University of Cambridge
Department of Physics

Computational Physics

Self-study guide 2

Programming in Fortran 95

Dr. Rachael Padman
Michaelmas 2007

Contents

Acknowledgements:
This handout was originally prepared by Dr. Paul Alexander, and has been updated and maintained by Dr Peter Haynes of the TCM group.

1. The Basics

In this section we will look at the basics of what a program is and how to make the program run or *execute*.

The non-trivial example programs can be found in the directory:

```
$PHYTEACH/part_2/examples
```

with the name of the file the same as that of the program discussed in this guide.

Some sections are more advanced and are indicated clearly indicated by a thick black line to the right of the text. These can be skipped certainly on a first reading and indeed you will be able to tackle the problems without using the material they discuss.

1.1 A very simple program

A program is a set of instructions to the computer to perform a series of operations. Those operations will often be mathematical calculations, decisions based on equalities and inequalities, or special instructions to say write output to the screen. The program consists of "source code" which is "stored" in a text file. This code contains the instructions in a highly structured form. Each computer language has a different set of rules (or *syntax*) for specifying these operations. Here we will only consider the Fortran 90/95 (F95 for short) programming language and syntax.

- Using `emacs` enter the following text into a file called `ex1.f90`, the `.f90` part of the file name is the extension indicating that this is program source code written in the Fortran 90/95 language

```
program ex1

  !
  ! My first program
  !
  write(*,*) 'Hello there'

end program ex1
```

This is a complete F95 program.

The first and last lines introduce the start of the program and show where it ends. Between the first and last lines are the program "statements". The lines beginning with an exclamation mark are special statements called comments. They are not instructions to the computer, but instead are there to enable us (the programmer) to improve the readability of the program and help explain what the program is doing.

The line beginning `write` is a statement giving a specific instruction to print to the screen.

Note that except within quotes:

⇒ Upper and lower case are NOT significant (different from Unix commands and files)

⇒ Blank lines and spaces are not significant.

1.2 Running the program

Before we can run the program we must get the computer to convert this symbolic language (F95) into instructions it can understand directly. This process is called "*compilation*". At the same time the computer will check our program source for errors in the syntax, but not for errors in our logic! In general programs will be assembled from source in many files; bringing all of these instructions together is called "*linking*". We perform both of these tasks using the Unix command `f95`.

- Type the following, the `-o` is an option saying where to place the output which in this case is a program which is ready to run, we call this an *executable*. (The default executable name is `a.out`).

```
f95 -o ex1 ex1.f90
```

  - If you haven't made any typing errors there should be no output to the screen from this command, but the file `ex1` should have been created. By convention executable programs under Unix do not normally have a file extension (i.e. no "`.xxx`" in the file name).

- To run the program type:

```
./ex1
```

  - Most Unix commands are files which are executed. The shell has a list of directories to search for such files, but for security reasons this list does not contain the current directory. The '`./`' (dot slash) before `ex1` tells the shell explicitly to look in the current directory for this file.
  - The output should be the words "`Hello there`".

- What happens if you make an error in the program? To see this let's make a deliberate error. Modify the line beginning `write` to read:

```
write(*,*) 'Hello there'  'OK'
```

  - Save the file, and compile again :

```
f95 -o ex1 ex1.f90
```

  - This time you get errors indicating that the syntax was wrong; i.e. you have not followed the rules of the F95 language! Correct the error by changing the source back to the original, recompile and make sure the program is working again.

1.3 Variables and expressions

The most important concept in a program is the concept of a variable. Variables in a program are much like variables in an algebraic expression, we can use them to hold values and write mathematical expressions using them. As we will see later F95 allows us to have variables of different types, but for now we will consider only variables of type `real`. Variables should be declared before they are used at the start of the program. Let us use another example to illustrate the use of variables.

- Enter the following program and save it to the file `ex2.f90`

```
program vertical
  !
  ! Vertical motion under gravity
  !
  real :: g          ! acceleration due to gravity
  real :: s          ! displacement
  real :: t          ! time
  real :: u          ! initial speed ( m / s)

  ! set values of variables
  g = 9.8
  t = 6.0
  u = 60

  ! calculate displacement
  s = u * t - g * (t**2) / 2

  ! output results
  write(*,*) 'Time = ',t,'   Displacement = ',s

end program vertical
```

- Compile and run the program and check the output is what you expect

```
f95 -o ex2 ex2.f90
./ex2
```

This program uses four variables and has many more statements than our first example. The variables are "declared" at the start of the program before any executable statements by the four lines:

```
real :: g          ! acceleration due to gravity
real :: s          ! displacement
real :: t          ! time
real :: u          ! initial speed ( m / s)
```

After the declarations come the executable statements. Each statement is acted upon sequentially by the computer. Note how values are assigned to three of the variables and then an expression is used to calculate a value for the fourth (`s`).

Unlike in an algebraic expression it would be an error if, when the statement calculating the displacement was reached, the variables `g`, `t` and `u` had not already been assigned values.

Some other things to note:

1. Comments are used after the declarations of the variables to explain what each variable represents.
2. The '`*`' represents multiplication
3. The '`**`' is the operator meaning "raise to the power of", it is called technically *exponentiation*.
4. In this program we have used single letters to represent variables. You may (and should if it helps you to understand the program) use longer names. The variable names should start with a character (`A-Z`) and may contain any character (`A-Z`), digit (`0-9`), or the underscore (`_`) character.
5. Upper and lower case are not distinguished. For example therefore the variables `T` and `t`, and the program names `vertical and Vertical` are identical.

The usefulness of variables is that we can change their value as the program runs.

All the standard operators are available in expressions. An important question is if we have the expression

```
g * t **2
```

what gets evaluated first? Is it `g*t` raised to the power of `2` or `t` raised to the power `2` then multiplied by `g`? This is resolved by assigning to each operator a precedence; the highest precedence operations are evaluated first and so on. A full table of numeric operators is (in decreasing precedence)

Operator	**Precedence**	**Meaning**
`**`	1	Raise to the power of
`*`	2	Multiplication
`/`	2	Division
`+`	3	Addition or unary plus
`-`	3	Subtraction or unary minus

You can change the precedence by using brackets; sub-expressions within brackets are evaluated first.

Let's look at ways of improving this program. An important idea behind writing a good program is to do it in such a way so as to avoid errors that you may introduce yourself! Programming languages have ways of helping you not make mistakes. So let's identify some possible problems.

- The acceleration due to gravity is a constant, not a variable. We do not wish its value to change.

- We want to avoid using a variable which is not given a value; this could happen if we mistyped the name of a variable in one of the expressions.

Consider the following modified form of our program:

```
program vertical
  !
  ! Vertical motion under gravity
  !
  implicit none

  ! acceleration due to gravity
  real, parameter :: g = 9.8

  ! variables
  real :: s           ! displacement
  real :: t           ! time
  real :: u           ! initial speed ( m / s)

  ! set values of variables
  t = 6.0
  u = 60

  ! calculate displacement
  s = u * t - g * (t**2) / 2

  ! output results
  write(*,*) 'Time = ',t,'   Displacement = ',s

end program vertical
```

We have changed three lines and some of the comments. The line:

```
implicit none
```

is an important statement which says that all variables must be defined before use. You should always include this line in all programs.[1]

The second change is to the line:

```
real, parameter :: g = 9.8
```

This in fact defines `g` to be a *constant* equal to the value 9.8; an attempt to reassign `g` via a statement like the one in the original version (`g = 9.8` on a line by itself) will now lead to an error. The syntax of this statement is as follows:

> After the definition of the variable type `real` we give a series of options separated by commas up until the '`::`' after which we give the variable name with an optional assignment.

[1] It is an unfortunate legacy of older versions of Fortran that you could use variables without defining them, and in that case Fortran supplied rules to determine what the variable type was.

We will meet more options later.

Try out these new ideas:

- Make these changes and make sure the program compiles.
- Now make some deliberate errors and see what happens. Firstly add back in the line `g = 9.8` but retain the line containing the `parameter` statement.
- Compile and observe the error message.
- Now change one of the variables in the expression calculating `s,` say change `u` to `v.` Again try compiling.
- Fix the program.

1.4 Other variable types: `integer`, `complex` and `character`

As we have hinted at, there are other sorts of variables as well as `real` variables. Important other types are `integer`, `complex` and `character`.

Let's first consider `integer` variables; such variables can only hold integer values. This is important (and very useful) when we perform calculations. It is also worth pointing out now that F95 also distinguishes the type of values you include in your program. For example a values of '`3.0`' is a `real` value, whereas a value of '`3`' without the '`.0`' is an `integer` value. Some examples will illustrate this.

Enter the following program:

```
program arithmetic
  implicit none

  ! Define real and integer variables
  real :: d, r, rres
  integer :: i, j, ires

  ! Assign some values
  d = 2.0 ; r = 3.0
  i = 2 ; j = 3

  ! Now the examples
  rres = r / d
  ! Print the result, both text and a value.
  ! Note how the text and value are separated by
  ! a comma
  write(*,*) 'rres = r / d : ',rres

  ! now some more examples
  ires = j / i; write(*,*) 'ires = j / i : ',ires
  ires = r / i; write(*,*) 'ires = r / i : ',ires
  rres = r / i; write(*,*) 'rres = r / i : ',rres

end program arithmetic
```

First some things to note about the program:

1. We can declare more than one variable of the same type at a time by separating the variable names with commas:
```
real :: d, r, rres
```

2. We can place more than one statement on a line if we separate them with a semicolon:
```
d = 2.0 ; r = 3.0
```

- Compile and run the program. Note the different output. The rule is that for integer division the result is truncated towards zero. Note that the same rules apply to expressions containing a constant. Hence:
```
ires = 10.0 / 3     !  value of ires is 3
rres = 10 / 3       !  value of rres is 3.0
rres = 10.0 / 3.0   !  value of rres is 3.333333
```

- Make sure you are happy with these rules; alter the program and try other types of expression.

Some expressions look a little odd at first. Consider the following expression:

```
n = n + 1
```

where `n` is an integer. The equivalent algebraic expression is meaningless, but in a program this is a perfectly sensible expression. We should interpret as:

"*Evaluate the right hand side of the expression and set the variable on the left hand side to the value evaluated for the right hand side*".

The effect of the above expression is therefore to increment the value of `n` by `1`. Note the role played by the '=' sign here: it should be thought of not as an equality but instead as an "assignment".

The `complex` type represents complex numbers. You can do all the basic numerical expressions discussed above with complex numbers and mix `complex` and other data types in the same expression. The following program illustrates their use.

- Enter the program, compile and run it. Make sure you understand the output.

```
program complex1
  implicit none

  ! Define variables and constants
  complex, parameter :: i = (0, 1)   ! sqrt(-1)
  complex :: x, y

  x = (1, 1); y = (1, -1)
  write(*,*) i * x * y

end program complex1
```

The `character` data type is used to store strings of characters. To hold a string of characters we need to know how many characters in the string. The form of the definition of `character`s is as follows:

```
character (len = 10) :: word
! word can hold 10 characters
```

We will meet character variables again later.

Rules for evaluating expressions

The type of the result of evaluating an expression depends on the types of the variables. If an expression of the form **a** *operator* **b** is evaluated, where *operator* is one of the arithmetic operations above (`+, -, *, /, **`) the type of the result is given as follows with the obvious symmetric completion:

Type of a	**Type of b**	**Type of result**
`integer`	`integer`	`integer`
`integer`	`real`	`real`
`integer`	`complex`	`complex`
`real`	`real`	`real`
`real`	`complex`	`complex`
`complex`	`complex`	`complex`

N.B. The result of evaluating an `integer` expression is an `integer`, truncating as necessary. It is worth emphasising this again, although we met it above, since a very common error is to write '`1 / 2`' for example, which by the above rules evaluates to zero. This can lead to non-obvious errors if hidden in the middle of a calculation.

When a complex value is raised to a complex power, the principal value (argument in the range $-\pi$, π) is taken.

Assignments take the form **variable** = *expression*, where **variable** has been declared and therefore has a type. If the type of the two do not agree, the following table determines the result

Variable	**Expression**	**Value assigned**
`integer`	`real`	truncated value
`integer`	`complex`	truncated real part
`real`	`integer`	convert to real
`real`	`complex`	real part
`complex`	`integer`	real part assigned value, imaginary part zero
`complex`	`real`	real part assigned value, imaginary part zero

1.5 Intrinsic functions

So far we have seen how to perform simple arithmetic expressions on variables. Real problems will involve more complicated mathematical expressions. As we shall see later, F95 enables you to define your own functions which return values. However, some functions are so common and important that they are provided for us as part of the language; these are called *intrinsic* functions.

Let us consider a program to compute projectile motion. The program computes the horizontal, x, and vertical, y, position of the projectile after a time, t:

$$x = u\,t\cos a \qquad\qquad y = u\,t\sin a - g\,t^2/2$$

```
program projectile
  implicit none

  ! define constants
  real, parameter :: g = 9.8
  real, parameter :: pi = 3.1415927

  real :: a, t, u, x, y
  real :: theta, v, vx, vy

  ! Read values for a, t, and u from terminal
  read(*,*) a, t, u

  ! convert angle to radians
  a = a * pi / 180.0

  x = u * cos(a) * t
  y = u * sin(a) * t – 0.5 * g * t * t

  vx = u * cos(a)
  vy = u * sin(a) - g * t
  v = sqrt(vx * vx + vy * vy)
  theta = atan(vy / vx) * 180.0 / pi

  write(*,*) 'x: ',x,'  y: ',y
  write(*,*) 'v: ',v,'  theta: ',theta

end program projectile
```

- Compile and run the program. It will wait. The statement "`read(*,*)`…" is requesting input from you. Enter three values for a, t and u. You should now get some output.
- Examine this program carefully and make sure you understand how it works.
- Note especially how we use the functions `cos`, `sin`, `atan` and `sqrt` much as you would use them in algebraic expressions. As always upper and lower case are equivalent.

Common Intrinsic Functions

Name	Action
`ABS(A)`	absolute value of any `A`
`ACOS(X)`	inverse cosine in the range (0,π) in radians
`AIMAG(Z)`	imaginary part of `Z`
`AINT(X [,KIND])`	truncates fractional part towards zero, returning `real`
`ANINT(X [,KIND])`	nearest integer, returning `real`
`ASIN(X)`	inverse sine in the range (-π/2,π/2) in radians
`ATAN(X)`	inverse tangent in the range (-π/2,π/2) in radians
`ATAN2(Y,X)`	inverse tangent of `Y/X` in the range (-π,π) in radians
`CMPLX(X [,Y][,KIND]`	converts to `complex` `X+iY`; if `Y` is absent, 0 is used
`CONJG(Z)`	complex conjugate of `Z`
`COS(W)`	cosine of argument in radians
`COSH(X)`	hyperbolic cosine
`EXP(W)`	exponential function
`FLOOR(X)`	greatest `integer` less than `X`
`INT(A [,KIND])`	converts to integer, truncating (real part) towards zero
`KIND(A)`	`integer` function, returns the `KIND` of the argument
`LOG(W)`	natural logarithm: if `W` is `real` it must be positive, if `W` is `complex`, imaginary part of result lies in (-π,π)
`LOG10(X)`	logarithm to base 10
`MAX(R1,R2...)`	maximum of arguments, all of the same type
`MIN(R1,R2...)`	minimum of arguments, all of the same type
`MOD(R1,R2)`	remainder of `R1` on division by `R2`, both arguments being of the same type (`R1-INT(R1/R2)*R2`)
`MODULO(R1,R2)`	`R1` modulo `R2`: `(R1-FLOOR(R1/R2)*R2)`
`NINT(X [,KIND])`	nearest `integer`
`REAL(A [,KIND])`	converts to `real`
`SIGN(R1,R2)`	absolute value of `R1` multiplied by the sign of `R2`
`SIN(W)`	sine of argument in radians
`SINH(X)`	hyperbolic sine
`SQRT(W)`	square root function; for complex argument the result is in the right half-plane; a real argument must be positive
`TAN(X)`	tangent of argument in radians
`TANH(X)`	hyperbolic tangent

- F95 has a set of over a hundred intrinsic functions, those in the list above are the most useful for scientific applications.
- In this list `A` represents any type of numeric variable, `R` a `real` or `integer` variable, `X` and `Y` `real` variables, `Z` a `complex` variable, and `W` a `real` or `complex` variable.
- Arguments in square brackets are optional. For an explanation of `kind` see section 7.

1.6 Logical controls

So far all the programming statements we have met will simply enable us to produce efficient calculators. That is useful, but there is a lot more to programming. In this and Section 1.8 we introduce two crucial ideas. The first is the idea of taking an action conditional upon a certain criteria being met. An example will help to introduce this idea. For many years it was the case in Part IA of the Tripos that your maths mark was only included if it improved your overall result. Let us write a program to perform that simple sum. We read in four marks and output a final average.

```
program tripos1
  implicit none

  real :: p1, p2, p3, maths
  real :: av1, av2

  ! read in the marks
  read(*,*) p1, p2, p3, maths

  ! work out two averages
  av1 = p1 + p2 + p3
  av2 = av1 + maths
  av1 = av1 / 3.0 ; av2 = av2 / 4.0

  ! use an if statement
  if (av2 > av1) then
     write(*,*) 'Final average = ',av2
  else
     write(*,*) 'Final average = ',av1
  end if

end program tripos1
```

- Compile and run this program and make sure you understand how it works.
- Note how the statements are indented. We use indenting to help show the logical structure of the program; indented statements are executed depending on the output of the test done by the `if` statement. The indenting is not essential, but it leads to a program which is much easier to follow. If you choose this style you can indent each level by any number of spaces as you wish.

The `if` statement is the simplest, but most important, of a number of ways of changing what happens in a program depending on what has gone before. It has the general form:

`if (`*logical expression*`)` *action*

As another example we can use it to check for negative values:

```
if (x < 0) x=0  ! replace negative x with zero
```

The `if` construct may also be used in more extended contexts (as above), such as:

```
if (logical expression) then
     xxx
else
     xxx
end if
```

Here if the condition is false the statements following the `else` are executed. We can also include additional tests which are treated sequentially; the statements following the *first* logical test to be reached which is true are executed:

```
if (logical expression) then
     xxx
else if (logical expression) then
     xxx
else
     xxx
end if
```

Operators which may occur in logical expression are as follows:

`.lt.`	or	`<`	less than
`.le.`	or	`<=`	less than or equal
`.eq.`	or	`==`	equal
`.ge.`	or	`>=`	greater than or equal
`.gt.`	or	`>`	greater than
`.ne.`	or	`/=`	not equal
`.not.`			not
`.and.`			and
`.or.`			inclusive or

and of course, brackets. Using brackets and the `.not.`, `.and.` and `.or.` forms we can build up complicated logical expressions.

- As an exercise consider the following. Suppose the rules for Part IA of the Tripos were changed so that:
    1. The full maths course is always counted in the average
    2. Quantitative biology mark is only counted if it improves the average
    3. Elementary maths for biology is never counted.

- Modify the program `tripos1` to compute the average mark. One further piece of information is required which is an `integer` code indicating the type of maths paper taken. This integer code can be assumed to take the values:

Full maths	1
Quantitative biology	2
Elementary maths	3

- One possible solution is available in the examples directory as `tripos2.f90`

`if` clauses may appear nested, that is one inside another. Suppose we wish to compute the expression $x = \left(b\sqrt{d}\right)/a$ which fails if $d < 0$ or a is zero. If these were entered by a user then they could (incorrectly) take on these values. A good program should check this. Here is some code to do this which illustrates nested `if` clauses

```
if (a /= 0.0) then
   if (d < 0.0) then
      write(*,*) 'Invalid input data d negative'
   else
      x = b * sqrt(d) / a
   end if
else
   write(*,*) 'Invalid input data a zero'
end if
```

1.7 Advanced use of `if` and `logical` comparisons

In a large program it is likely that `if` clauses will be nested, i.e. appear one within another. This causes us no problems, but might make it less clear which `end if` goes with which `if`. To overcome this we can name the `if` clauses. An example illustrates the syntax. Let's use the example we have just met:

```
outer: if (a /= 0.0) then
   inner: if (d < 0.0) then
      write(*,*) 'Invalid input data d negative'
   else inner
      x = b * sqrt(d) / a
   end if inner
else outer
   write(*,*) 'Invalid input data a zero'
end if outer
```

The names are `outer` and `inner`; note the syntax, especially the colon. Named `if` clauses are useful when you want to make your intention clear, but are not essential.

The logical expressions we have met in `if` clauses can be used more generally with a logical variable. Logical variables take on the value of `.true.` or `.false.`. Here is a simple example which illustrates their use.

```
logical :: l1, l2

l1 = x > 0.0
l2 = y /= 1.0
if (l1 .and. l2) then…
```

This program fragment could equally well have been written

```
if ((x > 0.0) .and. (y /= 1.0)) then
```

Using logical variables may make some things easier to understand.

1.8 Repeating ourselves with loops: `do`

Loops are the second very important concept needed in a program. If a set of instructions needs to be repeated, a loop can be used to do this repetition. As we shall see we have a lot of control over the loop and this makes them extremely powerful; this is especially true when combined with the `if` clauses we have just met.

The general form of the `do` loop is:

```
do var = start, stop [,step]
   xxx
end do
```

where as before the parts in square brackets are optional.

- *var* is an `integer` variable
- *start* is the initial value *var* is given
- *stop* is the final value
- *step* is the increment by which *var* is changed. If it is omitted, unity is assumed

The loop works by setting *var* to *start*. If *var* ≤ *stop* the statements up to the `end do` are executed. Then *var* is incremented by *step*. The process then repeats testing *var* against *stop* each time around the loop.

- It is possible for the included statements never to be executed, for instance if *start* > *stop* and *step* is 1.

This program is an example which computes factorials:

```
program factorial
  implicit none

  ! define variables, some with initial values
  integer :: nfact = 1
  integer :: n

  ! compute factorials
  do n = 1, 10
     nfact = nfact * n
     write(*,*) n, nfact
  end do
end program factorial
```

- Modify the `factorial` program as follows. Change `10` to `100` and insert the following line before the `end do`.

  ```
  if (n > 10) exit
  ```

  - What output do you get? Why? The `exit` command terminates the loop.

- Write a program to calculate the binomial coefficient ${}_nC_r$. The program should read in values from the user for *n* and *r* and write out the answer.

1.9 The `stop` statement

We have just seen how the `exit` command can be used to terminate a `do` loop. If you wish execution of your program to cease, you can insert a `stop` statement; this can incorporate some text, which is output when your program halts and identifies where this happened, e.g.

```
stop 'this is where it all ends up'
```

1.10 Arrays

A great deal of scientific computation involves the manipulation of vectors, matrices and more general arrays of numbers. In F95 we can have an array of variables set up in the declarations statements.

How do we specify arrays? The simplest way is to give the dimension in parentheses.

```
real :: a(3)   ! a is an array of 3 values: a vector
real :: m(3,3) ! m is a rank 2 array: a matrix
```

We call the part in parentheses a *shape*. Each element of the array can be addressed in the program using a similar notation. Here is a simple example:

```
program vector
  implicit none

  real :: v(3)
  real :: x
  integer :: i

  v(1) = 0.25
  v(2) = 1.2
  v(3) = 0.2

  ! compute the modulus squared of the vector
  x = 0.0
  do i=1,3
     x = x + v(i)*v(i)
  end do
  write(*,*) 'Modulus squared = ',x

end program vector
```

Notice how we use a loop to compute the sum over all elements of the vector.

A second example will show us how we can implement simple vector and matrix operations:

```
program linalg
  implicit none

  real :: v1(3), v2(3), m(3,3)
  integer :: i,j

  v1(1) = 0.25
  v1(2) = 1.2
  v1(3) = 0.2

  ! use nested do loops to initialise the matrix
  ! to the unit matrix
  do i=1,3
     do j=1,3
        m(j,i) = 0.0
     end do
     m(i,i) = 1.0
  end do

  ! do a matrix multiplication of a vector
  ! equivalent to v2_i = m_ij v1_j
  do i=1,3
     v2(i) = 0.0
     do j = 1,3
        v2(i) = v2(i) + m(i,j)*v1(j)
     end do
  end do
  write(*,*) 'v2 = ',v2

end program linalg
```

- Enter this program, compile and run it. Make sure you understand the output.
- Try modifying the program to multiply two matrices.

We can also have arrays of `integer`, `complex` or any other data types declared in analogous ways.

Arrays may be declared using other forms than those given above which can be useful for different situations. The `dimension` option to the declaration may be used to set a shape for all declared variables which do not have a particular shape specified for them. The `dimension` statement serves several purposes in a F95 declaration. In the following, note the critical nature of the punctuation, particularly '`,`', '`:`' and '`::`'.

An example of the simplest form of an array declaration for a matrix might be:

```
real, dimension(10,11) :: a  ! a is a rank 2 array
```

The array subscripts are assumed to start at unity, but this can be altered by using the explicit form of the declaration in which the range of the array subscripts is given separated by a colon:

```
real, dimension(0:9) :: a ! vector of 10 elements
                          ! starting at 0
```

We can also declare a variable to be an array of unknown size. We do this as follows

```
real, dimension(:), allocatable :: a
```

and at an appropriate time, when it is known how big this array needs to be, we use the following (say to create an array of 10 elements):

```
m=10
allocate(a(m))
```

where `m` is an `integer` variable. When the use of the array in the program is finished, the space can be released by using

```
deallocate(a)
```

1.11 Array arithmetic

One very useful feature of F95 is the ability to work with whole arrays. In most programming languages one can, say, add two arrays using a loop. For example

```
real :: a(10), b(10), c(10)
integer :: i

[some statements to setup the arrays]

do i=1,10
   c(i) = a(i) + b(i)
end do
```

F95 allows you to perform whole array operations in a natural way. Most of the normal arithmetic operations can be carried out on arrays, where they apply in an element by element fashion. The above example can be written as:

```
real :: a(10), b(10), c(10)

 [some statements to setup the arrays]

c = a + b
```

- Here are some more examples which illustrate array arithmetic:

```
real, dimension(3,3) :: a, b
real, dimension(3) :: x, y, z
integer, dimension(10) :: idx

idx = 1          ! set all elements of idx to 1
a=b              ! copies the array b into a
x=y+1            ! x(i) = y(i)+1 for i=1,2,3
z=atan2(y,x)     ! z(i) = atan2(y(i),x(i)) for i=1,2,3
```

- You can refer to a subset of an array and treat it as another array:
  - `z( (/1,3,6/) )` a length 3 array with elements set to `z(1)`, `z(3)`, `z(6)`
  - `z(m:n)` is an array of length `(n-m+1)` formed from the elements of `z` starting at `m` and ending at `n`
  - `z(m:n:c)` is an array of length `(n-m+1)/c` formed from the elements of `z` starting at `m` and ending at `n` incremented by `c`
  - `x(1:5) = y(2:6)` copies elements 2 to 6 of `y` into elements 1 to 5 of `x`
  - `z(1:3) = y(1:5:2)` copies elements 1,3,5 of `y` into elements 1,2,3 of `z`
  - `a(2,:) = z` copies the vector `z` into the second row of `a`

- There is a conditional, `where`, which operates on an array for simple function forms e.g. to replace negative elements of an array `z` with their absolute values:

```
where (z < 0.0) z=-z
```

  - More generally it has the form:

```
where (logical array test)
      [statements if test true]
elsewhere
      [statements if test false]
end where
```

  - For example to take the logarithm of the positive elements of an array:

```
real, dimension(1000) :: a
where (a > 0.0)
     a = log(a)
elsewhere
     a = 0.0
end where
```

- There are a number of intrinsic procedures taking array arguments e.g.

`dot_product`	takes 2 arguments of rank 1 and the same size and returns their inner product
`matmul`	performs matrix multiplication on 2 array arguments with compatible size and rank
`maxval`	returns maximum element of an integer or real array
`minval`	returns minimum element of an integer or real array
`product`	returns the product of the elements of an array
`sum`	returns the sum of the elements of an array

2 Good Programming Style

In section 1 we have covered some of the basics of programming. We will return to programming later when we look in even more detail at F95.

In this section we will briefly consider some rules for good practice in developing programs. When you come to tackle the computing exercise we will be looking for how you have tackled some of the issues we shall now discuss.

2.1 Readability

Your program should be as easy to follow in terms of its logical structure as possible. There are a number of ways we have already met that help us do this. Let us recap some of them.

First use comments. A general rule for comments is that you should use a comment when the F95 statements you write are not self explanatory. There is no need, for example, to add comments to obvious computational expressions. However you may want to add comments to the top of a block of expressions explaining how the following code relates to the physical problem.

- Similarly if you have a loop, a comment of the form below is of no help:

```
! loop from 1 to 10
do i=1,10
```

  - But a comment of the following form, say in a program calculating a binomial might be very useful:

```
! loop to calculate nCr
do k=1,r
```

So use comments sensibly to make the code understandable.

- What we don't want to hear is:
  "I have written my program, now I just need to comment it before handing it in"

  This is bad practice because comments are part of the program and should be there as much to help you follow your own intentions in programming as for the head of class to follow it.

- Another aspect of readability is indenting code blocks between `do` loops and in `if` clauses. This is very good practice. It uses the layout of the program to show at a glance the logical structure. Our strong advice is to make good use of indenting. Again it helps as much in program development as it does in presenting the final program.

2.2 Self-checking code

We have already seen in some of the examples how we can use checks to avoid numerical errors. There are a number of numerical operations which are "poorly defined". These include, among many others:

a) division by zero
b) taking the square root or logarithm of a negative `real` number

Alternatively we may know the range of possible allowed values for a variable and can include checks to make sure this is not violated.

Sometimes we can be sure that variables cannot take on illegal values, other times we cannot. For example values may be supplied to the program by a user and the values may be wrong. Alternatively we may know that under certain conditions a variable may, for example, become negative and all this really means is that it should be set equal to zero; in fact the formula we are computing may explicitly state something like:

$$z = \begin{cases} \ldots & x > 0 \\ 0 & \text{otherwise} \end{cases}.$$

In either case you must be careful to check arguments to make sure they are "in range". We have seen examples of this already and you should go back now and revise these methods.

Once again it is essential in program design to be sensible. Do not check a variable if it cannot be out of range; this just slows your code down. For example the following would be bad programming style:

```
real :: x
[some statements]

x = sin(y) +  1.0
if (x >= 0.0) z = sqrt(x)
```

Here `x` can never be less than zero; the test is not wrong, but clearly unnecessary and indicates a poor appreciation of the logic of the program.

2.3 Write clear code that relates to the physics

We are not aiming in this course to develop ultra-efficient programs or the shortest possible program etc. Our aim is for you to learn the basics of computational physics. Therefore you should aim to write your code so that it relates as clearly as possible to the physics and computational physics algorithms you are using as possible. You can split long expressions over many lines, for example, by using the continuation marker.

If the last character of a line is an ampersand '&', then it is as if the next line was joined onto the current one (with the '&' removed). Use this to lay out long expressions as clearly as possible.

- Another technique is to split long expressions using intermediate calculations. A simple example would be replacing something like:

```
res = sqrt(a + b*x + c*x*x + d*x*x*x) + &
     log(e * f / (2.345*h + b*x))
```

with

```
t1 = a + b*x + c*x*x + d*x*x*x
t2 = E * F / (2.345*h + b*x)
res = sqrt(t1) + log(t2)
```

- Think about the choice of variable names. You can make the variable names very clear with names such as `energy` or `momentum`. This can be very helpful, but also cumbersome in long expressions. A useful rule of thumb is that if there is an accepted symbol for a physical quantity consider using that (e.g. *E* and *p*); use longer more descriptive names if one does not exist.

We will return to the topic of programming style later when we consider how the program can be broken up into smaller units. This will be the main job in the next section of the course.

3. Input to and output from a F95 program

We have already seen some examples of outputting information from a program (`write`) and reading information from the terminal (`read`). In this section we will look in detail at input and output and explain those strange '`*`'s. To save some writing let's introduce some jargon: we will call input and output I/O.

3.1 F95 statements for I/O

Input and output to a F95 program are controlled by the `read` and `write` statements. The manner in which this is done is controlled by format descriptors, which may be given as character variables or provided in a `format` statement. For economy of effort we will only outline the latter method, together with the default mechanism.

The form of the I/O statements is as follows:

`read(`*stream*`, `*label*`[, end=`*end*`][, err=`*err*`])` *list*

and

`write(`*stream*`, `*label*`)` *list*

where

- *stream* is a number previously linked to a file, or a character variable, or `*`, where `*` here indicates the default value, usually the screen of a terminal session. If *stream* is a character variable, the result of the `write` is stored in that variable, and can be manipulated as such within the program.
- *label* is the number of a `format` statement, or `*` for free format.
- *list* is a list of items to be transferred, separated by commas, possibly including text strings enclosed in quotation marks.
- The optional items `end` and `err` are so that you can provide statement labels *end* and *err* to which control moves in the event that the end of data is reached prematurely (`end`) , or some error is encountered (`err`).

The precise details of how the output should look are governed by the format definition. This takes the form:

label `format (`*format descriptors*`)`

- *label* is an integer, corresponding to the label appearing in the `read` or `write` statement. More than one `read` or `write` can refer to the same label.
- *format descriptors* is a comma-separated list of items describing how the output is to be presented, possibly including text items. The latter should be enclosed in single quotation marks as in character strings.

Possible formats for numeric items are

*n*I*w*	to output `integer`s
*n*F*w*.*d*	to output `real` or `complex` in fixed-point form
*n*E*w*.*d*	to output `real` or `complex` in floating-point form

n is an optional repeat count (how many of each item is in the I/O list).
w is the total number of characters per number, including signs and spaces: the field width.
d is the number of decimal digits to be output within *w*.

For non-numeric items the descriptor

A*w* is available, which causes the next *w* `character`s to be output

To access a file for input or output you can use the `open` statement:

`open([unit=]`*stream*`, err=`*escape*`, action=`*action*`, file=`*name*`)`

There are further possible arguments which should not be needed for this course, but which you may look up in a text book.

- *stream* is the identifier which appears in the `read` or `write` statement
- *escape* is a statement label, to which control is transferred if there is a problem opening the file.
- *action* is one of '`read`', '`write`' or '`readwrite`', depending how you intend to use the file.
- *name* is the name of the file (in quotes) and may be held in a character variable.

Having opened a file, linking it to a *stream*, and read through it, you can move back to the beginning using:

`rewind(`*stream*`)`

When you have completed I/O to a particular file, you can use the `close` instruction to close the file and tidy things up:

`close(`*stream*`)`

- Try entering the following example, compile and run it and examine the output

```
program format1
  implicit none
  integer  :: i

  do i=1,20
     write(*,1) i, i*i, i**3
  end do
1 format(i4,i6,i8)

end program format1
```

- Modify the program to use “free-format” output and compare the results (you could output both in one program for comparison of course).

- The next example illustrates how to send the output from your program to a file

```
program fileio
  implicit none
  integer :: i

  open(20, file='cubes.dat')
  do i=1,100
     write(20,1) i, i*i, i**3
  end do
  close(20)
1 format(i4,i6,i8)

end program fileio
```

- Modify the program to send a copy of the output to two files simultaneously.

4 Graphics and Visualisation

There are many ways in which to plot data. It is very difficult to write graphics applications in F95, so generally it is easier (and better) to use an application. One such application is `gnuplot`, which is a free plotting program that can plot data files and user-defined functions. It can't do everything you might possibly want, but it is very easy to use. We introduce `gnuplot` here – there is documentation available via '`help`' within the program and on the course web site.

`gnuplot` is a command-line driven program. Typing `gnuplot` at the terminal you will see that the prompt changes. You will want to use the `help` command to find out more information. You will also be able to output graphs from `gnuplot` in a form you can import into, say, Microsoft Word, when you produce your report.

4.1 Plotting a data file

`gnuplot` expects data to be arranged in columns in an ordinary text file, e.g.

```
# Gnu population in Antarctica since 1965
         1965   103
         1970   55
         1975   34
         1980   24
         1985   10
```

You can have as many columns as you like. Comments are indicated by '#'. The recommended way you use `gnuplot` to produce results from your F95 program is therefore to write out results to a file and use `gnuplot` to plot them. Let's look at a simple F95 program to do just that.

```
program outputdata
  implicit none
  real, dimension(100) :: x, y
  integer :: i

  ! setup x and y with some data
  do i=1,100
     x(i) = i * 0.1
     y(i) = sin(x(i)) * (1-cos(x(i)/3.0))
  end do

  ! output data to a file
  open(1, file='data1.dat', status='new')
  do i=1,100
     write(1,*) x(i), y(i)
  end do
  close(1)

end program outputdata
```

The file `data1.dat` should contain the two columns of numbers, exactly the format needed by `gnuplot`. To plot the data is very easy:

- Enter this program, compile and run it and produce the data file `data1.dat`.
- Start up `gnuplot`.
- Within `gnuplot` give the command:
```
plot 'data1.dat'
```

4.2 Getting help

You can get help by typing '`?`' or '`help`' within `gnuplot`. The on-line help is very good. You can also abbreviate commands to save typing.

4.3 Further examples

As well as plotting data we can plot functions in `gnuplot`.

- For example to plot one of the trigonometric functions type the following:
```
plot sin(x)*cos(x)
```

- In fact, `gnuplot` lets us define a function:
```
pop(x) = sin(x)*(1-cos(x/3.0))
```

- Then we can plot this function, for $0 \leq x \leq 10$, say, as follows:
```
plot [0:10] pop(x)
```

- To plot the data file created using the F95 program of the previous section we can use:
```
plot 'data1.dat'
```

- And you can also plot both the function and the data together:
```
plot [0:10] 'data1.dat', pop(x)
```

- By default, data files are plotted point by point. If you want lines joining the points:
```
plot 'data1.dat' with linesp
```

- If you want lines only:
```
plot 'data1.dat' w lines
```

- To control which colour each set of lines and points comes out, see `help plot`. For example, to make the data come out with colour 2 (dotted lines), and `pop(x)` with colour 1,
```
plot [0:10] 'data1.dat' w lines 2 2, pop(x) \
          w lines 1 1
```

- The backslash enables you to continue on to another line if the commands become long.

- To plot column 4 of '`flib.dat`' against column 2 of the same file:
```
plot 'flib.dat' u 2:4 w linesp
```

- this gives column 2 on the x axis and 4 on the y axis. You can also plot points with error bars. The following command plots column 4 versus column 2, with columns 5 and 6 defining the upper and lower error bars:
```
plot 'flib.dat' u 2:4:5:6 w errorbars
```

4.4 Printing graphs into PostScript files

- The following sequence changes the terminal type to PostScript and replots the most recent plot to a file called `file.ps`:
```
set term post
set output 'file.ps'
replot
```

- Don't forget to set the terminal type back to X11 when you are done plotting to the file.
```
set term X
```

- In order to get graphs that are readable when included in papers, I recommend:
```
set size 0.6,0.6
```

  before plotting to the file. This reduces the size of the graph while keeping the font size and point size constant.

There are other output types from `gnuplot`. In particular you may want to use the CGM terminal type instead of the PostScript terminal type as above (use `cgm` instead of `post`). This produces a file which can be read directly into Microsoft Word and converted to a Word drawing.

Suggested Exercise 1

Modify the projectile program to output the path of a projectile launched from the ground.

The program should read in the initial speed, the angle to the horizontal and a time interval at which to output a series of *x*,*y* values (as two columns) suitable for plotting with `gnuplot`. Points should be output until the projectile reaches the horizontal plane in which it was launched.

It is a good idea to show this program to a demonstrator or head of class and discuss with them how it might be improved.

You can find a solution to this problem in the examples directory

```
$PHYTEACH/part_2/examples/exercise1.f90
```

5. Program Organisation: Functions and Subroutines

In this section we will consider ways to structure your program, and in particular the use of functions and subroutines. We shall start off by assuming that the functions are to be defined in the same file as the main program. However this is rather cumbersome for a large program, and in the next section we will consider how to split the program between multiple files. Splitting the program in this way results in different program *segments*.

Subroutines and functions are the main way in which you can structure your F95 program. The main difference between them is that a function returns a value through its name, while the subroutine does not (a function must have at least one argument, while a subroutine may have none). The type of the function name must be declared both where it is defined and in the segment from which it is called. The function is used like any of the intrinsic functions while the subroutine is accessed via a `call` statement as we shall see below. We will use the term routine to refer to both subroutines and functions.

Direct communication between one program segment and a routine is through the *arguments*. Operations coded in the routine act directly on the variables referred to via the arguments in the calling statement. Other variables declared within the routine are local to that routine and not in general accessible elsewhere, or indeed in a subsequent call to the routine.

In what follows we will introduce functions and subroutines and then discuss how grouping functions and subroutines in different files can be an additional way of structuring your program.

5.1 Functions

We have already met the idea of functions in the form of intrinsic functions, i.e. functions which are part of `F95` and which are used to calculate standard mathematical functions. We can also define our own functions for use in a program; this is a very powerful feature of all programming languages. Using functions has a number of advantages:

1. The code to implement the calculation can be written just once, but used many times.
2. The functions can be tested separately from the rest of the program to make sure they give the correct result; we can then be confident when using them in the larger program.

Let's consider an example where we define two functions; in fact this is a simple program to find the root of an equation using Newton's method.

The functions we define are for the mathematical function we are seeking the root of and its first derivative.

```
program newton
  !
  ! Solves f(x) = 0 by Newton's method
  !

  implicit none

  integer  ::  its = 0        ! iteration counter
  integer  ::  maxits = 20    ! maximum iterations
  integer  ::  converged = 0  ! convergence flag
  real :: eps = 1.0e-6        ! maximum error
  real :: x = 2               ! starting guess

  ! introduce a new form of the do loop
  do while (converged == 0 .and. its < maxits)
     x = x - f(x) / df(x)
     write(*,*) x, f(x)
     its = its + 1
     if (abs(f(x)) <= eps) converged = 1
  end do
  if (converged == 1) then
     write(*,*) 'Newton converged'
  else
     write(*,*) 'Newton did not converge'
  end if

contains
  function f(x)
    real :: f, x
       f = x**3 + x - 3.0
  end function f

  function df(x)
    ! first derivative of f(x)
    real :: df, x
       df = 3 * x**2 + 1
  end function df
end program newton
```

There are lots of things to notice about this example.

- We have introduced a modified form of the `do` statement. Instead of looping a fixed number of times we loop while a condition is true. We shall return to this later.
- The functions `f` and `df` are defined. They appear in the program after the word `contains`. `contains` marks the end of the main code of the program and indicates the start of the definition of `function`s and, as we shall see in a moment, `subroutine`s.

- Each function starts with the keyword `function` and ends with `end function` and the function name; this is quite like the syntax we have already met introducing the program itself.
- Notice how within the `function` the function name is treated like a variable; we define its type and assign it a value; this is the value returned from the `function` and used when the `function` is called.
- The functions defined in this way are called internal functions since they are defined within the program itself. We shall see the nature of this in a little while.
- Values are supplied to the `function` via its arguments, i.e. the variables specified in parentheses (the `x` in this example); note each argument must have its type specified.

  - Enter this program, compile and run it. Try to understand the output. Modify the functions to some other form to find the roots of different functions using Newton's method.

5.2 Formal definition

The structure of a routine is the same as for a main program, except that the first line defines the routine name. Thus the form for a function is:

```
function name(arg1, arg2, ....)
   [declarations, including those for the arguments]
   [executable statements]
end function [name]
```

and for a subroutine:

```
subroutine name(arg1, arg2, ....)
   [declarations, including those for the arguments]
   [executable statements]
end subroutine [name]
```

Additionally for functions `name` must occur in a declaration statement in the function segment itself. The value the function takes is defined by a statement where `name` is assigned a value just like any other variable in the function segment.

The arguments must appear in the declarations as will any local variables; if you change the value of one of the arguments then the corresponding variable in the program "calling" the function or subroutine will also change. For this reason it is imperative that the type of dummy arguments agree with that of the variables in the call of the routine.

If you wish to terminate execution of a `function` or `subroutine` before the last statement in the subroutine (e.g. you may test some condition with an `if` clause and decide there is no more to do) then you can use the `return` statement to terminate the routine.

5.3 Subroutines

`Subroutine`s are very similar to `function`s except that they do not return a value. Instead they have an effect on the program in which they are invoked by modifying the arguments to the `subroutine`. Here is a simple example which defines a subroutine called `swap` which swaps the values of its two arguments:

```
program swapmain
  implicit none

  real :: a, b

  ! Read in two values
  read(*,*) a, b

  call swap(a,b)
  write(*,*) a,b

contains
  subroutine swap(x, y)
     real :: x, y, temp
     temp = x
     x = y
     y = temp
  end subroutine swap
end program swapmain
```

- Compile this program and see what happens to the values typed in. You should find the values are interchanged.
- Note how the `subroutine` is called and that its action is to modify its arguments.
- One important difference between a `function` and `subroutine` is that a `function` must always have at least one argument, whereas a `subroutine` need not have any.

5.4 Local and global variables

Variables that are defined within a given routine (and are not arguments to the routine) are local to that routine, that is to say they are only defined within the routine itself. If a variable is defined in the main program then it is also available within any internal routine (i.e. one defined after the `contains` statement).

- We say that these variables are *global*.

Here is a simple example:

```
program set
  implicit none
  real :: a, b

  ! Read in value of a
  read(*,*) a

  call setval(b)
  write(*,*) b

contains
  subroutine setval(x)
     real :: x
     x = a ! value of a is from main program
  end subroutine setval
end program set
```

This program is very contrived, it simply sets the value of the argument of the call to `setval` to the value of the variable `a` in the main program. In this example `a` is a variable global to the main program and the subroutine `setval`.

5.5 Passing arrays to subroutines and functions

Arrays can of course be used as arguments to `subroutine`s and `function`s, however there are one or two special considerations.

5.5.1 Size and shape of array known

If the size and shape of the array are known, one can simply repeat the definition in the subroutine when the arguments are declared. For example, if `v` and `r` are vectors of length 3 a `subroutine` declaration in which they are passed would look like:

```
subroutine arr1(v, r)
  real :: v(3)
  real :: r(3)
```

and of course you could have used the alternative form using `dimension` instead:

```
subroutine arr1(v, r)
  real, dimension(3) :: v, r
```

5.5.2 Arrays of unknown shape and size

The problem is how to deal with arrays of unknown length. This is important when for example we wish to write a `subroutine` which will work with arrays of many different lengths. For example, we may wish to calculate various statistics of an array. We can in this case use the following definition:

```
subroutine stats(y, ybar)

   real, dimension(:) :: y
   real :: ybar

! local variables
   integer :: i, n

   n = size(y)
   ybar = 0.0
   do i=1,n
      ybar = ybar + y(i)
   end do
   ybar = ybar / n

end subroutine stats
```

- Here we have declared the array `y` to be of unknown size, but of an assumed shape; that is it is a vector and not a matrix. The intrinsic function `size` returns the number of elements in the array i.e. the length of the vector.
- We can do the same for a two-dimensional array, a matrix. The corresponding definition would be:

```
real, dimension(:,:) :: a2d
```

If you use this form for passing arrays and declare the `subroutine`s in a different file from the calling program, it is essential that you use the concept of modules discussed in the section 6.

5.6 The `intent` and `save` attributes

We have seen how `subroutine`s can modify their arguments, but often we will want some of the arguments to be modifiable and others just values supplied to the routine. Recall how we discussed the use of `parameter`s as a way of protecting us from misusing something which should be a constant. In a similar way F95 provides a way of stating which arguments to a routine may be modified and which not. By default all arguments can be modified. The extra syntactic protection offered is by the `intent` attribute, which can be added to the declaration. It can have the values `out`, `inout` or `in` depending on whether you wish to modify, or not, the value of the argument in the routine call. This is another example of an option to a declaration statement (again c.f. the `parameter` definition earlier).

Here is the specification:

```
subroutine name(arg1, arg2, .....)
real, intent(in) :: arg1
real, intent(out) :: arg2
```

in the following example any attempt to change the variable `x` would result in a compiler error.

```
subroutine setval(x, y)
   ! set y to value of x

   real, intent(in) :: x
   real, intent(out) :: y

   y = x

end subroutine setval
```

- Modify the `setval` example above to use this `subroutine`.
- Deliberately introduce an error and within the `subroutine` attempt to change the value of the first argument; try compiling the program and see what happens.

Local variables which are declared within a routine will in general not retain their values between successive calls to the routine. It is possible to preserve values after one return into the next call of the routine by specifying the `save` attribute in the declaration of that variable, e.g.

```
real, save :: local
```

This of course cannot be used for the arguments of the routine.

6. Using Modules

As programs grow larger it becomes less convenient to use a single file for the source code. Also it is very often the case that the `function`s and `subroutine`s you have written can be used in more than one program. We need a way to organise the source code to make this easy to do. In this section we will look at modules which enable the program to be split between multiple files and make it easy to "re-use" the code you write in different programs.

6.1 Modules

Modules are a very useful tool when we split programs between multiple files. Modules give us a number of advantages:

1. The module is useful for defining global data, and may be used as an alternative way of transmitting information to a routine.
2. The variables, etc., declared in a module may be made available within any routines at the choice of the programmer.
3. Modules can be imported for use into another program or subroutine. Functions and variables defined in the module become available for use. The compiler is also able to cross-check the calls to subroutines and functions and use of variables just as if they had been defined as internal routines.

The definition of a `module` is

```
module name
    [statement declarations]
[contains
    [subroutine and function definitions] ]
end module [name]
```

The module is incorporated in the program segment or routine in which it is to be used by the `use` statement:

```
use name
```

- You can have as many different modules as you like. Each module will be contained in a separate file and may be compiled (and checked) separately.

- Modules can be re-used between different programs and may be used multiple times in the same program.

- The variables declared in the `module` before the `contains` statement are global to the module and become global variables in any routine in which the `use` statement appears.

- The `use` statement can appear in the main program or any other `subroutine` or `module` which needs access to the routines or variables of another module.

We should now have a look at some examples. Let's return to the `swapmain` program from section 5.3 and implement this in two files using modules. First edit the file `swapmain.f90` and to contain:

```
program swapmain

  use swapmod  ! use statements must come first

  implicit none

  real :: a, b

  ! Read in two values
  read(*,*) a, b

  call swap(a,b)
  write(*,*) a, b

end program swapmain
```

Now in the second file, called `swapmod.f90` the module is defined:

```
module swapmod

  implicit none  ! you need this in every module
  !
  ! Global declarations would appear here if any
  !
contains
  ! routines provided by this module
  subroutine swap(x, y)
    real :: x, y, temp

    temp = x
    x = y
    y = temp

  end subroutine swap
end module swapmod
```

- Now compile the `swapmod.f90` file:
  `f95 -c swapmod.f90`

  - The `-c` option is a request to compile only and not try to produce an executable program.
  - Use `ls` to see what files you have. You should have `swapmod.mod` and `swapmod.o` files.

The `swapmod.o` file contains the compiled code for all the routines in the `swapmod.f90` file. When you compile a module an extra file called `name.mod` is created. This file is needed for as long as you want to include the module in programs.

- Now compile the complete program, making available the compiled code from the `swapmod` module so that the "linker" can bring together all the machine instructions necessary to define a complete working program. This is done by giving the name of the compiled routine on the `f95` command line:

```
f95 -o swap swapmain.f90 swapmod.o
```

  - Run the program and check the output.

Within the module you may define as many routines as you like; they are all included with the same `use` statement.

Let us now look at an example which makes use of modules to provide global data. Again this is a rather contrived example designed to illustrate as simply as possible the use of modules. This time let's have three files.

- Let's start with a file called `gd.f90` which defines a module called `gd`; this defines some global data but no routines:

```
module gd
  implicit none

  ! define global data
  real :: a_g, b_g

end module gd
```

- Note how we include an `implicit none` statement in the module; each program element (main program and each module) must have the `implicit none` statement.

- Now the second file, `setval.f90` defines a `setv` routine which is part of a module `setval`

```
module setval
  ! make the global data available
  use gd
  implicit none
contains
  subroutine setv(a, b)
     real :: a, b

     ! set arguments to global values
     a = a_g; b = b_g
  end subroutine setv
end module setval
```

- Finally the main program `testset.f90`

```
program testset

  ! include global data
  use gd
  ! make available the setval routine
  use setval

  real :: x, y

  ! read values from terminal
  read(*,*) a_g, b_g

  ! set x and y and check output
  call setv(x, y)
  write(*,*) x, y

end program testset
```

- Compile the two modules:

```
f95 -c gd.f90
f95 -c setval.f90
```

- Now the main program, creating an executable program:

```
f95 -o testset testset.f90 gd.o setval.o
```

- Run the program and check the output.

These two examples, although simple, illustrate the usefulness of modules.

6.2 `public` and `private` attributes

The examples of modules in the previous section were quite simple. In general you are likely to want a module which not only declares some global data, but also defines routines in the same module. However we may want to differentiate variables which are global and available in all routines in which the module is `use`d, as we saw in the last section, and variables which are to be global only within the module (i.e. used by all the routines in the module, but not available in the routines or program segments which include the module via the `use` statement). We can in fact achieve this using the `private` and `public` attributes. For example the variable `a_g` in the following will be available in all program segments including the module in which this definition occurs, whereas the variable `b` is a global variable only within the current module

```
real, public :: a_g
real, private :: b
```

The default is for variables is to be `public`.

7 Numerical precision and more about variables

7.1 Entering numerical values

We have already seen how to enter simple numerical values into a program. What happens if the values need to be expressed in the form $a \times 10^b$? F95 provides a way of doing it. For example the numbers on the left of the following table are entered in the form indicated on the right:

1.345	`1.345`		
2.34×10^{7}	`2.34e7`	or	`2.34d7`
2.34×10^{-7}	`2.34e-7`	or	`2.34d-7`

7.2 Numerical Accuracy

Variables in any programming language use a fixed amount of computer memory to store values. The real variables we have discussed so far have surprisingly little accuracy (about 6-7 significant figures at best). We will often require more accuracy than this, especially for any problem involving a complicated numerical routine (e.g. a NAG routine). In that case you will need to use what is called a "double precision variable". Such variables have twice the accuracy of normal `real`s.

F95 in fact provides a very flexible system for specifying the accuracy of all types of variables. To do this it uses the idea of `kind`; this really means the accuracy of a particular type of variable. You do not need to go into this in much detail, we will give a recipe for obtaining high precision, "double precision", variables.

Each variable type may have a number of different `kind`s which differ in the precision with which they hold a number for `real` and `complex` numbers, or in the largest number they can hold for `integer` variables. When declaring variables of a given type you may optionally specify the `kind` of that variable as we shall see below. A few intrinsic functions are useful:

`kind(x)`	Returns the `kind` (an `integer` variable) of the argument `x`. `x` may be any type.
`selected_int_kind(n)`	Return the kind necessary to hold an `integer` as large as 10^n.
`selected_real_kind(p[,r])`	Return the kind necessary to hold a `real` in the range 10^{-r} to 10^r with precision `p` (number of significant decimals).

Here we want to specify variables of high precision and there is an easy way of doing this. The notation for a double precision constant is defined; so that for instance `1.0` is designated `1.0d0` if you wish it to be held to twice the accuracy as ordinary single precision. To define variables as double precision, you first need to establish the

appropriate `kind` parameter and then use it in the declaration of the variable. Here is a useful recipe:

```
! find the kind of a high precision variable, by finding
! the kind of 1.0d0
     integer,  parameter :: dp=kind(1.0d0)

! use dp to specify new real variables of high precision
     real(kind=dp) :: x,y

! and complex variables
     complex(kind=dp) :: z

! and arrays
     real(kind=dp), dimension(30) :: table
```

The integer parameter `dp` can now also be used to qualify constants in the context of expressions, defining them as double precision, e.g. `0.6_dp`

8 Use of numerical libraries: NAG

There are many libraries of subroutines performing mathematical and graphics operations. The library you are encouraged to use here is the NAG library, which has an on-line documentation system. One important feature of the NAG library in particular is that all the variables are of double precision type, and all arguments must be declared as such when NAG routines are called. The on-line information is available via the link on the course web page.

When using the NAG library you must tell the linker where to find the library code. This is done by adding some extra information to the `f95` command line; e.g. let's imagine we were compiling and linking (to form an executable program) a simple program in the file `ex.f90`. The command we need is:

```
f95 -o ex ex.f90 -lnag
```

The `-lnag` tells the linker to search the NAG library file for the code that it needs.

8.1 A simple NAG example

- Enter this simple program and try compiling and running it; it should report some information about the NAG library we are going to use

```
program naginfo

  use nag_f77_a_chapter
  implicit none

  write(*,*) 'Calling NAG identification routine'
  write(*,*)
  call a00aaf

end program naginfo
```

8.2 A non-trivial NAG example: matrix determinant

Use the link on the course web page to bring up the NAG documentation. Find the documentation for the routine F03AAF.

The first page of the documentation appears slightly cryptic:

F03AAF – NAG Fortran Library Routine Document

Note. Before using this routine, please read the Users' Note for your implementation to check the interpretation of bold italicised terms and other implementation-dependent details.

1 **Purpose**

F03AAF calculates the determinant of a real matrix using an *LU* factorization with partial pivoting.

2 **Specification**

```
SUBROUTINE F03AAF(A, IA, N, DET, WKSPCE, IFAIL)
INTEGER           IA, N, IFAIL
real              A(IA,*), DET, WKSPCE(*)
```

3 **Description**

This routine calculates the determinant of A using the LU factorization with partial pivoting, $PA = LU$, where P is a permutation matrix, L is lower triangular and U is unit upper triangular. The determinant of A is the product of the diagonal elements of L with the correct sign determined by the row interchanges.

4 **References**

[1] Wilkinson J H and Reinsch C (1971) *Handbook for Automatic Computation II, Linear Algebra* Springer-Verlag

5 **Parameters**

1: A(IA,*) – ***real*** array *Input/Output*
Note: the second dimension of A must be at least max(1,N).
On entry: the n by n matrix A.
On exit: A is overwritten by the factors L and U, except that the unit diagonal elements of U are not stored.

2: IA – INTEGER *Input*
On entry: the first dimension of the array A as declared in the (sub)program from which F03AAF is called.
Constraint: IA ≥ max(1,N).

3: N – INTEGER *Input*
On entry: n, the order of the matrix A.
Constraint: N ≥ 0.

4: DET – ***real*** *Output*
On exit: the determinant of A.

5: WKSPCE(*) – ***real*** array *Workspace*
Note: the dimension of `WKSPCE` must be at least `max(1,N)`.

6: IFAIL – INTEGER *Input/Output*
On entry: IFAIL must be set to 0, -1 or 1.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

The note at the beginning draws attention to the bold italicised term ***real***, which you should simply interpret as "double precision" or `real(kind(1.0d0))` in all NAG documentation.

One would expect a subroutine which calculates a determinant to need arguments including the array in which the matrix is stored, the size of the matrix, and a variable into which to place the answer. Here there are just a couple extra.

- Because the library is written in Fortran 77 (F77, an earlier version of Fortran than we are using), which does not support dynamic memory allocation, it is often necessary to pass "workspace" to a routine. The routine will use this space for its own internal temporary requirements. Hence `WKSPCE(*)`.

- Array dimensions of unknown size are denoted by a '`*`', not a '`:`'.

- Most NAG routines have the `ifail` argument which is used to control error handling and communicate error information back to the calling routine. In general you should set `ifail = 0` before calling a NAG routine, and test it on return (zero indicates success).

Here is the example, enter this program compile and test and make sure it gives the result you expect!

```
program nagdet

  ! we make available the chapter f module of nag
  use nag_f77_f_chapter

  implicit none

  ! variables used in the program
  real(kind(1.0d0)) :: m(3,3), d, wrk(2)
  integer i, n, ifail

  ! assign values to only the upper 2x2 portion of
  ! the 3x3 array
  m(1,1)=2 ; m(1,2)=0
  m(2,1)=0 ; m(2,2)=2

  ! set up input values and call the NAG routine
  ! note the difference between i and n values
  i=3 ; n=2 ; ifail=0
  call f03aaf(m,i,n,d,wrk,ifail)
  if (ifail == 0) then
     write(*,*) 'Determinant is ',d
  else
     write(*,*) 'F03AAF error:',ifail
  end if

end program nagdet
```

Further examples of F95 programs calling the NAG library can be found in the lecture handout on the Physics of Computational Physics.

9 Some more topics

This section can be skipped at a first reading.

9.1 The `case` statement and more about `if`

If you have several options to choose between in a program, you can use the `case` statement, which takes the form:

```
select case (expression)
   case(value1,value2)
          .
          .
   case(value3)
          .
          .
   case default
          .
          .
end select
```

- `expression` must be an expression which evaluates to an `integer`, `character` or `logical` value.
- `value1, value2, ...` must be possible values for the expression.
- if the expression matches either of `value1` or `value2` then the code following the `case` statement listing that value is executed.
- If no match is found, the code following the optional `case default` statement is executed.

A simple example is the use of a `case` statement to take action on user input to a program:

```
program choice
  implicit none
  character(len=1) :: x

  write(*,*) 'Select a or b'
  read(*,*) x

  select case(x)
     case('A','a')
        write(*,*) 'A selected'
     case('B','b')
        write(*,*) 'B selected'
     case default
        write(*,*) 'Error: unknown option'
  end select
end program choice
```

The same outcome could have been achieved using an `if` clause as follows:

```
if (logical expression) then
…
else if (logical expression) then
…
else if (logical expression) then
…
else
…
end if
```

- Try writing the above `case` statement using this form of the `if` clause. Ask a demonstrator to check that it is correct.

9.2 Other forms of `do` loops

We have already met the `do` loop in which the number of times round the `do` loop we go is determined at the start:

```
[name:] do var = start, stop [,step]
     xxx
end do [name]
```

There are two other forms of the `do` loop which are useful; one we have seen already in a example:

```
[name:] do while (logical expression)
     xxx
end do [name]
```

Here the loop is repeated while the logical expression evaluates to `.true.`.

The final form of the `do` loop requires that you use an `exit` statement somewhere within it, as the `do` loop is set up to loop indefinitely:

```
[name:] do
     xxx
     need some test which will result in an exit
end do [name]
```

Suggested Exercise 2

Write a program which initialises two arrays

```
real(kind(1.0d0)) :: x(128), y(128)
```

as the real and imaginary parts of a complex vector. The real part x should contain $x(i) = \sin(i/a)$ where a is a parameter to be read in. The imaginary part, y, should be initialised to zero. The arrays should be initialised in a subroutine called from the main program.

Now extend the program to use a NAG routine to take the Fourier transform of the values and write the result out to disc so that the data can be plotted in `gnuplot`. For the purposes of this exercise you can simply adopt an arbitrary scale for the frequency.

You can find a solution to this problem in the examples directory

`$PHYTEACH/part_2/examples/exercise2.f90`