1.

Introduction to Modern Fortran

Since the publication of the FORTRAN 77 standard in 1978, the Fortran language has undergone a large number of revisions [61].[1] The changes that were introduced reflect both new insights in programming techniques and new developments in computer hardware. From the very start, the language has been designed with computing efficiency in mind. The latest standard as of this writing, Fortran 2008, puts even more emphasis on this aspect by introducing explicit support for parallel processing [71].

This first chapter gives an overview of the various standards that have appeared after FORTRAN 77. There is no attempt to be complete or even to describe all major features, as that would mean a whole book or even a series of books. Consult Metcalf [63], [65] or Brainerd et al. [36] for a detailed description of the standards.

1.1 The Flavor of Modern Fortran

The Fortran 90 standard introduced some very significant changes with respect to the widespread FORTRAN 77 standard: free form source code, array operations, modules, and derived types to name a few. To give an impression of what this means for the programmer, consider this simple problem: you have a file with numbers, one per line (to keep it simple), and you want to determine the distribution of these numbers to produce a simple histogram. In FORTRAN 77, a program that does this might look like the following:

```
*
* Produce a simple histogram
*
      PROGRAM HIST

      INTEGER MAXDATA
      PARAMETER (MAXDATA = 1000)
```

[1] Officially, Fortran 77 should be written as *FORTRAN 77*. Since the *Fortran 90* standard, the name is written in lowercase.

```
      INTEGER NOBND
      PARAMETER (NOBND = 9)
      REAL BOUND(NOBND)

      REAL DATA(MAXDATA)
      INTEGER I, NODATA

      DATA BOUND /0.1, 0.3, 1.0, 3.0, 10.0, 30.0,
     &            100.0, 300.0, 1000.0/

      OPEN( 10, FILE = 'histogram.data',
                       STATUS = 'OLD', ERR = 900 )
      OPEN( 20, FILE = 'histogram.out' )

      DO 110 I = 1,MAXDATA
          READ( 10, *, END = 120, ERR = 900 ) DATA(I)
  110 CONTINUE
*
  120 CONTINUE
      CLOSE( 10 )
      NODATA = I - 1

      CALL PRHIST( DATA, NODATA, BOUND, NOBND )
      STOP
*
*     File not found, and other errors
*
  900 CONTINUE
      WRITE( *, * ) 'File histogram.data could not be opened'
     &              'or some reading error'

      END

*
* Subroutine to print the histogram
*
      SUBROUTINE PRHIST( DATA, NODATA, BOUND, NOBND )
      REAL DATA(*), BOUND(*)
      INTEGER NODATA, NOBND

      INTEGER I, J, NOHIST

      DO 120 I = 1,NOBND
         NOHIST = 0
         DO 110 J = 1,NODATA
             IF ( DATA(J) .LE. BOUND(I) ) THEN
                 NOHIST = NOHIST + 1
             ENDIF
```

```
  110    CONTINUE

         WRITE( 20, '(F10.2,I10)' ) BOUND(I), NOHIST
  120 CONTINUE

      END
```

Since Fortran 90, this program can be rewritten in the so-called *free form*, using various inquiry functions and array operations:

```
! Produce a simple histogram
!
program hist
    implicit none

    integer, parameter :: maxdata = 1000
    integer, parameter :: nobnd   = 9

    real, dimension(maxdata) :: data
    real, dimension(nobnd)   :: &
        bound = (/0.1, 0.3, 1.0, 3.0, 10.0, &
                30.0, 100.0, 300.0, 1000.0/)

    integer                  :: i, nodata, ierr

    open( 10, file = 'histogram.data', status = 'old', &
        iostat = ierr )

    if ( ierr /= 0 ) then
        write( *, * ) 'file histogram.data could not be opened'
        stop
    endif

    open( 20, file = 'histogram.out' )

    do i = 1,size(data)
        read( 10, *, iostat = ierr ) data(i)

        if ( ierr > 0 ) then
            write( *, * ) 'Error reading the data!'
            stop
        elseif ( ierr < 0 ) then
            exit ! Reached the end of the file
        endif
    enddo

    close( 10 )
    nodata = i - 1
```

```
    call print_history( data(1:nodata), bound )

contains

! subroutine to print the histogram
!
subroutine print_history( data, bound )
    real, dimension(:), intent(in) :: data, bound

    integer :: i

    do i = 1,size(bound)
        write( 20, '(f10.2,i10)' ) &
            bound(i), count( data <= bound(i) )
    enddo
end subroutine print_history

end program hist
```

The main differences are:

- Fortran 90 and later allow the free form and lower-case program text, though many FORTRAN 77 compilers did allow for this as well as an extension.
- The introduction of the statement `implicit none` causes the compiler to check that all variables are actually declared with an explicit type, removing a whole class of programming errors.
- By using an *internal* routine (indicated by the `contains` statement), you can ensure that the compiler checks the correctness of the actual arguments so far as number and types are concerned.[2]
- You have no more need for the infamous `GOTO` statement in this program, therefore, it can be replaced by its more structured counterpart `exit` to terminate the do loop.
- You can use array sections (such as `data(1:nodata)`) to pass only that part of the array `data` that is of interest, and the inquiry function `size()` allows you to get the appropriate number of elements. This also means you can remove the two arguments that indicate the sizes.
- Finally, you have eliminated an entire do loop in the code by using the standard function `count()` to determine the histogram data.

Note, however, that the first program is still completely valid as far as modern Fortran standards are concerned. This is a very important aspect of Fortran. It means that you can gradually introduce modern features, rather than rewrite an entire program.

[2] This is actually only one effect of internal routines – (see Section 1.2).

1.2 Fortran 90

The Fortran 90 standard introduced a large number of features. The best known are perhaps those involving array operations, but there are many more:

- The `implicit none` statement requires the user to explicitly declare all variables and it requires the compiler to check this. Using this feature means typing errors are much less likely to inadvertently introduce new variables.
- *Modules*, together with the *public* and *private* statements or attributes, provide an effective means to partition the entire program into smaller units that can only be accessed when explicitly stated in the source code. Modules furthermore provide explicit interfaces. to the routines they contain, which makes it possible for the compiler to perform all manner of checks and optimizations.

  Should this not be possible (such as when dealing with routines in a different language), you can use *interface blocks*.
- The main program but also subroutines and functions can contain so-called *internal routines*. These routines have access to the variables of the containing routine but provide a new scope in which to define local variables. It is an additional method for modularizing the code.
- As modern computers allow a flexible memory management that was not at all ubiquitous when the FORTRAN 77 standard was published, several features of Fortran 90 relate to managing the memory:
  - *Recursive routines* are now allowed, making it far easier to implement recursive algorithms.
  - Via the *allocate* and *deallocate* statements, programmers can adjust the size of arrays to fit the problem at hand. Arrays whose size can be adjusted come in two flavors: *allocatable* and *pointer*. The former offers more opportunities for optimization, whereas the latter is much more flexible.
  - Besides explicit allocation, the programmer can also use *automatic* arrays – arrays that get their size from the dummy arguments and that are automatically created and destroyed upon entry or exit of a subroutine or function.
- *Array operations* are an important aspect of the Fortran 90 standard, as they allow a concise style of programming and make the optimization task much simpler for the compiler. These operations are supported in arithmetic expressions but also via a collection of generic standard functions.

  Not only can you manipulate an entire array, but you can also select a part, defined by the start, stop, and stride in any dimension.

  User-defined functions can also return arrays as a result, adding to the flexibility and usefulness of array operations.
- Besides the obsolete *fixed form* that characterized Fortran from the start, there is the *free form*, where columns 1 to 6 no longer have a special meaning. This makes source code look much more modern.

- Arrays of values can be constructed on the fly, via so-called array constructors. This is a powerful mechanism that can be put to good use to fill an array with values.
- Just as in most other modern languages, Fortran allows the definition of new data structures. The basic mechanism is that of *derived types*. Derived types consist of components of different types – either basic types, like integers or reals, or other derived types. You can use them in much the same way as the basic types – pass them to subroutines, use them as return values, or create arrays of such types.

  Moreover, you can use derived types to create linked lists, trees, and other well-known abstract data types.
- *Overloading* of routines and operations, such as addition and subtraction, makes it possible to extend the language with new full-blown types. The idea is that you define generic names for specific routines or define `+`, `-`, and so forth for numerical types that are not defined by the standard (for instance, rational numbers).

  It is in fact possible to define your own operations. A simple example: suppose your program deals with planar geometrical objects, then it may make sense to define an *intersection* operation on two such objects, `object_a .intersects. object_b`, to replace the call to a function `intersect_objects( object_a, object_b )`.
- Functions and subroutines can now have *optional* arguments, where the function `present()` determines if such an argument is present or not. You can also call functions and subroutines with the arguments in an arbitrary order, as long as you add the names of the dummy arguments, so the compiler can match the actual arguments with the dummy arguments.

  A further enhancement is that you can specify the *intent* of an argument: whether it is input only, output only, or both input and output. This is an aid to documentation as well as an aid to the compiler for certain optimizations.
- *Kinds* are Fortran's way of determining the characteristics of the basic types. For example, you can select single or double precision for real variables not via a different type (`real` or `double precision`) but via the `kind`:

```
integer, parameter :: double = &
    select_kind_real( precision, range )
real(kind=double)  :: value
```

- Besides the introduction of the `select/case` construct, `do while` loops and `do` loops without a condition, you can use a name for all control structures. This makes it easier to document the beginning and end of such structures.

  To skip the rest of a do loop's body you can now use the `cycle` statement – possibly with the name of the do loop – so that you can skip more than one do loop in a nested construction. Similarly, the `exit` statement terminates the do loop.

- The Fortran standard defines a large number of functions and subroutines:
  - Numerical inquiry functions for retrieving the properties of the floating-point model.
  - Array manipulation functions that often take an array expression as one of their arguments. For instance, you can count the number of positive elements in an array using:

```
integer, dimension(100,100) :: array
...
write(*,*) 'Number of positive elements:', &
    count( array > 0 )
```

  - Character functions and bit manipulation functions
- Enhancements of the I/O system include nonadvancing I/O. That is, rather than whole records, a program can read or write a part of a record in one statement and the rest in another one.

1.3 Fortran 95

As the Fortran 95 standard is a minor revision of the previous one, the differences mostly concern details. However, these details are important:

- In Fortran 90, variables with the *pointer* attribute could not be initialized. Their initial status was explicitly undefined: neither associated nor not associated. Fortran 95 introduces the *null()* function to initialize pointers explicitly:

```
real, dimension(:), pointer :: ptr => null()
```

- A further enhancement is that local *allocatable* variables without the *save* attribute are automatically deallocated, when they go out of scope (upon returning from a subroutine or function). It is safe for the compiler to do so, as the memory cannot be reached by any means afterwards.
- As a preparation for the next standard, a technical report describes how derived types may contain allocatable components and how functions can return allocatable results. (This technical report has become part of the Fortran 2003 standard with some additions.)
- New features in Fortran 95 are the *pure* and *elemental* routines. Elemental routines release the programmer from having to write versions of a routine for arrays of all the dimensions they want to support. Many functions and subroutines in the standard library already were elemental, meaning they work on individual elements of the arrays that are passed without regard for any order. With the advent of Fortran 95, programmers themselves can also write such routines.

  Pure routines are routines that provide the compiler more opportunities to optimize, as, roughly, they cause no side effects.

- Finally, the *forall* statement must be mentioned, known from High Performance Fortran. It was designed to enhance the capabilities of array operations, but in practice it turns out to be difficult to use properly. (In Fortran 2008, a more flexible construct, *do concurrent*, is introduced.)

1.4 Fortran 2003

Fortran 2003 is also a major revision and its main theme is the introduction of object-oriented programming.[3] However, there are more enhancements, for instance, a standardization of interacting with C routines.

- The support for **object-oriented programming** comes from several new features:
  - Derived types can now contain procedures (functions and subroutines). These are either bound to the type, so that all variables of a particular type have the same actual procedure, or they can be procedures particular to a variable. Such procedures automatically get passed the variable (or object if you like) as one of their arguments, but you can control which one. (We will discuss these matters extensively in Chapter 11.)
  - Derived types can be extended into new types. This is Fortran's inheritance mechanism. An extended type can have new components (both data and procedures), it can redefine the procedures that are bound to the parent type, but it cannot change their signature.
  - The *select* statement has been enhanced to select on the *type* of the variable. This is important if you use so-called *polymorphic* variables – pointer variables whose type comes from the variable they are associated with while the program is running. Polymorphic variables are declared via the *class* keyword instead of the *type* keyword.[4]
  - To enhance the flexibility of the extension mechanism, procedures can be characterized by *abstract interfaces*. Rather than defining a particular routine, these interfaces define what a routine should look like in terms of its argument list and return value. This can then be applied as a mold for actual routines.
- While *procedure pointers* will most often occur within derived types, you can use them as "ordinary" variables as well.
- To achieve certain special effects, the concept of *intrinsic modules* has been introduced. Examples of these special effects are the control of the floating-point environment: rounding mode, the effect of floating-point exceptions,

[3] Fortran 90 possesses quite a few features that allow a programmer to come close to that style of programming, but it lacks inheritance, often viewed as one of the important characteristics of object-oriented programming. Fortran 90 is, therefore, sometimes called **object-based.**

[4] Rouson and Adalsteinsson [73] compare the terminology for object-oriented concepts in Fortran and C++.

but also the interfacing to C, as that sometimes requires a different calling and naming convention.

- Memory management has been enhanced:
  - The length of character strings can now be determined by the `allocate` statement.
  - Under some circumstances an allocatable array can be automatically reallocated to the correct size.
  - It is possible to move the allocated memory from one variable to the next via the *move_alloc* routine. This makes it easier to expand an array.
- Another welcome new feature is that of *stream access* to files. This makes it possible to read (and write) files without attention to complete records. In particular, it makes it possible to read and write binary files that contain no intrinsic structure, unlike the traditional *unformatted* files that do have a record structure and are cumbersome to deal with in other programming languages.
- Fortran 2003 also standardizes the access to the system's environment in the form of *environment variables* and to the command-line arguments that were used to start the program. Previously, you had to use all manner of compiler-specific solutions to access this information.

1.5 Fortran 2008

The Fortran 2008 standard is a minor revision to the language, even though it defines a rather large new feature: coarrays. These are discussed more fully in Chapter 12 [72], [71]. Besides these, the standard defines a number of new constructs and keywords as well as new standard functions:

- Coarrays are a mechanism for parallel computing that falls in the category of *Partitioned Global Address Space* (PGAS). Essentially, it makes data available on various copies of the program without the programmer having to worry about the exact methods that have to be used to transfer the data. It is the compiler's job to generate code that does this efficiently and effectively.
- Arrays can be defined to be *contiguous*. This enables the compiler to generate more efficient code, as the array elements are adjacent in memory.
- The *block – end block* construct defines a local scope within a program or routine, so that new variables can be declared that are present only within that block.
- Modules as introduced in Fortran 90 are an important mechanism to modularize a program. However, the module mechanism itself does not allow much modularization, as the entire module must be compiled. Fortran 2003 introduces *submodules* to overcome the problem of very large source files containing a large module. An additional enhancement is the *import* feature. This is used in interface blocks to import a definition from the containing module, instead of having to define a second module containing just the definitions you require.

- To further reduce the need for a GOTO statement, the *exit* statement can be used to jump to the end of if blocks and select blocks.
- The *do concurrent* statement may be regarded as a more flexible alternative to the *forall* statement/block. It indicates to the compiler that the code can be run in parallel (note that this is parallellism within the same *copy*, not between copies as with coarrays).
- *Internal procedures* may now be passed as actual arguments, thus giving access to the variables in the routine that contains them. It makes certain interface problems easier to solve (see Chapter 5).
- New standard functions include various Bessel functions and bit inquiry functions.

1.6 What Has Not Changed?

The introduction of all these new features to the Fortran language does not have consequences for existing code that adheres to one or the other standards, with the exception of a few features that have been deleted or obsoleted.[5]

Apart from such deleted, and perhaps almost forgotten features, like the assigned GOTO, old code should still work.[6]

There are in fact more *invariants* in the design of modern Fortran, besides this support for old code:

- Fortran is case-insensitive, contrary to many other programming languages in use today.
- There is no concept of a *file* as an organizational unit for the source code. In particular, there can be no code outside the program, subroutine, function, or module.
- The design of Fortran is biased to efficient execution. This is especially so in the latest Fortran 2008 standard, in which features have been added to help the compiler in the creation of fast and efficient programs.

  The details may surprise programmers who are used to languages like C (see Appendix B).
- The data type of an expression or subexpression depends solely on the operands and the operation, not on the context. This makes reasoning about a program much easier, but again it may cause surprises (Appendix B). Here is an example:

```
real :: r

r = 1 / 3
```

[5] The most important of these deleted features is the use of real variables to control a do loop ([68], Section 2.1.5).
[6] Most compilers continue to support such features, so that old programs can still be compiled and run.