Chapter 4

Flow Control

The programs presented so far are examples of straight-line code in which the executable statements are performed exactly once in the order they appear in. Occasionally we need the computer to process the statement out-of-order, depending on the value of the input or of intermediate calculations, or more than once. The process of controlling the order of code execution is called flow control. We will visit two types of flow controls in the present lecture: branching and looping.

4.1 Conditionals, Branching, and Logical

4.1.1 Quadratic Roots

To make our discussion more concrete we will focus on the simple example of calculating the roots of a quadratic equation

\[ ax^2 + bx + c = 0 \] (4.1)

The two roots of the equations are given by

\[ x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \] (4.2)

Clearly the nature of the two roots depend on the sign of the discriminant \( d = b^2 - 4ac \). For \( d > 0 \) there are two distinct real roots, for \( d = 0 \) there is a double root, and for \( d < 0 \) there are pairs of complex roots.

Before we launch on how we code a solution of the quadratic equation we illustrate the process of turning the mathematical steps into FORTRAN code. The original charting relied on visual diagrams. Here we will use the simple alternative of pseudo code.

1. input a,b,c
2. form discriminant \( d = b^2 - 4ac \)

3. (a) if \( d < 0 \) print a message that no real exist and exit.
   (b) if \( d = 0 \) print \( x_\pm = -b/2a \)
   (c) if \( d > 0 \) print \( x_\pm = -b \pm \sqrt{d}/2a \)

### 4.1.2 IF statements

Clearly the execution flow depends on the value of \( d \) and a test is required to determine which FORTRAN statements should be executed. The if statement is designed to do just that. It takes the form:

```fortran
if (logical_expression) then
  conditional execution statement 1
  conditional execution statement 2
  .
  .
endif
```

The logical_expression is the test to be performed. If the test returns a value of true the statements between the if and endif are executed, otherwise they are skipped and execution continues after the endif statement.

Hence one code that can perform the above calculation is as follows:

```fortran
program quadraticroots
implicit none
real :: a,b,c,d,x1,x2
read *, a,b,c
print *, 'a=',a, ' b=', b, 'c=',c
d = b**2 - 4.0*a*c
if (d > 0.0) then
  d = sqrt(d)
x1 = (-b+d)/(2.0*a)
x2 = (-b-d)/(2.0*a)
  print *,'x1=',x1
  print *,'x2=',x2
endif
if (d == 0.0) then
  x1 = (-b+sqrt(d))/(2.0*a)
x2 = x1
  print *,'x1=x2=',x1
endif
if (d < 0.0) then
```

```fortran```
print *, ’No real root exist’
endif

stop
end program quadraticroots

The if statement allows multiple forms to be used to build more complicated tests. A first alternative form is the simple if-elseif-else form.

if (logical_expressionA) then
  StatA1
  StatA2
  .
elseif (logical_expressionB) then
  StatB1
  StatB2
  .
elself (logical_expressionC) then
  StatC1
  StatC2
  .
else
  StatD1
  StatD2
  .
endif

The expressions logical_expressionA, logical_expressionB, logical_expressionB, are tested in order, and the first one to test true will have its enclosed statement executed, all others will be skipped. If none of the expressions evaluates to .true. the statements after the else are executed.

The following relational operations are built into Fortran

<table>
<thead>
<tr>
<th>mathematics</th>
<th>F77 or F90</th>
<th>F90</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>a.lt.b</td>
<td>a &lt; b</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>a.gt.b</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>a ≤ b</td>
<td>a.le.b</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td>a ≥ b</td>
<td>a.ge.b</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>a = b</td>
<td>a.eq.b</td>
<td>a == b</td>
</tr>
<tr>
<td>a ≠ b</td>
<td>a.ne.b</td>
<td>a /= b</td>
</tr>
</tbody>
</table>
program quadraticroots
  implicit none
  real :: a,b,c,d,x1,x2
  read *, a,b,c
  print *, 'a=',a, ' b=', b, 'c=',c
  d = b**2 - 4.0*a*c
  if (d > 0.0) then
    d = sqrt(d)
    x1 = (-b+d)/(2.0*a)
    x2 = (-b-d)/(2.0*a)
    print *, 'x1=',x1
    print *, 'x2=',x2
  elseif (d == 0.0) then
    x1 = (-b+sqrt(d))/(2.0*a)
    x2 = x1
    print *, 'x1=x2=',x1
  else
    print *, 'No real root exist'
  endif
  stop
end program quadraticroots

This form allows us to skip one unnecessary test. Finally it is important to note that if statements can be nested. Each level however, should have its own well defined endif. A third version of the quadratic roots program would read:

if (d < 0.0) then
  print *, 'No real root exist'
else
  if (d == 0.0) then
    x1 = (-b+sqrt(d))/(2.0*a)
    x2 = x1
    print *, 'x1=x2=',x1
  else
    d = sqrt(d)
    x1 = (-b+d)/(2.0*a)
    x2 = (-b-d)/(2.0*a)
    print *, 'x1=',x1
    print *, 'x2=',x2
  endif
endif
4.1. CONDITIONALS, BRANCHING, AND LOGICAL

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.true.</td>
<td>.true.</td>
<td>.false.</td>
<td>.true.</td>
<td>.true.</td>
<td>.true.</td>
<td>.false.</td>
</tr>
<tr>
<td>.true.</td>
<td>.false.</td>
<td>.false.</td>
<td>.true.</td>
<td>.false.</td>
<td>.false.</td>
<td>.true.</td>
</tr>
<tr>
<td>.false.</td>
<td>.true.</td>
<td>.true.</td>
<td>.true.</td>
<td>.false.</td>
<td>.false.</td>
<td>.true.</td>
</tr>
<tr>
<td>.false.</td>
<td>.false.</td>
<td>.true.</td>
<td>.false.</td>
<td>.false.</td>
<td>.true.</td>
<td>.false.</td>
</tr>
</tbody>
</table>

4.1.3 logical data type and associated operators

We mentioned above that the test expression must evaluate to true for the enclosed statements to be executed. To go along with these logical operations of true or false, FORTRAN has an intrinsic data type, called logical to express these ideas. The logical data types can only have one of two values, .true. or .false.. An example of their use is:

```fortran
program quadlogic
  implicit none
  real :: a,b,c,d,x1,x2
  logical :: positive
  read *, a,b,c
d = b**2 - 4.0*a*c
if (d < 0.0) then
  positive = .false.
else
  positive = .true.
endif
if (positive) then
d = sqrt(d)
x1 =(-b+d)/(2.0*a)
x2 =(-b-d)/(2.0*a)
print *, x1,x2
else
  print *,'No real roots'
endif
stop
end program quadlogic
```

Certain operations are allowed on logical statements, they are: The .not. operator flips the state. The P.and.Q is .true. if and only P and Q are .true., it is false otherwise. The P.and.Q is .true. if P and/or Q is .true.; it is false otherwise. The P.eqv.Q is .true. if P and Q have the same value, it is false otherwise. The P.neqv.Q is .true. if P and Q have different values, it is false otherwise.
4.1.4 select case statement

FORTRAN 90 allows a slightly neater form of branching called the case statement. It takes the form:

```fortran
select case (expression)
  case (case_selector_a)
    fortran-statements_a
  case (case_selector_b)
    fortran-statements_b
  .
  .
  default ! this is optional
    defaults-fortran-statements
end select
```

The major difference between the if and case statements is that the case_selector must consists of constants or named constants of type integers, logical or character.

4.2 Do-Loops

4.2.1 Numerical Integration

Occasionally the need arises to go over some statements multiple times, most often when implementing iterative type procedures. We will take the simple example of calculating a definite integral of the form

\[ A = \int_{a}^{b} f(x) \, dx \] (4.3)

and recalling that the integral is the area under the curve \( f(x) \). The simple trapezoidal rule can be used for this calculation whereby the interval \( a \leq x \leq b \) is divided into small segments denoted by \( x_{i-1} \leq x \leq x_i \) where \( x_i = i \frac{b-a}{N}, i = 0, 1, 2, \ldots, N \). Note that the width of the interval is constant and is \( \Delta x = \frac{b-a}{N} \). The area under the trapezoids is tabulated below. We will use the simple shorthand notation of \( f_i = f(x_i) \). We now need merely to add up the small trapezoids \( \Delta A_i \) to get an approximation \( A_N \) to the area under the curve. This addition yields:

\[
A_N = \sum_{i=1}^{N} \Delta A_i = \Delta x \left[ f_0 + 2f_1 + 2f_2 + 2f_3 + \cdots + 2f_i + \cdots + 2f_{N-2} + 2f_{N-1} + f_N \right] / 2 \] (4.4)

\[
A_N = \Delta x \left[ \frac{f_0}{2} + f_1 + f_2 + f_3 + \cdots + f_i + \cdots + f_{N-1} + \frac{f_N}{2} \right] \] (4.5)

\[
A_N = \Delta x \left[ \frac{f_0}{2} + f_1 + f_2 + f_3 + \cdots + f_i + \cdots + f_{N-1} + \frac{f_N}{2} \right] \] (4.6)
Figure 4.1: Graphical illustration of the trapezoidal rule integration. The graph of the continuous approximation is shown in red and the discrete trapezoidal area are shown in black. The error between the two area estimate for a single discrete interval is shown in gray.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\Delta A_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\frac{f(x_0) + f(x_1)}{2} \Delta x$</td>
</tr>
<tr>
<td>2</td>
<td>$\frac{f(x_1) + f(x_2)}{2} \Delta x$</td>
</tr>
<tr>
<td>3</td>
<td>$\frac{f(x_2) + f(x_3)}{2} \Delta x$</td>
</tr>
<tr>
<td>4</td>
<td>$\frac{f(x_3) + f(x_4)}{2} \Delta x$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$i-1$</td>
<td>$\frac{f(x_{i-2}) + f(x_{i-1})}{2} \Delta x$</td>
</tr>
<tr>
<td>$i$</td>
<td>$\frac{f(x_{i-1}) + f(x_i)}{2} \Delta x$</td>
</tr>
<tr>
<td>$i+1$</td>
<td>$\frac{f(x_i) + f(x_{i+1})}{2} \Delta x$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$N-2$</td>
<td>$\frac{f(x_{N-3}) + f(x_{N-2})}{2} \Delta x$</td>
</tr>
<tr>
<td>$N-1$</td>
<td>$\frac{f(x_{N-2}) + f(x_{N-1})}{2} \Delta x$</td>
</tr>
<tr>
<td>$N$</td>
<td>$\frac{f(x_{N-1}) + f(x_N)}{2} \Delta x$</td>
</tr>
</tbody>
</table>

Table 4.1: Trapezoidal integration
It is known that the error in approximating the integral can be bounded by

\[ E = |A_N - A| \leq \frac{(b - a)^3}{12N^2} \max_{a \leq x \leq b} \left| \frac{d^2f}{dx^2} \right| \]

(4.7)

This is an example of an error bound, and reading it is somewhat tricky for non-mathematician. The theorem asserts that the error depends on the width of the interval and the maximum curvature of the function in the interval, both intrinsic properties of the function \( f \) and not of the numerical parameter. These will not change if the numerical parameters are changed. The only numerical parameter is \( N \), and the error decreases like the square of the number of interval. This is an example of quadratic convergence since doubling the number of interval to \( 2N \) will decrease the error by a factor of 4.

### 4.2.2 Indexed Do-loop

Clearly the trapezoidal rule is repetitive in nature and what we need to do is initialize the estimate, evaluate the function and sum it up. Here is a code to do just that using the do construct.

```
program trapezoid
  implicit none
  integer :: i,N
  real :: a,b,dx,x
  read *,a,b,N
  dx = (b-a)/real(N)
  aint = 0.5*(fun(a)+fun(b))
  do i = 1,N
    x = i*dx + a
    aint = aint + fun(x)
  enddo
  print *,'Integral = ', aint
  stop
end program trapezoid
```

The statements between the do and enddo statement will be repeated \( N \) times. The variable \( i \) is the **loop index**, and ranges from 1 to \( N \) in the present instance. In the first run through the loop \( i \) will be initialized to 1, and will be incremented by 1 after passing the enddo statement. Execution then resumes at the beginning of the loop, and the loop index \( i \) is compared to the ending index \( N \), if \( i \leq N \) the loop is repeated, each time increment the loop index \( i \) by 1, otherwise it is skipped and execution resumes at the first executable statement after the enddo statement. The general form of the looping statement is
4.2. DO-LOOPS

```fortran
   do i = ibegin,iend,istride
       fortran_statements
   enddo
```

Here `ibegin` and `iend` are the lower and upper limit, and `istride` is the **increment**. When the last parameter is absent, the increment defaults to 1; the increment must be specified for all other values. All these must be integer variables. All the following are valid loops:

```fortran
   do i = 1,10,2
       print *,i ! print the first 4 odd numbers in ascending order
   enddo
   do i = 20,2,-2
       print *,i ! print the first 10 even numbers in descending order
   enddo
```

The loop index must never be modified in the body of the loop. The following is not permissible:

```fortran
   do i = 1,100
       i = 2*i-1 ! PROBLEM, loop index modified
       print *,i
   enddo
```

Although it is impossible to branch into a do-loop, it is possible to branch out of it in one of two ways; either by using the **exit** or the **cycle** statement. The **exit** transfers control out of the loop (to the statement after the **enddo**) and interrupts the count; while **cycle** just skip the count. Here are two examples:

```fortran
   do i = 1,200
       if (mod(i,2)==0) then
           cycle ! skip to the end of the loop and restart
       endif
       if (i>100) then
           exit ! skip to the end of the loop and breakout
       endif
       print *, i ! prints only odd i
   enddo
```

4.2.3 Do while loops

Not all repetitive calculations have their iteration count known, sometimes it depends on the result of intermediate calculations. An example is to decide on the number of intervals needed to compute the trapezoidal rule to a given tolerance.
One strategy is to calculate the integral first using \( N \) interval to obtain a first approximation \( A_N \); the number of interval is then doubled and the results would then be \( A_{2N} \). If the difference between \( A_N \) and \( A_{2N} \) is small we can be assured that we have reached an error level we can live with; otherwise the number of interval is doubled yet again. The FORTRAN construct needed to carry out the logic is a do while (expression)
.
.
!expression must involve variables that are modified during the loop
dono
Thus the loop keeps being repeated as long as the test expression evaluates to \( \text{.true.} \); the loop terminates when evaluates to \( \text{.false.} \). For the integration example we would have, for example:

\[
\begin{align*}
\text{iter} &= 0; \quad \text{convergence} = \text{.false.}; \quad \text{goon} = \text{.true.}; \quad \text{aintold} = 0.0 \\
\text{do while (goon)} \\
! \text{insert code for trapezoidal calculation.} \\
& \quad \text{iter} = \text{iter} + 1 \\
& \quad \text{err} = \text{abs(aint-aintold)} \\
& \quad \text{if (err < tolerance) then} \\
& \quad \quad \text{convergence} = \text{.true.} \\
& \quad \quad \text{goon} = \text{.false.} \\
& \quad \text{else} \\
& \quad \quad \text{if (iter > maxiter) then} \\
& \quad \quad \quad \text{goon} = \text{.false.} \\
& \quad \quad \quad \text{endif} \\
& \quad \text{endif} \\
\text{endo}
\end{align*}
\]

4.2.4 Infinite do loops

A slightly different version of the loop uses triggers an infinite do-loop with the exit mechanism embedded within the body of the loop. The format is do
.
.
.
if (loop_expression_test) then
.exit
endif
endo
4.3 The infamous GoTo statement

In the middle ages of computing, programmers frequently used the infamous GoTo statement to skip executable code. The format is usually of the form `goto labelnb` where `labelnb` is the numeric label of a statement label. For example the executable statements in the quadratic root program can be written as:

```plaintext
if (d < 0.0 ) goto 30
if (d == 0.0) goto 20
10 d = sqrt(d)  
x1 = (-b+d)/(2.0*a)  
x2 = (-b-d)/(2.0*a)  
print *,'x1=’,x1,’x2=’,x2  
goto 40
20 x1 = -b/(2.0*a)  
print *,’x1=x2=’,x1
30 print *,'no real root’
40 stop
```

The above example is a relatively easy read as the code is short and the listing flows mostly downward. A slightly more contorted code can use the `goto` to effect loops as in:

```plaintext
program looping
  implicit none
  integer :: i,is
  is = 0
  i = 0
10 if (i < 10) goto 40
   i = i + 1
   is = is + i
   go to 10
40 print *,’is =’,is
   stop
end program looping
```

This code is already spaghetti code in that the downward flow is interrupted by an upward jolt with the `goto` 10 statement. The logic of the algorithm is obfuscated. The `goto` statement can be easily abused and the resulting code ends up looking
like a pasta dish with entwined coding noodles everywhere. It is to be avoided as much as possible.