A Simple Fortran Primer

2nd Edition

Rosemary Mardling and Daniel Price

School of Physics & Astronomy

Monash University

2016

Contents

1 Introduction

The aim of this primer is to get you fearlessly writing simple programs in Fortran. As with any language, the vocabulary and grammar (called *syntax* in a computer language) seem vast at first, but with practice soon become manageable. We have followed the philosophy that learning by example is an efficient way of learning. We have also approached problems from a mathematical point of view rather than a computer science point of view. After all, the language name Fortran comes from FORmula TRANslation.

Each section consists of a series of Examples, each of which is set out as follows:

- A mathematical problem is stated.
- A sample program is given which solves the problem and introduces some new elements of Fortran.
- Each new Fortran element is pointed out and discussed.

At the end of each section, a series of Exercises is given for which the student must write her/his own program. Attempting these Exercises will consolidate what has been learned so far.

Since this primer covers only the basics of Fortran programming, we often suggest reference to a Fortran manual. The best reference is the book by Metcalf, Cohen & Reid:

Modern Fortran Explained
Metcalf, M., Cohen, J., Reid, M. (Oxford University Press)
(The 3rd edition is called "Fortran 90/2003 explained")

in which most Fortran statements are defined and examples are given. It is our hope that after having worked through this primer, the student will feel confident consulting such a manual.

The current version of of Fortran is simply called 'Fortran' which is essentially "Fortran 90" with some minor updates to the standard in 2003, 2008 and 2015. Modern Fortran is backwards compatible with older languages called "FORTRAN 77", "FORTRAN 66" (66 $\equiv$ 1966, 77 $\equiv$ 1977). These older standards are indicated by filenames ending in `.f`, while filenames in modern Fortran end in `.f90`. Older Fortran is referred to as FORTRAN (in SHOUT CASE) and required some quirks like indenting all code by 6 characters along with other features which are regarded as 'deprecated' and should no longer be used.

Finally, there is no one "right" way to write a Fortran program. The examples presented in this primer indicate the programming style of the author, but you may find after some practice that you prefer to organise and present your programs differently.

2 Simple Mathematical Expressions

Try as much as possible to make your program "look" like your mathematics.

Example 2.1

Write a program to evaluate the following expression for $x = -3$, printing out the answer on the screen;

$$y = 2x + 3,$$

```
1  program myprog
2
3   x = -3.0
4   y = 2.0*x + 3.0
5   print*,y
6
7  end program
```

Things to note:

1. Each expression begins on a new line. Each line is called a *statement*.
2. The indentation of each line is optional in modern Fortran (i.e. if the filename ends in `.f90`).
3. *You must tell the computer the value of every variable*, either directly as in `x=-3.0` or indirectly, as in `y=2.0*x+3.0`.
4. The numbers are written with a decimal point. Although this is not necessary here, it is good practice to do this because Fortran distinguishes between `real` and `integer` variables (and also `complex` variables, but we won't look at them in this primer). We will look at this in more detail in Section 3.
5. In Fortran, multiplication is performed by the `*` symbol.
6. The simplest way to see the value of a variable on the computer is to use the `print*,` statement. The `*` means the computer can print out the answer the way it wants (this is called *free format*) instead of how you might like it.
7. Fortran doesn't distinguish between upper and lower case letters.
8. Note the last line: it is the `end` statement; every Fortran program (including `functions` and `subroutines`) must end with this statement.

Example 2.2

Evaluate

$$z = \frac{3\alpha - 4\beta}{2\alpha}$$

for $\alpha = 1$ and $\beta = 2$.

```
program myprog

 alpha = 1.0
 beta = 2.0
 z = (3.0*alpha - 4.0*beta)/(2.0*alpha)
 print*,' z = ',z

end program myprog
```

Things to note:

1. Variable names can be of any length, should start with a letter of the alphabet, but can contain numbers (and the underscore character _). Although we could call the variables anything we like, it is good practice to try and mimick the original mathematical expression. Of course, `a` and `b` would have done just as well.

2. In Fortran, division is performed by the / symbol (don't get this mixed up with the backslash symbol \).

3. Note the placement of the brackets. We would not get the correct answer if we didn't bracket the *whole* denominator, i.e. if we put `z=(3.0*alpha-4.0*beta)/2.0*alpha`, or worse still, `z=3.0*alpha-4.0*beta/2.0*alpha`.

4. If you try this exercise, you will see that the computer prints out `z=-2.500000`, whereas in Example 2.1, only the number is printed out. You can get the computer to print out things which make it easier for you to read the answers (such as the `z=` part in this example) by putting such things between single quotes. Note the comma between `'z='` and `z` in the `print*,` statement.

5. See what happens when you run the program if you replace `alpha=1.0` by `alpha=0.0` - the computer does not like to divide by zero!

Example 2.3

Evaluate the following expression for several values of x and a:

$$y = \log\left(x + \sqrt{x^2 + a^2}\right)$$

```
program log

 print*,'x,a?'
 read*,x,a
 y = log(x + sqrt(x**2 + a**2))
 print*,'y',y

end program log
```

Things to note:

1. You can read in data by typing it in directly using the `read*,` statement. When you run the program, it will pause at this statement until you have typed in as much data as it expects separated by commas, spaces or <*return*> (in this example, it will wait for 2 numbers).
2. The statement `print*,'x,a?'` is not actually necessary for the running of the program, but it is used as a "prompt" so that we know the program is waiting for values of `x`and `a`.
3. In Fortran, you can raise a variable to a power using `**`. For example, $x^{1/2}$ is written `x**0.5`.
4. The two "Fortran intrinsic functions" `log(...)` and `sqrt(...)` have been used here. The argument of these functions must be enclosed in brackets. Other library functions may be found in the appendix.

Example 2.4

Generate the first 4 numbers in the sequence defined as follows:

$$n_{i+1} = 4 - n_i, \quad n_1 = 1.$$

Ans: $\{1, 3, 1, 3\}$

```
program seq

 n = 1

 n = 4-n
 print*,n

 n = 4-n
 print*,n

 n = 4-n
 print*,n

end program seq
```

Things to note:

1. The symbol `'='` means *replace* or *assign*, *ie.* in this case, the *old* value for `n` is replaced by the *new* value for `n`. Each variable has a space assigned to it in the memory of the computer, and this is filled with the present value of the variable.

2. You can see how cumbersome it is to repeat the statements `n=n-4` and `print*,n` several times. We will address this when we look at *do loops* in Section 4.

Exercises

Evaluate the following expressions by writing a Fortran program for each, printing out the answers on the screen.

1. $y = s^2 + r^3$, where $s = \cos x$, $r = e^x$ and $x = 0.5$. *Ans:* $y = 5.251840$

2. $y = \tanh(f(z))$, where $f(z) = \log\sqrt{\frac{1+z}{1-z}}$, and $z = 1/2$. *Ans:* $y = 1/2$

3. $y = \cos^2\theta - \sin^2\theta$, where $\theta = \pi/2$. *Ans:* $y = -1$

4. $y = \mathrm{Sin}^{-1}x + \mathrm{Cos}^{-1}x$, for any value of x. *Ans:* $y = \pi/2$

Notes to Exercises

Q. 2: You will not get the correct answer if you put `z=1/2`; you must put either `z=0.5` or `z=1.0/2.0`. See the following section.

Q. 3: You must tell the computer the value of π. For instance you can say `pi=3.141593` or `pi=4.0*atan(1.0)`. Of course you needn't call π `pi`, but if you do, you will know immediately what this variable stands for.

3 Real and Integer Variables; Vectors and Subscripted Variables

3.1 Data Types

As mentioned earlier, Fortran distinguishes between *real* and *integer* variables[1], and we have three options to follow:

1. We can let the computer follow the Fortran convention of assuming all variables which start with the letters `i,j,k,l,m,n` are *integer* type variables, while all others are real, and risk forgetting about this convention (we'll see shortly what happens if you do this);
2. We can partly follow this convention making sure that the data types of any variables which don't follow the convention are *declared* at the top of the program; forgetting will incur an error message when the program is compiled, or
3. We can put the statement `implicit none` at the *top* of the program and declare the data type of *all* the variables appearing in the program. If we forget to declare any, we will get an error message when we compile the program.

[1] As well as *complex* variables, *double precision* variables, *logical* variables and *character* variables; see the Fortran manual for more information.

Example 3.1

Here is an example of the first option where the programmer has forgotten two things:

```
1  program vars
2
3   noddy = 3.7
4   bigears = -2.8
5
6   m = 2
7   n = 3
8
9   golly = noddy + bigears
10  j = m/n
11
12  print*,'golly=',golly
13  print*,'j = ',j
14
15 end program vars
```

When you try running this program, you might expect to get the answers `golly=0.9` and `j=0.666667`; instead you will find the computer comes up with `golly=0.2` and `j=0`. In the first case, the programmer has forgotten to declare the variable `noddy` as *real*, so the computer assumes it is of type *integer* and will round it *down* to the nearest integer (no matter what the decimal part of the number is). The second mistake was to forget that integer division only gives the correct answer when the denominator is a factor of the numerator. Otherwise it again rounds *down*[2].

• Note the blank lines in this program; while the computer ignores them, they help to make the program easier to read.

[2]Integer division can be used to obtain the integer part of a number.

Example 3.2

Now we will repeat Example 3.2 using the `implicit none` statement at the top of the program.

```
1  program vars
2   implicit none
3   real :: noddy,bigears,golly,j
4   integer :: m,n
5
6   noddy = 3.7
7   bigears = -2.8
8
9   m = 2
10  n = 3
11
12  golly = noddy + bigears
13  j = m/n
14
15  print*,'golly=',golly
16  print*,'j = ',j
17
18 end program vars
```

Although the variable `n` is an integer, we can obtain the correct answer for the division if we temporarily take `n` to be `real` by replacing `j=m/n` by `j=m/real(n)`. Similarly, if we wanted to temporarily treat a real variable as integer, or if we wanted to take the integer part of a real number, we would use `int(...)` as in `y=int(x)`.

- **From now on we will use `implicit none` at the top of the program and declare the data type of all our variables.**
- **Statements which declare the data type of variables go at the top of the program** *before* **the** *executable statements***. Executable statements such as `x=3.1` are statements which get the computer to do something.**

3.2 Arrays

Arrays are used for subscripted variables such as the components of a vector or of a matrix.

Example 3.3

Evaluate $\mathbf{u} \cdot \mathbf{v}$, where $\mathbf{u} = 3\mathbf{i} + 4\mathbf{j}$ and $\mathbf{v} = -\mathbf{i} + 2\mathbf{j}$.

```
1  program arrays
2   implicit none
3   real :: u(2), v(2), y, z
4
5   u(1) = 3.0
6   u(2) = 4.0
7
8   ! this is a comment
9   v = (/-1.0, 2.0/)
10
11  y = u(1)*v(1) + u(2)*v(2)
12
13  print*,'u.v = ',y
14
15  z = dot_product(u,v)
16  print*,'u.v = ',z
17
18 end program arrays
```

Things to note:

1. The second *statement* plays two roles: it tells the computer that the array variables `u` and `v` as well as the scalar variable `y` are of data type `real`, and that the computer should leave 2 "spaces" of memory for the array `u` and 2 spaces for the array `v`. Generally the *array size* has to be *at least* as many as the program will need; for example we could have put `real u(10),v(23)`, and the program would work. On the other hand, putting `real u(1),v(1)` would not work. **The computer assumes the arrays start with subscript 1, unless specified otherwise** (see the Fortran manual).

2. The arrays in this example are 1-dimensional; Fortran allows arrays of up to 7 dimensions. For example, a 2-dimensional array would be used to represent a 3×3 matrix as in the following example.

3. We can define arrays by either specifying each component `u(1)=`, `u(2)=`, or by setting all components at once `v = (/-1.0, 2.0/)`.

4. The `dot_product` intrinsic function computes the dot product between two arrays

5. Comments are lines starting with an exclamation mark. You should use them copiously. They are mainly for your future self, so you can remember what you were trying to do.

Example 3.4

Find the value for x such that the following matrix has zero determinant:

$$\begin{pmatrix} 3 & 1 & 1 \\ 6 & -2 & -1 \\ x & 2 & 3 \end{pmatrix}$$

Ans: $x = 18$.

```
program get_x
 implicit none
 real :: A(3,3),x,det1,det2,det3

 A(1,1) = 3
 A(1,2) = 1
 A(1,3) = 1
 A(2,1) = 6
 A(2,2) = -2
 A(2,3) = -1
 A(3,2) = 3
 A(3,3) = 3

 det1 = A(1,2)*A(2,3)-A(1,3)*A(2,2)
 det2 = A(1,1)*A(2,3)-A(1,3)*A(2,1)
 det3 = A(1,1)*A(2,2)-A(1,2)*A(2,1)

 x = (A(3,2)*det2-A(3,3)*det3)/det1

 print*,' x = ',x

end program get_x
```

Things to note:

1. A matrix consisting of n rows and m columns must be allocated *at least* $n \times m$ spaces of memory, with the first dimension being at least n and the second being at least m. Thus a 3×4 matrix must be dimensioned at least `A(3,4)`, although `A(5,5)` would also be correct while `A(4,3)` would not.

2. Rather than evaluate an expression for x in one line, we have calculated each subdeterminant (`det1, det2, det2`) separately. This has several advantages including making the mathematics more obvious and making the program easy to check for errors.

3. The last row has been used to evaluate the determinant; recall that *any* row or column may be used for this purpose.

Example 3.5

Repeat example 3.4, but using some modern features of Fortran to make the code shorter:

```
program get_x
 implicit none
 real :: A(3,3),x,det1,det2,det3

 ! define each row separately
 A(1,:) = (/3,1,1/)
 A(2,:) = (/6,-2,-1/)
 A(3,2:3) = 3

 det1 = A(1,2)*A(2,3)-A(1,3)*A(2,2)
 det2 = A(1,1)*A(2,3)-A(1,3)*A(2,1)
 det3 = A(1,1)*A(2,2)-A(1,2)*A(2,1)

 x = (A(3,2)*det2-A(3,3)*det3)/det1

 print*,' x = ',x

end program get_x
```

Things to note:

1. We can define whole columns or rows using a colon as the index, e.g. `A(:,1)`. We can also refer to just part of a row or column by specifying a range of numbers, e.g. `A(3,2:3)`.
2. Setting an array equal to a scalar sets the *every* component of the array equal to that number. For example, using `A(3,2:3) = 3` sets both `A(3,2)` and `A(3,3)` to the value `3`.
3. An alternative way to assign the whole array in one line would be to use the `reshape` intrinsic function. Try the following in the program above:

   ```
   A = reshape((/3,6,0,1,-2,3,1,-1,3/),shape=(/3,3/))
   ```

   Notice that here we have to set the *whole* array, including setting the blank entry to zero. Also note that the entries are listed in *column-major* order, which is the order in which the actual numbers are stored in the computer in Fortran.
4. In modern Fortran you can perform operations on arrays just as if they were numbers. For example you can compute the sum of two matrices, `A` and `B`, by simply typing `A  B+` (see exercise on next page). For this to work both arrays must have the same *rank*.

Exercises

1. Find the magnitude of the vectors $\mathbf{a} = 2\mathbf{i} - 3\mathbf{j} + \mathbf{k}$ and $\mathbf{b} = -\mathbf{i} + 4\mathbf{j} + 2\mathbf{k}$. *Ans*: 3.741657, 4.582576

2. Find the angle between the two vectors in Question 1 in degrees. *Ans:* 134.415°

3. Find the component of $\mathbf{a}$ in the direction of $\mathbf{b}$. *Ans:* $-4/7\mathbf{b}$

4. Evaluate the following:

$$\begin{pmatrix} -3 & 1 \\ 1 & 2 \\ 4 & 1 \end{pmatrix} + \begin{pmatrix} 2 & 1 \\ 0 & 3 \\ -1 & -1 \end{pmatrix}$$

4 Sums, Products, and Other Repetitive Operations: Do Loops

This section looks at what we call *looping*. This can be used when we need to repeat an operation more than once, such as defining the components of a vector via a formula. There are several ways to "loop"; here are two — a third way is illustrated in Ex. 8.3.

4.1 Do loops

We will use the following kind of *do loop* most often in this primer; it is up to you to choose which you prefer.

Example 4.1

Convert the polar coordinates $(r = 2, \theta = \pi/6)$, $(r = 1, \theta = \pi/4)$, $(r = 3, \theta = \pi/2)$ into Cartesian coordinates (x_i, y_i), $i = 1, 2, 3$.

```
1  program loops
2   implicit none
3   real :: x(3),y(3),r(3),theta(3)
4   integer :: i
5   real, parameter :: pi = 4.*atan(1.0)
6
7   r = (/2.0, 1.0, 3.0/)
8   theta = (/pi/6.,0.25*pi,0.5*pi/)
9
10  do i=1,3
11     x(i) = r(i)*cos(theta(i))
12     y(i) = r(i)*sin(theta(i))
13     print*,'x(',i,') = ',x(i)
14     print*,'y(',i,') = ',y(i)
15  enddo
16
17 end program loops
```

Things to note:

1. We must define π: Fortran doesn't have such numbers inbuilt. We used the `parameter` attribute to specify that `pi` should have a fixed value throughout the program execution
2. The loop starts with `do i=1,3` and ends with `enddo`. The computer repeats the statements in between for each value of the *counter* variable `i`. Although it is possible to use non-integers for counter variables, it is good practice to use integers.
3. To make the *do loop* easier to read, we have indented the statements between `do i=1,3` and `enddo` by 3 spaces.
4. You can *nest* do loops, *ie.* you can have do loops inside do loops.

4.2 Implied *do loops*

We can simplify the previous example by using the array notation in modern Fortran

Example 4.2

Repeat example 4.1 but using array notation instead of *do loops*.

```
program loops
 implicit none
 real :: x(3),y(3),r(3),theta(3)
 integer :: i
 real, parameter :: pi = 4.*atan(1.0)

 r = (/2.0, 1.0, 3.0)
 theta = (/pi/6.,0.25*pi,0.5*pi/)

 x = r*cos(theta)
 y = r*sin(theta)

 print*,'x = ',x
 print*,'y = ',y

end program loops
```

Things to note:

1. The *do loop* is no longer *explicit* in the code, but it is *implied* by the array operations on x and y.
2. We could equivalently use

   `x = r*cos(theta)`

   or

   `x(:) = r(:)*cos(theta(:))`

   or

   `x(1:3) = r(1:3)*cos(theta(1:3))`
3. An alternative to do loops for array assignment with conditions is the `where/end where` construct, e.g.:

```
where (r > 0)
   x = r*cos(theta)
   y = r*sin(theta)
elsewhere
   x = 0.
   y = 0.
end where
```

4.3 Another kind of *do loop*

Example 4.3

Compute the sum

$$\sum_{n=1}^{10}(n^2+n)$$

```
1  program loopy
2   implicit none
3   integer :: n, sum
4
5   sum = 0.
6
7   myloop: do n=1,10
8      sum = sum + n**2 + n
9   enddo myloop
10
11  print*,' sum = ',sum
12
13 end program loopy
```

Things to note:

1. Do loops can have labels, which can be useful if you want to exit or skip over part of the loop. For example, try inserting the following statement inside the loop:

   `if (n==9) exit myloop`

   or the following to skip $n = 8$:

   `if (n==8) cycle myloop`

   These statements should be used in place of the deprecated `goto`, `continue` and `do` <*line number*> statements that were used in older FORTRAN programs.

2. Note the way the sum is evaluated. We *initialize* the variable `sum` by setting it equal to zero. The sum is then built up each time the *do loop* is executed.

Example 4.4

For your choice of z, evaluate the product

$$\prod_{j=1}^{3} \frac{z - x_j}{y_j},$$

and (x_i, y_i) are given in the following table:

i	x_i	y_i
1	3.1	5.7
2	−2.2	0.1
3	1.1	−4.8

```
1  program myprod
2   implicit none
3   real :: z,x(3),y(3),prod
4   integer :: i
5
6   x = (/3.1,-2.2,1.1/)
7   y = (/5.7,0.1,-4.8/)
8
9   print*,' z?'
10  read*,z
11
12  prod = 1.0
13
14  do i=1,3
15     prod = prod*(z- x(i))/y(i)
16  enddo
17
18  print*,'product=',prod
19
20 end program myprod
```

A thing to note:

- The product is evaluated by initializing the variable `prod` to 1. The product is then built up as the *do loop* is executed.

Example 4.5

Generate the next 8 numbers of the following *Fibonacci sequence* using the rule:

$$x_{n+1} = x_n + x_{n-1}$$

with $x_1 = 1$ and $x_2 = 1$. *Ans:* {2,3,5,8,13,21,34,55}.

```
program fib
 implicit none
 integer :: x(10),n

 x(1) = 1
 x(2) = 1

 do n=2,9
    x(n+1) = x(n) + x(n-1)
    print*,n+1,x(n+1)
 enddo

end program fib
```

A thing to note:

- The print statement `print*,n+1,x(n+1)` involves the computer evaluating `n+1` before it can print out the answer.

Exercises

1. Verify the formula

$$\sum_{n=0}^{N} ar^n = \frac{a(1-r^{N+1})}{1-r}, \qquad r \neq 1$$

for various values of a, r and N.

2. Verify the formula

$$\prod_{k=1}^{n-1} \left(x^2 - 2x \cos \frac{k\pi}{n} + 1 \right) = \frac{x^{2n}-1}{x^2-1}$$

for various values of x and n.

3. How many terms of the following product do you need to calculate to obtain three figure accuracy?

$$\prod_{k=2}^{\infty} \left(1 - \frac{1}{k^2} \right) = \frac{1}{2}.$$

Ans: about 1000.

5 Logical Decisions: If Statements

It is possible to make logical decisions in Fortran using the `if` statement. There are several ways this can be used:

Example 5.1

Find the minimum number in the following set: $\{n_i, i = 1, 9\} = \{5, 7, 2, 9, 5, 3, 9, 1, 8\}$.

```
program mymin
 implicit none
 integer :: i,n(9),min

 n = (/5,7,2,9,5,3,9,1,8/)
 min = n(1)

 do i=1,size(n)
    if (n(i) < min) min = n(i)
 enddo

 print*,'minimum = ',min
 print*,'minimum = ',minval(n)

end program mymin
```

Things to note:

1. This is the simplest form of the *if* statement. It has the structure

   `if(`*logical expression*`)`*statement*

   Some logical expressions are listed here:

Expression	Modern Fortran	Old-style FORTRAN
$a < b$	`a < b`	`a.lt.b`
$a \leq b$	`a <= b`	`a.le.b`
$a > b$	`a > b`	`a.gt.b`
$a \geq b$	`a >= b`	`a.ge.b`
$n = m$	`n == m`	`n.eq.m`
$n \neq m$	`n /= m`	`n.ne.m`
$a < b$ & $x \geq y$	`(a < b).and.(x >= y)`	`a.lt.b.and.x.ge.y`
$n = m$ or $\alpha > \beta$	`n==m .or.  alpha > beta`	`n.eq.m.or.alpha.gt.beta`

2. We initialised the variable `min` by setting it to a number bigger than any in the set.

3. An alternative to the code above is the array intrinsic function *minval*. You can also use `minloc(n)` to find the *location* of the minimum in the array..

Example 5.2

Find all integers less than 100 which are divisible by 7 or 17.

```
program div
 implicit none
 integer :: n

 do n=1,99
    if ((n==(n/7)*7).or. &
        (n==(n/17)*17)) print*,n
 enddo

end program div
```

Things to note:

1. We have taken advantage of the way the computer does integer division; `(n/3)*3` will only equal `n` when n is divisible by 3.
2. Lines in old FORTRAN programs were not allowed be more than 72 spaces long. This was historical; it came from the days when computing was done with punch cards! This is no longer necessary but for readability it is still a good idea to avoid lines that are too long. We have broken the `if` statement into two lines to show you how to *continue* a statement on a new line. You simply put an ampersand at the end of the line you want to continue.
3. You must always check that your brackets match!
4. An alternative way to achieve the above is with the `mod` intrinsic function, which returns the *remainder* of an integer division operation. Thus we could replace the checks in the `if` statement with the following:

   ```
   if (mod(n,7)==0.or.mod(n,17)==0)
   ```

   Try this!

Example 5.3

Evaluate the following function for $x = -2.3, -e, 3\pi/4, 0$:

$$f(x) = \begin{cases} e^x - 1, & x < 0 \\ \tan x, & x \geq 0. \end{cases}$$

```
program fun
 implicit none
 real :: f,x(4)
 real, parameter :: e = exp(1.0)
 real, parameter :: pi = 4.0*atan(1.0)

 x = (/-2.3,-e,0.75*pi,0.0/)

 do i=1,4
    if (x(i) < 0.) then
       f = exp(x(i)) - 1.0
    else
       f = tan(x(i))
    endif

    print*,'f(',x(i),') = ',f
 enddo

end program fun
```

Things to note:

1. You don't need to declare all your variables of the same type in one line (note the two `real` statements).
2. Note the way we have defined e and π - you don't have to look them up or remember them! We also used the `parameter` attribute to specify that they should not be changed by the code.
3. In Example 5.1, we used a single line `if` statement because there was only one thing the computer had to do if the *logical expression* was true. If there are several things the computer must do if the *logical expression* is true, the structure is as follows:

```
if (<logical statement>) then
   <statement>
   <statement>
   ..
   ..
endif
```

If the *logical expression* is NOT true, you may wish the computer to execute some other statements, as in the previous example. In this case, the structure is as follows:

```
if (<logical statement>) then
   <statement>
   <statement>
   ..
elseif (<logical statement>) then
   <statement>
   <statement>
   ..
else
   <statement>
   <statement>
   ..
   ..
endif
```

You can *nest* `if` statements, just as you can *nest* do loops, *ie.* you can have `if` statements inside `if` statements to cover more than 2 alternatives.

Exercise

1. The function sgn x gives the sign of its argument, and thus is defined as follows:

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0 \\ 1, & x \geq 0. \end{cases}$$

Write a program which tabulates values of the function $f(x) = \operatorname{sgn}(\sin x)$ for 10 equally spaced values of x in the range $[0, 2\pi]$.

6 Subprograms

It is often convenient to separate a program into sections, with each section performing a specific task. For instance, it may be that a particular calculation needs to be performed several times, and if such a calculation involves several steps it is convenient to avoid repeating these steps each time we need to do the calculation.

We thus can have a *main* program and several *subprograms*, each of which (except for function statements) must follow the same rules regarding dimensioning arrays *etc.*, as will be made clear in the following examples.

Subprograms which return only one number to the main or subprogram which called it are called *functions*. Subprograms which return more than one number and/or arrays are called *subroutines*.

6.1 Functions

Sometimes it is necessary to evaluate a function at several points in a program. Rather than repeating the definition of the function at each point, we can use a *function statement* if it is possible to define the function in only one statement, or a *function subprogram* if we need more than one statement to define the function. The following example is rather trivial in that the definition of the function could have gone in the main body of the program, since it is only used once. However, it illustrates how a function is used.

Example 6.1

Evaluate the function $f(x) = (\sin x + 1)^{1/4}$ for $x = \pi/3, 2\pi/3, \pi$.

```
1  program morefun
2   implicit none
3   integer :: i
4   real :: x(3),y
5   real, parameter :: pi = 4.*atan(1.)
6
7   do i=1,3
8      x(i) = i*pi/3.
9      y = f(x(i))
10     print*,i,x(i),y
11  enddo
12
13 contains
14
15  real function f(z)
16   real, intent(in) :: z
17
18   f = (sin(z) + 1.0)**0.25
19
20  end function f
21
22 end program morefun
```

Things to note:

1. The function is defined after the *contains* statement and has the form

   `f(a,b,c,...)`=expression involving the variables `a,b,c,...`. We use the `intent(in)` attribute to specify that the arguments are *input* only.

2. As in this example, the argument of the function need not be the same as the one used in the main part of the program. This is because the function is treated like a separate sub-program, as in Example 6.3.

3. In the above we have included the function as part of the program. In general it is better to put functions inside a *module*, which are separate from the main program. This would be structured as follows:

```
1  module mymod
2   implicit none
3   public :: f
4
5  contains
6
7   real function f(z)
8    real, intent(in) :: z
9
10   f = (sin(z) + 1.0)**0.25
11
12  end function f
13
14 end module mymod
15
16 program morefun
17  use mymod
18  implicit none
19  integer :: i
20  real :: x(3),y
21  real, parameter :: pi = 4.*atan(1.)
22
23  do i=1,3
24     x(i) = i*pi/3.
25     y = f(x(i))
26     print*,i,x(i),y
27  enddo
28
29 end program morefun
```

   Modules were introduced in Fortran 90 and are very powerful. From now on we will put all functions and subroutines inside modules. Even better is to put modules into separate source files, which makes them re-usable by many different programs.

Example 6.2

Write a general program to calculate the first 3 terms of the Taylor series for a function $f(x)$ expanded about $x = a$, *ie.* calculate

$$f(x) = f(a) + f^{'}(a)(x-a) + \frac{f^{''}(a)}{2}(x-a)^2$$

Test it by calculating an approximate value for $\ln 1.1$ using the Taylor series for $\ln x$ expanded around $x = 1$ with $a = 1$ and $x = 1.1$.

```
module kanye
 implicit none
 public :: f,fd,fdd

contains
 real function f(x)
  real, intent(in) :: x

  f(x) = log(x)

 end function f

 real function fd(x)
  real, intent(in) :: x

  fd(x) = 1.0/x

 end function fd

 real function fdd(x)
  real, intent(in) :: x

  fdd(x) = -1.0/x**2

 end function fdd

end module kanye

program taytay
 use kanye
 implicit none
 real :: a,x,y

 a = 1.
 x = 1.1
 y = f(a) + fd(a)*(x-a) + 0.5*fdd(a)*(x-a)**2
 print*,'x = ',x,' f(x) = ',y

end program taytay
```

Example 6.3

Calculate an approximate value for $e^{3-\pi}$ using the first 5 terms of the following Taylor series expansion:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

```
module factorial
 implicit none
 real, parameter :: pi = 4.*atan(1.)
 public :: fac, pi

 private

contains

 function fac(n)
  integer, intent(in) :: n
  real :: fac, prod

  if (n==0) then
     fac = 1.
     return
  endif
  prod = 1.0
  do i=1,n
     prod = prod*i
  enddo
  fac = prod

 end function fac

end module factorial

program myfac
 use factorial
 implicit none
 real :: x,sum
 integer :: n

 x = 3.-pi
 sum = 0.
 do n=0,4
    sum = sum + x**n/fac(n)
 enddo
 print*,x,sum,' exact = ',exp(x)

end program myfac
```

Things to note:

1. A *function* is treated as a separate sub-program, so you must declare the data type of all the variables in it.
2. You must declare the *data type* of the *function*. You can either do this when defining the function, as in `real function fac` or on a separate line as done in the example above
3. A *real* function may take an *integer* variable as an argument.
4. The statements where the function is defined takes the form

   `fac=`*statement*.

   Note that it is NOT `fac(n)=.....`
5. For the special case `n=0`, `fac` is defined separately. If this option is taken, the program must then return to the main program. Generally, if you need to return to the main program *before* the end of the subprogram, you use the statement `return` (this statement is sometimes used before the `end` statement as well, but is redundant here).
6. Modules can have *public* and *private* components. A *public* component is one that can be seen by things *outside* the module, whereas *private* variables and functions are only visible to other functions inside the module.
7. Public parts of modules are included in the main program via the `use` statement (e.g. `use factorial` in the above example).
8. Modules can contain *shared variables* and constants. For example in the above we defined π within the module and used this in the main program via the `use` statement.

6.2 Subroutines

We often need to return several numbers or even arrays to the program which called the subprogram. For this purpose, we use *subroutines*.

Example 6.4

Verify that matrix multiplication is not commutative for the following pair of matrices:

$$\begin{pmatrix} 1 & 2 \\ 3 & -1 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 4 & 0 \\ 1 & 1 \end{pmatrix}$$

```
1  module mat
2   implicit none
3
4  contains
5   subroutine multiply(C,nc,mc,D,md,P)
6    real, intent(in) :: C(nc,mc), D(mc,md)
7    real, intent(out) :: P(nc,md)
8    real :: sum
9
10   do i=1,nc
11      do j=1,md
12         sum = 0.
13         do k=1,mc
14            sum = sum + C(i,k)*D(k,j)
15         enddo
16         P(i,j) = sum
17      enddo
18   enddo
19
20  end subroutine multiply
21
22 end module mat
23
24 program mult
25  implicit none
26  integer, parameter :: na = 2, ma = 2, nb = 2, mb = 2
27  real :: A(2,2), B(2,2), P(2,2)
28
29  A = reshape((/1,3,2,-1/),shape=(/2,2/))
30  B = reshape(/4,1,0,1/),shape=(/2,2/))
31
32  call multiply(A,na,ma,B,mb,P)
33
34  do i=1,2
35     print*,(P(i,j),j=1,2)
36  enddo
37
```

```
38  call multiply(B,nb,mb,A,ma,P)
39
40  do i=1,2
41     print*,(P(i,j),j=1,2)
42  enddo
43
44 end program mult
```

Things to note:

1. The main program uses the `call` statement to access the subroutine. It is of the form `call subroutine _name`(*parameter list*), where the parameter list consists of any constants/variables/arrays which the subroutine needs, as well as the required constants/variables/arrays which the subroutine calculates.

2. Although the variable names in the parameter lists need not be the same, they must match in data type and dimension. Thus in this example, the parameter list in both the main program and subroutine must have in the exact same order: a 2×2 array, 2 scalars, a 2×2 array, 1 scalar, a 2×2 array.

3. It is acceptable to have constants in the parameter list of the call statement. In this example, we could have used the statement `call multiply(A,2,2,B,2,P)`.

4. Each variable and array must be declared in the subroutine.

5. The dimensions of the arrays in the main program and the subroutine must match.

6. Note that the second dimension of the matrix `C` is the same as the first dimension of the matrix `D`. Matrix multiplication is not defined otherwise. Thus we only pass through to the subroutine both dimensions for the left-hand matrix and the first dimension for the right-hand matrix.

7. You should check that you understand how the nested *do loops* work in the subroutine - for example, invent a 2×3 and a 3×2 matrix and try multiplying them together.

8. Unlike in the case of function subprograms, it is meaningless to talk of the data type of a subroutine. Thus you can call a subroutine anything you like - in this case the name starts with the letter m. It must be a single string of characters, so if you want to involve more than one word, you can join them with the underscore character _ as in `subroutine very_interesting(a,b,c)`.

9. The `print` statement in the main program uses an *implied do loop*. Try it and see how it works!

10. The above subroutine is not actually necessary, since Fortran 90 introduced the `matmul` intrinsic function. Try rewriting your code to use this function instead of your `multiply` subroutine. The syntax is `P = matmul(A,B)`.

Exercise

Verify that the following matrices are the inverse of each other, *ie.*, that $AB = BA = I$. In this case matrix multiplication *is* commutative!

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & -1 & 1 \\ 0 & 2 & 1 \end{pmatrix}, \qquad B = \begin{pmatrix} -3 & 4 & 5 \\ -1 & 1 & 2 \\ 2 & -2 & -3 \end{pmatrix}$$

7 Reading From Files; Writing To Files

Often it is more convenient to read data from a file than type it in when you run the program. As well, you may need to store data you have calculated so that another program can use it, or so that you can print it out.

We have already used the `read*,` statement which pauses the program until you have read in the data, as well as the `print*,` statement which prints the answers on the screen. In the following example, data is read from a file called `polar.dat` and the data calculated is written to a file called `polar.out`. These names are arbitrary, including the *filename extensions* `.dat` and `.out` which here are used to distinguish between the data files. We could have used something like `polar1.dat` and `polar2.dat` if we wanted. The *filename extension* of the Fortran file itself *must* have the extension `.f90`.

Finally, in order to read from a data file, you must first create a data file!

Example 7.1

Convert from spherical polar to cartesian coordinates the following points:

r	θ	φ
2.34	0.44	0.62
1.02	1.02	−3.14
0.56	−3.76	−1.21
3.91	−0.60	6.03

```
program files
 implicit none
 real :: x,y,z,r(4),theta(4),phi(4)
 integer :: i

 open(1,file='polar.dat',status='old',action='read')
 open(2,file='polar.out',status='replace',action='write')

 do i=1,4
     read(1,*) r(i),theta(i),phi(i)
     x = r(i)*sin(theta(i))*cos(phi(i))
     y = r(i)*sin(theta(i))*sin(phi(i))
     z = r(i)*cos(theta(i))

     write(2,*) 'x(',i,') = ',x
     write(2,*) 'y(',i,') = ',y
     write(2,*) 'z(',i,') = ',z
 enddo
 close(1)
 close(2)

end program files
```

Things to note:

1. The `open` statement tells the program to associate a *unit number* with a particular data file. In this case, *unit number* `1` is associated with a file called `polar.dat` and as it happens, the program will `read` from this file. *Unit number* `2` is associated with a file called `polar.out` and the program will `write` to this file. The file names must be placed between single quotes.

2. We gave some options to the `open` command to indicate that the file should either already exist (`status='old'`) or be replaced (`status='replace'`), and whether we are opening the file for reading or writing using the `action=` specifier. These are optional, but help to tell the program what your intention is so that any mistakes can be caught automatically.

3. The `read` statement has the form `read(`*unit number*`,*)a,b,c,...` where the `*` again means *free format*. The data file the program reads from must have the data set out *exactly* as in the read statement. For the present example, the following file shows how `polar.dat` should look. There are two alternatives: you may separate your data with commas or spaces. The computer uses spaces when it `writes` to a file.

4. Files should be closed with the `close(`*unit number*`)` statement.

2.34,0.44,0.62 1.02,1.02,-3.14 0.56,-3.76,-1.21 3.91,-0.60,6.03	or	2.34 0.44 0.62 1.02 1.02 -3.14 0.56 -3.76 -1.21 3.91 -0.60 6.03

5. The `write` statement has the same format as the `read` statement, *ie.,* `write(`*unit number*`,*)a,b,c,...`

6. If the `open` statements for units 5 or 6 are omitted, these unit numbers can be used for for reading from and writing to the screen. In other words, the following are equivalent:

   `read*,` and `read(5,*)`, and

   `print*,` and `write(6,*)`.

7. Notice how we don't need to define arrays for x, y and z because they are written to `polar.out` as soon as they are calculated. If they had been needed later on in the program, we would have had to store their values in arrays.

8. The fourth `write(2,*)` statement will leave a blank line after every 3 lines in `polar.out`. This makes large data files easier to read.

8 Putting It All Together...

Example 8.1

Write a general program to approximately evaluate a definite integral using the *Trapezoidal Rule*:

$$\int_a^b f(x)dx \simeq \frac{\Delta x}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(a+i\Delta x) + f(b)\right),$$

where $\Delta x = (b-a)/n$ is the width of an interval and n is the number of such intervals; see Figure 1.

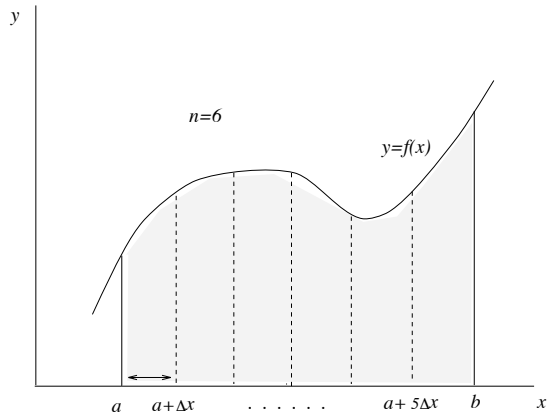

Figure 1: Approximating $\int_a^b f(x)dx$.

```
module myfun
 implicit none

contains
 real function f(x)
  real, intent(in) :: x

  f = ...

 end function f

end module myfun

program trapezoidal
 use myfun
 implicit none
 real :: a,b,dx,integral,sum
 integer :: i,n

 ! define terms
 a = ...
 b = ...
 n = ...
 sum = 0.
 dx = (b-a)/real(n)

 do i=1,n-1
    sum = sum + f(a+i*dx)
 enddo
 integral = 0.5*dx*(f(a) + 2.*sum + f(b))
 print*,' integral = ',integral

end program trapezoidal
```

Things to note:

1. The line starting with an exclamation mark `!` is called a *comment*. The computer ignores this line; *comments* are used for your own information. It is good practice to put *comments* in your programs; when you look at a program you have written in the past, it is sometimes hard to remember what you have done. It is also helpful to other people who might use your program.

2. To run this program, you need to supply values for `a, b` and `n` as well as a function (the integrand of the integral).

Example 8.2

Create a table of values for the *error function* $\operatorname{erf} x$, defined as follows:

$$\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Use $x = 0, 0.2, 0.4, \ldots, 2.0$. You should get:

x	$\operatorname{erf} x$
0.0	0.0000
0.2	0.2227
0.4	0.4284
0.6	0.6039
0.8	0.7421
1.0	0.8427
1.2	0.9103
1.4	0.9523
1.6	0.9763
1.8	0.9891
2.0	0.9953

Note that $\lim_{x \to \infty} \operatorname{erf} x = 1$, *ie.,* the *normalizing* factor $2/\sqrt{\pi}$ ensures that

$$\frac{2}{\sqrt{\pi}} \int_0^\infty e^{-t^2} dt = 1.$$

```
module myfun
 implicit none
 real, parameter :: pi = 4.*atan(1.)
 integer, parameter :: n = 100

 public  :: pi, trap
 private :: n, f

contains
 real function f(x)
  real, intent(in) :: x

  f = exp(-x**2)

 end function f

 real function trap(a,b)
  real, intent(in) :: a,b
  integer :: i
  real :: sum,dx

  sum = 0.
  dx = (b-a)/real(n)

  do i=1,n-1
     sum = sum + f(a+i*dx)
  enddo
  trap = 0.5*dx*(f(a) + 2.0*sum + f(b))

 end function trap

end module myfun

program nerfgun
 use myfun
 implicit none
 real :: x,myerf
 integer :: i

 print*,' x ',' erf(x) '
 do i=0,10
    x = 0.2*i
    myerf = (2.0/sqrt(pi))*trap(0.,x)
    print*,x,myerf,' correct answer = ',erf(x)
 enddo

end program nerfgun
```

Things to note:

1. We have used the program in Example 8.1 to create a *function subprogram* to evaluate the integral in the definition of erf x.
2. The intrinsic function `erf` was added in the Fortran 2008 standard, so we can use `erf(x)` to check our answer. Try increasing the value of `n` to see if your answer gets closer to the answer provided by the Fortran intrinsic routine.
3. The limits of the integral have been passed through to the function subprogram via the *parameter list* of `trap`. Note that you need not use the same variable names in the subprogram as in the main program, in fact in this example the number `0` in the main program corresponds to the variable `a` in the subprogram.
4. Variables and parameters defined at the top of the module are *shared* by all the subprograms of the module (e.g. the value of `n`)
5. We used the `public` and `private` attributes to show which parts of the module should be visible to the program, and which should be kept hidden

Example 8.3

Write a general program to find the root of the equation $f(x) = 0$ using the Newton- Raphson iteration formula

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})},$$

where x_n is the n^{th} approximation of the solution to $f(x) = 0$ and x_0 is an initial guess. Test your program by finding the root of $\cos x = x$, *ie.*, put $f(x) = \cos x - x$ (solution $x = 0.73908$). You should stop the iteration process when $|f(x)| < \epsilon$, where $\epsilon = 10^{-5}$ (say).

```
module myfun
 implicit none

contains

 real function f(x)
  real, intent(in) :: x

  f = cos(x) - x

 end function f

 real function fd(x)
  real, intent(in) :: x

  fd = -sin(x) - 1.

 end function fd
end module myfun

program newtraph
 use myfun
 implicit none
 real,    parameter :: eps = 1.e-5
 integer, parameter :: maxits = 20
 real    :: x
 integer :: k

 print*,' guess for x?'
 read*,x

 k = 0
 do while (abs(f(x)) > eps)
    k = k + 1
    x = x - f(x)/fd(x)
    if (k > maxits) then
       print*,' too many iterations'
       stop
    endif
 enddo

 print*,' root = ',x,' in ',k,' iterations'

end program newtraph
```

We have introduced two new statements in this example:

1. the `do while` loop. This continues to iterate the loop until the condition in brackets is satisfied or the `exit` command is issued
2. The `stop` statement. This causes the program to stop running at this point.
3. We have introduced a safeguard against getting stuck in an infinite loop by defining a *counter* variable `k`; this is initialized to zero and is increased by 1 every iteration.
4. Note the statement `x=x-f(x)/fd(x)`. Again we emphasize that the value assigned to the variable `x` is *replaced* by a new value, *ie.,* the = symbol means *replace*.
5. What happens if you put, say, $\epsilon = 10^{-10}$?
6. What happens if you have as your initial guess $x_0 = -\pi/2$?

Example 8.4

Given $n+1$ points $\{(x_i, y_i), i = 0, 2, \ldots, n\}$, an n^{th} order polynomial can be defined which passes through each point. Such a polynomial $P_n(x)$ is given by the *Lagrange Interpolation* formula

$$P_n(x) = \sum_{k=0}^{n} y_k \prod_{\substack{i=0 \\ i \neq k}}^{n} \left(\frac{x - x_i}{x_k - x_i} \right).$$

Evaluate the second order Lagrange polynomial which passes through the points given in Example 4.3 for several values of x. Terminate the program by reading in a value of x which is greater than, say, 99.

```
1  program lagrange
2   implicit none
3   real :: z,x(3),y(3),prod,sum,xx
4   integer :: i,k,n
5
6   n = 3
7   open(1,file='lagrange.dat',status='old',action='read')
8   do i=1,n
9      read(1,*) x(i),y(i)
10  enddo
11  close(1)
12
13  myloop: do
14     print*,'x ?'
15     read*,xx
16     if (xx > 99) exit myloop
17
18     sum = 0.
19     do k=1,n
20        prod = 1.0
21        do i=1,n
22           if (i /= k) prod = prod*(xx - x(i))/(x(k) - x(i))
23        enddo
24
25        sum = sum + y(k)*prod
26     enddo
27
28     print*,'Pn(',xx,')=',sum
29
30 enddo myloop
31 print*,' thanks for coming'
32
33 end program lagrange
```

Things to note

1. In order to exit the loop, we have included the statement `if (xx > 99) break`. You can "kill" the program from the unix command line using `ctrl-c`, which will stop the execution of the program.
2. Note that the scalar x is called `xx` and the subscripted x_i is called `x(i)`. We would have been in trouble if they had both been called `x` (try doing this and see what happens).

Example 8.5

By modifying the program for the *Trapezoidal Rule* (Example 8.1), write a program to evaluate a definite integral using *Simpson's Rule*, given by

$$\int_a^b f(x)dx \simeq \frac{\Delta x}{3}\left(f(a) + 4\sum_{i=1,2}^{n-1} f(a+i\Delta x) + 2\sum_{i=2,2}^{n-2} f(a+i\Delta x) + f(b)\right),$$

where Δx is the width of an interval, n is the number of such intervals and $\sum_{i=1,2}^{n-1}$ means sum from 1 to $n-1$ in steps of 2 (so n must be *even*).

Use it to verify (approximately) the following for various m:

$$\int_0^\pi \frac{\sin mx}{\sin x}dx = \begin{cases} 0, & m \text{ even} \\ \pi, & m \text{ odd.} \end{cases}$$

Try varying the number of intervals and notice how the accuracy increases with increasing n (although there comes a point where *roundoff errors* will start to contaminate your solution). Note also that straightforward use of the function statement for $f(a)$ and $f(b)$ will lead to a `divide by zero` error; you will have to take advantage of the limit results

$$\lim_{x\to 0}\frac{\sin mx}{\sin x} = m$$

and

$$\lim_{x\to \pi}\frac{\sin mx}{\sin x} = (-1)^{m+1}m$$

(try deriving these yourself!)

```
module myfun
 implicit none
 real, parameter :: pi = 4.*atan(1.)
 integer, parameter :: n = 100

 public  :: pi, simpson
 private :: n

contains
 real function f(x,m)
  real, intent(in) :: x
  integer, intent(in) :: m

  if (abs(sin(x)) < tiny(0.)) then
     f = (-1)**(m+1)*m
  else
     f = sin(m*x)/sin(x)
  endif

 end function f

 real function simpson(a,b,m)
  real, intent(in) :: a,b
  integer, intent(in) :: m
  integer :: i
  real :: sum1,sum2,dx,fa,fb

  fa = f(a,m)
  fb = f(b,n)

  sum1 = 0.
  sum2 = 0.
  dx = (b-a)/real(n)

  do i=1,n-1,2
     sum1 = sum1 + f(a+i*dx,m)
  enddo
  do i=2,n-2,2
     sum2 = sum2 + f(a+i*dx,m)
  enddo
  simpson = dx/3.*(fa + 4.0*sum1 + 2.0*sum2 + fb)

 end function simpson

end module myfun
```

```
50
51 program simp
52  use myfun
53  implicit none
54  integer :: m
55  real :: integral
56
57  do
58     print*,'m ?'
59     read*,m
60
61     integral = simpson(0.,pi,m)
62     print*,' integral = ',integral
63  enddo
64
65 end program simp
```

Things to note:

1. You should know how to "kill" the program before you attempt this; there is no exit point!
2. You can make a *do loop* increment in steps other than 1. For example, `do k=10,-6,-2` will step `k` from `10` back to `-6` in steps of `-2`, *ie.,* `k` will take the values `10,8,6,4,2,0,-2,-4,-6`.

A Appendix: Some Intrinsic Functions

All the following *intrinsic functions* must take a `real` number as their argument except `abs(x)`, `int(x)`, `min(x)` and `max(x)` which can all take both `real` and `integer` type arguments. Search online for *Fortran intrinsic functions* for other other intrinsic functions.

$\sin x$	`sin(x)`
$\cos x$	`cos(x)`
$\tan x$	`tan(x)`
$\sinh x$	`sinh(x)`
$\cosh x$	`cosh(x)`
$\tanh x$	`tanh(x)`
$\mathrm{Sin}^{-1} x$	`asin(x)`
$\mathrm{Cos}^{-1} x$	`acos(x)`
$\mathrm{Tan}^{-1} x$	`atan(x)`, `atan2(x,y)`
$\sqrt{x}$	`sqrt(x)`
e^x	`exp(x)`
$\ln x$	`log(x)`
$\log_{10} x$	`log10(x)`
$\lvert x \rvert$	`abs(x)`
$[x]$	`int(x)`
$\max(x, y, z, \ldots)$	`max(x,y,z,...)`
$\min(x, y, z, \ldots)$	`min(x,y,z,...)`
$\mathrm{Erf}(x)$	`erf(x)`
$\mathrm{J}_n(x)$	`bessel_jn(n,x)`
$\Gamma(x)$	`gamma(x)`