Getting Started With Fortran 90

Tabish Qureshi

First Edition, August 2011

Center for Theoretical Physics
Jamia Millia Islamia
New Delhi - 110025.

Contents

1 Programming Languages and Fortran 2

2 Fortran Basics 3

3 Variables, types, and declarations 4

4 Expressions and assignment 5

5 Simple Input/Output 7

6 The `if` statements 8

7 Loops 8

8 Arrays and Matrices 10

9 Subprograms 12

10 Random numbers and Monte Carlo simulations 13

11 Some tit-bits 15

1 Programming Languages and Fortran

Programming Languages

We all know that computers are made of digital circuitary, where only two states are important, the high state, and the low state. That makes it possible to simulate binary numbers, where any number can be represented by a sequence of zeros and ones. High state of the digital circuits become 1, and low states 0. This way, a computer can store numbers and calculate anything it is asked to. However, there is one downside to this - all numbers and all instructions have to be supplied in the binary format, what we will loosely call, the machine language. Human mind is not used to binary instructions, but that is the only language the computers understand.

So, the solution people came up with, is as follows. Create a language which is easy to understand for human beings, what we will call, a *high-level language*. It is called high-level, because it doesn't go down to the level of the computer. In order for the computer to understand the instructions (program) written in this language, it has to be translated into the machine language, something which the computer understands. This process of translation to the machine language is called *compilation*. A special program which does it, is called a *compiler*. Once compiled, the computer understands the instructions written by human beings, and executes them.

There are several high-level programming languages, developed for specific purposes. C and C++ are general-purpose programming languages, COBOL is for business related tasks, BASIC, Pascal are again general purpose languages.

What is Fortran?

Fortran is a general purpose programming language, mainly intended for mathematical computations in science applications (e.g. physics). Fortran is an acronym for FORmula TRANslation. Fortran was the first high-level programming language. The work on Fortran started in the 1950's at IBM and there have been many versions since. By convention, a Fortran version is denoted by the last two digits of the year the standard was proposed. Thus we have Fortran 66, Fortran 77 and Fortran 90 (95). The most common Fortran version today is Fortran 90.

Users should be aware that most Fortran 90 compilers also allow Fortran 77, i.e. any Fortran 77 program is also a valid program in Fortran 90.

Why learn Fortran?

The most natural reason to learn Fortran is that it is the easiest programming language, and well-suited for scientific computation. Fortran is the dominant programming language used in scientific applications. It is therefore important for physics (or engineering) students to be able to read and modify Fortran code. A major advantage Fortran has is that it is standardized by ANSI (American National Standards Institute) and ISO (International Standards Organization). As a resultm if your program is written in ANSI Fortran 77 or ANSI Fortran 90, then it will run on any computer that has a Fortran 90 compiler. Thus, Fortran programs are portable across computer platforms

From time to time, so-called experts predict that Fortran will die out and soon become extinct. However, previous predictions of the downfall of Fortran have always been wrong. Fortran is the most enduring computer programming language in history. So let us learn to program in Fortran.

2 Fortran Basics

A Fortran program is just a sequence of lines of text. The text has to follow a certain *syntax* to be a valid Fortran program. We start by looking at a simple example where we calculate the area of a circle:

```
      program circle
      real r, area
c This program reads a real number r and prints
c the area of a circle with radius r.
      write(*,*) 'Give radius r:'
      read(*,*) r
      area = 3.14159*r*r
      write(*,*) 'Area = ', area
      stop
      end
```

The lines that begin with a "c" are *comments* and have no purpose other than to make the program more readable for humans. We type the program in a file called circle.f90, the ".f90" file-extension being the convention in Linux. Compiling and running the program in Linux typically looks like the following:

```
tabish@mutinao:~$ gfortran circle.f90
tabish@mutinao:~$ ./a.out
 Give radius r:
3
 Area =    28.274311
tabish@mutinao:~$
```

Program organization

A Fortran program generally consists of a main program and possibly several subprograms (or functions or subroutines). For now we will assume all the statements are in the main program; subprograms will be treated later. The structure of a main program is:

```
program name
non-executable statements (declarations)
statements
stop
end
```

All non-executable statements should come before the executable statements. In this tutorial, words that are in *italics* should not be taken as literal text, but rather as a generic description. The `stop` statement is optional and may seem superfluous since the program will stop when it reaches the end anyway but it is recommended to always terminate a program with the `stop` statement to emphasize that the execution flow stops there.

Column position rules

Fortran 77 is *not* a free-format language, but has a very strict set of rules for how the source code should be formatted. The most important rules are the column position rules:

```
Col.  1    :  Blank, or a "c" or "*" for comments
Col.  2-5  :  Statement label (optional)
Col.  6    :  Continuation of previous line (optional)
```

```
Col.  7-72 :  Statements
Col.  73-80:  Sequence number (optional, rarely used today)
```

Most lines in a Fortran 77 program starts with 6 blanks and ends before column 72, i.e. only the statement field is used.
Fortran 90 allows free format, so you may ignore the above restrictions completely, if you name your file something.f90 and are using *gfortran*, the GNU Fortran 90 compiler.

Apart from putting a "c" in the first column, there is another way of adding comments to your program. Any text after an exclamation mark (!) is considered a comment. The exclamation mark may appear anywhere on a line (except in positions 2-6). Comments may appear anywhere in the program. Well-written comments help you read your program better at a later stage.

Continuation

Occasionally, a statement does not fit into one single line. One can then break the statement into two or more lines, and use the continuation mark in position 6. Example:

```
c23456789 (This is a comment to just indicate column positions)
       area = 3.14159265358979
     & *r*r
```

Any character can be used instead of the plus sign as a continuation character. It is considered good programming style to use either the plus sign, an ampersand, or numbers (2 for the second line, 3 for the third, and so on).

3 Variables, types, and declarations

Variable names

Variable names in Fortran consist of characters chosen from the letters a-z and the digits 0-9. The first character must be a letter. (Note: Fortran 77 only allows variable names of length 1-6).

Types and declarations

Every variable *should* be defined in a *declaration*. This establishes the *type* of the variable. The most common declarations are:

```
integer   list of variables
real      list of variables
double precision  list of variables
complex   list of variables
logical   list of variables
character list of variables
```

The list of variables should consist of variable names separated by commas. Each variable should be declared exactly once. If a variable is undeclared, Fortran 77 uses a set of *implicit rules* to establish the type. This means all **variables starting with the letters `i-n` are integers and all others are real**. Most compilers still allow these implicit rules, but *you should not!* If you do not consistently declare your variables, you in danger of making errors which are hard to debug.

Integers and floating point variables

Fortran has only one type for integer variables. Integers are usually stored as 32 bits (4 bytes) variables. Fortran

has two different types for floating point variables, called `real` and `double precision`. While `real` is often adequate, some numerical calculations need very high precision and `double precision` should be used. Usually a real is a 4 byte variable and the double precision is 8 bytes, but this is machine dependent. Some non-standard Fortran versions use the syntax `real*8` to denote 8 byte floating point variables.

4 Expressions and assignment

Constants

The simplest form of an expression is a *constant*. There are 6 types of constants, corresponding to the 6 data types. Here are some integer constants:

```
3
0
-105
+15
```

Then we have real constants:

```
3.0
-0.25
-2.7E6
3.73E-2
```

The E-notation will be clear from the following: `-2.7E6` means -2.7×10^6, and `3.33E-2` means 0.0373.

Assignment

The assignment has the form
variable_name = expression
or, for example

```
a = b
```

The above statement should not be read as *a is equal to b*. Rather, it should be interpreted as follows: Evaluate the right hand side and assign the resulting value to the variable on the left. With this meaning, the following two statements will make perfect sense

```
a = 5
a = a + 1
```

The variable a takes the value 5, and then 1 is added to that value, to make the value of a as 6. If you don't take this interpretation, you will cancel a from both sides, and will get a nonsensical answer!

Expressions

The simplest expressions are of the form

operand operator operand

and an example is

```
x + y
```

The result of an expression is itself an operand, hence we can nest expressions together like

```
x + 2 * y
```

This raises the question of precedence: Does the last expression mean `x + (2*y)` or `(x+2)*y`? The precedence of arithmetic operators in Fortran 77 are (from highest to lowest):

```
**    {exponentiation} - x**3 means x^3
*,/   {multiplication, division}
+,-   {addition, subtraction}
```

All these operators are calculated left-to-right, except the exponentiation operator **, which has right-to-left precedence. If you want to change the default evaluation order, you can use parentheses.

The above operators are all binary operators. there is also the unary operator - for negation, which takes precedence over the others. Hence an expression like `-x+y` means what you would expect.

Extreme caution must be taken when using the division operator, which has a quite different meaning for integers and reals. If the operands are both integers, an integer division is performed, otherwise a real arithmetic division is performed. For example, 3/2 equals 1, while 3./2. equals 1.5.

Built-in functions

There are many built-in functions in Fortran. Some of the most common are:

```
abs     absolute value
min     minimum value
max     maximum value
sqrt    square root
sin     sine
cos     cosine
tan     tangent
atan    arctangent
exp     exponential (natural)
log     logarithm (natural)
```

For example `y = log(3.*x**2)` will be a valid Fortran statement.

Logical expressions

Logical expressions can only have the value `.TRUE.` or `.FALSE.`. A logical expression can be formed by comparing arithmetic expressions using the following *relational operators*:

Operator (Fortran 77)	Fortran 90	Meaning
.LT.	<	less than ($<$)
.LE.	<=	less than or equal ($\leq$)
.GT.	>	greater than ($>$)
.GE.	>=	greater than or equal ($\geq$)
.EQ.	==	equal ($=$)
.NE.	/=	not equal ($\neq$)

These operators are generally used inside a "conditional statement". Logical expressions can be combined by the *logical operators* `.AND.` `.OR.` `.NOT.` which have the obvious meaning.

Logical variables and assignment

Truth values can be stored in *logical variables*. The assignment is analogous to the arithmetic assignment. Example:

```
logical a, b

a = .TRUE.

b = a .AND. 3 .LT. 5/2
```

The order of precedence is important, as the last example shows. The rule is that arithmetic expressions are evaluated first, then relational operators, and finally logical operators. Hence b will be assigned `.FALSE.` in the example above.

Logical variables are seldom used in Fortran. But logical expressions are frequently used in conditional statements like the `if` statement.

5 Simple Input/Output

An important part of any computer program is to handle input and output. For example, a program should be able to read data from the keyboard, or from a file. And it should be able to print the result to the computer screen, or to a file. In our examples so far, we have already used the two most common Fortran constructs for this: `read` and `write`. Fortran I/O can be quite complicated, so we will only describe simple cases enough for most purposes.

Read and write

`Read` is used for input, while `write` is used for output. A simple form is

```
read (unit no, format no) list-of-variables
write(unit no, format no) list-of-variables
```

The unit number can refer to either standard input, standard output, or a file. This will be described in later section. The format number refers to a label for a format statement, which you may not bother about, to start with.

It is possible to simplify these statements further by using asterisks (*) for some arguments, like we have done in all our examples so far. This is sometimes called *list directed* read/write.

```
read (*,*) list-of-variables
write(*,*) list-of-variables
```

The first statement will read values from the standard input and assign the values to the variables in the variable list, while the second one writes to the standard output, which is the screen in most cases.

Example of I/O

```
integer m, n
real x, y
read(*,*) m, n
x = 1.0*(m+n)/(m*n)
y = 1.0*(m-n)/(m*n)
write(2,*) x, y
stop
end
```

In the example on the right, the program when run, will wait for the user to give the values of `m` and `n` from the keyboard, either separated by comma or by a space. The resulting values of `x, y` will be printed in a file `fort.2`, a name which comes from the value "2" of the unit-number. If we had used the value "10" instead of 2, the result would have been written in a file `fort.10`.

6 The `if` statements

An important part of any programming language are the *conditional statements*. The most common such statement in Fortran is the `if` statement, which actually has several forms. The simplest one is the logical `if` statement:

```
if (logical expression) executable statement
```

This has to be written on one line. This example finds the absolute value of x:

```
if (x < 0) x = -x
```

If more than one statement should be executed inside the `if`, then the following syntax should be used:

```
if (logical expression) then
   statements
endif
```

The most general form of the `if` statement has the following form:

```
if (logical expression) then
   statements
elseif (logical expression) then
   statements
 :
 :
else
   statements
endif
```

The execution flow is from top to bottom. *The conditional expressions are evaluated in sequence until one is found to be true.* Then the associated code is executed and the control jumps to the next statement after the `endif`.

Example of if statement

```
real x, r
r = 3.141592654
read(*,*) x, y
if (sqrt(x**2 + y**2) > r) then
   write(*,*) 'yes'
else
   write(*,*) 'no'
endif
stop
end
```

Example of nested if statement

```
real x, y
read(*,*) x, y
if (x > 0) then
   if (x >= y) then
      write(*,*) 'x is +ve and x >= y'
   else
      write(*,*) 'x is +ve but x < y'
   endif
elseif (x < 0) then
   write(*,*) 'x is negative'
else
   write(*,*) 'x is zero'
endif
stop
end
```

Nested `if` statements

`if` statements can be nested in several levels. To ensure readability, it is important to use proper indentation. You should avoid nesting many levels of `if` statements since things get hard to follow.

7 Loops

For repeated execution of similar group of statements, *loops* are used. If you are familiar with other programming languages you have probably heard about *for*-loops, *while*-loops, and *until*-loops. Fortran has one loop con-

struct, called the do-loop, which is most commonly used. The do-loop corresponds to what is known as a *for*-loop in other languages. Other loop constructs have to be simulated using the `if` and `goto` statements.

do-loops

The do-loop is used for simple counting. Here is a simple example that prints the cumulative sums of the integers from 1 through n (assume n has been assigned a value elsewhere):

```
integer i, n, sum
sum = 0
do i = 1, n
   sum = sum + i
   write(*,*) 'i = ', i
   write(*,*) 'sum = ', sum
enddo
```

Factorial of a number using do loop

```
integer n, fac
read(*,*) n
fac = 1
do i = 1, n
   fac = fac*i
enddo
write(*,*) n, '! is ', fac
stop
end
```

The variable defined in the do-statement is incremented by 1 by default. However, you can define any other integer to be the *step*. This program segment prints the even numbers between 1 and 10 in decreasing order:

```
integer i
do i = 10, 1, -2
   write(*,*) 'i = ', i
enddo
```

The general form of the do loop is as follows:

```
do var = start, end, step
   statements
enddo
```

var is the loop variable (often called the *loop index*) which must be integer. *start* specifies the initial value of *var*, *end* is the terminating bound, and *step* is the increment. All the three of these need not always be variables and constants, they can also be expressions.

Note: The do-loop variable must never be changed by other statements within the loop! This will cause great confusion.

In how many steps does a random-walker travel a distance 20

```
integer n, step, L, seed
L = 0
n = 0
read(*,*) seed  ! +ve integer
call srand(seed) ! random no. setup
do while(.true.)
   if (rand(0) > 0.5) then
      step = +1
   else
      step = -1
   endif
   L = L + step
   n = n + 1
   if (iabs(L) == 20) exit
enddo
write(*,*) n, L
stop
end
```

do while loop

There is another kind of loop in Fortran 90, the `do while` loop. The syntax of the loop is as follows

```
do while (logical expression)
   executable statements
enddo
```

Here the executable statements will be continually repeated in sequence, as long as the logical expression is true. This

can be used in situations where you want to continue repeating some procedure until a particular condition is satisfied. For example, the following program

```
integer i
i = 0
do while(i < n)
   i = i + 1
   write(*,*) 'i = ', i
enddo
stop
end
```

will work exactly like a do loop with `do i=1,n`.

8 Arrays and Matrices

There are many computational problems in which one needs to use subscripted variables, like vectors and matrices. The variable type (called *data type*) Fortran uses for representing such objects is called *array*. A one-dimensional array is like a vector, while a two-dimensional can be thought of as a matrix. For example a 1-d array of size n can store n numbers at a time, and a 2-d array of size $m \times n$ can store $m \times n$ numbers at a time. Like normal variables, arrays are also real, integer, double precision, or even logical.

One-dimensional arrays

The simplest array is the one-dimensional array, which is just a linear sequence of numbers stored consecutively in memory. All arrays, and their size, have to be declared in the beginning of the program. For example, the declaration

```
real a(10)
```

declares a as a real array of length 10. By convention, Fortran arrays are indexed starting from 1. Thus the first element of the array is denoted by `a(1)` and the last by `a(10)`. However, you may define an arbitrary index range for your arrays using the following syntax:

```
real b(0:9), gamma(-100:100)
```

Here, b is exactly similar to a from the previous example, except the index runs from 0 through 9. gamma is an array of length 201.

Each element of an array can be thought of as a separate variable, and can be used as such. You reference the i'th element of array a by `a(i)`. Here is a code segment that stores the 10 first square numbers in the array sq:

```
integer i, sq(10)
do i = 1, 10
   sq(i) = i**2
enddo
```

A common bug in Fortran programs is that the program tries to access array elements that are out of bounds or undefined. This

Create a normalized random vector

```
real a(100), norm, scale
integer n
n = 100
a(1) = rand(1) ! initialize..
norm = 0.0
do i = 1, n
   a(i) = rand(0)
   norm = norm + a(i)*a(i)
enddo
scale = 1./sqrt(norm)
  do i = 1, n
   a(i) = scale*a(i)
enddo
stop
end
```

is the responsibility of the programmer, and the Fortran compiler will not detect any such bugs!

Matrices: 2-dimensional arrays

Matrices are very important in linear algebra. Matrices are usually represented by two-dimensional arrays. For example, the declaration

```
real A(3,5)
```

defines a two-dimensional array of 3*5=15 real numbers. It is useful to think of the first index as the row index, and the second as the column index. Hence we get the graphical picture:

```
A(1,1)  A(1,2)  A(1,3)  A(1,4)  A(1,5)

A(2,1)  A(2,2)  A(2,3)  A(2,4)  A(2,5)

A(3,1)  A(3,2)  A(3,3)  A(3,4)  A(3,5)
```

The row and column indices of matrices start from 1 by default, but like 1-d arrays, they can also be defined to start from 0 or a negative value. For example

```
integer spin(-10:10, -20:20)
```

is a valid 21×41 matrix.

Multiplying two matrices

```
integer n, i, j, k
real a(3,3), b(3,3), c(3,3)
n = 3
read(*,*) a
read(*,*) b
do i = 1, n
   do j = 1, n
      c(i,j) = 0.0
      do k = 1, n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
   enddo
enddo
write(*,*) c
stop
end
```

It is quite common in Fortran to declare arrays that are larger than the matrix we want to store. This is perfectly legal. Do not assume that all these elements of a defined matrix are initialized to zero by the compiler (some compilers will do this, but not all). So, you have to initialize all elements of an array to zero, if the program is such that it will try to *use* some elements before assigning values to them.

There is an alternate way to declare arrays in Fortran. The statements

```
real A, x
dimension x(50), A(10,20)}
```

are equivalent to

```
real A(10,20), x(50)
```

This `dimension` statement is considered old-fashioned style today.

Multi-dimensional arrays

Fortran allows arrays of higher dimensions too. The syntax and storage format are analogous to the two-dimensional case, so we will not spend time on this.

9 Subprograms

When a programs is more than a few hundred lines long, it gets hard to follow. Fortran codes that solve real engineering problems often have tens of thousands of lines. The only way to handle such big codes, is to use a *modular* approach and split the program into many separate smaller units called *subprograms*.

A subprogram is a (small) piece of code that solves a well defined subproblem. In a large program, one often has to solve the same subproblems with many different data. Instead of replicating code, these tasks should be solved by subprograms . The same subprogram can be invoked many times with different input data.

Fortran has two different types of subprograms, called *functions* and *subroutines*.

Functions

Fortran functions are quite similar to mathematical functions: They both take a set of input arguments (parameters) and return a value of some type.

A simple example illustrates how to use a built-in function:

```
x = cos(pi/3.0)
```

Here `cos` is the cosine function, so `x` will be assigned the value 0.5 (if `pi` has been correctly defined; by the way, a quick way to define pi is `pi = 355./113.`).

In general, a function always has a *type*, i.e., real, integer. double precision etc. For example, if x is a double precision variable, the previous example should read as

```
x = dcos(pi/3.0)
```

where `dcos` is the double-precision version of the cosine function.

Quite often one needs to define one's own function, depending on the problem one is dealing with. For example, if x and y are the coordinates of the position of a moving particle, one might need to know the distance of the particle from the origin every now and then. Instead of typing the formula at every place, one can define the following function:

```
real function r(x,y)
real x, y
r = sqrt(x*x+y*y)
return
end
```

We see that the structure of a function closely resembles that of the main program. The main differences are:

1. Functions have a type. This type must also be declared in the calling program.
2. The return value should be stored in a variable with the same name as the function.
3. Functions are terminated by the *return* statement instead of *stop*.

To sum up, the general syntax of a Fortran function is:

```
type function name (list-of-variables)
declarations
statements
return
end
```

The function has to be declared with the correct type in the calling program unit. The function is then called by simply using the function name and listing the parameters in parenthesis.

Important notes
1. Variables used *inside* a function remain inside, including the variables that take values passed by the main program - the main program does'nt know about those variables.
2. Vice-versa also holds true. For example, if `pi` has been defined in the main program, a function cannot use it. Either it is redefined inside the function, or the main program passes it to the function through one of the arguments of the function.
3. A function can be called by just giving constants, instead of variables. For example the function `r(x,y)` described above, could be called from the main program by the statement `dist = r(3.2, 7.5)`.
4. A function can call another function from inside.

Subroutines

A Fortran function can essentially only return one value. Often we want to return two or more values (or sometimes none!). For this purpose we use the `subroutine` construct. The syntax is as follows:

```
subroutine name (list-of-arguments)
declarations
statements
return
end
```

Note that subroutines have no type and consequently should not (cannot) be declared in the calling program unit. We give an example of a very simple subroutine. The purpose of the subroutine is to swap two integers.

```
subroutine swap (a, b)
integer a, b, tmp
tmp = a
a = b
b = tmp
return
end
```

Note that, unlike the case of function, the variables passed from the main program, are *not* local. Their values will be passed back to the main program. You have to be careful about this when writing Fortran code, because it is easy to introduce undesired *side effects*. For example, sometimes you may mistakenly treat an input parameter in a subprogram as a local variable and change its value. You should *never* do this since the new value will then propagate back to the calling program with an unexpected value!

10 Random numbers and Monte Carlo simulations

Very often in computer programs, we need to simulate a coin-toss, or throw of a dice. For this, we need random number. For example, if we had a function which randomly returned either value +1 or -1, we could take it as an ideal coin-toss. Unfortunately, it turns out that computers cannot generate true random numbers. What they can generate from various complicated algorithms, are *pseudorandom number*, i.e., numbers which *appear to be* random. Most Fortran compilers have a built-in function which returns a random value, uniformly distributed in the interval [0,1), i.e. including 0, but not 1. In GNU Fortran 90 compiler, this function is called `rand()`.

In order to generate a random number, the function should be called with an argument 0, for example

```
x = rand(0)
```

assigns a random value to x. However, in order to use the random number effectively, it has to be *initialized* with an integer *seed* number. In GNU Fortran 90 compiler, the seeding is done by calling the subroutine `srand(seed)`, once before starting the calls to `rand(0)`. As you can guess, `srand` stands for seed random number generator.

One can look at the example program for a random-walker, in order to understand the use of `rand(0)`.

Note: If you rerun the program with the same seed, it will generate exactly the same sequence of random numbers! In order to generate a new sequence everytime, you should use a different value of the integer seed evertime.

Some people use a clever method of generating a new set of random numbers on every run, by using the built-in function `secnds(x)`, which returns the number of seconds (minus x) since midnight. This value is assigned to the seed.

The random number generator function only gives numbers in the interval [0,1). Sometimes we want random numbers in a different interval, e.g. [-1,1). A simple transformation can be used to change intervals. For example, if we want a random number (x) in the interval [a,b) we can do so using:

```
x=(b-a)*rand(0)-a
```

Thus for the interval [-1,1) we get: `x=2*rand(0)-1`.

In fact, we can take our set of numbers from the `rand(0)` function which have a uniform probability distribution in the interval [0,1) and turn them into a set of numbers that look like they come from just about any probability distribution with any interval that one can imagine!

A few examples:

```
dice=int(1+6*rand(0))     !  This generates the roll of a 6 sided die.
```

```
g=sqrt(-2*log(rand(0)))*cos(2*pi*rand(0))
```

This generates a random number g from a gaussian distribution with mean=0 and variance=1. We assume that `pi` is already initialized to 3.14159 in the program.

```
t= -a*log(rand(0))
```

This generates a random number from an exponential distribution with a decay time = a.

Being able to transform the random numbers obtained from ran(seed) into any probability distribution function we want is extremely useful and forms the basis of all computer simulations.

Flipping a spin with a probability

```
integer S(100,100)
:
prob = exp(-DelE/kT)
if (rand(0) < prob) then
   S(i,j) = -S(i,j)
endif
:
:
stop
end
```

Monte-Carlo simulation

In some types of computer simulation, one sometimes wants to do a particular operation *with a probability*. For example, in the simulation of Ising Model in physics, one may want to "flip" a spin with some probability. The way to do that is illustrated in the program on the right. The probability with which one wants to flip a spin, is calculated with the formula `prob = exp(-DelE/kT)`. Then a random number is generated and compared with `prob`. One can see that if `prob = 1`, then the random number will always be less than it, and the spin will be flipped with probability 1. If `prob = 0.5`, 50% of the time the random number will be greater than 0.5, and 50% of the time it will be less than that. So, spin will be flipped with probability 0.5.

This type of simulation often goes by the name "Monte Carlo". Why Monte Carlo? In the pre-computer era a popular way to obtain a set of random numbers was to use a roulette wheel, just the type found in the Mediterranean city of Monte Carlo, famous for its gambling casino.

11 Some tit-bits

The `parameter` statement

Some constants appear many times in a program. It is then often desirable to define them only once, in the beginning of the program. This is what the `parameter` statement is for. It also makes programs more readable. Moreover, if you are defining the dimensions of many arrays in the main program, you cannot use a variable for that:

```
n = 100
real a(n,n), b(n,n), c(n,n), vec(n)      ! This is WRONG
:
```

This is wrong because `n = 100` is an executable statement and the declaration of arrays is a non-executable statement, which should come *before* any executable statement. The right way to do it is

```
parameter(n=100, pi=3.14159)
real a(n,n), b(n,n), c(n,n), vec(n)      ! This is correct
:
```

Remember that the"variable" defined in the `parameter` statement is not a variable but rather a constant whose value can never change.

The `exit` and `cycle` statements

The `exit` statement can be used to come out of a loop, if a condition is satisfied. See the random-walker example.

The `cycle` statement can be used to skip the rest of the statements in a loop cycle, and go to the next value of the loop index.

The `continue` and `goto` statements

The `continue` statement does absolutely nothing. It is mostly used to specify a location in a program, together with a statement label, like
`10 continue`

One may occasionally want to use the `goto` statement to break the flow of the program, and make it go to a particular location, specified by a labelled `continue` statement.

The use of `goto` statement is considered a very clumsy programming practice, and should be avoided. It makes the logic of the program hard to follow.

goto and continue statements

```
      :
      if (abs(a) < epsil) goto 15
      :
      :
15    continue
      :
      :
```

Reading input from files, writing output to files

Although normally the reading/writing of files is handled via the `open` and `close` statements in Fortran, there is a quick and dirty way to do it without those statements.

If you are satisfied that running a program is generating the right data in the output, you can use the following trick (in Linux) to redirect the out to a data file. The following way of running the program

```
./a.out > energy.dat
```

will redirect the output of the program, which was supposed to be displayed on the screen, to the file `energy.dat`. Same way, sometimes you want some data to be read from a file, instead of giving it from the keyboard, like a 3×3 matrix. This can be done by a command like this

```
./a.out < matrix.dat
```

Editing, compiling and running Fortran 90 programs

Although Fortran programs can be written in any text editor (not a word-processor!), it is better to do it in an editor which is made for programming. One example of such an editor is `geany`. Install `geany` in Linux, and use it to type your program. Once you have saved the file with the extension `.f90`, it knows that it is a Fortran program and will color various Fortran statements to make it more readable (this is called syntax-highlighting). You can also compile and run your programs from inside `geany`.

Normally programs are compiled using the command

```
gfortran myprog.f90
```

which generates an executable file `a.out`, which can be executed by the command

```
./a.out
```

However, you can generate an executable file by another name, e.g.

```
gfortran myprog.f90 -o myprog
```

which will generate an executable file `myprog`, which can then be executed.