Chapter 5: Arrays

An advantage fortran has over other programming languages is the ease with which it handles arrays. Because arrays are so easy to handle, fortran is an ideal choice when writing code to handle vector and matrix operations. Fortran also offers a great deal of freedom when indexing arrays, but keep in mind that the default initial index is 1.

It is worth noting that ***fortran stores array information by column***, often referred to as ***column-major.*** Knowing this helps when printing multi-dimensional arrays. Also, understanding how a programming language manages memory can help you write faster, more efficient code. We won't go into any more detail on this topic, but it is something to keep in mind when writing programs later.

Note, now that we know how to use modules, that, ***if necessary, all the code examples from this point on will use the*** `Constants` ***module.***

5.1 Array Basics

We'll give some examples of how to declare, manipulate, and print arrays. But first we list some of the useful intrinsic functions we can use with arrays:

- SIZE – Returns the number of elements in an array.
- SHAPE – Returns the number of elements in each direction in an integer vector.
- LBOUND – Returns the lower index of each dimension of an array.
- UBOUND – Returns the upper index of each dimension of an array.
- MAXVAL – Returns the largest value in the array.
- MINVAL – Returns the smallest value in the array.
- MAXLOC – Returns the location of the largest value in an array.
- MINLOC – Returns the location of the smallest value in an array.
- SUM – Returns the sum of the elements of an array
- TRANSPOSE – Returns the transpose of a matrix.
- DOT_PRODUCT – Returns the dot product of two vectors
- MATMUL – Returns the product of two matrices, dimensions must be consistent, i.e., (M, K) and (K, N)

arrayExampleA.f90

```
PROGRAM arrayExampleA
   USE Constants
   IMPLICIT NONE
! Can use the DIMENSION command or put an array's size after the variable name
   REAL(KIND=RP),DIMENSION(0:3)  :: array1
   REAL(KIND=RP),DIMENSION(4)    :: array2 ! same as 1:4
   REAL(KIND=RP),DIMENSION(-2:2) :: array3
   REAL(KIND=RP)                 :: array4(-1:1,0:2)
   REAL(KIND=RP)                 :: x,y
   INTEGER                       :: j,k

   x = 3.44_RP
   y = 1.25_RP

   WRITE(*,*)'size of array1',SIZE(array1)
   WRITE(*,*)'size of array2',SIZE(array2)
```

```fortran
   WRITE(*,*)'size of array3',SIZE(array3)
   WRITE(*,*)'size of array4',SIZE(array4)
   WRITE(*,*)
   WRITE(*,*)'shape of array1',SHAPE(array1)
   WRITE(*,*)'shape of array2',SHAPE(array2)
   WRITE(*,*)'shape of array3',SHAPE(array3)
   WRITE(*,*)'shape of array4',SHAPE(array4)
   WRITE(*,*)
   WRITE(*,*)'lower index of array1',LBOUND(array1)
   WRITE(*,*)'lower index of array2',LBOUND(array2)
   WRITE(*,*)'lower index of array3',LBOUND(array3)
   WRITE(*,*)'lower indices of array4',LBOUND(array4)
   WRITE(*,*)
   WRITE(*,*)'upper index of array1',UBOUND(array1)
   WRITE(*,*)'upper index of array2',UBOUND(array2)
   WRITE(*,*)'upper index of array3',UBOUND(array3)
   WRITE(*,*)'upper indices of array4',UBOUND(array4)
   WRITE(*,*)
! Syntax to assign specific values to an array
   array1 = (/ -2.0_RP, 6.0_RP, pi, 1.1_RP /)
   array2 = (/ x-y, x+y, SIN(x)-EXP(y), y**x /)
! Assign values in a loop
   DO j = -2,2
      array3(j) = x**j
   END DO
! Can do array slicing using a colon, we assign each column of an array
   array4(-1,:) = (/  1.0_RP,-0.5_RP ,12.0_RP  /)
   array4(0,:)  = (/ -3.0_RP, 0.5_RP , 1.11_RP /)
   array4(1,:)  = (/  2.0_RP,-0.35_RP, 8.8_RP  /)
! Can print the array without loops
   WRITE(*,*)'array3 w/o loop',array3
   WRITE(*,*)'array4 w/o loop',array4
   WRITE(*,*)
! But it is always more readable if you print with loops and slice
   WRITE(*,*)'array3 w/loop'
   DO j = -2,2
      WRITE(*,*)array3(j)
   END DO
   WRITE(*,*)'array4 w/loop'
   DO j = -1,1
      WRITE(*,*),array4(j,:)
   END DO

   WRITE(*,*)'max of array1',MAXVAL(array1)
   WRITE(*,*)'location of max in array2',MAXLOC(array2)
   WRITE(*,*)'min of array3',MINVAL(array3)
   WRITE(*,*)'location of min in array4',MINLOC(array4)
END PROGRAM arrayExampleA
```

We compile and run the first array example and show the results. Note that the way you print arrays can turn gibberish in a nice 2D output.

arrayExampleA.f90 - Output

```
 size of array1           4
 size of array2           4
 size of array3           5
 size of array4           9
```

```
shape of array1           4
shape of array2           4
shape of array3           5
shape of array4           3           3

lower index of array1           0
lower index of array2           1
lower index of array3          -2
lower indices of array4          -1           0

upper index of array1           3
upper index of array2           4
upper index of array3           2
upper indices of array4           1           2

array3 w/o loop  8.45051379123850782E-002  0.29069767441860467        1.0000000000000000
  3.4399999999999999        11.833599999999999
array4 w/o loop   1.0000000000000000       -3.0000000000000000        2.0000000000000000
-0.50000000000000000        0.50000000000000000       -0.34999999999999998        12.0000000000
00000        1.1100000000000001        8.8000000000000007

array3 w/loop
 8.45051379123850782E-002
 0.29069767441860467
  1.0000000000000000
  3.4399999999999999
  11.833599999999999
array4 w/loop
  1.0000000000000000      -0.50000000000000000        12.000000000000000
 -3.0000000000000000       0.50000000000000000        1.1100000000000001
  2.0000000000000000      -0.34999999999999998        8.8000000000000007
max of array1   6.0000000000000000
location of max in array2           2
min of array3  8.45051379123850782E-002
location of min in array4           2           1
```

Next, we provide an example when we manipulate arrays which represent matrices. In the next example we also write a quick subroutine to help print the arrays in the style we want.

arrayExampleB.f90

```fortran
PROGRAM arrayExampleB
   IMPLICIT NONE
   INTEGER,DIMENSION(2,2) :: A,B
   INTEGER                :: i,j

   A(1,:) = (/ 1 , 2 /)
   A(2,:) = (/ 3 , 4 /)
   B      = 1 ! sets every element in B to 1

   WRITE(*,*)'A='
   CALL PrintIntegerMatrix(A,2,2)
   WRITE(*,*)'B='
   CALL PrintIntegerMatrix(B,2,2)

   B(2,2) = B(2,2) + 3
   WRITE(*,*)'Add 3 to B(2,2)'
   CALL PrintIntegerMatrix(B,2,2)
```

```
   A = A*2
   WRITE(*,*)'A=A*2'
   CALL PrintIntegerMatrix(A,2,2)

   A = A**2 - 1 ! This works for intrinsics like SIN, EXP, etc... as well
   WRITE(*,*)'A=A**2-1'
   CALL PrintIntegerMatrix(A,2,2)

   A = A+B
   WRITE(*,*)'A+B'
   CALL PrintIntegerMatrix(A,2,2)

   A = MATMUL(A,B)
   WRITE(*,*)'A=AB'
   CALL PrintIntegerMatrix(A,2,2)

   A = TRANSPOSE(A)
   WRITE(*,*)'A = A^T'
   CALL PrintIntegerMatrix(A,2,2)

   WRITE(*,*)'Dot product of col 1 of A with col 2 of A',DOT_PRODUCT(A(:,1),A(:,2))
   WRITE(*,*)'Dot product of row 1 of A with row 2 of A',DOT_PRODUCT(A(1,:),A(2,:))
END PROGRAM arrayExampleB

SUBROUTINE PrintIntegerMatrix(mat,N,M)
   IMPLICIT NONE
   INTEGER,INTENT(IN) :: N,M
   INTEGER,INTENT(IN) :: mat(N,M)
! Local Variable
   INTEGER            :: i

   DO i = 1,N
      WRITE(*,*)mat(i,:)
   END DO
   WRITE(*,*)
   RETURN
END SUBROUTINE PrintIntegerMatrix
```

Again, we provide the output of the second array example for instructive purposes.

arrayExampleB.f90 - Commands and Output

```
gfortran arrayExampleB.f90 -o arrayB
./arrayB
 A=
           1           2
           3           4

 B=
           1           1
           1           1

 Add 3 to B(2,2)
           1           1
           1           4

 A=A*2
           2           4
```

```
           6           8

A=A**2-1
           3          15
          35          63

A+B
           4          16
          36          67

A=AB
          20          68
         103         304

A = A^T
          20         103
          68         304

Dot product of col 1 of A with col 2 of A       22732
Dot product of row 1 of A with row 2 of A       32672
```

5.2 Example: Interpolation

We wish to interpolate a function f given two set of points $\{x_i\}_{i=0}^{N}$ and $\{y_i\}_{i=0}^{N}$, where $y_i = f(x_i)$. Omitting some detail, we know that Newton divided difference provide an efficient way to specify an interpolation rule. Consider the two sets of points (x_0, y_0) and (x_1, y_1). The polynomial of order 1 is given by the straight line:

$$p_1(x) = y_0 + \underbrace{\frac{y_1 - y_0}{x_1 - x_0}}_{y[x_0,x_1]}(x - x_0).$$

Notice that $p_1(x_0) = y_0$ and $p_1(x_1) = y_1$. A series of polynomials may then be constructed depending on their order, i.e., depending on the number of points available.

$$\begin{aligned}
p_0(x) &= y_0, \\
p_1(x) &= y_0 + y[x_0, x_1](x - x_0), \\
p_2(x) &= y_0 + y[x_0, x_1](x - x_0) + y[x_0, x_1, x_2](x - x_0)(x - x_1) \\
&\vdots
\end{aligned}$$

Then the interpolating polynomial,P_N, is given by the sum of the preceding sequence, i.e.,

$$P_N(x) = \sum_{i=0}^{N} c_i \prod_{j=0}^{i-1} (x - x_j)$$

Next, we detail the procedures necessary to have a proper interpolation routine based on Newton divided differences. How we obtained and wrote the pseudocode code on the next page may not necessary make sense. Don't fret! The topic of interpolation, using Newton divided differences as well as other methods, is covered in great detail when you take Dr. Gallivan's FCM II course. For now, be assured that the procedures are correct, you'll learn exactly why later. We will collect all our interpolation procedures into a module that is then used by a main program.

Algorithm 1: *Newton Coefficients*: Compute interpolation coefficients from Newton divided differences.

Procedure Newton Coefficients
Input: $\{x_i\}_{i=0}^N, \{y_i\}_{i=0}^N, N$

$c_0 \leftarrow y_0$
for $k = 1$ **to** N **do**
  $d \leftarrow x_k - x_{k-1}$
  $u \leftarrow c_{k-1}$
  for $j = k - 2$ **to** 0 **do**
    $u \leftarrow u(x_k - x_j) + c_j$
    $d \leftarrow d(x_k - x_j)$

  $c_k \leftarrow (y_k - u)/d$

Output: $\{c_i\}_{i=0}^N$

End Procedure Newton Coefficients

Algorithm 2: *Newton Interpolating Polynomial*: Evaluate interpolant at a point

Procedure Newton Interpolating Polynomial
Input: $x, \{x_i\}_{i=0}^N, \{c_i\}_{i=0}^N, N$

$P(x) \leftarrow 0$
for $k = 0$ **to** N **do**
  $p \leftarrow 1$
  for $j = 0$ **to** $k - 1$ **do**
    $p \leftarrow p(x_k - x_j)$

  $P(x) \leftarrow P(x) + c_i p$

Output: $P(x)$

End Procedure Newton Interpolating Polynomial

Algorithm 3: *Interpolated Values*: Interpolate to a set of nodes stored in $\mathbf{x}^{new}$.

Procedure Interpolated Values
Uses: **Algorithm 5**
Input: $\{c_i\}_{i=0}^N, \{x_i\}_{i=0}^N, \{x_i^{new}\}_{i=0}^N, N$

for $k = 0$ **to** N **do**
  $y_k^{new} \leftarrow$ Newton Interpolating Polynomial$(x_k^{new}, \{x_i\}_{i=0}^N, \{c_i\}_{i=0}^N, N)$

Output: $\{y_i^{new}\}_{i=0}^N$

End Procedure Interpolated Values

We amalgamate the interpolation procedures and convenience procedures for printing results to a file, for plotting purposes, into a module.

interpolationRoutines.f90

```
MODULE InterpolationRoutines
   USE Constants
   IMPLICIT NONE
```

```
! No variable declarations
CONTAINS
   SUBROUTINE Interpolate(C,X,Xnew,Ynew,N)
      IMPLICIT NONE
      INTEGER                      ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N)  ,INTENT(IN)  :: C,X
      REAL(KIND=RP),DIMENSION(0:2*N),INTENT(OUT) :: Xnew,Ynew
      REAL(KIND=RP),EXTERNAL                     :: Poly_Interpolant

      INTEGER                                    :: i

      DO i = 0,2*N
         Xnew(i) = i*(2.0_RP*pi)/N
         Poly_Interpolant(Xnew(i),X,C,N,Ynew(i))
      END DO
      RETURN
   END SUBROUTINE Interpolate
!
   SUBROUTINE NewtonCoeff(X,Y,C,N)
      IMPLICIT NONE
      INTEGER                     ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN)  :: X,Y
      REAL(KIND=RP),DIMENSION(0:N),INTENT(OUT) :: C
! Local Variables
      REAL(KIND=RP)                            :: d,u
      INTEGER                                  :: i,j

      C(0) = Y(0)
      DO j = 1,N
         d = X(j) - X(j-1)
         u = C(j-1)
         DO i = j-2,0,-1
            u = u*(X(j)-X(i))+C(i)
            d = d*(X(j)-X(i))
         END DO
         C(j) = (Y(j)-u)/d
      END DO
      RETURN
   END SUBROUTINE NewtonCoeff
!
   SUBROUTINE Poly_Interpolant(x,nodes,coeff,N,y)
      IMPLICIT NONE
      INTEGER                     ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN)  :: nodes,coeff
      REAL(KIND=RP)               ,INTENT(IN)  :: x
      REAL(KIND=RP)               ,INTENT(OUT) :: y
! Local variables
      INTEGER       :: i,j
      REAL(KIND=RP) :: temp

      y = 0.0_RP
      DO i = 0,N
         temp = 1.0_RP
         DO j = 0,i-1
            temp = temp*(x-nodes(j))
         END DO
         y = y + coeff(i)*temp
```

```fortran
      END DO
   END SUBROUTINE Poly_Interpolant
!
   SUBROUTINE ReadInArrays(X,Y,N)
      IMPLICIT NONE
      INTEGER                       ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(OUT) :: X,Y
! Local variables
      INTEGER :: i

      OPEN(12,FILE="poly.dat")
      DO i = 0,N
         READ(12,*)X(i),Y(i)
      END DO
      RETURN
   END SUBROUTINE ReadInArrays
!
   SUBROUTINE WriteData(X,Y,N)
      IMPLICIT NONE
      INTEGER                       ,INTENT(IN) :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN) :: X,Y
! Local Variables
      INTEGER :: i

      OPEN(12,FILE='output.dat')
      DO i = 0,N
         WRITE(12,*)X(i),Y(i)
      END DO
      CLOSE(12)
   END SUBROUTINE WriteData
END MODULE InterpolationRoutines
```

With the interpolation module in place, we have the tools necessary to write and compile a driver program

interpolationDriver.f90

```fortran
PROGRAM interpolationDriver
   USE InterpolationRoutines
   IMPLICIT NONE
   INTEGER                           :: N = 12
   REAL(KIND=RP),DIMENSION(0:N)      :: X,Y,c
   REAL(KIND=RP),DIMENSION(0:2*N)   :: newX,newY
   INTEGER                           :: i

   OPEN(13,FILE='poly.dat')
   DO i = 0,N
      X(i) = i*(2.0_RP*pi)/N
      Y(i) = COS(X(i))
      WRITE(13,*)X(i),Y(i)
   END DO
   CLOSE(13)

   CALL ReadInArrays(X,Y,N)
   CALL NewtonCoeff(X,Y,c,N)
   CALL Interpolate(c,X,newX,newY,N)
   CALL WriteData(newX,newY,N)
END PROGRAM interpolationDriver
```

We compile and run the interpolation program and plot the result on the next page. The result passes the so called "eyeball" norm, because the answer looks correct. To fully verify the code functions properly we would compare the numerical errors, using multiple functions, to theoretical predictions and ensure consistency (something we would do in an FCM II code).

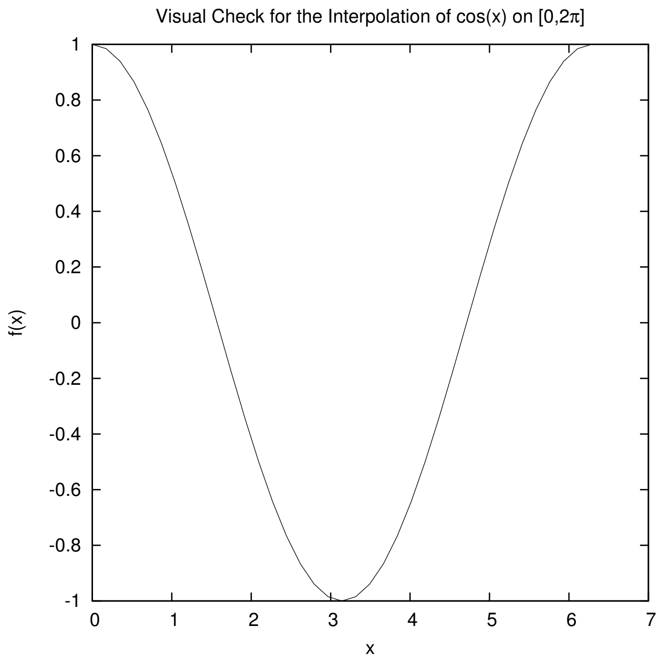

Figure 1: Newton interpolating polynomial run with $N = 12$.