Fortran 95

Dr MJ Rutter
mjr19@cam.ac.uk

Michaelmas 2001

Typeset by FoilTEX

What are (Computer) Languages?

A useful computer language is:

- 'easy' for a human to understand
- 'easy' for a compiler to convert to the obscure machine instructions and OS calls that the computer understands – a process called 'compilation'.

Of course one may argue about what the word 'easy' above means.

Portability

It should not be necessary to rewrite programs merely because one wishes to run them on a different type of computer or on a different operating system.

A mere recompilation should be sufficient.

A language with a well-defined and well-supported standard tends to achieve this ideal (e.g. F90, C, Java). A program written in a language without a widely accepted standard (e.g. BASIC) often requires a major rewrite in order to use it on a different computer.

FORmula TRANslation

Fortran is a venerable language designed to translate scientific formulae into a form usable by computers. It was not designed as a general-purpose language.

Fortran was originally developed by IBM in the late 1950's. The first official standard was published by ANSI in 1966, followed by significant revisions in 1977. The next standard, published in 1990, represented a more dramatic change, and that published in 1995 a very minor set of changes. The relevant standards committee still exists, and further revisions are expected.

ANSI: American National Standards Institute.

C is also quite an old language. So-called K&R C was developed in Bell Labs in the early 1970s, and was launched on the world in 1978, with the first ANSI standard published in 1989, thus creating ANSI C.

Fortran Today

Fortran may be nearing fifty years old, but it is far from dead!

Most large numerical codes are still written in Fortran: examples include most weather forecasting programs, many 'numerical wind-tunnels' and most quantum mechanical electronic structure codes. Thus the majority of the CPU time of big supercomputers, both national supercomputers and those in Cambridge, is spent running Fortran.

Not being a general-purpose language, there are some things Fortran is very bad at. Anyone trying to write a compiler or operating system in Fortran is probably mad or about to become mad. However the 'old' University Library electronic catalogue, which is soon to be replaced, is written in Fortran, even though this is far from the sort of application for which Fortran was designed.

Fortran's historic advantage is its simplicity: easy for humans to learn, easy for compilers to translate into the optimal sequence of instructions for a computer to execute. On some machines (such as vector Crays from the early to mid 1990s), the same algorithm written in Fortran and C would typically run ten to twenty times faster when written in Fortran.

Hello

```
$ less hello.f90
write(*,*) 'Hello, world.'
end
$ f95 -o hello hello.f90
$ ./hello
Hello, world.
$
```

First one must create the above two-line program using an editor.

The program consists of statements, one per line. The first writes a message back to the screen, the second, `end`, finishes each and every Fortran program.

The UNIX `less` command shows us the contents of the text file we have created: press 'q' to exit.

The UNIX `f95` command compiles that file, assuming it is valid Fortran. We have asked it to name the resulting binary program, which we will be able to run, '`hello`'. Omitting '`-o hello`' would result in the default name of '`a.out`' being used.

The UNIX command `./hello` runs this new program we have just created. The initial '`./`' indicates that this is not a pre-installed system program, but something to be found in our current directory.

Although `print*` can be used as a synonym for `write(*,*)`, the more general form will be used throughout these notes.

Variables

A variable is simply a label for an area of memory reserved for storing some data.

```
integer i
i=3
write(*,*) 'i is ',i
i=i+i
write(*,*) 'i is ',i
end
```

Here we start by choosing to use '`i`' as the label to refer to an area of memory in which a single integer value can be stored.

The next line stores the value 3 in this location.

The following line prints out i. Notice that '`write(*,*)`' happily accepts a comma-separated list of things to print out.

The next line appears to be mathematical nonsense, but $=$ should be interpreted as meaning *assignment* not equality. Thus the line calculates $i + i$, and stores the answer back into the original location that `i` occupied. The old value of `i` is overwritten.

The final two lines should need no explanation.

More variables

We may define as many variables as needed. Their names can be up to 31 characters long, the first must be alphabetic, the others alphanumeric or underscore.

Unlike UNIX, Fortran is not case-sensitive. In a Fortran program, 'small', 'SMALL' and 'Small' are considered equivalent. Inconsistency will confuse humans though.

The operators `+,-,*,/ and **` are available, the last being exponentiation.

```
integer small, smaller
small=3
smaller=2
write(*,*)small-smaller, small/2
small=smaller
smaller=small
write(*,*)smaller
end
```

This program prints 1 twice, then 2. Note that 3/2 is 1: so far we have met only integers. Also, the two assignments near the end do not swap the contents of `small` and `smaller`: indeed, the antepenultimate line is utterly pointless here.

Interacting

Examples always appear a little pointless until they perform something slightly non-trivial. Thus a way of getting a program to prompt for input is introduced at this early stage:

```
real a,b,c,x,y
write(*,*) 'Please input coeffs of quadratic'
read(*,*) a,b,c
x=(sqrt(b*b-4*a*c))/(2*a)
y=-b/(2*a)
write(*,*)'Roots are ',y-x,' and ',y+x
end
```

`real` defines a variable to be floating point, rather than integer.

`read(*,*)` reads a list of values from the keyboard. The values read can be separated by spaces or be on separate lines, but not separated by commas.

`b*b-4*a*c` is equivalent to `(b*b)-(4*a*c)`, just like normal algebra. It is not `(b*b-4)*a*c`.

The `sqrt` *function* calculates the square root of its *argument*.

Less Variable

Occasionally it is useful to define a variable in such a way that it cannot be changed when the program runs. Such a variable is called a *parameter*, and everything about it can be determined when the program is compiled.

```
real pi
parameter (pi=3.141593)
```

Here `pi` can be used like any other variable, except that any attempt to change its value will result in an error.

N.B. All declarations and parameter statements must precede the executable lines of a program.

Conditional execution

The above program has many deficiencies, one of which becomes apparent if it is given the input '`1 0 1`'.

This can be solved by adding the following lines immediately after the `read`:

```
if ((b*b-4*a*c)<0) then
  write(*,*)'No real roots'
  stop
endif
```

The lines between the `if` and `endif` are executed only if the condition on the `if` line is true. The `stop` instruction will terminate the program.

Humans enjoy seeing parts of the code indented in order to emphasis the structure, as above.

Computers don't care, and will not check that the indenting is consistent with the meaning.

More Conditions

Fortran has the 'standard' set of comparison operators: $<$, $<=$, $==$, $>=$, $>$ and $/=$ (not equal).

One can also combine the results of comparisons using the operators `.and.`, `.or.` and `.neqv.` (xor).

```
if ((a==0).and.(b==0).and.(c==0)) then
```

might be useful. As might

```
if (a<0) then
  write(*,*)'Quadratic has a maximum'
else
  write(*,*)'Quadratic has a minimum'
endif
```

The above errs for $a = 0$.

N.B. $=$ is assignment, $==$ is equality test.

More Control

So far all the examples have shown programs executing line by line from the first line to the last, maybe skipping some lines when `if` is used. One very useful addition to this model is that of repeated execution: looping.

```
integer i
do i=1,20
  write(*,*)i,i*i,i**3
enddo
end
```

The code between the `do` and `enddo` is executed twenty times, with the variable `i` being set to each of the values from 1 to 20 in turn for each *iteration* of the loop. The output looks like:

```
 1 1 1
 2 4 8
 3 9 27
 4 16 64
 . .. ..
```

Neatness

The output of the above program does not fall into the neat columns that one might hope for. This can be improved by using a `format` statement to tell the `write` statement precisely what to do.

```
   integer i
   do i=1,20
     write(*,10)i,i*i,i**3
   enddo
10 format(i4,i6,i8)
   end
```

The second `*` of the `write` statement has beeen replaced by a 10. This tells the compiler not to guess what layout is required, but rather to look for a `format` statement somewhere in the program and labelled by a 10. This statement then says: one integer, padded to a width of four characters, ditto six, ditto eight.

The `format` statement may occur before or after the `write`. Many `write`s may refer to the same `format` statement. The `format` statement is completely ignored at the point when it would be due for execution. The label can be any integer of five or fewer digits.

The complete list of options for `format` is large.

Infinite loops

```
integer i

do
  write(*,*)'How many burgers, sir?'
  read(*,*)i
  write(*,*)'Have a nice day!'
enddo
end
```

This is bad practice: there ought to be some way of stopping a program. Adding

```
if (i==-1) exit
```

to the loop after the `read` would achieve this.

`exit` causes the program to jump to the statement following the current loop, whether the loop is infinite or finite.

Factorisation

```
integer i,p,sqrt_p
write(*,*)'Input no. to test'
read(*,*)p
sqrt_p=sqrt(real(p))
do i=2,sqrt_p
  if (mod(p,i)==0) then
    write(*,*)p,' is divisible by ',i
    stop
  endif
enddo
write(*,*)p,' is prime'
end
```

`mod(a,b)` calculates the remainder after dividing `a` by `b`

As `sqrt` does not accept integer arguments, the function `real`, which converts integers to reals, is called first.

The above algorithm is not optimally fast!

Faster Factorisation

```
integer i,p,sqrt_p
write(*,*)'Input no. to test'
read(*,*)p

if (mod(p,2)==0) then
  write(*,*)p,' is divisible by 2'
  stop
endif

sqrt_p=sqrt(real(p))
do i=3,sqrt_p,2
  if (mod(p,i)==0) then
    write(*,*)p,' is divisible by ',i
    stop
  endif
enddo
write(*,*)p,' is prime'
end
```

The full syntax of a `do` statement is `do i=start,end,step`, where the '`,step`' is optional. If given, it specifies how much to add to `i` between each iteration of the loop. The final iteration will be the last one which does not cause `i` to exceed `end`.

The structure of a simple program

- declarations of variables & parameters
- executable statements
- the `end` statement

Comments (read by humans, ignored by computers) can be added by placing a '!' on a line. Everything after the exclamation mark will be ignored by the compiler.

A long statement can be split across two physical lines by placing a '&' as the last character of the first line:

```
write(*,*) &
 'A somewhat wordy greeting to you all'
end
```

Terminology

Operator:	$+$, $-$, $*$, $/$, $**$
Constant:	data whose value is known at compile time
Variable:	data whose value may change during program execution
Function:	`sqrt, cos, sin,` etc.
Expression:	mathematically valid combination of the above
Argument:	expression on which function acts
Declaration:	reservation of memory for a variable

Rather informal definitions. As examples:

```
real a,b,c         ! declarations
parameter (c=5.0) ! c is a real constant, as is 5.0

b=sqrt(c)          ! c is the argument of the function sqrt
                   ! brackets are required around arguments
a=sqrt(0.4*c)      ! 0.4 and c are real constants, * is an operator
                   ! 0.4*c is a real constant expression (=2.0)
a=a/b              ! a/b is a real variable expression
a=a**2             ! 2 is an integer constant, a**2 a real variable expression
...
```

Data Types

So far we have seen integers and reals as data types.

Integers have a range of about $\pm 2 \times 10^9$.

Reals have a range of about $\pm 1.7 \times 10^{38}$ and about 7 decimal digits of precision.

However, another type of real exists with a range of about $\pm 9 \times 10^{307}$ and about 14 decimal digits of precision. This is far more useful.

There is also a type called `complex` for dealing with complex numbers (stored simply as two reals: real part and imaginary part).

Compilers may offer more than the above two precision of reals.

The above assumes standard IEEE-754 data types. Almost every computer uses these. Elderly Crays and IBM mainframes might not, PCs do.

Constants

A constant is simply a number occuring directly in the program:

```
i=i+3
```

`i` is a variable, and 3 a constant.

The type of a constant is integer, unless it contains a decimal point or an exponent, when it is real.

Exponents can be specified as `1.3E12` meaning 1.3×10^{12}.

Using a `D` rather than an `E` for the exponent causes the constant to be double precision, that is, enjoy the larger of the above ranges. Hence `5E50` is (probably) an error, whereas `5D50` is fine.

What's your kind?

One can enquire about the kind of a real number, and define another variable to be of a suitable precision:

```
integer r12
parameter (r12=selected_real_kind(12))
real (r12) :: x,y
```

The integer parameter `r12` is given the value the the compiler uses to distinguish its sort of `real` that has at least 12 decimal digits of precision. The variables `x` and `y` are declared to be one of these. The double colons are required to separate the variables declared from the list things modifying the `real`.

```
integer dp
parameter (dp=kind(1d0))
real (dp) :: x,y
```

In a similar fashion, here `x` and `y` are each declared to be of the same kind of real as `1d0`, i.e. double precision.

N.B. I did not invent this syntax.

The Difference

```
integer ix,iy,dp
real rx,ry
parameter(dp=kind(1.0d0))
real (dp) :: dx,dy

ix=5 ; rx=5 ; dx=5
iy=3 ; ry=3 ; dy=3

write(*,*)ix/iy
write(*,*)rx/ry
write(*,*)dx/dy
end
```

The output from compiling and running the above is:

```
 1
   1.6666666
   1.6666666666666667
```

A semicolon can separate statements, just as using separate lines does.

Complex

The `complex` type is fully supported, and all the standard functions can take complex arguments and return complex values. It has different kinds in the same way that `real` does.

```
integer dp
parameter(dp=kind(1.0d0))
complex c
complex (dp) :: dc

c=-2      ; dc=-2
c=sqrt(c) ; dc=sqrt(dc)

write(*,*)c
write(*,*)dc
write(*,*)dc-c
end
```

Other functions supporting the `complex` type include: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `log`, `exp` and others.

Conversions

Conversions between different types and kinds occur automatically at the last possible moment.

```
integer i,j
real a,b,c
i=2 ; j=3

a=i           ! a is 2.0
b=i/j         ! b is 0.0
c=real(i)/j   ! c is 0.66..
...
```

The operators +, -, * and / all convert their operands to the same type and kind before carrying out the operation. The type and kind of the answer is the type and kind of the converted operands. The conversion always involves promotion, i.e. movement to the right in the sequence integer, real, complex and also in the sequence single precision, double precision.

An assignment will convert the type and kind of the right hand side to that of the left immediately before the assignment is done. Promotion or demotion may occur.

Most built-in functions (`sqrt`, trig functions) return the same type and kind as their argument, and do not accept integer arguments.

Vagueness

You may discover that programs compile happily even if none of the variables used are declared. Simply referencing a variable causes it to be declared automatically.

This is bad.

Firstly the type may not be what you expect: integer if the first letter of the variable is in the range i-n, real otherwise.

Secondly, some common mistakes cannot be spotted automatically:

```
real salary, tax
salary=15000
tax=(salary-4000)*0.23
salery=salary-tax
write(*,*) salary
end
```

Being precise

This unhelpful default behaviour can be suppressed by the use of the statement `implicit none`. This turns off the implicit declarations which caused referencing the misspelt '`salery`' above not to be an error.

```
implicit none
real salary, tax
salary=15000
tax=(salary-4000)*0.23
salery=salary-tax
write(*,*) salary
end
```

Attempts to compile this example will fail.

`implicit none` must precede any declarations.

Arrays

Simple scalar variables do not cover every need in Physics. Thus there exist *arrays*: variables containing multiple scalars.

One dimensional arrays, or vectors, are used as follows:

```
real position(2)

position(1)=-0.02
position(2)=52.2
```

and thus the two element one dimensional variable `position` stores our latitude and longitude. Similarly

```
integer marks(150)
```

could store the marks of a whole class.

More dimensions

```
real v1(3),v2(3),mat(3,3)
...
do i=1,3
  v2(i)=0
  do j=1,3
    v2(i)=v2(i)+v1(j)*mat(i,j)
  enddo
enddo
```

A simple form of matrix-vector multiplication.

Fortran permits up to seven dimensions.

Array terminology

```
integer i,j
real x, mat(100,100)
```

defines `x` to be a (scalar) variable and `mat` to be a two dimensional array.

`mat(i,j)` is an *element* of the array, and `i` and `j` are the array *indices* or *subscripts*.

Indices must be integers or integer expressions. `mat(2*i,j+7)` is quite valid.

The *bounds* of the array are `(1-100,1-100)` and any indices must lie within this range.

Exceeding the bounds of an array is a common error, and is often referred to as 'falling off the end' of the array. A reference to `mat(101,100)` would be such a mistake. The program will not necessarily stop at this point. . .

An array element can be used anywhere that a scalar variable is valid.

Memory

Clearly `real mat(3,3)` must take nine times as much memory as simply `real x`. Indeed, the matrix will be laid out in memory simply as `mat(1,1), mat(2,1), mat(3,1), mat(1,2), mat(2,2) ...mat(3,3)`.

Thus if `mat(i,j)` is suddenly requested, the computer knows where `mat(1,1)` is stored, and adds an offset of `(i-1)+3*(j-1)` to find `mat(i,j)`.

Beware! `real space(1000,1000,1000)` looks innocent, until one realises that it will require 4GB of memory (assuming 4 bytes per real).

The indices usually start at one, but this can be specified:

```
integer map(-20:20,-20:20)
```

is a 1681 element two-dimensional array whose indices range between ± 20.

Dynamic Arrays

The arrays shown so far are called *static*, because their size is specified at compile time. They cannot grow or shrink.

Dynamically allocated arrays can respond to precise storage needs at run time.

```
integer msize
integer, allocatable :: map(:,:)

read(*,*) msize

allocate(map(-msize:msize,-msize:msize))
...
deallocate(map)
```

This will be similar to the previous snippet if 20 is input.

The program will stop with an error if an allocate fails (e.g. not enough memory).

Notice the use of the colon as a placeholder for the as yet unknown dimension sizes in the declaration. The keyword 'allocatable' defines this to be a dynamic array.

The `end` statement will automatically deallocate any remaining arrays, though one may do so explicitly anywhere after the last point at which each array is used.

Implicit loops

An array element is just like any other variable, so one way of copying one array to another would be:

```
do i=1,n
  a(i)=b(i)
enddo
```

Fortran90 provides an abbreviated syntax for this:

```
a=b
```

Similarly

```
a=b*c
```

sets `a(i)=b(i)*c(i)` for the whole array.

This syntax requires `a`, `b` and `c` to be the same length, or for `b` and/or `c` to be simple scalar variables or constants (e.g. `a=2*a` doubles all the elements of `a`).

There is no magic here: the execution time will be as long as if the loop were written explicitly.

Array Functions

Fortran90 also provides some functions which act directly on arrays. These include:

`sum(a)`	$\sum a(i)$
`product(a)`	$\prod a(i)$
`dot_product(a,b)`	$\sum a(i)b(i)$
`maxval(a)`	value of largest element
`maxloc(a)`	indices of largest element
`matmul(a,b)`	vector or matrix product

and `minval` and `minloc` too.

`maxval` and `minval` work on multidimensional arrays, and hence `maxloc` and `minloc` return a one dimensional integer array giving the list of indices corresponding to the location. E.g.

```
integer a(3,3),i(2)
a=1 ; a(2,3)=2
i=maxloc(a)
```

sets `i(1)`=2 and `i(2)`=3.

`matmul` returns a vector or matrix, and one argument should be a matrix, the other a vector or matrix of suitable shape that the multiplication is mathematically reasonable. Note that `c=a*b` performs element-wise multiplication, $c_{ij} = a_{ij}b_{ij}$, whereas `c=matmul(a,b)` performs standard matrix multiplication $c_{ij} = \sum a_{ik}b_{kj}$.

Eratosthenes

```
integer i,j,imax,prime_max
integer, allocatable :: primes(:)

write(*,*)'Input largest number for search'
read(*,*)prime_max
allocate(primes(prime_max))
primes=1  ! Sets all elements of array to 1
imax=sqrt(real(prime_max))

do i=2,imax
  do j=2*i,prime_max,i
    primes(j)=0
  enddo
enddo

do i=2,prime_max
  if (primes(i)==1) write(*,*)i
enddo
end
```

Note the use of a non-unit stride in the loop over `j`: `j` takes the values `2*i`, `3*i`, `4*i`. . .

Note the shortened form of the `if` statement: the `then` is replaced by a single statement to execute if the condition is true, and there can be no `else` section.

Random Answers

```
integer i,trials,hits
real x,y

trials=1000000 ; hits=0

do i=1,trials
  call random_number(x)
  call random_number(y)
  x=2*x-1 ; y=2*y-1
  if ((x*x+y*y)<1.0) hits=hits+1
enddo

write(*,*)'pi is approx ',(4.0*hits)/trials
end
```

This program calculates π by calculating the area of the unit circle by testing points at random in the ± 1 square.

The built-in subroutine `random_number` sets its argument to a random value in the range $0 \leq x < 1$. If passed an array, it sets all the elements to different random values.

This program will give a different answer each time it is run: try it!

Decisions

```
integer die
real x

call random_number(x)
die=1+6*x

select case (die)
case (1:3)
  write(*,*)'Stay in bed'
case (4,5)
  write(*,*)'Skip lecture'
case (6)
  write(*,*)'Accuse linguist of laziness'
case default  ! This should never happen
  write(*,*)'Attend lecture'
end select
end
```

Although the above is possible using `if ... else if ... else if ... else ... endif`, the above construction can look tidier. In the case statement, `(a:b)` specifies a range, `(a,b)` two discrete values, and `default` what will happen if none of the `case` conditions matches.

A more practical use of the above would be controlling the motion of a particle performing a random walk.

Factorials

No set of programming examples is complete without this:

```
integer i,n,dp
parameter (dp=kind(1.0d0))
real (dp) :: fact

write(*,*)'Please input n'
read(*,*)n

fact=1
do i=2,n
  fact=fact*i
enddo

write(*,*)n,' factorial is ',fact
end
```

The double precision `real` is used for its greater range than `integer` or `real`.

Extending Fortran

```
module fact_mod
contains
  function factorial(n)
    integer i,n
    real (kind(1.0d0)) :: factorial
    factorial=1
    do i=2,n
      factorial=factorial*i
    enddo
  end function
end module

program test
  use fact_mod
  integer i
  write(*,*)'Please input i'
  read(*,*) i
  write(*,*)i,' factorial is ',factorial(i)
end
```

The Details

The program now starts with a *module* which contains one *function* definition.

The function definition is reasonably straightforward. Notice that here the function is going to return a `real` value of `kind` that of `1.0d0`, so the function name is declared as though it were a variable of that type and kind, and the value assigned to that variable when the function ends is the value which the function will return to the calling program. The type of the function arguments must also be declared.

The unnecessary but clarifying `program` statement marks the start of our program. It immediately specifies that it wishes to use the module just declared, and then it calls `factorial` just as it would any other FORTRAN function.

Note that the module precedes the program in the source file, and that the module name is not the same as any variable or function name.

`factorial`, as defined here, takes an integer argument and returns a real. A function need not return the same type as its argument(s).

The Broader Picture

Variables declared within a function are independent from any declared elsewhere. In the above example `i` is used for different purposes in the main program and the function.

Thus two different people can write the main program and the function, and, provided they agree on what arguments the function takes, and on what it returns, neither need know anything about the internal details of the other's code.

This makes managing largish programs with multiple programmers possible!

Subroutines

A subroutine is very like a function. Consider swap.f90:

```
module swap_mod
contains
  subroutine swap(i,j)
    integer i,j,tmp
    tmp=i ; i=j ; j=tmp
  end subroutine
end module

program swapit
  use swap_mod
  integer small,smaller
  small=3 ; smaller=2
  write(*,*) small,smaller
  call swap(small,smaller)
  write(*,*) small,smaller
end
```

Subroutines in more detail

A subroutine returns no value itself, and has to be called explicitly with a `call` statement. It can, however, alter its arguments.

So can a function, but it is considered very bad practice.

How are the arguments modified? When the above subroutine was called, it was not passed '3' and '2' directly, but rather the addresses in memory of '`small`' and '`smaller`'.

Simply the addresses. Not their names.

The subroutine read from these addresses, assuming it would find integers stored there. And then it wrote back to those addresses. When the main program continued, the values stored in the memory locations it calls '`small`' and '`smaller`' had thus changed.

Arguments are passed to functions in the same way.

Libraries: the Refuge of the Lazy?

Laziness in programming can be a virtue. Rewriting an algorithm that someone else has already written is pointless (unless one does it better). People sell collections of well-written functions and subroutines in bundles called `libraries`.

We have already used libraries implicitly, for the standard Fortran library includes definitions of the trig functions amongst other things.

To use extra libraries we must include something in our program to tell the first phase of the compilation process to expect to find that something has been left undefined, and then we must tell the final phase, the linker, where to find it.

For the NAG library we are to use, this is quite straightforward.

The NAG library, developed by the Oxford-based Numerical Algorithms Group, is copyrighted commercial software. The version we shall use was developed for FORTRAN77, so does not make use of all the features of F90. Most noticeably, its routine names suffer from a seven character limit, rather than a 31 character limit.

Very simple NAG

First a test just to show successfully compiling and running a program calling NAG.

```
use nag_f77_a_chapter
write(*,*)'Calling NAG identification routine'
write(*,*)
call a00aaf
end
```

which should be compiled with

```
f95 nag_test.f90 -lnag
```

The output produced by this program is:

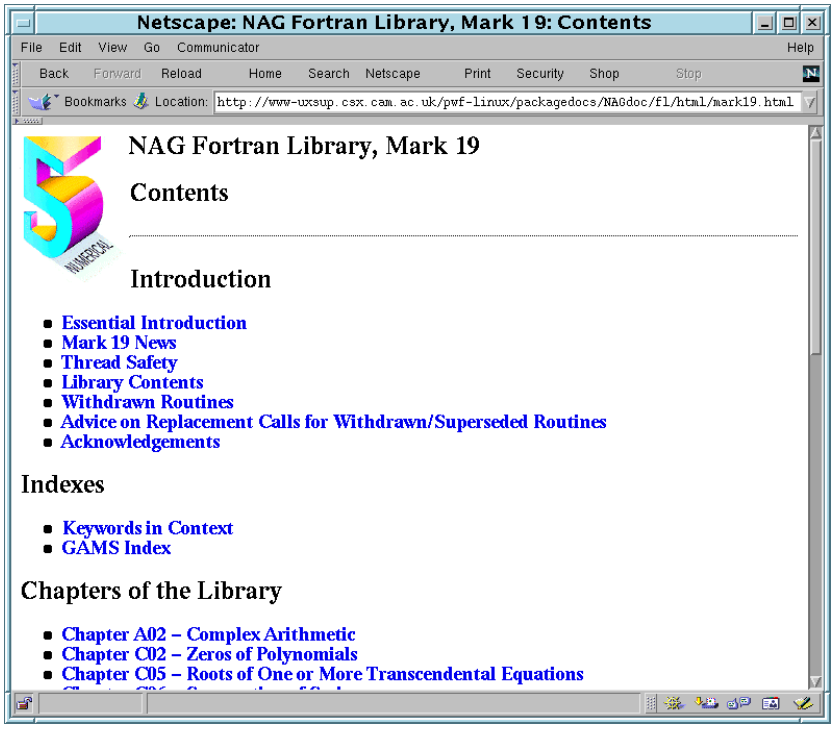
```
 Calling NAG identification routine

 *** Start of NAG Library implementation details ***

 Implementation title: Linux (Intel) NAGWare f95 (RedHat 6.x)
            Precision: double
         Product Code: FLLUX19D9
                 Mark: 19A

 *** End of NAG Library implementation details ***
```

The structure of the NAG library

The library is divided into 12 chapters. Each covers a different area of numerical methods and has its own module, but the one library includes all chapters.

When using PWF Linux, the library and the modules are in standard places which the compiler will search.

The NAG library is documented by paper manuals in many PWFs, and by online documentation accessible from the WWW page for this course.



N.B. This version of the NAG library never uses single precision reals, nor any sort of complex.

A non-trivial NAG example: matrix determinant

A quick scan of the documentation produces the slightly cryptic:

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.

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$\geq$max(1,N).

3: N – INTEGER Input
On entry: n, the order of the matrix A.
Constraint: N$\geq$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.

Interpretation

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 F77, which does not support dynamic memory allocation, it is often necessary to pass 'workspace' to a routine. This space the routine will use for its own internal temporary requirements. Hence `WKSPCE(*)`.

Because F77 has no decent definition of what 'single precision' and 'double precision' mean, all reals are described as *real*. In this version, this means `real (kind(1.0d0))`.

Finally, array dimensions of unknown size are denoted by a '*', not a ':' – another F77ism.

An example

```
use nag_f77_f_chapter
real (kind(1.0d0)) :: m(3,3),d,wrk(2)
integer i,n,ifail

m(1,1)=2   ;   m(1,2)=0
m(2,1)=0   ;   m(2,2)=2

i=3 ; n=2 ; ifail=0

call f03aaf(m,i,n,d,wrk,ifail)

write(*,*)'Determinant is ',d
end
```

Notes on F03AAF example

First the relevant NAG module is included, so that the compiler can do some checking of arguments to the subroutine.

Try swapping the `m` and `i` on the call to `f03aaf` and recompiling.

The array which will hold the matrix, the subroutine's workspace and the output variable are all declared. The arrays could have been dynamic if we so chose.

The variables are initialised. The matrix is simply twice the identity.

The routine is called, and the result written out. Notice that the 2x2 matrix is held in a 3x3 array, so the array size and the matrix size differ. This was done on a whim.

The full NAG documentation offers example programs too, albeit in F77.

Try finding the determinants of:
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$

Function maximisation

NAG provides routines for minimising functions, but not maximising. If we wish to maximise xe^{-x} for small positive x, this is the same as minimising $-xe^{-x}$.

A suitable NAG subroutine is `E04ABF`, whose specification is:

```
SUBROUTINE E04ABF (FUNCT, E1, E2, A, B, MAXCAL,
                                        X, F, IFAIL)
INTEGER MAXCAL, IFAIL
real E1, E2, A, B, X, F
EXTERNAL FUNCT
```

`E1` and `E2` define the accuracy required, and `A` and `B` the range to search. `MAXCAL` is the maximum number of trials, and `IFAIL` acts as before. `X` will be set to the position of the minimum, and `F` to the function value there.

This leaves `FUNCT`, which is the function to minimise, and which we must supply.

Passing subroutines to subroutines

The NAG documentation for this reads:

1: `FUNCT` – SUBROUTINE, supplied by user External Procedure

This routine must be supplied by the user to calculate the value of the function $F(x)$ at any point x in $[a, b]$. It should be tested separately before being used in conjunction with E04ABF. Its specification is:

`SUBROUTINE FUNCT(XC, FC)`
real `XC,FC`

1: `XC` – *real* Input
On entry: the point at which the value of F is required.

2: `FC` – *real* Output
On exit: `FC` must be set to the value of the function F at the current point x.

`FUNCT` must be declared as `EXTERNAL` in the (sub)program from which `E04ABF` is called. Parameters denoted as *Input* must **not** be changed by this procedure.

Writing for NAG

So NAG is asking us to supply a subroutine (not a function) which takes two arguments, and returns the value of the function it is evaluating in the second.

The references to `EXTERNAL` refer to the old F77 style of function and subroutine declarations: we can continue to use modules as before.

Of course this example is trivial analytically, the result being a minimum of $-1/e$ at $x = 1$. However, it is good to start with examples whose results one can check!

Function minimisation: E04ABF

```
module x_ex_mod
contains
  subroutine x_ex(x,f)
    real (kind(1.0d0))::x,f
    f=-x*exp(-x)
  end subroutine
end module

program minimize
  use nag_f77_e_chapter
  use x_ex_mod
  integer ifail,maxit
  real (kind(1.0d0))::e1,e2,a,b,x,f

  e1=0; e2=0 ! Routine will use defaults
  a=0.5 ; b=1.5 ; maxit=50 ; ifail=-1

  call e04abf(x_ex,e1,e2,a,b,maxit,x,f,ifail)

  if (ifail==0) then
    write(*,*)'Minimum of ',f,' at ',x
  else
    write(*,*)'Minimum not found ',ifail
  endif
end
```

Variables in modules

Sometimes it is useful to have variables (or constants) declared once and used in different functions and subroutines. One example would be:

```
module constants
  real (kind(1.0d0)), parameter ::  &
      pi=3.1415926535897 93d0, &
       h=6.62606891d-34
end module

module print_pi_mod
contains
  subroutine print_pi
    use constants
    write(*,*)'pi is ',pi
  end subroutine
end module

program hbar
  use constants
  use print_pi_mod
  write(*,*)'hbar is ',h/(2*pi)
  call print_pi
end
```

Integration

Numerical integration, often called quadrature, is a common task: analytic integration can quickly become impossible!

As an example we shall consider a situation where the analytic result is (relatively) easy, and where the numerical approach is hard as the range of integration is infinite. However, NAG can cope with this.

The integral to consider is:

$$\int_{-\infty}^{\infty} e^{-\lambda x^2} dx$$

So that this can readily be evaluated for different parameters λ, we shall pass λ via a module.

The NAG routine we shall use is D01AMF.

```
module lambda_mod
  real (kind(1.0d0)) :: lambda
end module

module gauss_mod
contains
  function gaussian(x)
    use lambda_mod
    real (kind(1.0d0)) :: gaussian, x
    gaussian=exp(-lambda*x**2)
  end function
end module

program integrate
  use lambda_mod ; use gauss_mod
  use nag_f77_d_chapter
  integer, parameter :: lw=1000, liw=lw/4
  integer i, inf, ifail, iwrk(liw)
  real (kind(1.0d0)) :: b, result, err, epsabs, &
                                    epsrel, wrk(lw)
  inf=2 ; ifail=1
  epsabs=-1 ; epsrel=1d-8

  do i=1,20
    lambda=0.1d0*i
    call d01amf(gaussian,b,inf,epsabs,epsrel,  &
                result,err,wrk,lw,iwrk,liw,ifail)
    write(*,*) lambda,result
  end do
end
```

File I/O

Writing directly to a file is convenient when the output is too much to fit on the screen! This can be achieved thus:

```
   integer i
   open(20,file='cubes.dat')
   do i=1,100
     write(20,10)i,i*i,i**3
   enddo
   close(20)
10 format(i6,i8,i10)
   end
```

The `open` statement creates a file called '`cubes.dat`' and associates it with I/O *unit* number 20. This statement must come before the first use of that unit number in a `write` statement.

The first `*` in the `write` statement is now revealed to be the unit number. The `close` is good practice, and should follow the last use of that I/O unit. The `end` statement should also close any units still open. More details are given at the end of these notes.

When it all goes wrong

Programs inevitably have bugs, arising from a variety of causes.

- Syntax of F90 misunderstood
- Details of library call misunderstood
- Brain forgot what it did a few lines back
- Fingers improvised

It is therefore essential to test programs thoroughly on data for which the correct results are known: not all bugs cause instant crashes!

Debugging by `write` statements

The most basic form of debugging, yet quite effective, is placing extra `write` statements in the code. These can be used either to show how far a program has got before crashing:

```
write(*,*)'Third checkpoint reached'
```

or to check what values variables are taking, particularly before evaluating a complex expression.

```
write(*,*)'x=',x
y=1.0/exp(x*x)
```

N.B. The above will fail for quite modest values of `x`.

Run-time checks

```
real    x(2,10)
integer i
do i=1,1000
  x(1,i)=i
  x(2,i)=i*i
enddo
...
end
```

This program writes beyond the end of an array: `x(1,11)` does not exist. The compiled code does not check every array reference to ensure that it is in the declared bounds: doing so would slow the code down. Most compilers can be told to add the checks to the code by compiling with the extra option '`-C`'.

```
$ f95 -o bug bounds.f90
$ ./bug
Segmentation fault
$ f95 -C -o bug bounds.f90
$ ./bug
Subscript 2 of X (value 11) is out of range (1:10)
Program terminated by fatal error
Abort
```

Debuggers

An alternative approach is to use a debugger. This can be much more versatile than the above methods, and the following only scratches the surface of what is possible.

Firstly one must compile the code with the '-g90' flag. This causes extra information to be placed in the executable file about variable names and source line numbers.

```
$ f95 -o bug -g90 bounds.f90
```

Then one starts the debugger, specifying which program one wishes to debug. The debugger used here is called 'dbx90'.

```
$ dbx90 bug
```

Finally, one asks the debugger to run the program:

```
(dbx90) run
```

dbx90

```
(dbx90) run
Program received signal SIGSEGV, Segmentation fault.
(dbx90) where
    $MAIN$(argc, argv) at line 5 in "bounds.f90"
[C] ??() in "memory"
(dbx90) list 5,5
5         x(1,i)=i
(dbx90) print i
I = 133
(dbx90) whatis x
REAL::X(2,10)
(dbx90) quit
```

So using a debugger one can make simple enquiries about the value of variables and how they were declared.

Cores

When a program crashes, it often produces a file called 'core' in the directory it was running in. This file can be very large: it contains a snapshot of the whole of that program's memory when the crash occurred. The creation of such files can be turned off with the UNIX command
`ulimit -c 0`
and on with the command
`ulimit -c unlimited`

There is a use for these files. Provided that the program was compiled with the -g option, a debugger should be able to read the core file and show where the program was and what values variables had at the time of the crash. Most debuggers can be told to read such files by specifying 'core' after the program name on the command line. `dbx90` will read in any core file automatically.

Core files can also do nasty things to one's quota if not removed!

The Good Style Guide

Most of the examples given are not particularly good: the requirement to fit on one overhead is rather constraining. Good advice would include:

- Do use `implict none`.
- Use comments liberally.
- Indent structures neatly
- Use names which make sense.

And remember: however good it looks, always test it.

A good program starts with comments describing what it does. Further comments describe what the variables declared will do (unless it is very obvious). Likewise any function or subroutine, for which describing precisely what input variables are required and what output is given is very important.

Calling variables 'a', 'b' and 'c' is generally much less helpful than '`energy`', '`charge`', '`electron_mass`' etc.

Exercises

These exercises are not meant to be prescriptive. They are merely intended to stimulate ideas and to save you from discussing the weather with the demonstrators. They range from the very basic to a level a little beyond what is required to complete this Part II course successfully. Those parts which are certainly beyond what is required are indented from the left and right.

The Basics

Try creating the two-line program on page 5 and compiling it, following the instructions there. Note that the quote mark used at each end of the text `'Hello, world.'` is that produced by the key close to the right hand shift key, and is not a double-quote.

Use the UNIX `ls` command to see that the *executable* file `hello` does indeed appear alongside the *source* file `hello.f90` once you have successfully run the compiler. Use the `-l` option to `ls` to see how much bigger the executable is than the original source. Do *not* attempt to read the executable using the UNIX `cat` command: if you must be curious, use `less` instead. (Using `cat` will damage little apart from to your nerves and may cause you to need to kill the window in which you typed the command.)

Try creating a few more lines of output by adding some more `write(*,*)` commands to the program. Notice that `write(*,*)` with no text following it is valid and produces a blank line.

Type in the six line program on page 6. Lines three and five are identical. Can you persuade your editor to copy line three, rather than type it twice?

This time there are no instructions on what to call the source file, nor the executable, nor is the compile command explicitly given. But it should be possible to work something out from the previous example.

Try changing any of lines two to five and see if your changes produce the expected results. Do remember that after changing the source, you must save it and recompile if you want to run the new version of the program, rather than the old! Until you save it, the source file in your directory will be unchanged, and until you recreate the executable by recompiling the new source file, the old executable will remain and can be run as many times as you please. Also remember that pressing the 'up' cursor key will recall your last commands, so you need not keep on typing the compile command: pressing ↑ a couple of times should usually find one, which can then be edited using the left and right cursor keys if necessary.

Look at the program on page 7, and try changing the program you have just been working on to demonstrate some of the extra features of this longer code. Ensure you understand that `10/4 = 2`. Verify that `3**2`, `3**3` and `3**4` are what you expect. Is `3**30` what you expect? Check using a calculator. (No calculator? Type '`xcalc &`' at a UNIX prompt, and use the mouse to click on `3, y^x, 3, 0, =`.) You have just discovered that the range of integers is somewhat finite. In fact, the range is $-2^{31} \leq x \leq 2^{31} - 1$.

The example programs are beginning to get longer. Rather than typing them all in, one can either download them from the course WWW pages, or copy them from the shared filespace on the PWF using the UNIX `cp` command. The UNIX *environment variable* `$PHYTEACH` has been set up to point to this, which enables one to view this directory by typing `ls $PHYTEACH`. Similarly, to copy an example from a directory there to your current directory, a command such as

```
cp $PHYTEACH/examples/quadratic.f90 .
```

would suffice. Alternatively, one can try using `cd` to change to this directory `cd $PHYTEACH`, have a look around, using `less` to inspect files, but without trying to compile anything directly here, as you cannot write the executable into this directory. Use

```
cp quadratic.f90 ~
```

to copy a file back to your home directory (both `$HOME` and $\sim$ are ways of specifying your home directory), and then simply typing `cd` on its own will take you back to your home directory once more.

Copy (or type in) the example quadratic solver on page 8. Compile and run it. Test it on a few simple examples, such as:

$$\begin{aligned} x^2 - 1 &= 0 \\ 2x^2 + 3x - 10 &= 0 \\ x^2 - 2 &= 0 \\ x^2 + 1 &= 0 \\ x + 1 &= 0 \end{aligned}$$

Notice that you do not need to recompile each time you wish to run the program.

It should fail on the last two examples. Why?

Look at page 10. Can you work out how to insert this code fragment into the program so that it fails more gracefully when there are no real roots? Try it.

Can you work out something to add so that it deals correctly with the case of the coefficient of x^2 being zero?

Finally, look at the example of repeated execution on page 12. Type in and run this short program. Try changing the length of the loop by replacing the '`1,20`' by '`1,1`' or '`1,200`'. Try '`1,0`': what happened? Try `1,20000`. Recall that in UNIX pressing the control key (marked `Ctrl`) and 'C' simultaneously should stop a runaway job. Curse gently that although the job should/will stop immediately, several hundred lines of output may be buffered up between it and your terminal: a job can easily get some way ahead of the point which your terminal displays if the output is arriving particularly rapidly.

Try '`1,1300`': what is happening after `i=1290`? (Hint: $1291^3 > 2^{31} - 1$) Try the suggestions for neater output on page 13. What happens between `i=215` and `i=216` now? At `i=317`? At `i=465`?

Congratulations: you have survived the basic examples. Remember that you should take frequent breaks when using computers extensively in order to avoid RSI, eye-strain, etc., and then move on to the next section.

Simple programs

Look at the simple code for factorisation on page 15. Try compiling it and running it. Which of 1 234 567, 1 021, 8 388 607 and 524 287 are prime? Test it also on some smaller numbers where you know the correct answer!

What is the largest number which can be tested with this program?

Can you modify the code so that it prints all factors of the number input?

Why is testing up to $\sqrt{p}$ rather than p (still) sufficient?

What happens if one tries to write `sqrt_p=sqrt(p)`?

The program on the following page tests all odd numbers less than $\sqrt{p}$, rather than all numbers less than p. It is thus twice as fast, and a

demonstration of using a `do` loop with an increment other than one. Notice that the 'slow' program can factorise the largest number it can deal with in under a second, so the speed enhancement is hardly necessary.

Data types

Data types may appear to be tedious, but their misuse is a cause of many bugs. Look at the example on page 22 and ensure you understand it. The two kinds of real used, single and double precision, correspond to the two kinds of real that most modern CPUs can deal with automatically.

What is (probably) wrong with the following programs?

```
program pi_bug
integer, parameter   :: dp=kind(1.0d0)
real (dp), parameter :: pi=3.14159265358979

write(*,*)pi
end
```

and

```
program sqrt2_bug
integer, parameter :: dp=kind(1.0d0)
real (dp) :: sqrt2

sqrt2=sqrt(2)

write(*,*)sqrt2
end
```

Notice that one of these examples (which?) compiles and runs, but probably does not do what one might naively expect.

What does the following program do, and why?

```
program equality
real a,b
integer i,j

i=123456789
j=i+1

a=i
b=j

if (i==j) write(*,*)'i==j'
if (a==b) write(*,*)'a==b'

end
```

Which of the following expressions would, according to Fortran, be zero at the end of the above program?

```
a-i
b-j
a-123456789
a-123456789.
a-123456789d0
a-real(i)
a-real(i,kind(1.0d0))
```

In the last expression, `real(i,kind(1.0d0))` converts its argument directly to a double-precision real, rather than the single precision that `real(i)` produces.

Declarations

Look at the example of the variable `salery` being automatically declared on page 25. Convince yourself that it really is better if a compiler spots these typos rather than accepting them, and therefore that using `implicit none` at the beginning of a program is a very good thing.

For the record, neither C nor C++ provide any mechanism for such automatic declarations, whereas languages like Basic and Perl use them almost exclusively.

Arrays

An array can be a very convenient way of storing large amounts of data. A two-dimensional array could be used to store atmospheric temperature or pressure at ground level on a 10km grid covering the UK. A more complicated requirement would be to store wind velocity on a 3D grid. This would require a four-dimensional array, as the velocity itself is a vector.

Try rewriting the simple program on page 12 so that one loop fills an array with `i`, `i*i` and `i**3`, and a second loop writes out the contents of the array.

Look at the example of dynamic arrays on page 31. Can you change the program from the above paragraph to prompt for a maximum value for `i` and to declare an array of the appropriate size? Remember not to deallocate an array before you have finished using it!

More Simple Programs

Herewith some simple programs with no subroutines, functions or libraries.

Look at Eratosthenes' sieve on page 34. Eratosthenes was a Greek mathematician who lived in the third century BC, and invented the well-known method for finding prime numbers. The method is to cross out all multiples of 2 except 2, all of 3 except 3, all of 4 except 4, etc. The numbers not struck out are prime.

The program creates the array `primes` and sets all elements to one. Elements are then struck out by setting them to zero. Finally, the elements left are printed.

Try running it for numbers up to about 100 000. You will soon see that the calculation time is almost instantaneous, but the time to write out the answer is quite long! Indeed, you may need to recall that pressing the control key (marked '`Ctrl`') and '`C`' together can be useful for aborting such output, although the effect will not be immediate.

One could simply print the highest prime found, by making the final loop

```
do i=prime_max,2,-1
  if (primes(i)==1) then
    write(*,*)i
    stop
  endif
enddo
```

This is now a program which potentially takes more than an instant to run, and which is not delayed by writing to the screen. One can find how long it takes by using the UNIX `time` command:

```
bash$ f95 -o sieve eratosthenes_largest.f90
bash$ time ./sieve
 Input largest number for search
```

```
10000
 9973
0.00user 0.00system 0:02.16elapsed 0%CPU (...)k
0inputs+0outputs (128major+36minor)pagefaults 0swaps
```

That tells us that the program ran for 2.16 seconds of 'wall-clock' time, and consumed 0.00s of CPU (user) time. The difference is the time it took me to enter the number '10000'.

However, entering a million makes the user time about 0.87s, ten million takes 10.6s, and one hundred million causes the program to run out of memory (at least on the machine I was testing this on). Even ten million will require forty million bytes (40MB) to hold the array `primes`, which is becoming somewhat large, and antisocial on a multiuser machine!

The next program to examine is that to calculate π on page 35. Set trials to 1000, and run the program a few times. Notice that the answer varies: different random sequences are used each time. If you run the program several times in very quick succession (hint: either use the up arrow key to recall the previous command rapidly, or make use of the ';' character which separates multiple UNIX commands on the same line, just like FORTRAN, so '`./pi ; ./pi ; ./pi`' is valid) it gives the same answer each time. The time of day is being used to set the initial value produced by the random number generator. This is not good, but getting genuinely random numbers from a computer can be hard.

Alter the code so that it prompts for the number of trials. Notice that although the estimates for π get better with increasing numbers of trials, they do so rather slowly. A million trials get three figures fairly reliably, and ten million four figures.

Notice too that although the code becomes rather slow when several million trials are requested, the memory use stays constant and tiny: there are no

large arrays here. The use of an integer for the loop counter restricts us to about two thousand million trials, which would take around 10 minutes on a fastish PC.

Some compilers produce code which gives the same 'random' sequence each time it is run. This can be very useful for debugging. Though the default in Fortran is undefined, either behaviour can be explicitly requested by setting the seed the generator uses. The following code is usually good enough.

```
integer i(1)
i(1)=...
call random_seed(put=i)
```

sets the seed. Set `i(1)` to a constant for the same sequence each time, or use

```
integer i,j(1)
call system_clock(i)
j(1)=i
call random_seed(put=j)
```

for a different sequence each time.

The potential problem with the above code is that the seed is not necessarily an integer array of length one: some compilers may require a longer integer array. The sort of seed a given compiler uses can be determined using

`call random_seed(i)`

which sets `i` (an integer) to the length of the array of integers which constitutes a seed. It is almost always one, and certainly is for the compiler used for this course. Completely correct code would be:

```
integer, allocatable :: seed(:)
integer i
call random_seed(i) ! Get size of seed array
```

```
allocate(seed(i))
! Next line if same sequence each time desired
seed=1
! Next two lines if different sequence each time desired
call system_clock(i)
seed=i
call random_seed(put=seed)
deallocate(seed)
```

The next example (on page 36) is given in a particularly trivial form: that of working out what to do in the morning. However, (why) is

```
call random_number(x)
die=1+6*x
```

the correct code for setting `die` to a random integer between 1 and 6, thus simulating the roll of a normal six-sided die?

The final example program in this section is that which calculates factorials (see page 37). Why is `n` declared integer, but `fact` double precision real? What is the largest number this program can cope with? Does the program stop or produce wrong answers for larger inputs?

Structured Programming

So far our programs have had a very simple structure: just a single piece of code, with, perhaps, some loops or conditional statements. Now we shall consider breaking our code into independent self-contained pieces, and thus introduce the concepts of functions and subroutines. First we consider defining a function to calculate factorials, just like the previous example program.

The example concerned is on page 38, and looks somewhat complex. Try deleting the four lines:

```
module fact_mod
contains
end module
use fact_mod
```

and adding the line
`real (kind(1.0d0)) ::  factorial`
immediately after the `program test` statement.

This reduces the program to something close to Fortran77 syntax, and it will still compile and run.

The points to notice are that the name of the function is unique within the program, and that within the function itself the function name is declared just like any other variable, and has values assigned to it like any other variable. The value of that variable when execution reaches the `end function` statement, or when it reaches a `return` statement, is the value the function returns. The current function returns one for all negative inputs. If one wished it to return -1 to indicate the error, one could add

```
if (n<0) then
  factorial=-1
  return
endif
```

before (or after) the loop. Notice that the variables `i` and `n` in the function are not the same as variables of the same name in the main program.

The addition of the module (the four lines which are potentially removeable) serves an important purpose. It encourages the compiler to check that we

are calling our function correctly: that is we are passing it one integer value and we expect a double precision value back. The NAG compiler used in this course is quite good at checking itself when it is able, but many other compilers do not unless the language requires them to do so. The module also serves to declare the name `factorial` within the main program, so it should not be (re)declared explicitly here anymore. It is good practice to use modules.

> Even the NAG compiler will fail to do argument checking if we chose to split this program into two input files without using a module. With the module, it is perfectly reasonable. Simply create two files, the first containing the lines up to and including the `end module` statement, the second containing the main program. Then compile using:
> ```
> $ f95 -c file1.f90
> $ f95 -o myprog file2.f90 file1.o
> ```
> Notice the first command will create two files, one called `fact_mod.mod` and one called `file1.o`. The first is required by the `use fact_mod` statement in the main program. It contains information about the definition of the factorial function (i.e. that it wants an integer and returns a double precision). The second contains the actual factorial function in compiled binary form.

A subroutine is very like a function, except that it returns no value. It may, however, modify any of the variables passed to it. It is usual practice to use a function only if the routine is to return a single value and print nothing, and to use a subroutine otherwise. A function thus restricted behaves very much like all the other functions which one is used to: `log`, `sin`, etc.

Look at the example of using a subroutine on page 41. Consider that when `i` and `j` are passed to the subroutine, actually the locations in memory which hold `i` and `j` are passed. Thus the subroutine can modify them, by

storing new values there before it returns. However, as the contents of the memory will be just a collection of binary bits, it is important to ensure that one passes the correct type to the subroutine. If the subroutine is told to expect a four-byte integer, and you have actually stored an eight-byte double precision real at the address, the subroutine has no way of knowing this, and strange things will occur. Correct use of modules will prevent most of this chaos, as the compiler will notice and complain.

Libraries

The climax of this course, and most of the problems, is successfully calling a library. This is the sane way to write code: break your task down into general tasks most (all?) of which have been solved before, and call the relevant routines. This you have already been doing to a large extent: if you wish to evaluate `sin(2.6)` you write `sin(2.6)`. You do not worry about reducing 2.6 to the first quadrant and then looking up some exciting polynomial or polynomial fraction which will usefully approximate the sine function in this region: that is all done for you. The library functions we shall consider now are more complex and comprehensive than sine, but conceptually similar.

The first example on page 44 simply calls NAG's identifaction routine. The Fortran starts with a `use` statement, for modules are provided describing all NAG routines. It then calls the subroutine `a00aaf` which asks the library to identify itself. Unfortunately all NAG routines have these bizarrely unmemorable names. Notice that as `a00aaf` takes no arguments, not even brackets are needed after its name.

When compiling we must ask the compiler to look in the NAG library for any functions or subroutines we have called and yet not provided code for ourselves. This is done with the '`-lnag`' on the end of the compile line. The result should be roughly as shown: try it.

A more worth-while example is given starting on page 49. Finding a determinant is actually quite hard: there are several simple algorithms which are algebraically correct, but which do not cope well when executed with the limited precision and range of a computer's arithmetic. Such problems rarely occur with small matrices, but do readily occur once the matrix has even a few tens of rows and columns. The people at NAG will have produced a solver which is both fairly robust and which gives an error if it finds the problem too hard (rather than simply giving a wrong answer).

The NAG documentation talks about LU factorisation and partial pivoting. It is not the purpose of this part of this course to explain or worry about such techniques.

It then describes in detail what parameters should be passed to the routine, and what it will return. Remember that when a matrix is passed to a subroutine, all the subroutine is given is the address in memory at which the first element of that matrix is stored. Thus it is necessary to pass separately information about how large a matrix to expect. (Thus 'traditional' languages like C and Fortran77. More 'modern' languages, such as C++ and Fortran90, are capable of passing a single complex object such as an array complete with size and shape information. Unfortunately the library which it has been decreed we shall use was written for Fortran77. In some ways this is an advantage: the language is simpler, and no 'magic' is occurring behind one's back.)

NAG's library offers three choices of error detection, which are chosen by setting the value `ifail` on entry. These are:
`ifail=-1` print error message and continue
`ifail=0` print error message and stop
`ifail=1` continue
If you choose `ifail` to be other than zero, it is *essential* to check its value after calling the NAG routine. Any value apart from zero at this point

means that some form of error occurred (i.e. on success the NAG routine will reset it to zero).

The final examples

The most complex examples in this course, which certainly go as far as you will need to solve any of the problems set, are those which call a NAG routine which itself needs a user-defined function or subroutine as an argument. The first example is on page 54, where we determine numerically the maximum of the function xe^{-x}.

Careful reading of the NAG documentation is always required to ensure that the function or subroutine provided by the user does precisely what the library requires. In this case the subroutine is easily written. The NAG routine itself has a large number of arguments. This is not unreasonable: when performing numerical minimisation, integration, root finding, etc., one often wishes to trade off speed against accuracy, and NAG therefore provides mechanisms for specifying the required accuracy. For a problem this simple, using the defaults is quite reasonable. NAG also expects a hint as to where to look for the minimum. Again, this is very useful: our subroutine would fail if called to evaluate the function at $x = -10000$ (why?), but, by telling NAG to look for a minimum between 0.5 and 1.5 we get a guarantee that NAG will not ask for the function to be evaluated outside of this range.

Try modifying the program to find that minimum of the sinc(x) function (defined as $\sin(x)/x$) which lies between 3 and 5. Try also calling your sinc function at $x = 0$: what trap exists here? Notice that unlike the previous example, this minimisation is rather hard if attempted algebraically.

As a prelude to the final NAG example, we consider one further use of modules. So far a module has contained either a function or a subroutine, and the module name and function / subroutine name must be distinct.

In fact a module may contain multiple functions and subroutines and also variables (including parameters). Thus one can have a module containing the value of π or $\hbar$ and include it whenever one needs that constant. A simple example is given on page 55. By using modules in this fashion we can arrange that certain identically-named variables in the main program and a subroutine (or in multiple functions / subroutines) are identical. This can be very useful as we shall see in the final NAG example.

That final example is to evaluate numerically

$$I = \int_{-\infty}^{\infty} e^{-\lambda x^2} dx$$

for values of λ from 0.1 to 2 in steps of 0.1.

NAG has a routine for doing integration over an infinite or semi-infinite interval, at least for functions as well-behaved as this one, and, as usual, we can specify the precision we require and we are required to provide some workspace. However, NAG wishes to call a function of x only, and we wish to define a function of λ and x. This is resolved by letting NAG vary x in the obvious manner, and using a module to contain λ. By using this module both in our main program and the function, λ is shared between the two without the NAG routine knowing, or needing to know, anything about it.

The analytic result of

$$\begin{aligned} I^2 &= 2\pi \int_0^{\infty} x e^{-\lambda x^2} dx \\ &= \pi/\lambda \end{aligned}$$

is familiar. Change the program to print both the numerical result and the result of evaluating directly $\sqrt{\pi/\lambda}$.

Debugging

Bugs are inevitable and unpleasant. Try experimenting with the three debugging techniques mentioned starting at page 59.

Usually an error of 'segmentation fault (or violation)' or 'sigsegv' means that an array bound has been exceeded, whereas a 'floating point (or numeric or arithmetic) exception' or 'sigfpe' means that an impossible piece of floating point arithmetic (e.g. division by zero, square root of negative number, exponentiation of something large) has been attempted.

Change the gaussian integration routine so that the function reads
`gaussian=1/exp(lambda*x**2)`
and investigate the crash which will follow.

For instance, adding:
`write(*,*)lambda,x`
produces the output:

```
   0.1000000000000000   1.0000000000000000
   0.1000000000000000  -1.0000000000000000
   0.1000000000000000   2.3306516868994831E+02
*** Arithmetic exception:  - aborting
Aborted (core dumped)
```

In other words, NAG requested function evaluations at $x = 1$, $x = -1$ and $x = 233.065$. Calculating `exp(5431)` is (far) beyond the permitted range of about 10^{308}.

NAG's compiler also has the useful option '`-gline`' which causes the following output:

```
*** Arithmetic exception:  - aborting
file: gaussian_bug.f90, line 10
Aborted (core dumped)
```

Using the debugger results in the following:

```
$ f95 -g90 -o bug2 gaussian_bug.f90 -lnag
$ dbx90 bug2
NAGWare dbx90 Version 4.1(15)
Copyright 1995-2000 The Numerical Algorithms Group Ltd.
(dbx90) run

Program received signal SIGFPE, Arithmetic exception.
(dbx90) where
[C] __exp(x) at line 45 in "w_exp.c"
    GAUSS_MOD‘GAUSSIAN(GAUSS_MOD‘GAUSSIAN‘X) at line 10
    in "gaussian_bug.f90"
[C] d01amz_()
[C] d01amv_()
[C] d01amf_()
    INTEGRATE(argc, argv) at line 26 in "gaussian_bug.f90"
[C] __libc_start_main(...) at line 129 in "libc-start.c"
(dbx90) up
Current scope is GAUSS_MOD‘GAUSSIAN
(dbx90) list 10,10
10          gaussian=1/exp(lambda*x**2)
(dbx90) print x
X = 233.06516868994831
(dbx90) print lambda
LAMBDA = 0.10000000000000001
(dbx90) print lambda*x*x
LAMBDA*X*X = 5431.9372856474065
```

```
(dbx90) quit
$
```

This tells us that the program died whilst attempting the numerically impossible

```
Program received signal SIGFPE, Arithmetic exception.
```

and that this was in a routine called `__exp` for which there is no source available. Typing `where` shows the current call stack. Our program (`main`) called `d01amf_` (no surprise there: we did call the NAG routine `d01amf`). It then called `d01amv_` and `d01amz_` before calling `GAUSS_MOD_MPG̀AUSSIAN`, which is our gaussian function with its name embellished to show that it is contained in the `gauss_mod` module. This called `__exp` and the program died.

By typing `up` we ask the debugger to move its attention one step up this stack, that is from the `exp` function to our `gaussian` function. We can now list the line which called `exp`, and print the values of `x` and `lambda`, or even simple expressions involving these (however note ** for exponentiation is not understood by the debugger).

Note that not all bugs cause programs to fail, or even to produce obviously wrong answers.

The `format` statement

An incomplete list of options to the `format` statement for reference.

`i`x	integer, at most x characters
`i`x`.`y	integer, at most x characters and at least y (padded with leading zeros)
`f`x`.`y	floating point, at most x characters and precisely y characters after the decimal point
`e`x`.`y	exponential, at most x characters and precisely y characters after the decimal point
`d`x`.`y	ditto, but use 'D' not 'E' to represent exponent
`a`	ASCII string
`a`x	ASCII string, at most x characters
`g`x`.`y	general – treated as `f`x`.`y for numbers close to one, as `e`x`.`y otherwise, and as `i`x if used on an integer
`'any text'`	literal text

Complex numbers are printed by using two `e`, `f`, `g` or `d` descriptors.

Any of the above may be preceded by a repeat count.

Examples:

```
   complex c
   write(*,10) c
10 format(f10.4,' + ',f10.4,'i')

   integer i
   write(*,10)i,i*i,i**3
10 format(3i8)

   integer i
   write(*,10)'Number ',i,' your time is up.'
10 format(a,i3,a)
```

The `open` statement

An incomplete list of options to the `open` statement for reference.

`open(`*x*`, file=`*cc*`, status=`*stat*`, action=`*act*`)`

x	unit number. Integer expression.	
cc	file name. Character variable or constant.	
stat	one of:	`'OLD'` (file must exist)
		`'NEW'` (file must not exist)
		`'REPLACE'` (file will be overwritten)
	`status=`*stat* is optional.	
act	one of:	`'READ'` (writes will not be permitted)
		`'WRITE'` (reads will not be permitted)
		`'READWRITE'` (no restrictions)
	`action=`*act* is optional.	

Unit numbers 5 and 6 should not be used: 5 often corresponds to reading from the terminal (read(*,*)), 6 to writing to it (write(*,*)).

Simple examples:

```
open(10,file='output.dat')
open(50,file='more.data',status='REPLACE')
```

Complicated example (ignore if you wish):

```
   character(len=40) name
   integer i
   i=20

   write(name,10)i
10 format('output',i3.3,'.dat') ! OK for 0-999 except 5 and 6

   open(i,file=name,status='REPLACE')
```

(You have not formally been introduced to the `character` type, nor using `write` to fill it)

The `read` statement

The following shows how to read a file of unknown length, and introduces a few new features. The file is assumed to contain two columns of reals, and will be stored in the array `x`.

```
    integer i,len
    real dummy
    real, allocatable :: x(:,:)

    open(15,file='input.dat',status='OLD')
    len=0  ! Number of lines read succesfully
    do
      read(15,*,end=20,err=30) dummy
      len=len+1
    enddo

20  rewind(15)
    allocate (x(2,len))
    do i=1,len
      read(15,*,err=30) x(1,i),x(2,i)
    enddo
    close(15)
    write(*,*)len,' data elements read'
    stop

30  write(*,*)'I/O error occurred'
    end
```

The first loop reads in the file, looking for one data item on each line, which it discards by storing it in the variable `dummy`, which is overwritten on every cycle of the loop. The `end=` option of `read` tells the program to jump to label 20 when it reaches the end of the input file.

The `rewind` statement tells the program to go back to the beginning of the file for that unit number. Originally files would have been stored on tapes, hence the concept of rewind.

Now the number of lines is known, one can allocate an array to hold the data, and read in precisely the correct number of items. In the case that a line does not start with two items which can be interpreted as reals, an error will occur and the program will jump to label 30.

F90 Functions

An incomplete list of F90 functions for reference.

Notation: i – integer, r – real, d – double precision, c – complex, z – double precision complex.
Unless otherwise stated, functions return the same type and kind as their argument.

`abs(irdcz)`	Absolute value
`acos(rdcz)`	Arccosine
`aimag(cz)`	Imaginary part
`aint(rd)`	Truncate to integer
`anint(rd)`	Nearest integer
`asin(rdcz)`	Arcsine
`atan(rdcz)`	Arctangent
`atan2(rd,rd)`	`atan2(x,y)` is the arctangent of x/y
`ceiling(rd)`	Ceiling
`cmplx(ird,ird)`	$a + ib$ as type c
`cmplx(ird,ird,kind(1d0))`	$a + ib$ as type z
`conjg(cz)`	Complex conjugate
`cos(rdcz)`	Cosine
`cosh(rdcz)`	Hyperbolic cosine
`exp(rdcz)`	Exponential function
`floor(rd)`	Floor
`int(rd)`	Convert to integer type by truncation
`log(rdcz)`	Natural logarithm
`log10(rdcz)`	Base 10 logarithm
`mod(i,i)`	`mod(a,b)` is a modulo b
`nint(rd)`	Convert to nearest integer
`real(idcz)`	Convert to real (r)
`real(ircz,kind(1d0))`	Convert to real (d)
`sin(rdcz)`	Sine
`sinh(rdc)`	Hyperbolic sine
`sqrt(rdcz)`	Square root
`tan(rdcz)`	Tangent
`tanh(rdcz)`	Hyperbolic tangent

See also page 33 for functions taking arrays as arguments.

Reading F77

It is useful to be able to read Fortran77, and here are described those features of F77 which are best not used in F95, even though they are still valid syntax.

Fixed Form Source

The most striking and archaic feature of F77 is its fixed form source. A comment is indicated by the letter 'C' in the first column of a line, the first five columns may contain only statement labels, the sixth column contains the continuation mark (if column six does not contain a space, then this line should be considered a continuation of the previous one), and columns seven to seventy-two contain the 'real' statements. This made sense in a world of punched cards, but not today.

Conditionals

The conditional operators $<$, $<=$, $==$, $>=$, $>$ and `/=` are `.lt.`, `.le.`, `.eq.`, `.ge.`, `.gt.` and `.ne.` respectively. The characters $<$ and $>$ simply do not occur in F77.

Labels everywhere

We have used *statement labels* for `format` statements only. In F77 they were used more widely. Indeed, the only `do` loop was of the form

```
    do 10, i=1,20
      write(*,*)i,i*i
10  continue
```

The `continue` statement does nothing, except act as a placeholder for the label. The label specified immediately after the `do` is the last line within the body of the loop. Sometimes one will see the (bad) style of:

```
   do 15, i=1,10
     do 15, j=1,10
15     a(i,j)=0.0
```

F77 also makes use of the `goto` statement, which causes execution to jump immediately to the label given.

```
20  write(*,*)'Input a positive number'
    read(*,*) i
    if (i.lt.0) goto 20
```

The `goto` statement is considered ugly and untidy. It is almost always better to use a `do`, a multi-line `if` or a `select` statement instead.

double precision

Double precision variables could be declared as
`double precision x`
equivalent to the F90 declaration
`real (kind(1.d0)) ::  x`

Officially this was the only way of specifying double precision, and double precision complex simply did not exist! Most compilers accepted `real*8` as an alternate way of declaring double precision reals, and `complex*16` for the complex variety.

parameter statement

The declaration
```
integer i
parameter (i=5)
```
is equivalent to the F90 declaration
`integer, parameter ::  i=5`

common

The `common` statement allows specified variables to be shared by subroutines without being explicitly passed. It does nothing useful that modules do not do in a less error-prone fashion.

F95 Omissions

Many features of F95 have been intentionally omitted from this course. Time is limited, and the features covered should be quite sufficient for the purposes of this course. However, should you know of other features, you are free to use them.

So that a fair impression of the language is given, here is a list of the main omissions:

- Pointers
- User-defined data types
- Optional arguments to functions / subroutines
- `intent` attribute for arguments
- Private members of modules
- Array subobjects (automatic array slicing)
- Unformatted I/O (very necessary for large files)
- Operator overloading
- Recursion
- Bit manipulation functions
- String handling

F95 Syntax summary

A brief and rather incomplete summary of those bits of Fortran syntax covered in these notes. This is really intended for those already familiar with a programming language. If you have never programmed before, do not worry if these pages make no sense whatsoever.

Notation: *stmnt* one F90 statement, *stmnts* zero or more F90 statements, *expr* numeric expression, *const* expression which can be evaluated at compile time, *cond* condition (logical expression), [] enclose optional constructions.

General Structure

`module` *module_name*
`contains`
variable declarations
function and subroutine declarations
`end module`

`program`
use statements
`implicit none`
variable declarations
stmnts
`end`

There may be zero or multiple modules.

The `program` statement may carry a name, and the `end` may be written as
`end program program_name`

Names (of modules, functions and variables) can be up to 31 characters, case insensitive, first character alphabetic, others alphanumeric or underscore.

Variable declarations

`implicit none` should precede definitions!

Scalars

`real` *vars*
`complex` *vars*
`integer` *vars*
`real` [(`kind=`]*int*)] [`, parameter`] `::` *var*[=*const*][,*var...*]
`complex` [(`kind=`]*int*)] [`, parameter`] `::` *var*[=*const*][,*var...*]
`integer`[`, parameter`] `::` *var*[=*const*][,*var...*]

The first three are in the old F77 style, the last three in the F90 style. If `parameter` is specified, the variable should be initialised immediately by being defined as *var=const*.

Fixed size arrays

As above, but each variable specified as
var([*lower:*]*upper*[,[*lower:*]*upper,...*]*)*
Up to seven dimensions are permitted. If a lower bound for a dimension is omitted, one is assumed.

Allocatable arrays

As above, but the modifier `allocatable` must be present, and just a colon with no lower or upper bound is given for each dimension. E.g., for a 2D array:

```
integer, allocatable ::  grid(:,:)
```

Such an array can be the subject of an `allocate` statement, and, if allocated, of a `deallocate` statement.

Conditional Execution

If, short form

`if (`*cond*`) stmnt`

If, long form

`if (`*cond*`) then`
stmnts
[`else if (`*cond*`) then`
stmnts]
[`else`
stmnts]
`end if`

Any number of `else if` clauses may occur.

Select

`select case (`*expr*`)`
`case (`*const*[`:`*const*][`,`*const*]`)`
stmnts
[`case default`
stmnts]
`end select`

Where *const*`:`*const* denotes a range, and commas separate multiple single values or ranges. The range may be open-ended, such as '`:5`' for anything less than or equal to five.

Loops

`do var=`*expr1*,*expr2*[,*expr3*]
[*stmnts*]
[`if (` *cond* `) exit`]
[*stmnts*]
`enddo`

`var` must be defined as an integer variable, and will take the value *expr1* for the first iteration of the loop, and have *expr3* added to it at the end of each iteration. If *expr3* is omitted, it is taken to be one. The total number of iterations is `floor(1+(`*expr2*-*expr1*`)/`*expr3*`)`, or zero if this is negative. The expressions must be integer, and will be

evaluated just once on entry to the loop. No attempt may be made to change the value of `var` within the loop.

`do`
[*stmnts*]
`if (` *cond* `) exit`
[*stmnts*]
`enddo`

An infinite loop. Some form of exit is effectively required.

Bibliography

The following resources may be helpful, and were used in preparing this course.

FORTRAN 90/95 explained, Metcalf and Reid, Oxford Science Publications, £18.
Fortran 90 for Scientists and Engineers, Hahn, Arnold, £23.

Metcalf and Reid covers the language only, Hahn also includes some background on numerical methods. As neither is cheap, you may prefer to seek out library copies.

(Fortran 90 and Fortran 95 are almost indistinguishable: do not worry if a book mentions F90 only).

The course WWW site, found from `http://www.phy.cam.ac.uk/`, or, more directly, at `http://www.mrao.cam.ac.uk/teaching/comp_phy/`, has some pointers to other useful F90 resources too, as well as an errata for this document, should one be needed.

!, 17
**, 7
:, 31
::, 21
;, 22
=, 6, 11
==, 11
&, 17

algebra, 8
allocatable, 31
allocate, 31
ANSI, 3
antisocial, 75
argument, 8, 18
argument passing, 42
array
  bounds, 29, 61
  functions, 33
  operations, 32
arrays, 27–29, 73
  dynamic, 31
  static, 31
assignment, 6, 11

Bell Labs, 3

calculator, 68
call, 42
case, 36
case sensitivity, 7
close, 58
comments, 17, 65
common, 92
comparison, 11, 91
complex, 19, 23
complex, 23
conditions, 10
constants, 9, 18, 20
contains, 38
continuation line, 17
continue, 91
conversions, 24
core files, 64

dbx90, 62, 64, 85
deallocate, 31
debugger, 62, 85
debugging, 59–64, 84
declarations, 17, 18, 73

do, 12, 14, 16, 34, 91
double precision, 92

elements, 29
end, 5, 31, 38, 58
enddo, 12
endif, 10
Eratosthenes, 34, 74
examples, copying, 68
executable, 67
exit, 14
exponents, 20
expression, 18
external, 53

f95, 5
  -g90, 62, 85
  -gline, 84
  -lnag, 44, 80
factorials, 37, 38
factorisation, 15
file I/O, 58, 88, 89
format, 13, 70, 87
Fortran 77, 91
function, 8, 18
function, 38
functions, 39, 78
  list of, 90

goto, 92

IBM, 3
if, 10, 34
ifail, 81
implicit declarations, 25
implicit loops, 32
implicit none, 26, 65
indentation, 10, 65
indices, 29, 30
input, 8
integer, 19
integer, 6
integers, 7

kind, 21

label, 13, 89, 91
libraries, 43, 80
linker, 43
loops, 12, 14, 32

memory usage, 30

mod, 15
module, 38
modules, 39, 79
  variables in, 55

NAG compiler, 79
NAG library, 43–54
  documentation, 45
numerical integration, 56

omissions, 93
open, 58, 88
operator, 18

parameter, 9, 92
π, 35
portability, 2
precidence, 8
precision, 19, 21, 22, 72
print, 5
program, 38

random numbers, 35, 75
random_number, 35
random_seed, 76
range, 19, 20, 70
read, 8, 89
real, 19
real, 8, 15, 72
real*8, 92
rewind, 89
run-time checks, 61

scalar, 27
select case, 36
selected_real_kind, 21
SIGFPE, 84, 86
SIGSEGV, 84
speed, 71, 75
sqrt, 8, 15, 23, 71
stop, 10
stride, 34
subroutine, 41
subroutines, 42, 52, 79
subscripts, 29
supercomputers, 4

test, 53, 65, 84
time, 74
type conversions, 24

`ulimit`, 64
unit number, 58
`use`, 38

variable, 18
variables, 6, 7, 65
vectors, 27

`write`, 58, 88