

2

Introduction to Modern Fortran

This chapter and the next provide a crash course in modern Fortran. Some knowledge of programming, such as mild experience with FORTRAN 77, Basic, or C, will be helpful but is not absolutely required. These chapters cover the basic syntax of Fortran and features of the language that are most useful for statistical applications. We do not attempt to provide a comprehensive reference to Fortran 95. For example, we do not list all of the available edit descriptors or intrinsic functions and subroutines. Those details are readily available in reference manuals distributed with Fortran compilers. Rather, we focus on larger concepts and strategies to help the reader quickly build familiarity and fluency.

2.1 Getting Started

2.1.1 *A Very Simple Program*

A simple Fortran program that generates uniform random numbers is shown below.

```
----- uniform1.f90 -----  
!#####  
program uniform  
  ! Generates random numbers uniformly distributed between a and b  
  ! Version 1  
  implicit none  
  integer :: i, n  
  real :: a, b, u
```

```

print "(A)", "Enter the lower and upper bounds:"
read(*,*) a, b
print "(A)", "How many random numbers do ya want?"
read(*,*) n
print "(A)", "Here they are:"
do i = 1, n
    call random_number(u)
    print *, a + u*(b-a)
end do
end program uniform
!#####

```

We have displayed the source code within a box to indicate that this example appears within a file maintained on the book's Web site; in this case, the file is named `uniform1.f90`. The `integer` and `real` statements declare that the variables `a`, `b`, and `u` are to be regarded as floating-point real numbers, whereas `i` and `n` are integers. Understanding the differences between these types of variables is crucial. Generally speaking, real variables are used for data storage and computational arithmetic, whereas integer variables are used primarily for counting and for defining the dimensions of data arrays and indexing their elements. Readers with programming experience may already understand the purpose of the `read` and `print` statements and the meaning of the `do` construct, but these will be explained later in this section. We will also explain `random_number`, a new Fortran intrinsic procedure for generating pseudorandom variates.



Style tip

Notice the use of `implicit none` in this program. This statement overrides the implicit typing of variables used in many old-fashioned FORTRAN programs. Under implicit typing,

- a variable beginning with any of the letters `i`, `j`, `...`, `n` is assumed to be of type `integer` unless explicitly declared otherwise, and
- a variable beginning with any other letter is assumed to be `real` unless explicitly declared otherwise.

Modern Fortran still supports implicit typing, but we strongly discourage its use. With implicit typing, misspellings cause additional variables to be created, leading to programming errors that are difficult to detect. Placing the `implicit none` statement at the beginning forces the programmer to explicitly declare every variable. We will use this statement in all of our programs, subroutines, functions, and modules.

2.1.2 *Fixed and Free-Form Source Code*

Readers familiar with FORTRAN 77 may notice some differences in the appearance of the source code file. In old-fashioned FORTRAN, source lines could not exceed 72 characters. Program statements could not begin before column 7; column 6 was reserved for continuation symbols; columns 1–5 held statement labels; and variable names could have no more than six characters. These rules, which originated when programs were stored on punch cards, make little sense in today’s computing environments.

Modern Fortran compilers still accept source code in the old-fashioned fixed format, but that style is now considered obsolescent. New code should be written in the free-form style introduced in 1990. The major features of the free-form style are:

- program statements may begin in any column and may be up to 132 characters long;
- any text appearing on a line after an exclamation point (!) is regarded as a comment and ignored by the compiler;
- an ampersand (&) appearing as the last nonblank character on a line indicates that the statement will be continued on the next line;
- variable names may have up to 31 characters.

As a matter of taste, most programmers use indentation to improve the readability of their code, but this has no effect on program behavior. Fortran statements and names are not case-sensitive; the sixth line of the uniform generator could be replaced by

```
Integer :: I, N
```

without effect.

By convention, source files whose names end with the suffix `*.f`, `*.for` or `*.f77` are expected to use fixed format, whereas files named `*.f90` or `*.f95` are assumed to follow the free format. Programs consisting of multiple source files, some with fixed format and others with free format, are acceptable; however, you are not allowed to combine these two styles within a single file. Some compilers expect all free-format source files to have the `.f90` filename extension, even those that use new features introduced in Fortran 95. For this reason, all of the example source files associated with this book have names ending in `.f90`, even if they contain features of Fortran 95 that are not part of the Fortran 90 standard.

2.1.3 *Compiling, Linking and Running*

Before a program can be run, the Fortran code must be converted into sequences of simple instructions that can be carried out by the computer’s

processor. The conversion process is called building. Building an application requires two steps: compiling, in which each of the Fortran source-code files is transformed into machine-level instructions called object code; and linking, in which the multiple object-code files are collected and connected into a complete executable program.


Building can be done at a command prompt, but the details vary from one compiler to another. Some will compile and link with a single command. For example, if you are using Lahey/Fujitsu Fortran in a Windows environment, the program `uniform1.f90` can be compiled and linked by typing


```
lfc uniform1.f90
```

at the command line. Once the application is built, the file of executable code is ready to be run. In Windows, this file is typically given the `*.exe` suffix, whereas in Unix or Linux it may be given the default name `a.out`. The program is usually invoked by typing the name of the executable file (without the `*.exe` suffix, if present) at a command prompt.

Many compilers are accompanied by an Integrated Development Environment (IDE), a graphical system that assists the programmer in editing and building programs. A good IDE can be a handy tool for managing large, complex programs and can help with debugging. The IDE will typically include an intelligent editor specially customized for Fortran and providing automatic indentation, detection of syntax errors, and other visual aids such as coloring of words and symbols.

For example, to build the `uniform` program in the Microsoft Visual Studio .NET 2003 IDE, using Intel Visual Fortran 8.0, begin by creating a new project. Select **File** → **New** → **Project...** from the menu, and the New Project dialog window appears. Specify a Console Application project, and name the project `uniform` (Figure 2.1). The next window will prompt for further information; select “Application Settings” and choose “Empty Project” as the console application type (Figure 2.2). Next, add the file `uniform1.f90` to the project in the “Solution Explorer” under “Source Files.” To do this, right-click on the folder “Source Files,” choose **Add** → **Add existing item...** from the pop-up menu, and select the source file (or files) to add.

In the “Solution Explorer,” double-click on the source file’s name to open the file in the code editor (Figure 2.3). To build the program, select **Build** → **Build uniform** from the menu, or press the “Build” button, . The IDE will provide output from the build, indicating success or failure. If a compiler error occurred—due to incorrect syntax, for example—the IDE will direct you to the location of the error in the source-code editor.

Once the program has been successfully built, it can be executed from within the IDE by selecting **Debug** → **Start** from the menu, by pressing the “Start” button, , or by pressing the F5 keyboard key.

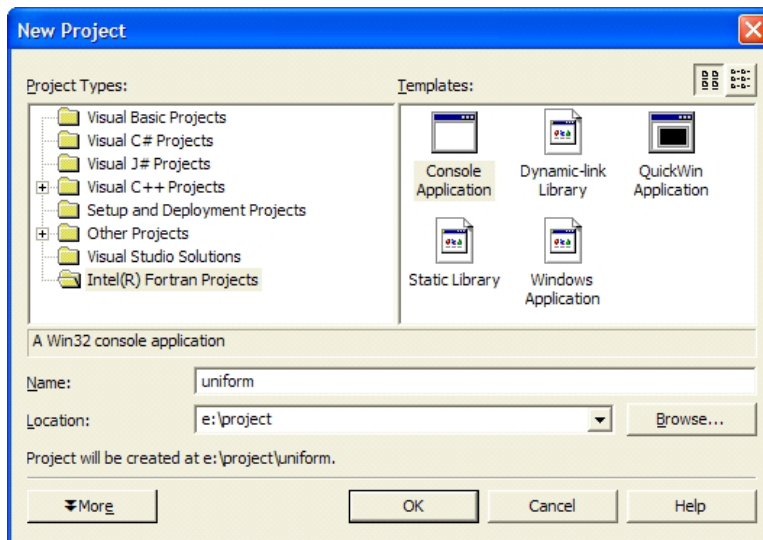


FIGURE 2.1. Intel Fortran New Project dialog.

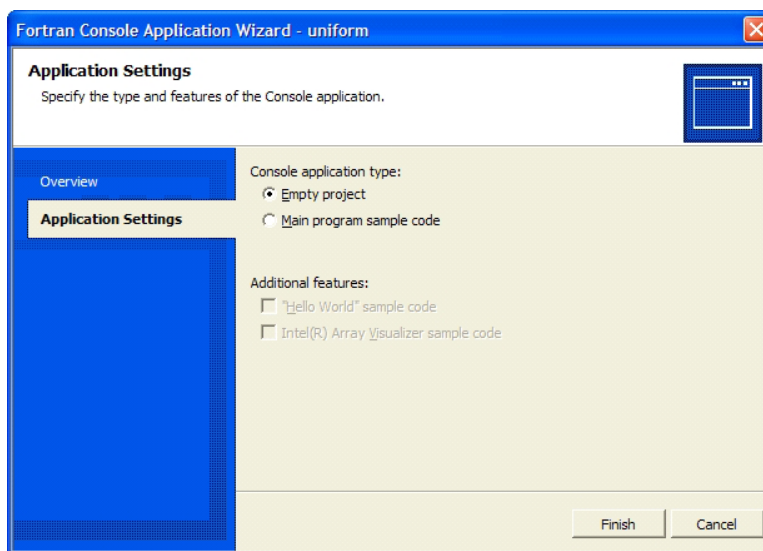


FIGURE 2.2. Selecting an “Empty Project”.

2.1.4 Compiler Options

Most implementations of Fortran allow the developer to choose among a wide variety of options when the code is compiled. Some of these options are of minor importance and primarily a matter of personal taste—for exam-

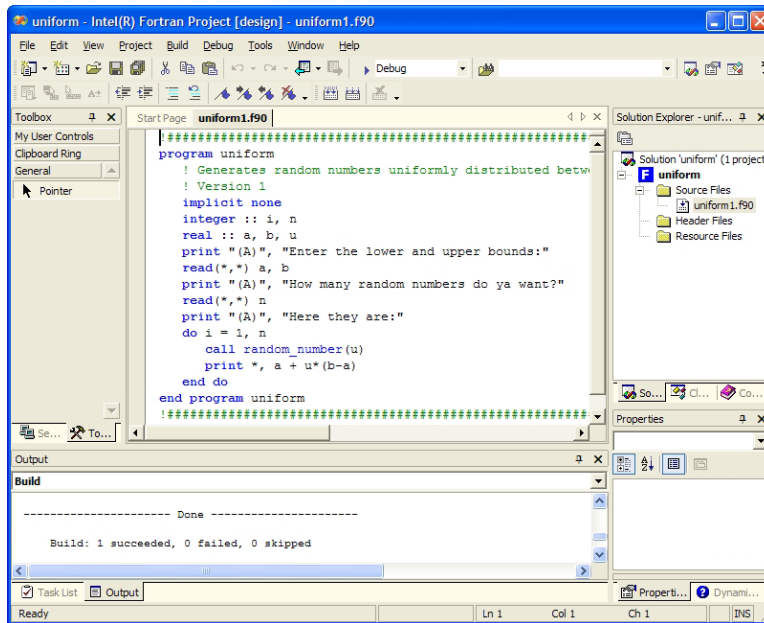


FIGURE 2.3. The Visual Studio IDE.

ple, whether the messages appearing on the screen during the compilation process will be brief or verbose. Other options are more crucial and can have a significant impact on program behavior. The optimal choice of options will vary depending on whether the program is under development and needs to be continually checked and modified or has been tested and validated and is basically ready for use and release. In the former situation, which may be called “debugging mode,” the following features tend to be extremely helpful:

- options that, when the program crashes, tell the programmer the exact line(s) in the source code that produced the infraction; and
- safeguards that check whether the physical bounds of an array are exceeded at run time (e.g., the program tries to access `x(4)` when `x` is an array of size 3).

An IDE, such as Microsoft Visual Studio, usually provides two possible build configurations: debug and release. The programmer can select the desired configuration before building and executing the application. Whenever the debug configuration is used, the IDE can step through your application’s source code one line at a time. It also allows the programmer to watch the values of the variables (including arrays) at each step. The programmer can set breakpoints in the editor window at which execution

will automatically pause and then step into subprocedures. These features are extremely helpful for identifying and fixing code errors.

Because these special features embed extra code into a program and cause it to run slowly, they should be turned off when the program is compiled for release. When compiling in release mode, the most useful options are the optimization settings that help to increase execution speed. Optimization features will be discussed in Chapter 4.

2.1.5 *Standard Input and Output*

Like many simple textbook example programs, **uniform** accepts information from standard input (the computer keyboard) and provides results to standard output (the screen). More elaborate programs that receive and send information via files will appear in Chapter 5. A command-line session that runs the program is displayed below.

```
d:\jls\software\demos>uniform1
Enter the lower and upper bounds:
0 100
How many random numbers do ya want?
5
Here they are:
3.9208680E-05
2.548044
35.25161
66.69144
96.30555
```

In this simple program, the statements that perform input/output (I/O) are easy to understand. In the statement

```
print "(A)", "Enter the lower and upper bounds:"
```

"(A)" is a format string indicating that the value of the subsequent expression will be printed verbatim as alphanumeric or character data. In

```
read(*,*) a, b
```

the first asterisk (*) indicates that the data will come via standard input, and the second means that the values for **a** and **b** will be read in a list-directed fashion. In list-directed input—also commonly known as “free format” input—multiple items may be separated by any amount of white space (blank spaces, tabs, commas, or new line characters). If one were to apply list-directed input to character variables, like

```
character(len=15) :: first_name, last_name
read(*,*) first_name, last_name
```

the user who interacts with the program would have to enclose the string values in single or double quotes:

```
"Bozo" "The Clown"
```

On the other hand, if one specifies alphanumeric input as in

```
read(*,"(A)") first_name
read(*,"(A)") last_name
```

then the user may type

```
Bozo
The Clown
```

without quotation marks, and any leading or trailing spaces on the input lines will be retained in the string variables. String variables will be discussed in greater detail in Section 2.4.

2.1.6 *Intrinsic Uniform Generator*

Program `uniform` includes a call to the intrinsic random number generator,

```
call random_number(u)
```

which sets the argument `u` to a pseudorandom real number x such that $0 \leq x < 1$. Every version of Fortran 95 has a uniform random generator that can be called using this syntax, but the details of the implementation vary; compiling and running this program on different platforms could produce different number streams. The argument to `random_number` may also be a real array, in which case the entire array would be filled with random numbers.

One feature of `random_number` is that, if the programmer does not explicitly initialize the generator with one or more integer seed values, it will be seeded with processor-dependent integers and produce the same stream of numbers each time the program is run. To produce different results each time, we may include the following line at the beginning of the program, which sets the seeds to values determined by a call to the system clock:

```
call random_seed()
```

This intrinsic subroutine `random_seed` can also be used to set the seeds to specific values and to query the system to find the current state of the seeds, but the details of this usage may vary slightly from one compiler to another; consult your reference manual for details.

2.1.7 *Integer and Real Kinds*

In old-fashioned FORTRAN programs, floating-point real variables were declared as `real` or `double precision`, the latter being used when round-

TABLE 2.1. Kind parameters for real variables in popular compilers.

	Kind	Bits	Precision (digits)	Comments
Lahey/Fujitsu	4	32	6	default kind
	8	64	15	double precision
	16	128	33	
Salford	1	24	6	default kind
	2	53	15	double precision
	3	64	18	
Intel	4	32	6	default kind
	8	64	15	double precision
	16	128	33	VMS/Unix/Linux only

off error was a potentially serious issue (e.g., when accumulating sufficient statistics over large datasets). A **real** variable typically occupied up to 32 bits of storage and was precise to 6 significant digits, whereas a **double precision** variable used up to 64 bits and was precise to at least 15 digits.

The **real** and **double precision** declarations still work, but now the preferred way to determine the precision of real variables is through a **real** statement with the optional **kind** parameter. A variable may now be declared as

```
real(kind=i) :: x
```

or, equivalently, as

```
real(i) :: x
```

where **i** is a positive integer that determines the overall length of the datum in memory. The available floating-point models and the values of **kind** that correspond to them vary from one compiler to another. The **kind** parameters for some popular compilers are shown in Table 2.1. Omitting the parameter produces a real variable of the default kind.

Integer variables also have **kind** parameters. On most compilers, the default representation is a 32-bit integer that, because the first bit is used for signing, has a maximum value of $2^{31} - 1 = 2,147,483,647$. (In older FORTRAN compilers, the 32-bit representation was often called a “long integer” and had to be specially requested; now it is typically given by default.) The **kind** parameters for integers in some popular compilers are shown in Table 2.2.

To write source code that migrates well from one compiler or platform to another, it is essential to avoid setting the **kind** parameters to specific numeric values such as 3 or 4. This leaves us with two options. The first is to leave **kind** unspecified and pray that the default data types are sufficiently large and precise to yield good results. The second, and clearly

TABLE 2.2. Kind parameters for integer variables in popular compilers.

	Kind	Bits	Maximum value	Comments
Lahey/Fujitsu	1	8	127	default kind
	2	16	32,767	
	4	32	2,147,483,647	
	8	64	9,223,372,036,854,775,807	
Salford	1	8	127	default kind
	2	16	32,767	
	3	32	2,147,483,647	
Intel	1	8	127	default kind
	2	16	32,767	
	4	32	2,147,483,647	
	8	64	9,223,372,036,854,775,807	

more desirable, option is to create your own global parameters for desired kinds and use them consistently throughout the source code. For example, if you are using Intel Visual Fortran, you can specify

```
integer, parameter :: my_int = 4
```

and then declare your integers as

```
integer(kind=my_int) :: i, j, k
```

in every procedure. If you want to compile the same code in Salford, you would only need to change the value of `my_int` from 4 to 3. To make the `kind` parameters accessible to all parts of the program, it is convenient to keep them in a module; this will be described in Section 3.2. Modern Fortran also has two intrinsic functions, `selected_real_kind` and `selected_int_kind`, which can be used to query the compiler and set the `kind` parameters automatically; use of these functions will be described in Chapter 4.

2.1.8 Do, if, case, goto

Iteration in the `uniform` program is carried out by a `do` loop, one of the most frequently used programming constructs:

```
do i = 1, n
  call random_number(u)
  print *, a + u*(b-a)
end do
```

In this `do` loop, `i` is the `do` variable. A Fortran 95 `do` variable must be an integer. The loop continues to execute as long as the value of the `do` variable is less than or equal to the upper limit `n`. By default, the `do` variable

is incremented by one at each cycle, but the increment may be changed by including an optional integer after the upper limit:

```
do i = 2, 2*n, 2    ! gives exactly the same result
  call random_number(u)
  print *, a + u*(b-a)
end do
```

A `do` loop may be written without a `do` variable, but the programmer must then provide another way to stop the execution at the desired moment. One way is to include a `while` statement followed by a logical expression:

```
i = 0
do while (i < n)
  i = i + 1
  call random_number(u)
  print *, a + u*(b-a)
end do
```

Another way is to use an `if` construct with `exit`:

```
i = 0
do
  i = i + 1
  call random_number(u)
  print *, a + u*(b-a)
  if( i == n ) exit
end do
```

The `exit` statement causes the point of execution to jump down to the line immediately following the next `end do` statement; therefore, `exit` within a nested `do` loop will cause the program to jump out of the innermost loop. A close relative of `exit` is `cycle`, which causes the point of execution to jump up to the `do` statement above it and begin another cycle of the innermost loop.



Style tip

Notice that in the last two examples, `i` and `n` were compared by the relational operators `<` and `==` rather than the old FORTRAN operators `.lt.` and `.eq.`. The old-fashioned operators still work, but the new ones are preferred. The old and new relational operators are shown in Table 2.3.

The syntax of the Fortran `if` construct is essentially unchanged from previous versions. Conditional execution of a single statement looks like

```
if( logical-expression ) statement
```

whereas conditional execution of a group of statements looks like

TABLE 2.3. Relational operators in Fortran.

Old	New	Meaning
.lt.	<	less than
.le.	<=	less than or equal to
.eq.	==	equal to
.ne.	/=	not equal to
.gt.	>	greater than
.ge.	>=	greater than or equal to

```

if( logical-expression ) then
    statement
    statement
    :
end if

```

or like this:

```

if( logical-expression ) then
    statement
    :
else
    statement
    :
end if

```

Because the `if` construct depends on a logical expression, it can conditionally execute only one or two groups of program statements. A more flexible structure that handles any number of conditions is the `case` construct,

```

select case( expression )
    case( value-1 )
        statement
        :
    case( value-2 )
        statement
        :
    case default
        statement
        :
end select

```

where *expression* is a variable or an expression that evaluates to an integer, logical value, or character string, and *value-1*, *value-2*, ... are some or all of the possible values that *expression* may take. The optional `case default` statement designates actions to be taken if *expression* evaluates to some-

thing other than *value-1*, *value-2*, A programming example that uses the **case** construct will appear in Section 3.6.

Numbered statements have become less common in Fortran, but they are still useful, particularly in conjunction with **goto**. The use of **goto** is commonly denounced, with some authors saying that it should never be used under any circumstances. For the most part, we agree. Frequent use of **goto** can make a program very difficult to understand and hard to debug. On the other hand, we find **goto** to be a prudent way to trap errors. An **if** statement with **goto** can be used to detect an error condition and immediately jump to the bottom of the procedure to report the error and exit. The same effect could be achieved without **goto**, of course, but **goto** enables us to collect all the error-handling statements into one section to keep our code clean and legible. The use of **goto** for error handling will be illustrated at the end of Chapter 3. In this book, we use **goto** for trapping errors but not for any other purpose.

2.1.9 Exercises

1. Write a simple program that converts temperatures from Fahrenheit to Celsius and vice versa using $C = 5(F - 32)/9$ and $F = 9C/5 + 32$.
2. Modify one of the uniform generator programs to generate random samples from an exponential density

$$f(y) = \lambda e^{-\lambda y}, \quad y > 0. \quad (2.1)$$

Use the inverse-cdf method $y = F^{-1}(u)$, $u \sim U(0, 1)$, where F is the cumulative distribution function.

3. Write a program that simulates rolling a pair of six-sided dice. It should produce 2 with probability $1/6$, 3 with probability $2/36$, and so on. Then describe a strategy for generalizing the result to rolling n dice, each with k sides.
4. Write a program that simulates random draws from the negative binomial distribution

$$P(Y = y) = \binom{y-1}{r-1} p^r (1-p)^{y-r},$$

$y = r, r+1, r+2, \dots$ for any given $p \in (0, 1)$ and positive integer r . Use the fact that Y is the number of Bernoulli trials required to obtain r successes, with p as the per-trial success rate.

5. Fortran 95 introduced the new intrinsic subroutines **date_and_time**, **system_clock**, and **cpu_time**. Learn how to use them and demonstrate each one with a simple programming example.

6. If the birthdays of n individuals are independently and uniformly distributed over 365 days of the year, then the probability that at least two individuals share the same birthday is

$$1 - \prod_{j=1}^n \left(\frac{365 - j + 1}{365} \right).$$

Write a program that computes and prints these probabilities for $n = 2, 3, \dots, 365$.

2.2 Arrays

2.2.1 Rank, Size and Shape

Like its predecessors, modern Fortran supports arrays of logical, integer, real, complex, and character data. A real vector $y = (y_1, y_2, \dots, y_{10})$ with ten elements may be declared as

```
real :: y
dimension :: y(10)
```

as

```
real, dimension(10) :: y
```

or like this:

```
real :: y(10)
```

Once this array has been created, we refer to the individual elements as $y(1)$, $y(2)$, \dots , $y(10)$. The dimension of an array can also be declared as an integer pair $i:j$, where i is the lower bound (the subscript of the first element) and j is the upper bound (the subscript of the last element). For example,

```
real :: y(0:14)
```

creates an array with elements $y(0)$, $y(1)$, \dots , $y(14)$. Even more generally, an array can be dimensioned as $i:j:k$, where k is the increment between the subscript values of successive elements; for example,

```
real :: y(0:6:2)
```

has elements $y(0)$, $y(2)$, $y(4)$, and $y(6)$.

Multiple dimensions of an array are separated by commas. For example, a matrix with five rows and three columns may be dimensioned as

```
real :: x(5,3)
```

and its elements will then be $x(1,1)$, $x(1,2)$, and so on. Fortran arrays may have up to seven dimensions. The number of dimensions is called the rank. The rank of an array is fixed by the compiler and cannot be changed while a program is running. The Fortran intrinsic function `rank` returns the rank of its argument; if x is declared as shown above, then `rank(x)` evaluates to the integer 2.

The functions `lbound` and `ubound` return the lower or upper bounds of an array along any or all dimensions. If we dimension an array as

```
integer :: iarr(2,0:50)
```

then `lbound(iarr,1)` is 1; `ubound(iarr,1)` is 2; `lbound(iarr,2)` is 0; `ubound(iarr,2)` is 50; `lbound(iarr)` is an integer array with elements 1 and 0; and `ubound(iarr)` is an integer array with elements 2 and 50.

The size of an array is the number of elements in the entire array or the number of elements along a single dimension. The size of an array can be queried through the intrinsic function `size`. If we declare

```
real :: mat(0:5,10)
```

then `size(mat)` returns 60, `size(mat,1)` returns 6, and `size(mat,2)` returns 10. Taken together, the sizes of all dimensions of an array are called its shape. The intrinsic function `shape` returns the shape of an array as a rank-one array of integers; in this case, `shape(mat)` is an integer array with elements 6 and 10.

2.2.2 Array Functions

The intrinsic functions `maxval`, `minval`, `sum`, and `product` can find the minimum, maximum, sum, or product of all the elements across a whole array or across any of its dimensions. These functions share the same basic syntax. For example, if x is a 2×3 real array, then:

- `sum(x,1)` returns a rank-one real array of size 3 whose elements are $x(1,1) + x(2,1)$, $x(1,2) + x(2,2)$ and $x(1,3) + x(2,3)$;
- `sum(x,2)` returns a rank-one real array of size 2 whose elements are $x(1,1) + x(1,2) + x(1,3)$ and $x(2,1) + x(2,2) + x(2,3)$; and
- `sum(x)` returns a real scalar equal to the sum of all six elements of x .

These functions can also be applied to integer arrays, in which case the values returned will be of the integer type.

The functions `all`, `any`, and `count` perform similar operations on logical arrays. If x is a logical array, then `all(x)` returns `.true.` if every element of x is `.true.`; `any(x)` returns `.true.` if at least one of the elements of x is `.true.`; and `count(x)` returns an integer value equal to the number of elements of x that are `.true.`

The generic intrinsic functions `transpose` and `matmul` return whole arrays whose size, type, and kind depend on those of the arguments you provide. If \mathbf{x} is a $q \times r$ array, then `transpose(x)` is an $r \times q$ array of the same type and kind. If \mathbf{x} is $q \times r$ and \mathbf{y} is $r \times s$, then `matmul(x,y)` is $q \times s$. Matrix multiplication usually involves real arrays, but it can also be applied to arrays of integers or logical values, with the result being integer or logical. Before invoking `transpose` or `matmul`, make sure that you have available an array of the correct type and shape to hold the result. The `matmul` function can also be used to multiply a matrix by a vector or a vector by a matrix, but at least one of the arguments must have rank two; multiplication of vectors of the same length is accomplished by `dot_product`, which returns a scalar. Note that `matmul` is designed for arbitrary arguments and may be inefficient for problems with special structure—for example, computing symmetric matrices such as $X^T X$ or multiplying matrices that are known to be diagonal, banded, or triangular.

2.2.3 Operations on Arrays and Array Sections

Modern Fortran can perform some intelligent assignment and comparison operations on entire arrays. If \mathbf{x} is an array, then

```
x = 0
```

sets each element of \mathbf{x} to zero. If \mathbf{x} and \mathbf{y} have the same shape, then

```
x = y
```

sets each element of \mathbf{x} equal to the corresponding element of \mathbf{y} . To check the arrays for conformity, the previous statement could be preceded by

```
if( .not. all( shape(x)==shape(y) ) ) goto 100
```

with subsequent statements to handle the error condition. The result of a comparison operation applied to two arrays of the same shape produces a logical array of that shape. In the previous example, `shape(x)==shape(y)` evaluates to a logical array indicating, for each dimension, whether the sizes of \mathbf{x} and \mathbf{y} match.

In a similar way, operations can be performed on array sections. The colon (`:`) denotes the entire extent of an array along a particular dimension; for example,

```
sum( x(i,:) * y(:,j) )
```

or

```
dot_product( x(i,:), y(:,j) )
```

computes the inner product of the i th row of \mathbf{x} with the j th column of \mathbf{y} . A colon preceded or followed by integers limits the range of elements along a dimension. If \mathbf{x} is a rank-one array of size \mathbf{n} , then

- `x(2:4)` refers to `x(2)`, `x(3)`, and `x(4)`,
- `x(:2)` refers to `x(1)` and `x(2)`, and
- `x(1:)`, `x(:n)`, `x(:)`, and `x` are equivalent.

Assignment statements that operate on arrays or array sections are convenient for keeping source code concise, but on some systems they may also yield benefits in terms of efficiency because they allow the compiler to choose an order for the elementwise operations to optimize execution. The new Fortran 95 statements **where** and **forall** also signal to the compiler that a group of elementwise operations may be performed in any order for optimal performance. The **forall** statement can replace any **do** loop in which the order of the operations is unimportant. For example, a sum of squared deviations $\sum_{i=1}^n (y_i - a)^2$ can be computed as

```
ssdev = 0.
do i = 1, n
    ssdev = ssdev + ( y(i) - a )**2
end do
```

as

```
ssdev = 0.
forall( i = 1:n ) ssdev = ssdev + ( y(i) - a )**2
```

or like this:

```
ssdev = sum( ( y - a )**2 )
```

In the last statement, the scalar value in `a` is implicitly broadcast to an array of the same shape as `y` prior to the computation of `y-a`.

The **where** statement carries out an operation on selected array elements corresponding to the **.true.** values in another array of logical values. For example, if `n` is an integer array and `x` is a real array of the same shape, then

```
where( n == 2 ) x = 0.
```

sets each element of `x` to zero only if the corresponding element of `n` is 2.

2.2.4 Your Mileage May Vary

Programmers who use array functions such as **sum**, **dot_product**, and **matmul** sometimes find that the resulting performance is slower than if they code these operations themselves using **do** loops, especially if they pay attention to issues of data storage, loop order, and stride (see Section 4.3). These Fortran intrinsics are guaranteed to work in implementations of the Fortran 95 standard, but they are not guaranteed to be fast. In response, some vendors have begun to offer specially optimized versions of

these functions for users who demand high performance. Similarly, the new statements **where** and **forall** were designed to promote efficiency, but depending on your compiler and your computer's architecture, you may not see any improvement over expressions involving whole arrays or array sections with colon notation. You may not even see any improvement over conventional **do** loops because many compilers already have optimization features that can reorganize these loops.

Given these complexities, we do not categorically advise programmers to apply all of these new features in all of their coding. On the other hand, we do not want to discourage their use either because the architects of the Fortran standard had good reasons for adding these features to the language; even when they do not increase efficiency, they may have other important benefits (e.g., keeping source code clean and concise). If you are migrating from FORTRAN 77 to Fortran 95, don't feel obligated to revise your existing code to use these features, as traditional **do** loops still work and are not going away in the foreseeable future.

2.2.5 Array Allocation

One of the major advances since FORTRAN 77 is that the size of an array no longer has to be determined at compilation time; it can now be established while the program is running. This feature, called dynamic allocation, is illustrated by the revised uniform generator program below.

uniform2.f90

```
#####
program uniform
  ! Generates random numbers uniformly distributed between a and b
  ! Version 2
  implicit none
  integer :: i, n
  real :: a, b
  real, allocatable :: x(:)
  print "(A)", "Enter the lower and upper bounds:"
  read(*,*) a, b
  print "(A)", "How many random numbers do ya want?"
  read(*,*) n
  allocate( x(n) )
  call random_number(x)
  x = a + x*(b-a)
  print "(A)", "Here they are:"
  do i = 1, n
    print *, x(i)
  end do
  deallocate(x) ! not really necessary
end program uniform
#####
```

In this program,

```
real, allocatable :: x(:)
```

declares **x** to be a rank-one array of real numbers whose size has not yet been determined. The statement

```
allocate( x(n) )
```

sets aside a section of the heap (the currently unused portion of memory) large enough to hold **n** real numbers. The call to **random_number** then fills the entire array with random variates.

If the **allocate** statement were removed, this program could crash at the **random_number** call because the size of **x** and its location in memory would be undefined. Similarly, applying the intrinsic functions **size**, **shape**, **lbound**, or **ubound** to an allocatable array that has not yet been allocated may cause a program to crash. To prevent crashes, you can test the array beforehand with the logical function **allocated(x)**, which returns **.true.** if **x** has been allocated and **.false.** otherwise.

Although **allocate** reserves memory locations, it may not initialize them, so the contents of an array immediately after allocation may not be meaningful. Therefore, after allocating an array, one should not attempt to view or print its contents, use it in an expression, or let it appear on the right-hand side of an assignment statement until the array has been filled with data; doing so could lead to unpredictable results.

The **deallocate** statement returns memory to the heap. Explicit deallocation of arrays at the end of a program is usually unnecessary because the operating system does that automatically. It tends to be good practice to explicitly deallocate arrays, however, as it forces the programmer to pay greater attention to issues of memory management. To change the size of an allocated array, you need to first deallocate and then reallocate. Applying **deallocate** to an array that has not yet been allocated will cause a crash, so if the allocation status is in doubt, be sure to check it first with the **allocated** function.

2.2.6 Exercises

1. Consider the arrays declared below.

```
real :: x(10,4)
integer, dimension(0:5,0:4,0:3) :: y
logical :: larr(0:100:4)
```

What are the returned values of **rank(y)**, **size(larr)**, **lbound(x,1)**, **ubound(y)**, and **shape(y)**?

2. Write a program that draws a random sample of size n without replacement from the population $\{1, 2, \dots, N\}$ for any $N \geq 1$ and $n \leq N$.

3. Using allocatable arrays, write a program that tabulates and prints the binomial probability mass function

$$f(y) = \binom{n}{y} p^y (1-p)^{n-y}, \quad y = 0, 1, \dots, n,$$

and the cumulative distribution $F(y) = \sum_{z=0}^y f(z)$ for user-specified values of n and p . Use double-precision arithmetic and take reasonable measures to avoid overflows in the computation of $n!$ for large n .

2.3 Basic Procedures

2.3.1 Subroutines

Only very short programs can be effectively written as a single unit. Subroutines became a part of FORTRAN in 1958 and are indispensable for program organization. A re-revised version of our uniform random generator that uses a subroutine is shown below.

```

                                uniform3.f90
#####
subroutine fill_with_uniforms(vec_len, vec, lower, upper)
  ! Fills the rank-one array vec with random numbers uniformly
  ! distributed from lower to upper
  implicit none
  ! declare arguments
  integer, intent(in) :: vec_len
  real, intent(in) :: lower, upper
  real, intent(out) :: vec(vec_len)
  ! begin
  call random_number(vec)
  vec = lower + vec * ( upper - lower )
end subroutine fill_with_uniforms
#####
program uniform
  ! Generates random numbers uniformly distributed between a and b
  ! Version 3
  implicit none
  integer :: i, n
  real :: a, b
  real, allocatable :: x(:)
  print "(A)", "Enter the lower and upper bounds:"
  read(*,*) a, b
  print "(A)", "How many random numbers do ya want?"
  read(*,*) n
  allocate( x(n) )
  call fill_with_uniforms(n, x, a, b)
  print "(A)", "Here they are:"

```

```

do i = 1, n
  print *, x(i)
end do
deallocate(x) ! not really necessary
end program uniform
#####

```

In this example, `vec_length`, `vec`, `lower`, and `upper` are the dummy arguments, whereas `n`, `x`, `a`, and `b` are the actual arguments. Notice that each of the dummy arguments in `fill_with_uniforms` has an `intent` attribute, a highly useful feature introduced in Fortran 90. A dummy argument may be declared to be:

- `intent(in)`, which means that it is considered to be an input and its value may not be changed within the procedure;
- `intent(out)`, which means that it functions only as an output, and any data values contained in the actual argument just before the subroutine is called are effectively wiped out; or
- `intent(inout)`, which means that the argument may serve as both input and output.

If the `intent` attribute is not specified, it is automatically taken to be `inout`. Explicitly declaring the `intent` for each dummy variable is an excellent practice because it forces the programmer to be more careful and instructs the compiler to prevent unwanted side effects. A program will not compile successfully if, for example, an `intent(in)` dummy argument appears on the left-hand side of an assignment statement. We will follow the practice of explicitly declaring the `intent` for all dummy arguments in our subroutines and functions. (The only exception to this rule is pointers, to be discussed in the next chapter; Fortran 95 does not allow `intent` for pointer arguments.)

2.3.2 Assumed-Shape and Optional Arguments

In the previous example, notice that `vec_length`, the size of `vec`, is being passed as an argument. In old-fashioned FORTRAN programs, the dimensions of all array arguments had to be arguments themselves. In modern Fortran, this is no longer the case; we can remove `vec_length` from the argument list and make it a local variable, like this:

```

subroutine fill_with_uniforms(vec, lower, upper)
  ! Fills the rank-one array vec with random numbers
  ! uniformly distributed from lower to upper
  implicit none
  ! declare arguments

```

```

real, intent(in) :: lower, upper
real, intent(out) :: vec(:)
! begin
call random_number(vec)
vec = lower + vec * ( upper - lower )
end subroutine fill_with_uniforms

```

The dummy argument `vec` is now called an assumed-shape array; it derives its shape from the actual argument being passed. If the size of the array had been needed within the procedure, we could have obtained it with the intrinsic function `size` or `shape`.

Using assumed-shape arrays is an excellent way to shorten argument lists, reducing the chance that the subroutine will be called incorrectly. However, if we simply insert this new version of `fill_with_uniforms` into `uniform3.f90` and remove the actual argument `n` from the `call` statement, the program will not work. The reason is that any function or subroutine that uses assumed-shape arrays must have an explicit interface. The explicit interface, a concept introduced in Fortran 90, is a new set of rules governing how actual and dummy arguments interact. Among other things, it forces the dummy and actual arguments to agree in type, kind, and rank. The implicit interface used by old-fashioned FORTRAN procedures had minimal checking, which made certain kinds of programming errors difficult to detect. For this reason, many experienced Fortran programmers now insist upon using explicit interfaces for all procedures.

It is possible to create an explicit interface for a subroutine by using an `interface` block. An easier way is to simply place the subroutine within a module because any function or subroutine placed in a module is automatically given an explicit interface by Fortran. Modules will be discussed at length in Section 3.2; for now, we simply illustrate how to place a subroutine within one. Here is yet another version of our uniform generator that uses a module.

```

----- uniform4.f90 -----
!#####
module my_mod
! a simple module that contains one subroutine
contains
!#####
subroutine fill_with_uniforms(vec, lower, upper)
! Fills the rank-one array vec with random numbers uniformly
! distributed from lower to upper, which default to 0.0 and
! 1.0, respectively
implicit none
! declare arguments
real, intent(out) :: vec(:)
real, intent(in), optional :: lower, upper
! declare locals
real :: a, b
! begin

```

```

a = 0.0
b = 1.0
if( present(lower) ) a = lower
if( present(upper) ) b = upper
call random_number(vec)
vec = a + vec * (b - a)
end subroutine fill_with_uniforms
!#####
end module my_mod
!#####
program uniform
! Generates random numbers uniformly distributed between a and b
! Version 4
use my_mod      ! allows the program to use fill_with_uniforms
implicit none
integer :: i, n
real :: a, b
real, allocatable :: x(:)
print "(A)", "Enter the lower and upper bounds:"
read(*,*) a, b
print "(A)", "How many random numbers do ya want?"
read(*,*) n
allocate( x(n) )
call fill_with_uniforms(x, upper=b, lower=a)
print "(A)", "Here they are:"
do i = 1, n
    print *, x(i)
end do
deallocate(x) ! not really necessary
end program uniform
!#####

```

Placing our subroutine in a module also allowed us to apply another new feature, the **optional** attribute, to the dummy arguments **lower** and **upper**. Declaring a dummy argument as **optional** means that the calling program no longer needs to supply an actual argument for it. The intrinsic function **present**, which returns **.true.** if the actual argument has been provided and **.false.** if it has not, is used within the subroutine to define default values for the optional arguments. In this example, if the calling statement were

```
call fill_with_uniforms(x)
```

then **x** would be filled with random variates uniformly distributed on the unit interval. Notice that in our actual calling statement,

```
call fill_with_uniforms(x, upper=b, lower=a)
```

the order of the actual arguments and dummy arguments differs; the program still works properly, however, because the names of the dummy arguments **lower** and **upper** appear as keywords, enabling the interface to associate the arguments correctly.

Optional arguments and keywords are very helpful when modifying or enhancing subroutines that are already in use by existing programs. It is now possible to add new arguments and options to a subroutine without having to change any of the preexisting `call` statements. These new features require an explicit interface, however, so when using optional arguments or keywords you should make sure that the subroutine is placed within a module.



Style tip

Optional arguments should usually be placed at the end of the argument list, after the required arguments.

Because the names of dummy arguments now function as keywords, it has become more important to choose these names carefully to make them descriptive and meaningful. Well-chosen names will make your source code easier to read and more understandable.

2.3.3 Functions

A function is a procedure that, when it is called, evaluates to a result. For a nontrivial example, let us imagine that the Fortran intrinsic function `sqrt` did not exist. How could we then compute the square root of a real number? The well-known Newton-Raphson procedure solves $f(y) = 0$ by computing

$$y^{(t+1)} = y^{(t)} - f(y^{(t)})/f'(y^{(t)})$$

for $t = 0, 1, 2, \dots$, where f' denotes the first derivative of f , and $y^{(0)}$ is a starting value. To compute $y = \sqrt{x}$, we can take $f(y) = y^2 - x$, and the iteration becomes

$$y^{(t+1)} = \frac{1}{2} \left(y^{(t)} + \frac{x}{y^{(t)}} \right).$$

This algorithm converges for any $x > 0$, but if $x = 0$ we must be careful to avoid division by zero. Here is a simple function that computes $y = \sqrt{x}$ to single precision using a starting value of $y^{(0)} = 1$:

```
real function square_root(x) result(answer)
  implicit none
  real, intent(in) :: x
  real :: old
  answer = 1.
  do
    old = answer
    answer = ( old + x/old ) / 2.
    if( answer == old ) exit
    if( answer == 0. ) exit
```



```

        end do
    end function square_root

```

Here is a very simple program that calls the function:

```

program i_will_root_you
    implicit none
    real :: x, square_root
    read(*,*) x
    print *, square_root(x)
end program i_will_root_you

```

Notice that `square_root` has to be explicitly declared as `real` in the main program because it is an external function and because `implicit none` has been used. If `square_root` had been placed in a module, Fortran would have given it an explicit interface and this type declaration would have been unnecessary.

Any function can be made into a subroutine by adding one more dummy argument with the attribute `intent(out)`. Similarly, any subroutine with a single `intent(out)` argument can be reexpressed as a function. Whether one uses a subroutine or a function to perform a task is primarily a matter of taste. In general, we will adopt the following conventions:

- If a procedure cannot fail to produce the desired result, we will express it as a subroutine.
- If a procedure has a possibility of failing for any reason, we will express it as a function that returns an integer 0, indicating successful completion, or a positive integer, indicating failure.

This rule will allow us to develop a unified approach to handling run-time errors, which we will introduce at the end of Chapter 3. In keeping with this rule, let us revise our square-root function to handle the possibility of a negative or zero input value.

```

integer function square_root(x,y) result(answer)
    implicit none
    real, intent(in) :: x
    real, intent(out) :: y
    real :: old
    if( x < 0. ) then
        goto 5
    else if( x == 0. ) then
        y = 0.
    else
        y = 1.
        do
            old = y

```

```

        y = ( old + x/old) / 2.
        if( y == old ) exit
    end do
end if
! normal exit
answer = 0
return
! error trap
5  answer = 1
end function square_root

```

The revised calling program is

```

program i_will_root_you
    implicit none
    real :: x, y
    integer :: square_root
    read(*,*) x
    if( square_root(x,y) > 0 ) then
        print "(A)", "I can't handle that."
    else
        print *, y
    end if
end program i_will_root_you

```

2.3.4 Pure, Elemental and Recursive Procedures

A function is called pure if all of its arguments are `intent(in)`. A pure function cannot modify the values of its actual arguments; results are communicated to the calling program only through the function's returned value. Our first version of `square_root` was a pure function. We could have notified the compiler that the function was intended to be pure by including the keyword `pure` in our function definition, like this:

```

pure real function square_root(x) result(answer)

```

If we had done this, the compiler would have generated an error message if the dummy argument `x` appeared on the left-hand side of an assignment operation. Any function called within a pure function must also be pure. Declaring functions as pure helps us to protect ourselves against unwanted side effects.

All of the intrinsic functions in Fortran 95 are pure. Many of these functions are also elemental, which means that they can be applied both to scalars and arrays. When applied to arrays, the result is the same as if the function had been applied to each element individually. For example, if x is a real array, then `log(x)` is a real array of the same shape containing

the logarithms of the elements of `x`. We can write our own elemental functions by including the `elemental` keyword in the definition. An elemental function must be given an interface, which we can easily do by placing it in a module. For example, if we define our square-root function as

```
module mymod
  contains
    elemental real function square_root(x) result(answer)
      implicit none
      real, intent(in) :: x
      real :: old
      answer = 1.
      do
        old = answer
        answer = ( old + x/old ) / 2.
        if( answer == old ) exit
      end do
    end function square_root
  end module mymod
```

then we can apply it to a real array of any shape. An elemental function is assumed to be pure.

A procedure is considered to be recursive if it calls itself. We can allow a procedure to call itself by including the keyword `recursive`. For example, here is a recursive function for calculating $x!$ with no argument checking or overflow protection:

```
recursive integer function factorial(x) result(answer)
  implicit none
  integer, intent(in) :: x
  if( x == 0 ) then
    answer = 1
  else
    answer = x * factorial( x-1 )
  end if
end function factorial
```

Recursive procedures can be useful for sorting data, finding sample medians and quantiles, and managing linked lists and other recursive data structures (Section 3.5.3).

2.3.5 *On the Behavior of Local Variables*

Any variable in a subroutine or function that is not passed as an argument is a local variable, defined only within the scope of that procedure. A local variable can be declared and initialized in the same statement, like this:

```
logical :: converged = .false.
```

This ensures that `converged` is `.false.` when the procedure is invoked the first time. In subsequent calls, however, the initial value of `converged` could vary. The reason is that any local variable initialized in a declaration statement is automatically given the `save` attribute, allowing its value to persist from one call to another. That is, the statement above is equivalent to

```
logical, save :: converged = .false.
```

because `save` is implied. If the procedure is called a second time in the same program, the initial value of `converged` will be the value it had upon completion of the first call. If you want the local variable to be initialized each time the procedure is invoked, you need to include the executable statement

```
converged = .false.
```

within the procedure. On the other hand, if you really do want the value of a variable to persist, it's better to pass the variable as an argument.



Style tip

Don't rely on `save` to store persistent data locally within a procedure because it may produce unexpected results if the procedure is embedded in a DLL or in a COM server.

Subroutines and functions may also have local allocatable arrays to serve as temporary workspaces. If the shape of the array is not known in advance but is determined inside the procedure, then the array should be explicitly dimensioned within the procedure by an `allocate` statement. It is not absolutely necessary to explicitly destroy the array with `deallocate` because unless the array has been given the `save` attribute, Fortran 95 automatically deallocates the local array when it exits the procedure. However, explicit deallocation is still a good practice.

Alternatively, if the dimensions of a local array enter the procedure as integer arguments, Fortran will allocate the array automatically. This is called an automatic array. For example, if `m` and `n` are dummy arguments, then declaring a local array

```
real :: workspace(m,n)
```

will cause `workspace` to be dynamically allocated when the procedure is called. In this case, Fortran will automatically deallocate the array when the procedure is finished.

2.3.6 Exercises

- Optional arguments to procedures were discussed in Section 2.3.2. A dummy argument that has been declared **optional** may be passed as an actual argument to another procedure, provided that the corresponding dummy argument in the latter procedure is also **optional**. Try this in an example.
- Create a procedure for matrix multiplication using assumed-shape arrays and write a simple program that calls it. Within your procedure, check the dimensions of the argument arrays for conformity.
- Write a recursive procedure for generating any term of the Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, \dots,$$

in which each element is the sum of the two preceding elements. Then use this procedure in a program that prints out the first n terms for a user-specified n . Does this procedure seem efficient? Explain.

- Write elemental functions for computing the logistic transformation

$$\text{logit}(p) = \log \left(\frac{p}{1-p} \right)$$

and its inverse

$$\text{expit}(x) = \frac{e^x}{1 + e^x}$$

for real arrays of arbitrary shape.

- Suppose that a procedure begins like this:

```
subroutine do_something( arg1, arg2 )
  integer, intent(in) :: arg1
  integer, intent(out) :: arg2
  real, allocatable, save :: x(:)
  allocate( x(arg1) )
```

Explain why this subroutine may cause a program to crash. What can be done to fix it?

- In a Fortran procedure, the name of a function may be passed as an argument. If we want to pass the name of a real-valued function, for example, the corresponding dummy argument should be declared as a **real** variable of the appropriate kind. Write a procedure that accepts the name of a function f as an argument and numerically approximates the first derivative,

$$f'(x) \approx \frac{f(x + \delta/2) - f(x - \delta/2)}{\delta},$$

for a given x and $\delta > 0$.

2.4 Manipulating Character Strings

2.4.1 Character Variables

Modern Fortran has many helpful features for storing and manipulating character data. In statistical applications, these features are useful for handling filenames, reading and processing data from files, interpreting user-supplied text expressions, handling error messages, and so on.

A character variable can be declared as a single string of a fixed length,

```
character(len=name_length) :: my_name
```

as an array of fixed-length strings,

```
character(len=name_length) :: my_parents_names(2)
```

or as an allocatable array of fixed-length strings:

```
character(len=name_length), allocatable :: &
    all_my_children(:)
```

In these three examples, the integer `name_length` must either be a constant,

```
integer, parameter :: name_length = 80
```

or, if the string or string array appears in a procedure, `name_length` must be included among the dummy arguments.

Under many circumstances, it is also possible to declare character variables of assumed length (`len=*`). A character-string constant of assumed length, created as

```
character(len=*), parameter :: &
    program_version = "Beta version"
```

automatically takes the length of the literal string on the right-hand side of the equal sign. A string that serves as a dummy argument to a procedure may have an assumed length, in which case it derives its length from that of the corresponding actual argument.



Style tip

When writing procedures for character strings, it's an excellent idea to use dummy arguments of assumed length. This makes the procedures easier to call and more general, reducing the chance that the procedure will need to be changed as the program develops.

The colon (`:`) allows us to access individual characters or substrings within a character string. Suppose we declare and set a character variable as follows:

```
character(len=10) :: my_name
```

```
my_name = "Charlie"
```

Then `my_name(2:2)` is "h", `my_name(2:4)` is "har", `my_name(:4)` is "Char", and `my_name(4:)` is "rlie ". If `strarray` is a rank-one string array, then `strarray(i)(j:j)` extracts the *j*th character from the *i*th element.

2.4.2 Assigning, Comparing, and Concatenating Strings

The equal sign may be used for character assignment as follows. If **a** and **b** are strings of equal length, then

```
a = b
```

copies the contents of **b** into **a**. If **a** is shorter than **b**, the initial part of **b** goes into **a** and the rest is discarded. If **a** is longer than **b**, then the contents of **b** go into the initial part of **a**, and the rest of **a** is padded with blank spaces. Therefore,

```
a = ""
```

has the effect of filling **a** with space, and

```
a(i:) = ""
```

has the effect of blanking out the *i*th and all subsequent characters of **a**.

Strings can also be compared by the relational operators shown in Table 2.3. If two strings are not of the same length, then the shorter one is implicitly padded with blank spaces on the right-hand side before the comparison is made. Two strings are judged to be equal if characters in every pair of corresponding positions agree; for example, `("boy" == "boy ")` evaluates to `.true.`, and `(a == "")` evaluates to `.true.` if and only if **a** is entirely blank.

Applying the inequality operators `<`, `<=`, `>`, and `>=` to character strings is not recommended because the outcome of these operations depends on a collating sequence that may vary from one compiler to another. Instead, we recommend using the intrinsic lexical functions `l1t` (less than), `l1e` (less than or equal to), `lgt` (greater than), and `lge` (greater than or equal to) because these functions rely on the universal ASCII collating sequence. Each of these functions takes two character strings as arguments and returns a logical value; for example, the expression `lgt(string1, string2)` evaluates to `.true.` if `string1` is lexically greater than `string2`. One character is considered to be greater than another if it appears later in the ASCII sequence; multicharacter strings are compared one character at a time, moving from left to right, with the first nonidentical pair of characters determining which string is the greater one.

Strings are concatenated (i.e., joined together) by the double forward slash (`//`) operator. For example, the expression

```
"Tom" // " " // "Sawyer" == "Tom Sawyer"
```

evaluates to `.true.`. Concatenation is useful for defining literal strings that are too long to comfortably appear in a single line of source code:

```
gettysburg_address = "Four score and seven years ago " &
// "our fathers brought forth on this continent " &
// "a new nation..."
```

2.4.3 More String Functions

The Fortran intrinsic function `len` returns the length of a string, whereas `len_trim` returns the length without counting trailing spaces. For example, `len("boy ")` returns the integer 4, but `len_trim("boy ")` returns 3. The function `adjustl` returns a character string of the same length as its input, with leading blanks removed and reinserted at the end; `adjustr` performs the opposite operation, removing trailing blanks and reinserting them at the beginning. Thus `adjustl(" boy")` evaluates to `"boy "` and `adjustr("boy ")` evaluates to `" boy"`.

An extremely useful function is `index`, which allows us to search for a single character or string within another string. More precisely,

```
index(string, substring)
```

returns an integer indicating the starting position of the first occurrence of `substring` within `string`. For example, `index("hello","l")` evaluates to 3, as does `index("hello","llo")`. If the substring does not occur within the string, the returned value is 0. This function has an optional third argument, `back`, whose default value is `.false.`. Specifying `back=.true.` will begin the search from the right side rather than the left, returning the starting position of the last occurrence of the substring within the string. The following function, which uses both `index` and `len_trim`, removes any existing suffix from a filename and replaces it with a new three-character suffix supplied by the user.

```
----- suffix.f90 -----
!#####
integer function add_suffix(file_name, suffix) result(answer)
! Returns 0 if operation is successful, 1 otherwise.
implicit none
character(len=*), intent(inout) :: file_name
character(len=3), intent(in) :: suffix
integer :: i
if( suffix(1:1)==" " ) goto 5          ! suffix begins with blank
i = index(file_name, ".", back=.true.) ! look for last period
if( i==0 ) i = len_trim(file_name) + 1 ! if no period was found
if( len(file_name) < i+3 ) goto 5      ! not enough room for suffix
file_name(i:) = "." // suffix
! normal exit
answer = 0
return
! error trap
```



```

5  answer = 1
   return
end function add_suffix
!#####

```

2.4.4 Internal Files

Fortran now allows character variables to be used for input and output; data may be written to them or read from them as if they were files. When character variables are used in this manner, they are called internal files. Internal files provide a convenient mechanism to convert numeric data to character representations and vice versa. If **str** is a character string and **a** is a variable, then

```
write(str, *) a
```

writes the value of **a** to **str** in free format, and

```
read(str, *) a
```

attempts to read a value of **a** from **str**. An error will result in the former case if the written value of **a** exceeds the length of the string, or in the latter case if the contents of **str** cannot be interpreted as data of the expected type.

For a more detailed example, suppose we are writing a statistical application that operates on a rectangular dataset consisting of **p** variables. The user may supply character-string names for the variables, but if names are not given, we will create the default names "VAR_1", "VAR_2", and so on. The code below shows how these default names can be created by writing integers to an internal file.

```

integer :: p                ! the number of variables
integer, parameter :: var_name_length = 8
character(len=var_name_length), allocatable :: var_names(:)
character(len=4) :: sInt
integer :: i

! lines omitted

allocate(var_names, p)
do i = 1, p
  write(sInt, "(I4)", err=5) i
  var_names(i) = "VAR_" // adjustl(sInt)
end do
! error trap
5 continue

```

In the `write` statement, the format string `"(I4)"` declares that the integer is to be written to a field four characters wide. In the unlikely event that 10,000 or more variables are present, the `err` specifier will cause the point of execution to jump to a trap where the problem can be handled gracefully.

In Fortran `read` and `write` statements, the format string does not need to be a literal character string such as `"(I4)"`; it may also be a character variable whose value is altered during program execution. This feature greatly facilitates run-time formatting, which was very awkward in FORTRAN 77. For example, in FORTRAN 77 it was difficult to print out a nicely formatted rectangular table for which the number of columns was not known in advance. By writing integers to internal files, one can now build a format string and then use it in a subsequent write statement.

2.4.5 Exercises

1. Write a function that accepts a character string of arbitrary length as input and centers the text within the string. For example, if the input string is `"aa "`, the result should be `" aa "`.
2. Write a program that reads a single line of text (up to, say, 128 characters long) and prints out the line with the words given in reverse order. For this purpose, words may be defined as strings of text delimited by any amount of blank space. For example, if the user types

```
hello,    my name  is Bob
```

then the response should be

```
Bob is name my hello,
```

3. The intrinsic function `achar` returns the character corresponding to any position in the ASCII collating sequence. That is, if `i` is an integer between 0 and 127 inclusive, then `achar(i)` is a character string of length one containing the character in ASCII position `i`. Write a program that prints to the screen all the characters in the ASCII sequence (but note that some characters are nonprintable). From this output, identify the ASCII positions of the lowercase letters (`a`, ..., `z`), the uppercase letters (`A`, ..., `Z`), and the numerals (`0`, ..., `9`).
4. The intrinsic function `iachar` accepts a character-string argument of length one and returns as an integer the ASCII position of that character. Using this function, write a procedure that converts all the lowercase letters within a string to uppercase or vice versa.

2.5 Additional Topics

2.5.1 Expressions with Mixed Types and Kinds

In general, one should be very cautious with integer division and with expressions or statements that mix integers with reals. Consider the simple test program below.

```
program test
  implicit none
  real :: x
  x = 2/3
  print *, x
end program test
```

Many novice programmers would be surprised by the result:

```
D:\jls\software\demos>test
0.0000000E+00
```

In this example, Fortran interprets the literals 2 and 3 as integers of the default kind. It then computes $2/3$ by integer division, which truncates the result to zero. Finally, the zero value is converted to a floating-point representation and stored in *x*. If $2/3$ were changed to $2./3$, $2/3.$, or $2./3.$, the result would become $0.666\dots$ because the literals 2. and 3. are interpreted as real numbers of the default kind (single precision).

Whenever an expression involving only integers is evaluated, Fortran returns the result as an integer. If an expression involves integers and reals of a given kind, the integers may be implicitly converted to reals of that kind, and the result is returned as real. If an expression involves reals of different kinds, those that are less precise may be converted to the higher precision, and the returned result is of the higher precision. If the expression is complicated, however, it may be partially evaluated before any conversion takes place. For example,

$$2/3 + 1.6$$

will evaluate to 1.6. Finally, when an assignment

$$variable = expression$$

is made, some conversion takes place if the value returned by *expression* does not match *variable* in type and kind. If *expression* is real and *variable* is an integer, the conversion will be made by truncation rather than rounding.

Large errors can arise in computational statements that mix data of different types or kinds because floating-point arithmetic is only approximate and because many integers have no exact representation in a floating-point model. For example, in

```
real :: a, b
```

```
integer :: i
a = 0.2
b = 3.8
i = a + b
```

the resulting value of `i` could be 3 or 4. For these reasons, mixed-type expressions should be avoided.

2.5.2 *Explicit Type Conversion*

To avert problems arising from mixing types and kinds, you can perform conversions yourself using the intrinsic functions `real` and `int`. If `x` is an integer or real variable, then `real(x,mykind)` returns a real value of the kind `mykind`. If the kind argument is omitted, the result will be a real value of the default kind. Similarly, `int(x,mykind)` converts the value in `x` to a `mykind` integer, and `int(x)` converts it to a default integer. Note that Fortran converts reals to integers by truncation toward zero, so `int(1.9989)` returns 1 and `int(-2.732)` returns -2. Both of these functions are elemental, so they can also be applied to arrays of arbitrary shape.

It is also important to understand how Fortran interprets literal constants in your source code. Numbers such as 0, 10, and -97 are regarded as integers. Anything involving a decimal point, such as 3.7, -9.0, and 6., is stored as a floating-point real number, but the kind may vary. Usually it will be of the default or single-precision kind, but if that model appears to be inadequate, the precision may or may not be increased. For example, some compilers will store 123456789.0 in double precision, noticing that this constant has more than six significant digits. However, others will disregard the extra digits and store it in single precision as approximately 1.23456×10^8 . Putting the value into a double-precision variable, as in

```
double precision :: k
k = 123456789.0
```

or turning it into a double-precision named constant as in

```
double precision, parameter :: k = 123456789.0
```

may not recover the extra digits because the compiler may still interpret the literal with single precision before converting it to double precision. The best way to guarantee greater accuracy is to specify the literal in exponential notation using D rather than E. That is, the constant could be written as 1.23456789D8, 1.23456789D+8, or 1.23456789D+08. Exponential notation using an E, as in 1.4E-06, is generally interpreted with single precision, whereas D is interpreted as double precision.

If you still have any doubts about what your compiler is doing, the intrinsic function `kind` may be used to query the system to determine

the actual `kind` parameter of any variable or constant. To see how your compiler handles the previous example, run this test program:

```
program test
  real :: a
  double precision :: b
  print *, "Single precision is kind", kind(a)
  print *, "Double precision is kind", kind(b)
  print *, "123456789.0 is kind", kind(123456789.0)
end program test
```

The way that literals are written may have important implications not only for accuracy but also for computational efficiency. For example, the statements

```
y = x**2
```

and

```
y = x**2.0
```

may seem equivalent, but a compiler implements them quite differently. The first simply computes `x*x`, whereas the second—raising a floating-point number to a floating-point power—is a far more complicated operation. Wherever possible, it is a good idea to use integer rather than floating-point exponents. It is also beneficial to express square roots as `sqrt(x)` rather than `x**0.5`, as the former allows the compiler to take advantage of highly efficient routines for computing square roots. More issues of numerical accuracy and efficiency will be taken up in Chapter 4.

2.5.3 Generic Procedures

Many Fortran intrinsic functions are generic. A generic function may accept arguments of different types and return values of different types. For example, the generic absolute value function `abs` may be applied to integer or real variables. If `x` is real, then `abs(x)` returns a real number of the same kind; if `x` is an integer, then `abs(x)` returns an integer of the same kind. Other commonly used generic functions include `sqrt`, `exp`, `log`, `log10`, `sign`, `sin`, `cos`, and `tan`. All of these functions are elemental and can thus be applied to arrays.

When using these generic functions, it is helpful to check the language reference material supplied with the compiler to see what types of arguments are expected and what type of value will be returned; not doing so could produce unexpected results. For example, the generic square root function `sqrt` expects to operate on real numbers of various kinds but not on integers. Depending on the options of your compiler, `sqrt` may accept an integer argument and perform a type conversion to real or it may not.

That is, the expression `sqrt(4)` might possibly evaluate to 2.000... as intended, or it might produce an error when the program is compiled. To help ensure consistency of results across compilers, it would be wise to explicitly convert the argument with `real` in this case.

Fortran allows you to write your own generic procedures. This will be discussed in the next chapter, when we take up the subject of modules.

2.5.4 *Don't Pause or Stop*

We strongly recommend that you never use the Fortran statement `stop` within any subroutine or function, or even in a main program. This statement, along with its diabolical cousin `pause` (which was declared obsolescent in Fortran 90 and deleted from Fortran 95), appear in old-fashioned FORTRAN programs to halt execution in the event of an error. These statements may have unfortunate consequences for procedures embedded in DLLs or in COM servers. Rather than using `stop`, it's better to anticipate the possible errors that may arise and structure your procedures to exit gracefully if one occurs.

2.6 Additional Exercises

1. Explain why this snippet of Fortran code prints three different values:

```
double precision :: x, a, b
print *, 2./7.
x = 2./7.
print *, x
a = 2.
b = 7.
x = a/b
print *, x
```

What result do you expect from this?

```
real :: x
x = 4.0
print *, x**(1/2)
```

2. Write a procedure that takes as input an $n \times p$ data matrix X and a vector of weights $w = (w_1, \dots, w_n)^T$ and computes the matrix of weighted sums of squares and cross products, $X^T W X$, where $W = \text{Diag}(w)$. Streamline the procedure by making use of the fact that $X^T W X$ is symmetric. Make the argument for w optional, so that if no weights are provided, the procedure computes $X^T X$ by default.

3. Write a procedure that accepts as input an $n \times p$ data matrix

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix}$$

and computes the $p \times 1$ vector of sample means

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

and the sample covariance matrix $S = k^{-1}A$, where

$$A = \sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})^T = X^T X - n\bar{x}\bar{x}^T$$

and $k = n - 1$ by default or $k = n$ by the caller's request. Use the one-pass method that updates \bar{x} and A by incorporating rows of the data matrix one at a time. That is, letting $\bar{x}_{(i)}$ and $A_{(i)}$ denote the values of \bar{x} and A based on the first i rows of the data matrix, initialize $\bar{x}_{(0)}$ and $A_{(0)}$ to zero and compute

$$\begin{aligned} \bar{x}_{(i)} &= \bar{x}_{(i-1)} + \frac{1}{i} (x_i - \bar{x}_{(i-1)}) \\ A_{(i)} &= A_{(i-1)} + \frac{i-1}{i} (x_i - \bar{x}_{(i-1)}) (x_i - \bar{x}_{(i-1)})^T \end{aligned}$$

for $i = 1, \dots, n$.

4. Explain why the expression `(2. == 2.D0)` may evaluate to `.true.`, whereas `(exp(2.) == exp(2.D0))` may not. If you are not sure, write a test program to print out the value and kind of each subexpression.
5. Expanding e^x in a Taylor series about $x = 0$ and substituting $x = 1$ yields

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

This suggests that the constant $e = 2.71828\dots$ may be computed by this simple algorithm: set a , n , and e equal to 1, and repeat

$$\begin{aligned} a &= a/n, \\ e &= e + a, \\ n &= n + 1, \end{aligned}$$

until a is indistinguishable from zero. Write a Fortran program to compute e using the most precise floating-point model offered by your compiler.

6. If (X, Y) is uniformly distributed over the unit square—that is, if X and Y are independent $U(0, 1)$ random variates—then the probability of falling within the unit circle $(X - 1/2)^2 + (Y - 1/2)^2 \leq 1$ is $\pi/4$. Therefore, we can simulate the value of $\pi = 3.14159\dots$ by generating a sample of n random points within the unit square, computing the proportion of the sample that falls within the circle, and multiplying by 4. Write a simple program that simulates the value of π in this manner for any given value of n .
7. Given a full-rank $n \times p$ data matrix X , there are many ways to transform it into another $n \times p$ matrix whose columns span the same linear space but are mutually orthogonal. One simple method is the Modified Gram-Schmidt (MGS) procedure (Golub and van Loan, 1996). This bit of Fortran code will decompose X into $X = QR$, where Q is an $n \times p$ orthonormal matrix and R is $p \times p$ upper-triangular. The X matrix is overwritten with Q .

```
double precision :: x(n,p), r(p,p)
integer :: n, p, j, k
r(:, :) = 0.D0
do j = 1, p
  r(j,j) = sqrt( sum( x(:,j)**2 ) )
  x(:,j) = x(:,j) / r(j,j)
  do k = (j+1), p
    r(j,k) = dot_product( x(:,j), x(:,k) )
    x(:,k) = x(:,k) - x(:,j) * r(j,k)
  end do
end do
```

Implement this MGS procedure in a function or subroutine, including a provision to prevent division by zero in case X is rank-deficient. Test your procedure on an X matrix filled with random numbers, and verify that $Q^T Q \approx I$ and $QR \approx X$.

8. Consider a two-way contingency table with elements x_{ij} , $i = 1, \dots, r$, $j = 1, \dots, c$. The estimated expected counts under a model of row-column independence are $\hat{x}_{ij} = x_{i+}x_{+j}/x_{++}$, where

$$x_{i+} = \sum_{j=1}^c x_{ij}, \quad x_{+j} = \sum_{i=1}^r x_{ij}, \quad x_{++} = \sum_{i=1}^r \sum_{j=1}^c x_{ij}.$$

A test for row-column independence is usually carried out by comparing the Pearson goodness-of-fit statistic

$$X^2 = \sum_{i=1}^r \sum_{j=1}^c \frac{(\hat{x}_{ij} - x_{ij})^2}{\hat{x}_{ij}}$$

or the deviance statistic

$$G^2 = 2 \sum_{i=1}^r \sum_{j=1}^c x_{ij} \log \frac{x_{ij}}{\hat{x}_{ij}}$$

to a chi-square distribution with $(r-1)(c-1)$ degrees of freedom. Write a procedure that accepts an assumed-shape two-way real array of nonnegative observed counts and computes the two-way array of estimated expected counts, X^2 , G^2 , and the degrees of freedom. When calculating G^2 , handle observed counts of zero by treating $0 \log 0$ as zero. If any expected count is zero, then X^2 and G^2 are undefined and the procedure should signal a failure.



<http://www.springer.com/978-0-387-23817-3>

Developing Statistical Software in Fortran 95

Lemmon, D.R.; Schafer, J.L.

2005, XVI, 324 p., Softcover

ISBN: 978-0-387-23817-3