(k,i)
 10          if(i.ne.k) a(i,k)=-a(i,k)*d

             do 20 i=1,m
             do 20 j=1,m
 20          if((i.ne.k).and.(j.ne.k)) a(i,j)=a(i,j)+a(i,k)*a(k,j)/d

 50    continue

       ier=0

       return
       end

       double precision function inprod(x,y,n)
       double precision x(n),y(n)

       inprod=0.0d0
       do 10 i=1,n
 10    inprod=inprod+x(i)*y(i)

       return
       end

       double precision function smean(x,n)
       double precision x(n)

       smean=0.0d0
       do 10 i=1,n
10     smean=smean+x(i)
       smean=smean/n

       return
       end

```

The External Statement and Numerical Integration

The Common and Named Common Statements and Newton-Raphson

See the chapter on "Iterative Methods for Parameter Estimation" for examples of using the common and named common statements.