1. Repetition (do loops)
2. Conditional processing (if; select)
3. Arrays
4. Functions and loops in different programming languages

1. Repetition (do Loops)

Example 1.1 Different types of do loop

Newton’s method for square roots:
\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{A}{x_n} \right) \rightarrow \sqrt{A} \]

(i) Deterministic do loop – repeat a fixed number of times

```fortran
program newton
  implicit none
  real a                                   ! number to be square-rooted
  real x                                   ! current value of root
  integer n                                ! loop counter
  print *, "Enter a number"                ! input number to be rooted
  read *, a                                ! input number to be rooted
  x = 1.0                                  ! initial value
  do n = 1, 10                             ! fixed number of iterations
    x = 0.5 * ( x + a / x )               ! update value
    print *, x                            ! output current value
  end do
end program newton
```

Example 2.1 Different types of if statement
(ii) Non-deterministic do loops – repeat until some criterion is met.

(a) Using if (...) exit

```fortran
program newton
  implicit none
  real a                           ! number to be square-rooted
  real x, xold                     ! current and previous value
  real change                      ! change during one iteration
  real, parameter :: tolerance = 1.0e-6 ! tolerance for convergence

  print *, "Enter a number"
  read *, a                       ! input number to be rooted
  x = 1.0                         ! initial value

  xold = x                        ! store previous value
  do
    x = 0.5 * ( x + a / x )       ! update value
    print *, x                    ! print current value
    change = abs( (x - xold) / x ) ! fractional change
    if ( change < tolerance ) exit ! criterion for stopping
  end do
end program newton
```

(b) Using do while (...)

```fortran
program newton
  implicit none
  real a                           ! number to be square-rooted
  real x, xold                     ! current and previous value
  real change                      ! change during one iteration
  real, parameter :: tolerance = 1.0e-6 ! tolerance for convergence

  print *, "Enter a number"
  read *, a                       ! input number to be rooted
  x = 1.0                         ! initial value
  change = 1.0                    ! anything big enough to
                                   ! make the first loop run
  do while ( change > tolerance ) ! criterion for continuing
    xold = x                      ! store previous value
    x = 0.5 * ( x + a / x )       ! update value
    print *, x                    ! print current value
    change = abs( (x - xold) / x ) ! fractional change
  end do
end program newton
```
Example 1.2 Summing power series.

\[ \exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \]

Note that each term is not worked out from scratch, but – more efficiently – as a multiple of the previous one:

\[ \frac{x^n}{n!} = \frac{x}{n} \cdot \frac{x^{n-1}}{(n-1)!} \]

program power_series
    implicit none
    real, external :: new_exp ! declare function used
    real value ! number to test
    print *, "Enter a number"
    read *, value
    print *, "Sum of series = ", new_exp( value ) ! our own function
    print *, "Actual exp(x) = ", exp( value ) ! standard function
end program power_series
!
!
real function new_exp( x ) ! Sum a power series for exp(x)
    implicit none
    real x ! argument of function
    integer n ! number of a term
    real term ! a term in the series
    real, parameter :: tolerance = 1.0e-07 ! truncation level
    ! First term
    n = 0; term = 1; new_exp = term
    ! Add successive terms until they become negligible
    do while ( abs( term ) > tolerance ) ! criterion to continue
        n = n + 1 ! index of next term
        term = term * x / n ! new term is multiple of last
        new_exp = new_exp + term ! add to sum
    end do
end function new_exp
!

Warning: the termination criterion used here:

\[ |term| < \text{small number} \]

is OK here because this power series can be shown to converge. This is not always a sufficient condition for convergence; for example, the harmonic series

\[ \sum \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \]

actually diverges, even though the terms tend to zero.
2. Conditional Processing (if; select)

Example 2.1 Comparing if and select.

```fortran
program exam
  implicit none
  integer mark
  character grade

  do
    write( *, "( 'Enter mark (negative to end): ')", advance = "no" )
    read *, mark
    if ( mark < 0 ) stop ! stop program with negative value
      if ( mark >= 70 ) then
        grade = 'A'
      else if ( mark >= 60 ) then
        grade = 'B'
      else if ( mark >= 50 ) then
        grade = 'C'
      else if ( mark >= 40 ) then
        grade = 'D'
      else if ( mark >= 30 ) then
        grade = 'E'
      else
        grade = 'F'
      end if
    end do
    print *, "Grade is ", grade
  end do
end program exam

program exam
  implicit none
  integer mark
  character grade

  do
    write( *, "( 'Enter mark (negative to end): ')", advance = "no" )
    read *, mark
    if ( mark < 0 ) stop ! stop program with negative value
      select case ( mark )
        case ( 70: )
          grade = 'A'
        case ( 60:69 )
          grade = 'B'
        case ( 50:59 )
          grade = 'C'
        case ( 40:49 )
          grade = 'D'
        case ( 30:39 )
          grade = 'E'
        case ( :29 )
          grade = 'F'
      end select
    end do
    print *, "Grade is ", grade
  end do
end program exam
```
3. Arrays

Example 3.1 Illustrating operations element-by-element with arrays.

```fortran
program matrix
implicit none
real, dimension(3,3) :: a, b, c ! declare size of a, b and c
! real a(3,3), b(3,3), c(3,3) ! alternative dimension statement
real PI ! the number pi
integer i, j ! counters
character (len=*) , parameter :: fmt = "( a, 3(/, 3(1x, f8.3)), / )" ! format string for output

! Basic initialisation of matrices by assigning all values (inefficient)
a(1,1) = 1.0;   a(1,2) = 2.0;   a(1,3) = 3.0
a(2,1) = 4.0;   a(2,2) = 5.0;   a(2,3) = 6.0
a(3,1) = 7.0;   a(3,2) = 8.0;   a(3,3) = 9.0
b(1,1) = 10.0;  b(1,2) = 20.0;  b(1,3) = 30.0
b(2,1) = 40.0;  b(2,2) = 50.0;  b(2,3) = 60.0
b(3,1) = 70.0;  b(3,2) = 80.0;  b(3,3) = 90.0

! Alternative initialisation using data statements (note order)
! data a / 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 /
! data b / 10.0, 40.0, 70.0, 20.0, 50.0, 80.0, 30.0, 60.0, 90.0 /

! Alternative initialisation computing each element of a
! do j = 1, 3
!     do i = 1, 3
!        a(i,j) = (i - 1) * 3 + j
!     end do
! end do
! Then whole-array operation for b
! b = 10.0 * a

! Write out matrices (using implied do loops)
write( *, fmt ) "a", ( ( a(i,j), j = 1, 3 ), i = 1, 3 )
write( *, fmt ) "b", ( ( b(i,j), j = 1, 3 ), i = 1, 3 )

! Matrix sum
c = a + b
write( *, fmt ) "a+b", ( ( c(i,j), j = 1, 3 ), i = 1, 3 )

! "element-by-element" multiplication
c = a * b
write( *, fmt ) "a*b", ( ( c(i,j), j = 1, 3 ), i = 1, 3 )

! "Proper" matrix multiplication
c = matmul(a, b)
write( *, fmt ) "matmul(a,b)", ( ( c(i,j), j = 1, 3 ), i = 1, 3 )

! Some operation applied to all elements of a matrix
PI = 4.0 * atan( 1.0 )
c = sin( b * PI / 180.0 )
write( *, fmt ) "sin(b)", ( ( c(i,j), j = 1, 3 ), i = 1, 3 )
end program matrix
```
**Functions and Loops in Different Programming Languages**

Consider a function

\[ \text{sumsq}(n) = 1^2 + 2^2 + \cdots + n^2 \]

(This could be worked out analytically, but the intention is to illustrate loops.)

**Fortran**

```fortran
integer function sumsq( n )
    integer n                     ! declare argument type
    integer i                     ! declare internal variables
    sumsq = 0                     ! initialise sum
    do i = 1, n                   ! start of loop
        sumsq = sumsq + i * i      ! add to sum
    end do                       ! end of loop
end function sumsq
```

**Visual Basic**

```vbnet
Function sumsq(n As Integer) As Integer
    Dim i As Integer             ' declare internal variables
    sumsq = 0                     ' initialise sum
    For i = 1 To n               ' range of loop counter
        sumsq = sumsq + i * i     ' add to sum
    Next i                       ' end of loop
End Function
```

**C++**

```cpp
int sumsq( int n )
{
    int i, value;                  // declare internal variables
    value = 0;                     // initialise sum
    for ( i = 1; i <= n; i++ )    // start of loop
    {
        value += i * i;            // add to sum
    }                               // end of loop
    return value;
}
```

Actually, the last is a little bit naughty (but typical of C++ shorthand):

- `i++` is equivalent to `i = i + 1`
- `value += i * i` is equivalent to `value = value + i * i`