Chapter 21.  Introduction to Fortran 90

Language Features

21.0 Introduction

Fortran 90 is in many respects a backwards-compatible modernization of the long-used (and much abused) Fortran 77 language, but it is also, in other respects, a new language for parallel programming on present and future multiprocessor machines. These twin design goals of the language sometimes add confusion to the process of becoming fluent in Fortran 90 programming.

In a certain trivial sense, Fortran 90 is strictly backwards-compatible with Fortran 77. That is, any Fortran 90 compiler is supposed to be able to compile any legacy Fortran 77 code without error. The reason for terming this compatibility trivial, however, is that you have to tell the compiler (usually via a source file name ending in "."f" or "."f00") that it is dealing with a Fortran 77 file. If you instead try to pass off Fortran 77 code as native Fortran 90 (e.g., by naming the source file something ending in "."f90") it will not always work correctly!

It is best, therefore, to approach Fortran 90 as a new computer language, albeit one with a lot in common with Fortran 77. Indeed, in such terms, Fortran 90 is a fairly big language, with a large number of new constructions and intrinsic functions. Here, in one short chapter, we do not pretend to provide a complete description of the language. Luckily, there are good books that do exactly that. Our favorite one is by Metcalf and Reid[1], cited throughout this chapter as “M&R.” Other good starting points include [2] and [3].

Our goal, in the remainder of this chapter, is to give a good, working description of those Fortran 90 language features that are not immediately self-explanatory to Fortran 77 programmers, with particular emphasis on those that occur most frequently in the Fortran 90 versions of the Numerical Recipes routines. This chapter, by itself, will not teach you to write Fortran 90 code. But it ought to help you acquire a reading knowledge of the language, and perhaps provide enough of a head start that you can rapidly pick up the rest of what you need to know from M&R or another Fortran 90 reference book.

CITED REFERENCES AND FURTHER READING:
21.1 Quick Start: Using the Fortran 90 Numerical Recipes Routines

This section is for people who want to jump right in. We’ll compute a Bessel function \( J_0(x) \), where \( x \) is equal to the fourth root of the Julian Day number of the 200th full moon since January 1900. (Now there’s a useful quantity!)

First, locate the important files nrtype.f90, nrutil.f90, and nr.f90, as listed in Appendices C1, C1, and C2, respectively. These contain modules that either are (i) used by our routines, or else (ii) describe the calling conventions of our routines to (your) user programs. Compile each of these files, producing (with most compilers) a .mod file and a .o (or similarly named) file for each one.

Second, create this main program file:

```fortran
PROGRAM hello_bessel
USE nrtype
USE nr, ONLY: flmoo0, bessj0
IMPLICIT NONE
INTEGER(4B) :: n=200,mph=2,jd
REAL(SP) :: x,frac,ans
call flmoo0(n,mph,jd,frac)
x=jd**0.25_SP
ans=bessj0(x)
write (*,*) 'Hello, Bessel: ', ans
END PROGRAM
```

Here is a quick explanation of some elements of the above program:

The first USE statement includes a module of ours named nrtype, whose purpose is to give symbolic names to some kinds of data types, among them single-precision reals ("sp") and four-byte integers ("4B"). The second USE statement includes a module of ours that defines the calling sequences, and variable types, expected by (in this case) the Numerical Recipes routines flmoo0 and bessj0.

The IMPLICIT NONE statement signals that we want the compiler to require us explicitly to declare all variable types. We strongly urge that you always take this option.

The next two lines declare integer and real variables of the desired kinds. The variable \( n \) is initialized to the value 200, \( \text{mph} \) to 2 (a value expected by flmoo0).

We call flmoo0, and take the fourth root of the answer it returns as \( jd \). Note that the constant 0.25 is typed to be single-precision by the appended _sp.

We call the bessj0 routine, and print the answer.

Third, compile the main program file, and also the files flmoo0.f90, bessj0.f90. Then, link the resulting object files with also nrutil.o (or similar system-dependent name, as produced in step 1). Some compilers will also require you to link with nr.o and nrtype.o.

Fourth, run the resulting executable file. Typical output is:

```
Hello, Bessel: 7.3096365E-02
```
21.2 Fortran 90 Language Concepts

The Fortran 90 language standard defines and uses a number of standard terms for concepts that occur in the language. Here we summarize briefly some of the most important concepts. Standard Fortran 90 terms are shown in italics. While by no means complete, the information in this section should help you get a quick start with your favorite Fortran 90 reference book or language manual.

A note on capitalization: Outside a character context, Fortran 90 is not case-sensitive, so you can use upper and lower case any way you want, to improve readability. A variable like SP (see below) is the same variable as the variable sp. We like to capitalize keywords whose use is primarily at compile-time (statements that delimit program and subprogram boundaries, declaration statements of variables, fixed parameter values), and use lower case for the bulk of run-time code. You can adopt any convention that you find helpful to your own programming style; but we strongly urge you to adopt and follow some convention.

Data Types and Kinds

Data types (also called simply types) can be either intrinsic data types (the familiar INTEGER, REAL, LOGICAL, and so forth) or else derived data types that are built up in the manner of what are called “structures” or “records” in other computer languages. (We’ll use derived data types very sparingly in this book.) Intrinsic data types are further specified by their kind parameter (or simply kind), which is simply an integer. Thus, on many machines, REAL(4) (with kind = 4) is a single-precision real, while REAL(8) (with kind = 8) is a double-precision real. Literal constants (or simply literals) are specified as to kind by appending an underscore, as 1.5_4 for single precision, or 1.5_8 for double precision. [M&R, §2.5–§2.6]

Unfortunately, the specific integer values that define the different kind types are not specified by the language, but can vary from machine to machine. For that reason, one almost never uses literal kind parameters like 4 or 8, but rather defines in some central file, and imports into all one’s programs, symbolic names for the kinds. For this book, that central file is the module named artype, and the chosen symbolic names include SP, DP (for reals); I2B, I4B (for two- and four-byte integers); and LGT for the default logical type. You will therefore see us consistently writing REAL(SP), or 1.5_sp, and so forth.

Here is an example of declaring some variables, including a one-dimensional array of length 500, and a two-dimensional array with 100 rows and 200 columns:

```fortran
USE artype
REAL(SP) :: x,y,z
INTEGER(I4B) :: i,j,k
REAL(SP), DIMENSION(500) :: arr
REAL(SP), DIMENSION(100,200) :: barr
REAL(SP) :: carr(500)
```

The last line shows an alternative form for array syntax. And yes, there are default kind parameters for each intrinsic type, but these vary from machine to machine and can get you into trouble when you try to move code. We therefore specify all kind parameters explicitly in almost all situations.
Chapter 21. Introduction to Fortran 90 Language Features

Array Shapes and Sizes

The shape of an array refers to both its dimensionality (called its rank), and also the lengths along each dimension (called the extents). The shape of an array is specified by a rank-one array whose elements are the extents along each dimension, and can be queried with the shape intrinsic (see p. 949). Thus, in the above example, \texttt{shape(barr)} returns an array of length 2 containing the values \((100, 200)\).

The size of an array is its total number of elements, so the intrinsic \texttt{size(barr)} would return \(20000\) in the above example. More often one wants to know the extents along each dimension, separately: \texttt{size(barr,1)} returns the value 100, while \texttt{size(barr,2)} returns the value 200. [M&R, §2.10]

Section §21.3, below, discusses additional aspects of arrays in Fortran 90.

Memory Management

Fortran 90 is greatly superior to Fortran 77 in its memory-management capabilities, seen by the user as the ability to create, expand, or contract workspace for programs. Within \textit{subprograms} (that is, \textit{subroutines} and \textit{functions}), one can have \textit{automatic arrays} (or other \textit{automatic data objects}) that come into existence each time the subprogram is entered, and disappear (returning their memory to the pool) when the subprogram is exited. The size of automatic objects can be specified by arbitrary expressions involving values passed as \textit{actual arguments} in the calling program, and thus received by the subprogram through its corresponding \textit{dummy arguments}. [M&R, §6.4]

Here is an example that creates some automatic workspace named \texttt{carr}:

```
SUBROUTINE doSomething(j,k)
USE nrttype
REAL(SP), DIMENSION(2*j,k**2) :: carr
```

Finer control on when workspace is created or destroyed can be achieved by declaring \textit{allocatable arrays}, which exist as names only, without associated memory, until they are \textit{allocated} within the program or subprogram. When no longer needed, they can be \textit{deallocated}. The \textit{allocation status} of an allocatable array can be tested by the program via the \texttt{allocated} intrinsic function (p. 952). [M&R, §6.5]

Here is an example in outline:

```
REAL(SP), DIMENSION(:,,:), ALLOCATABLE :: darr
... allocate(darr(10,20))
... deallocate(darr)
... allocate(darr(100,200))
... deallocate(darr)
```

Notice that \texttt{darr} is originally declared with only “slots” (colons) for its dimensions, and is then allocated/deallocated twice, with different sizes.

Yet finer control is achieved by the use of \textit{pointers}. Like an allocatable array, a pointer can be allocated, at will, its own associated memory. However, it has the additional flexibility of alternatively being \textit{pointer associated} with a \textit{target} that
already exists under another name. Thus, pointers can be used as redefinable aliases for other variables, arrays, or (see §21.3) array sections. [M&R, §6.12]

Here is an example that first associates the pointer parr with the array earr, then later cancels that association and allocates it its own storage of size 50:

```fortran
REAL(SP), DIMENSION(:,), POINTER :: parr
REAL(SP), DIMENSION(100), TARGET :: earr
...
parr => earr
...
nullify(parr)
allocate(parr(50))
...
deallocate(parr)
```

### Procedure Interfaces

When a procedure is referenced (e.g., called) from within a program or subprogram (examples of scoping units), the scoping unit must be told, or must deduce, the procedure's interface, that is, its calling sequence, including the types and kinds of all dummy arguments, returned values, etc. The recommended procedure is to specify this interface via an explicit interface, usually an interface block (essentially a declaration statement for subprograms) in the calling subprogram or in some module that the calling program includes via a USE statement. In this book all interfaces are explicit, and the module named nr contains interface blocks for all of the Numerical Recipes routines. [M&R, §5.11]

Here is a typical example of an interface block:

```fortran
INTERFACE
SUBROUTINE caldat(julian, mm, id, iyyy)
USE nrtype
INTEGER(I4B), INTENT(IN) :: julian
INTEGER(I4B), INTENT(OUT) :: mm, id, iyyy
END SUBROUTINE caldat
END INTERFACE
```

Once this interface is made known to a program that you are writing (by either explicit inclusion or a USE statement), then the compiler is able to flag for you a variety of otherwise difficult-to-find bugs. Although interface blocks can sometimes seem overly wordy, they give a big payoff in ultimately minimizing programmer time and frustration.

For compatibility with Fortran 77, the language also allows for implicit interfaces, where the calling program tries to figure out the interface by the old rules of Fortran 77. These rules are quite limited, and prone to producing devilishly obscure program bugs. We strongly recommend that implicit interfaces never be used.

### Elemental Procedures and Generic Interfaces

Many intrinsic procedures (those defined by the language standard and thus usable without any further definition or specification) are also generic. This means that a single procedure name, such as $\log(x)$, can be used with a variety of types and kind parameters for the argument $x$, and the result returned will have the same type and kind parameter as the argument. In this example, $\log(x)$ allows any real or complex argument type.
Better yet, most generic functions are also \textit{elemental}. The argument of an elemental function can be an array of arbitrary shape! Then, the returned result is an array of the same shape, with each element containing the result of applying the function to the corresponding element of the argument. (Hence the name \textit{elemental}, meaning “applied element by element.”) [M&R, §8.1] For example:

\begin{verbatim}
REAL(SP), DIMENSION(100,100) :: a,b
b=sin(a)
\end{verbatim}

Fortran 90 has no facility for creating new, user-defined elemental functions. It does have, however, the related facility of \textit{overloading} by the use of \textit{generic interfaces}. This is invoked by the use of an interface block that attaches a single \textit{generic name} to a number of distinct subprograms whose dummy arguments have different types or kinds. Then, when the generic name is referenced (e.g., called), the compiler chooses the specific subprogram that matches the types and kinds of the actual arguments used. [M&R, §5.18] Here is an example of a generic interface block:

\begin{verbatim}
INTERFACE myfunc
    FUNCTION myfunc_single(x) USE nrtype
        REAL(SP) :: x,myfunc_single
    END FUNCTION myfunc_single

    FUNCTION myfunc_double(x) USE nrtype
        REAL(DP) :: x,myfunc_double
    END FUNCTION myfunc_double
END INTERFACE
\end{verbatim}

A program with knowledge of this interface could then freely use the function reference \texttt{myfunc(x)} for \texttt{x}'s of both type \texttt{SP} and type \texttt{DP}.

We use overloading quite extensively in this book. A typical use is to provide, under the same name, both scalar and vector versions of a function such as a Bessel function, or to provide both single-precision and double-precision versions of procedures (as in the above example). Then, to the extent that we have provided all the versions that you need, you can pretend that our routine is elemental. In such a situation, if you ever call our function with a type or kind that we have not provided, the compiler will instantly flag the problem, because it is unable to resolve the generic interface.

\section*{Modules}

\textit{Modules}, already referred to several times above, are Fortran 90's generalization of Fortran 77's common blocks, \texttt{INCLUDED} files of parameter statements, and (to some extent) statement functions. Modules are \textit{program units}, like main programs or subprograms (subroutines and functions), that can be separately compiled. A module is a convenient place to stash global data, \textit{named constants} (what in Fortran 77 are called “symbolic constants” or “\texttt{PARAMETERS}”), interface blocks to subprograms and/or actual subprograms themselves (\textit{module subprograms}). The convenience is that a module’s information can be incorporated into another program unit via a simple, one-line \texttt{USE} statement. [M&R, §5.5]

Here is an example of a simple module that defines a few parameters, creates some global storage for an array named \texttt{arr} (as might be done with a Fortran 77 common block), and defines the interface to a function \texttt{yourfunc}:
More on Arrays and Array Sections

As mentioned earlier, the module nr contains INTERFACE declarations for all the Numerical Recipes. When we include a statement of the form

USE nr, ONLY: recipe1

it means that the program uses the additional routine recipe1. The compiler is able to use the explicit interface declaration in the module to check that recipe1 is invoked with arguments of the correct type, shape, and number. However, a weakness of Fortran 90 is that there is no fail-safe way to be sure that the interface module (here nr) stays synchronized with the underlying routine (here recipe1). You might think that you could accomplish this by putting USE nr, ONLY: recipe1 into the recipe1 program itself. Unfortunately, the compiler interprets this as an erroneous double definition of recipe1’s interface, rather than (as would be desirable) as an opportunity for a consistency check. (To achieve this kind of consistency check, you can put the procedures themselves, not just their interfaces, into the module.)

CITED REFERENCES AND FURTHER READING:

21.3 More on Arrays and Array Sections

Arrays are the central conceptual core of Fortran 90 as a parallel programming language, and thus worthy of some further discussion. We have already seen that arrays can “come into existence” in Fortran 90 in several ways, either directly declared, as

REAL(SP), DIMENSION(100,200) :: arr

or else allocated by an allocatable variable or a pointer variable,

REAL(SP), DIMENSION(:,,:), ALLOCATABLE :: arr
REAL(SP), DIMENSION(:,,:), POINTER :: barr
... allocate(arr(100,200),barr(100,200))

or else (not previously mentioned) passed into a subprogram through a dummy argument:

SUBROUTINE myroutine(carr)
USE nrtype
REAL(SP), DIMENSION(:,,:) :: carr
... i=size(carr,1)
Chapter 21. Introduction to Fortran 90 Language Features

\[ j = \text{size}(\text{carr}, 2) \]

In the above example we also show how the subprogram can find out the size of the actual array that is passed, using the \text{size} intrinsic. This routine is an example of the use of an \textit{assumed-shape array}, new to Fortran 90. The actual extents along each dimension are inherited from the calling routine at run time. A subroutine with assumed-shape array arguments \textit{must} have an explicit interface in the calling routine, otherwise the compiler doesn’t know about the extra information that must be passed. A typical setup for calling \textit{myroutine} would be:

```
PROGRAM use_myroutine
USE nrtype
REAL(SP), DIMENSION(10,10) :: arr
INTERFACE
  SUBROUTINE myroutine(carr)
    USE nrtype
    REAL(SP), DIMENSION(:, :) :: carr
  END SUBROUTINE myroutine
END INTERFACE

... call myroutine(a)
```

Of course, for the recipes we have provided all the interface blocks in the file \texttt{nr.f90}, and you need only a \texttt{USE} \texttt{nr} statement in your calling program.

Conformable Arrays

Two arrays are said to be \textit{conformable} if their shapes are the same. Fortran 90 allows practically all operations among conformable arrays and elemental functions that are allowed for scalar variables. Thus, if $\texttt{arr}$, $\texttt{barr}$, and $\texttt{carr}$ are mutually conformable, we can write,

\[
\texttt{arr} = \texttt{barr} + \texttt{cos(carr)} + 2.0_{-}\text{sp}
\]

and have the indicated operations performed, element by corresponding element, on the entire arrays. The above line also illustrates that a scalar (here the constant 2.0_{-}\text{sp}, but a scalar variable would also be fine) is deemed conformable with \textit{any} array — it gets “expanded” to the shape of the rest of the expression that it is in. \textcite{M&R, §3.11}

In Fortran 90, as in Fortran 77, the default lower bound for an array subscript is 1; however, it can be made some other value at the time that the array is declared:

\[
\text{REAL(SP), DIMENSION(100,200) :: farr}
\]

\[
\text{REAL(SP), DIMENSION(0:99,0:199) :: garr}
\]

\[
... 
\text{farr} = 3.0_{-}\text{sp} * \text{garr} + 1.0_{-}\text{sp}
\]

Notice that \texttt{farr} and \texttt{garr} are conformable, since they have the same shape, in this case (100, 200). Also note that when they are used in an array expression, the operations are taken between the corresponding elements \textit{of their shapes}, not necessarily the corresponding elements \textit{of their indices}. \textcite{M&R, §3.10} In other words, one of the components evaluated is,

\[
\text{farr}(1,1) = 3.0_{-}\text{sp} * \text{garr}(0,0) + 1.0_{-}\text{sp}
\]

This illustrates a fundamental aspect of array (or data) parallelism in Fortran 90. Array constructions should \textit{not} be thought of as merely abbreviations for do-loops
over indices, but rather as genuinely parallel operations on same-shaped objects, abstracted of their indices. This is why the standard makes no statement about the order in which the individual operations in an array expression are executed; they might in fact be carried out simultaneously, on parallel hardware.

By default, array expressions and assignments are performed for all elements of the same-shaped arrays referenced. This can be modified, however, by use of a where construction like this:

```fortran
where (harr > 0.0_sp)
    farr = 3.0_sp*garr + 1.0_sp
end where
```

Here harr must also be conformable to farr and garr. Analogously with the Fortran if-statement, there is also a one-line form of the where-statement. There is also a where ... elsewhere ... end where form of the statement, analogous to if ... else if ... end if. A significant language limitation in Fortran 90 is that nested where-statements are not allowed. [M&R, §6.8]

**Array Sections**

Much of the versatility of Fortran 90’s array facilities stems from the availability of array sections. An array section acts like an array, but its memory location, and thus the values of its elements, is actually a subset of the memory location of an already-declared array. Array sections are thus “windows into arrays,” and they can appear on either the left side, or the right side, or both, of a replacement statement. Some examples will clarify these ideas.

Let us presume the declarations

```fortran
REAL(SP), DIMENSION(100) :: arr
INTEGER(4B), DIMENSION(6) :: iarr=/(11,22,33,44,55,66)/
```

Note that iarr is not only declared, it is also initialized by an *initialization expression* (a replacement for Fortran 77’s DATA statement). [M&R, §7.5] Here are some array sections constructed from these arrays:

<table>
<thead>
<tr>
<th>Array Section</th>
<th>What It Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr(;)</td>
<td>same as arr</td>
</tr>
<tr>
<td>arr(1:100)</td>
<td>same as arr</td>
</tr>
<tr>
<td>arr(1:10)</td>
<td>one-dimensional array containing first 10 elements of arr</td>
</tr>
<tr>
<td>arr(51:100)</td>
<td>one-dimensional array containing second half of arr</td>
</tr>
<tr>
<td>arr(51;)</td>
<td>same as arr(51:100)</td>
</tr>
<tr>
<td>arr(10:1:-1)</td>
<td>one-dimensional array containing first 10 elements of arr, but in reverse order</td>
</tr>
<tr>
<td>arr( /10,99,1,6/ )</td>
<td>one-dimensional array containing elements 10, 99, 1, and 6 of arr, in that order</td>
</tr>
<tr>
<td>arr(iarr)</td>
<td>one-dimensional array containing elements 11, 22, 33, 44, 55, 66 of arr, in that order</td>
</tr>
</tbody>
</table>
Now let's try some array sections of the two-dimensional array

\[
\text{REAL(8P), DIMENSION(100,100) :: barr}
\]

<table>
<thead>
<tr>
<th>Array Section</th>
<th>What It Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>barr(:,:,              )</td>
<td>same as barr</td>
</tr>
<tr>
<td>barr(1:100,1:100)</td>
<td>same as barr</td>
</tr>
<tr>
<td>barr(7,:),</td>
<td>one-dimensional array containing the 7th row of barr</td>
</tr>
<tr>
<td>barr(7,1:100)</td>
<td>same as barr(7,:)</td>
</tr>
<tr>
<td>barr(:,7)</td>
<td>one-dimensional array containing the 7th column of barr</td>
</tr>
<tr>
<td>barr(21:30,71:90)</td>
<td>two-dimensional array containing the sub-block of barr with the indicated ranges of indices; the shape of this array section is (10,20)</td>
</tr>
<tr>
<td>barr(100:1:-1,100:1:-1)</td>
<td>two-dimensional array formed by flipping barr upside down and backwards</td>
</tr>
<tr>
<td>barr(2:100:2,2:100:2)</td>
<td>two-dimensional array of shape (50,50) containing the elements of barr whose row and column indices are both even</td>
</tr>
</tbody>
</table>

Some terminology: A construction like 2:100:2, above, is called a subscript triplet. Its integer pieces (which may be integer constants, or more general integer expressions) are called lower, upper, and stride. Any of the three may be omitted. An omitted stride defaults to the value 1. Notice that, if \((\text{upper} - \text{lower})\) has a different sign from \(\text{stride}\), then a subscript triplet defines an empty or zero-length array, e.g., \(1:5:-1\) or \(10:1:1\) (or its equivalent form, simply \(10:1\)). Zero-length arrays are not treated as errors in Fortran 90, but rather as “no-ops.” That is, no operation is performed in an expression or replacement statement among zero-length arrays. (This is essentially the same convention as in Fortran 77 for do-loop indices, which array expressions often replace.) [M&R, §6.10]

It is important to understand that array sections, when used in array expressions, match elements with other parts of the expression according to shape, not according to indices. (This is exactly the same principle that we applied, above, to arrays with subscript lower bounds different from the default value of 1.) One frequently exploits this feature in using array sections to carry out operations on arrays that access neighboring elements. For example,

\[
carr(1:n-1,1:n-1) = \text{barr}(1:n-1,1:n-1) + \text{barr}(2:n,2:n)
\]

constructs in the \((n-1) \times (n-1)\) matrix \(\text{carr}\) the sum of each of the corresponding elements in \(n \times n\) \(\text{barr}\) added to its diagonally lower-right neighbor.

Pointers are often used as aliases for array sections, especially if the same array sections are used repeatedly. [M&R, §6.12] For example, with the setup

\[
\text{REAL(8P), DIMENSION(:,::), POINTER :: lef,b, righb}
\]