Programming Techniques (Master in Astrophysics - ULL)

Ángel de Vicente

September 2019

Public draft - open for comments

These are the notes to be followed in the course “Programming Techniques” (“Técnicas de Programación”) of the Master in Astrophysics (ULL).

The objective of the course is to learn some programming techniques necessary in many scientific codes. In particular, we will study about dynamic data structures and MPI parallel programming with Fortran.

All the material taught in the course will be motivated by a sample N-body problem. We will start by developing a naïve and serial code; then we will study about the much more efficient Barnes-Hut algorithm and in order to implement this algorithm we will have to learn about dynamic data structures, recursion, trees, lists, etc. Once a Barnes-Hut serial implementation is finished, we will focus on learning the basics of parallel programming, in this case using the MPI library, and on parallelizing the Barnes-Hut version of our code.

We will also touch briefly on code debugging and profiling (both in serial and in parallel codes) and on other parallel programming models (OpenMP, CUDA, OpenACC) and how to use them together with MPI.

Contents

PART I

NAÏVE N-BODY SOLUTION WITH FORTRAN

Chapter 1

N-body problem formulation

1.1 Introduction

A nice introduction to the N-Body problem can be found in the documentation for the XStar[1] code: “The “N-Body problem” is the problem of trying to find how n objects will move under one of the physical forces, such as gravity.” [1]. The XStar code (written in the C language) solves what we will cover in this course and more, except for the parallelization part. Figure 1.1 shows an example screenshot of a simulation run with XStar.

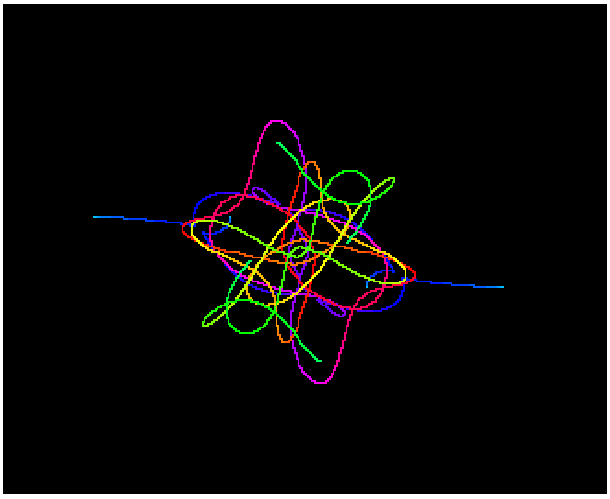


Figure 1.1: Sample XStar configuration simulation (from http://www.schlitt.net/xstar/screen_shots.html)

1.2 N-body equations

The equations to solve the N-body problem are very simple. See, for example, sections 1.2 and 1.3 of the Xstar guide (http://www.schlitt.net/xstar/n-body/nb-8.html), and page 65 of “Moving Stars Around”.

1. http://www.schlitt.net/xstar/

1.2 Newtonian Physics

Newton laid out the formulas needed to solve the N-body problem for gravity some 300 years ago. They are really fairly simple and the formulas are: (16:762-85,17:78-83)

x	The position of the body.
$v = x'$	Velocity is the rate of change of the position.
$a = v' = x''$	Acceleration is the rate of change of the velocity and is also the second derivative of the position.
$F = ma$	Force equals the mass times the acceleration.
$F = \frac{Gm_1m_2}{r_{12}^2}$	The force of gravity between two bodies (of mass m_1 and m_2) is equal to a constant G times the product of the masses, divided by the square of the distance r_{12} between the bodies. Technically, the formula looks more like $\vec{F} = \frac{Gm_1m_2}{\vec{r}_{12}^2}\frac{\vec{r}_{12}}{\lvert\vec{r}_{12}\rvert}$ where $\vec{r}_{12}$ is the vector between the two bodies and $\lvert\vec{r}_{12}\rvert$ is the length of the vector. That is, the force is projected along the line connecting the two bodies.

Figure 1.2: p.8 Guía Xstar

$$\mathbf{a}_i = G\sum_{\substack{j=1 \\ j\neq i}}^{N} \frac{M_j}{r_{ji}^2}\hat{\mathbf{r}}_{ji}$$

Figure 1.3: p. 65 “Moving Stars Around”

1.3 Time integration

To integrate in time, one can use several methods. The simplest one, of order 1 is Forward-Euler, as can be seen in page 24 of “Moving Stars Around”. The one we are going to use here is of order 2: the leapfrom algorithm (see pages 55-56 of “Moving Stars Around”).

$$
\begin{aligned}
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i dt
\end{aligned}
$$

Figure 1.4: p.24 “Moving Stars Around”

4.1 Two Ways to Write the Leapfrog

The name *leapfrog* comes from one of the ways to write this algorithm, where positions and velocities ‘leap over’ each other. Positions are defined at times $t_i, t_{i+1}, t_{i+2}, \ldots$, spaced at constant intervals dt, while the velocities are defined at times halfway in between, indicated by $t_{i-1/2}, t_{i+1/2}, t_{i+3/2}, \ldots$, where $t_{i+1} - t_{i+1/2} = t_{i+1/2} - t_i = dt/2$. The leapfrog integration scheme then reads:

$$
\begin{aligned}
\mathbf{r}_i &= \mathbf{r}_{i-1} + \mathbf{v}_{i-1/2} dt && (4.1) \\
\mathbf{v}_{i+1/2} &= \mathbf{v}_{i-1/2} + \mathbf{a}_i dt && (4.2)
\end{aligned}
$$

Note that the accelerations $\mathbf{a}$ are defined only on integer times, just like the positions,

Figure 1.5: p.55 “Moving Stars Around”

while the velocities are defined only on half-integer times. This makes sense, given that $\mathbf{a}(\mathbf{r}, \mathbf{v}) = \mathbf{a}(\mathbf{r})$: the acceleration on one particle depends only on its position with respect to all other particles, and not on its or their velocities. Only at the beginning of the integration do we have to set up the velocity at its first half-integer time step. Starting with initial conditions $\mathbf{r}_0$ and $\mathbf{v}_0$, we take the first term in the Taylor series expansion to compute the first leap value for $\mathbf{v}$:

$$\mathbf{v}_{1/2} = \mathbf{v}_0 + \mathbf{a}_0 dt/2. \tag{4.3}$$

We are then ready to apply Eq. 4.1 to compute the new position $\mathbf{r}_1$, using the first leap value for $\mathbf{v}_{1/2}$. Next we compute the acceleration $\mathbf{a}_1$, which enables us to compute the second leap value, $\mathbf{v}_{3/2}$, using Eq. 4.2, and so on.

A second way to write the leapfrog looks quite different at first sight. Defining all quantities only at integer times, we can write:

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2 \qquad (4.4) \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1}) dt/2 \qquad (4.5) \end{aligned}$$

Figure 1.6: p.56 “Moving Stars Around”

Chapter 2

Basic Fortran for first N-body implementation

2.1 Basic Fortran

In this chapter we will just cover the basic concepts of Fortran. Fortran is a complex language, so in this chapter we will only cover the surface of the language, only the necessary minimum to be able to write a first implementation of the N-body problem.

The following slides are taken from a 5-day intentive course on Fortran, given by the University of Liverpool.

THE UNIVERSITY
of LIVERPOOL

Fortran 90 Programming

(5 Day Course)

—

Dr. A C Marshall (funded by JISC/NTI)

with acknowledgements to Steve Morgan and Lawrie Schonfelder.

Fortran 90 New features

Fortran 90 supports,

1. free source form;
2. array syntax and many more (array) intrinsics;
3. dynamic storage and pointers;
4. portable data types (`KIND`s);
5. derived data types and operators;
6. recursion;
7. `MODULE`s

- procedure interfaces;
- enhanced control structures;
- user defined generic procedures;
- enhanced I/O.

Example

Example Fortran 90 program:

```
MODULE Triangle_Operations
 IMPLICIT NONE
CONTAINS
 FUNCTION Area(x,y,z)
  REAL :: Area       ! function type
  REAL, INTENT( IN ) :: x, y, z
  REAL :: theta, height
  theta=ACOS((x**2+y**2-z**2)/(2.0*x*y))
  height=x*SIN(theta); Area=0.5*y*height
 END FUNCTION Area
END MODULE Triangle_Operations

PROGRAM Triangle
 USE Triangle_Operations
 IMPLICIT NONE
  REAL :: a, b, c, Area
  PRINT*, 'Welcome, please enter the&
         &lengths of the 3 sides.'
  READ*, a, b, c
  PRINT*,'Triangle''s area: ',Area(a,b,c)
END PROGRAM Triangle
```

Coding Style

It is recommended that the following coding convention is adopted:

- *always* use `IMPLICIT NONE`.
- Fortran 90 keywords, intrinsic functions and user defined entities should be in upper case,
- other user entities should be in lower case but may start with a capital letter.
- indentation should be 1 or 2 spaces and should be applied to the bodies of program units, control blocks, `INTERFACE` blocks, etc.
- the names of program units are always included in their `END` statements,
- argument keywords are always used for optional arguments,

Please note: In order that a program fits onto a slide these rules are sometimes relaxed here.

Source Form

Free source form:

- 132 characters per line;
- '!' comment initiator;
- '&' line continuation character;
- ';' statement separator;
- significant blanks.

Example,

```
PRINT*, "This line is continued &
        &On the next line"; END ! of program
```

Intrinsic Types

Fortran 90 has three broad classes of object type,

- character;
- boolean;
- numeric.

these give rise to six simple intrinsic types, known as default types,

```
CHARACTER          :: sex  ! letter
CHARACTER(LEN=12)  :: name ! string
LOGICAL            :: wed  ! married?
REAL               :: height
DOUBLE PRECISION   :: pi   ! 3.14...
INTEGER            :: age  ! whole No.
COMPLEX            :: val  ! x + iy
```

Literal Constants

A literal constant is an entity with a fixed value:

```
12345      ! INTEGER
1.0        ! REAL
-6.6E-06   ! REAL: -6.6*10**(-6)
.FALSE.    ! LOGICAL
.TRUE.     ! LOGICAL
"Mau'dib"  ! CHARACTER
'Mau''dib' ! CHARACTER
```

Note,

- there are only two `LOGICAL` values;
- `REAL`s contain a decimal point, `INTEGER`s do not,
- `REAL`s have an exponential form
- character literals delimited by `"` and `'`;
- two occurrences of the delimiter inside a string produce one occurrence on output;
- there is only a finite range of values that numeric literals can take.

Implicit Typing

Undeclared variables have an implicit type,

- if first letter is `I, J, K, L, M` or `N` then type is `INTEGER`;
- any other letter then type is `REAL`s.

Implicit typing is potentially very dangerous and should **always** be turned off by adding:

```
    IMPLICIT NONE
```

as the first line after any `USE` statements.

Consider,

```
    DO 30 I = 1.1000
      ...
 30 CONTINUE
```

in fixed format with implicit typing this declares a `REAL` variable `DO30I` and sets it to `1.1000` instead of performing a loop 1000 times!

Numeric and Logical Declarations

With `IMPLICIT NONE` variables must be declared. A simplified syntax follows,

< *type* > [,< *attribute*-list >] ::< *variable*-list >&
[=< *value* >]

The following are all valid declarations,

```
REAL                        :: x
INTEGER                     :: i, j
LOGICAL, POINTER            :: ptr
REAL, DIMENSION(10,10)      :: y, z
INTEGER                     :: k = 4
```

The `DIMENSION` attribute declares an array (10×10).

Constants (Parameters)

Symbolic constants, oddly known as *parameters* in Fortran, can easily be set up either in an attributed declaration or parameter statement,

```
REAL, PARAMETER :: pi = 3.14159
CHARACTER(LEN=*), PARAMETER :: &
                  son = 'bart', dad = "Homer"
```

`CHARACTER` constants can assume their length from the associated literal (`LEN=*`).

Parameters should be used:

- if it is known that a variable will only take one value;
- for legibility where a 'magic value' occurs in a program such as π;
- for maintainability when a 'constant' value could feasibly be changed in the future.

Initialisation

Variables can be given initial values:

- can use *initialisation expressions*,
- may only contain PARAMETERs or literals.

```
REAL              :: x, y =1.0D5
INTEGER           :: i = 5, j = 100
CHARACTER(LEN=5) :: light = 'Amber'
CHARACTER(LEN=9) :: gumboot = 'Wellie'
LOGICAL  :: on = .TRUE., off = .FALSE.
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL :: circum = 2 * pi * radius
```

`gumboot` will be padded, to the right, with blanks.

In general, intrinsic functions *cannot* be used in initialisation expressions, the following can be: `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, `TRIM`, `LBOUND`, `UBOUND`, `SHAPE`, `SIZE`, `KIND`, `LEN`, `BIT_SIZE` and numeric inquiry intrinsics, for, example, `HUGE`, `TINY`, `EPSILON`.

Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, for example, `+`, `//` and `.AND.`,

The following are valid expressions,

- `NumBabiesBorn+1` — numeric valued
- `"Ward "//Ward` — character valued
- `TimeSinceLastBirth .GT. MaxTimeTwixtBirths` — logical valued

Expressions can be used in many contexts and can be of any intrinsic type.

Assignment

Assignment is defined between all expressions of the same type:

Examples,

```
a = b
c = SIN(.7)*12.7  ! SIN in radians
name = initials//surname
bool = (a.EQ.b.OR.c.NE.d)
```

The LHS is an object and the RHS is an expression.

Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- `**` exponentiation, dyadic operator, for example, `10**2`, (evaluated right to left);
- `*` and `/` multiply and divide, dyadic operators, for example, `10*7/4`;
- `+` and `-` plus and minus or add and subtract, monadic and dyadic operators, for example, `10+7-4` and `-3`;

Can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of `**` must be scalar.

Example,

```
a = b - c
f = -3*6/5
```

Relational Operators

The following *relational operators* deliver a `LOGICAL` result when combined with numeric operands,

`.GT.`	`>`	greater than
`.GE.`	`>=`	greater than or equal to
`.LE.`	`<=`	less than or equal to
`.LT.`	`<`	less than
`.NE.`	`/=`	not equal to
`.EQ.`	`==`	equal to

For example,

```
bool = i .GT. j
boule = i > j
IF (i .EQ. j) c = D
IF (i == j)   c = D
```

When using real-valued expressions (which are approximate) `.EQ.` and `.NE.` have no real meaning.

```
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```

Intrinsic Logical Operations

A `LOGICAL` or boolean expression returns a `.TRUE.` / `.FALSE.` result. The following are valid with `LOGICAL` operands,

- `.NOT.` — `.TRUE.` if operand is `.FALSE.`.
- `.AND.` — `.TRUE.` if both operands are `.TRUE.`;
- `.OR.` — `.TRUE.` if at least one operand is `.TRUE.`;
- `.EQV.` — `.TRUE.` if both operands are the same;
- `.NEQV.` — `.TRUE.` if both operands are different.

For example, if `T` is `.TRUE.` and `F` is `.FALSE.`

- `.NOT. T` is `.FALSE.`, `.NOT. F` is `.TRUE.`.
- `T .AND. F` is `.FALSE.`, `T .AND. T` is `.TRUE.`.
- `T .OR. F` is `.TRUE.`, `F .OR. F` is `.FALSE.`.
- `T .EQV. F` is `.FALSE.`, `F .EQV. F` is `.TRUE.`.
- `T .NEQV. F` is `.TRUE.`, `F .NEQV. F` is `.FALSE.`.

Control Flow

Control constructs allow the normal sequential order of execution to be changed.

Fortran 90 supports:

- conditional execution statements and constructs, (`IF` ... and `IF` ... `THEN` ... `ELSE` ... `END IF`);
- loops, (`DO` ... `END DO`);
- multi-way choice construct, (`SELECT CASE`);

`IF` Statement

Example,

```
IF (bool_val) A = 3
```

The basic syntax is,

`IF(`< *logical-expression* >`)`< *exec-stmt* >

If < *logical-expression* > evaluates to `.TRUE.` then execute < *exec-stmt* > otherwise do not.

For example,

```
IF (x .GT. y) Maxi = x
```

means 'if `x` is greater than `y` then set `Maxi` to be equal to the value of `x`'.

More examples,

```
IF (a*b+c <= 47) Boolie = .TRUE.
IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```

`IF ... THEN ... ELSE` Construct

The block-`IF` is a more flexible version of the single line `IF`. A simple example,

```
IF (i .EQ. 0) THEN
 PRINT*, "I is Zero"
ELSE
 PRINT*, "I is NOT Zero"
ENDIF
```

note the how indentation helps.

Can also have one or more `ELSEIF` branches:

```
IF (i .EQ. 0) THEN
 PRINT*, "I is Zero"
ELSE IF (i .GT. 0) THEN
 PRINT*, "I is greater than Zero"
ELSE
 PRINT*, "I must be less than Zero"
ENDIF
```

Both `ELSE` and `ELSEIF` are optional.

Conditional Exit Loops

Can set up a DO loop which is terminated by simply jumping out of it. Consider,

```
i = 0
DO
  i = i + 1
  IF (i .GT. 100) EXIT
  PRINT*, "I is", i
END DO
! if i>100 control jumps here
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
I is   3
  ....
I is   100
Loop finished. I now equals   101
```

The EXIT statement tells control to jump out of the current DO loop.

Conditional Cycle Loops

Can set up a DO loop which, on some iterations, only executes a subset of its statements. Consider,

```
i = 0
DO
  i = i + 1
  IF (i >= 50 .AND. i <= 59) CYCLE
  IF (i > 100) EXIT
  PRINT*, "I is", i
END DO
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
  ....
I is   49
I is   60
  ....
I is   100
Loop finished. I now equals   101
```

CYCLE forces control to the **innermost** active DO statement and the loop begins a new iteration.

Named and Nested Loops

Loops can be given names and an `EXIT` or `CYCLE` statement can be made to refer to a particular loop.

```
0|      outa: DO
1|       inna: DO
2|        ...
3|        IF (a.GT.b) EXIT outa  ! jump to line 9
4|        IF (a.EQ.b) CYCLE outa ! jump to line 0
5|        IF (c.GT.d) EXIT inna  ! jump to line 8
6|        IF (c.EQ.a) CYCLE      ! jump to line 1
7|       END DO inna
8|      END DO outa
9|      ...
```

The (optional) name following the `EXIT` or `CYCLE` highlights which loop the statement refers to.

Loop names can only be used once per program unit.

DO ... WHILE Loops

If a condition is to be tested at the top of a loop a DO ... WHILE loop could be used,

```
DO WHILE (a .EQ. b)
 ...
END DO
```

The loop only executes if the logical expression evaluates to .TRUE.. Clearly, here, the values of a or b must be modified within the loop otherwise it will never terminate.

The above loop is functionally equivalent to,

```
DO; IF (a .NE. b) EXIT
 ...
END DO
```

Indexed DO Loops

Loops can be written which cycle a fixed number of times. For example,

```
DO i1 = 1, 100, 1
  ... ! i is 1,2,3,...,100
  ... ! 100 iterations
END DO
```

The formal syntax is as follows,

DO $<$ DO-*var* $>$=$<$ *expr1* $>$,$<$ *expr2* $>$ [,$<$ *expr3* $>$]
    $<$ *exec-stmts* $>$
END DO

The number of iterations, which is evaluated **before** execution of the loop begins, is calculated as

MAX(INT(($<$ *expr2* $>$-$<$ *expr1* $>$+$<$ *expr3* $>$)/$<$ *expr3* $>$),0)

If this is zero or negative then the loop is not executed.

If $<$ *expr3* $>$ is absent it is assumed to be equal to 1.

Examples of Loop Counts

A few examples of different loops,

1. upper bound not exact,

```
loopy: DO i = 1, 30, 2
 ... ! i is 1,3,5,7,...,29
 ... ! 15 iterations
END DO loopy
```

2. negative stride,

```
DO j = 30, 1, -2
 ... ! j is 30,28,26,...,2
 ... ! 15 iterations
END DO
```

3. a zero-trip loop,

```
DO k = 30, 1, 2
 ... ! 0 iterations
 ... ! loop skipped
END DO
```

4. missing stride — assume it is `1`,

```
DO l = 1,30
 ... ! i = 1,2,3,...,30
 ... ! 30 iterations
END DO
```

`SELECT CASE` Construct I

Simple example

```
    SELECT CASE (i)
      CASE (3,5,7)
        PRINT*,"i is prime"
      CASE (10:)
        PRINT*,"i is > 10"
      CASE DEFAULT
        PRINT*, "i is not prime and is < 10"
    END SELECT
```

An `IF .. ENDIF` construct could have been used but a `SELECT CASE` is neater and more efficient. Another example,

```
    SELECT CASE (num)
      CASE (6,9,99,66)
!     IF(num==6.OR. .. .OR.num==66) THEN
        PRINT*, "Woof woof"
      CASE (10:65,67:98)
!     ELSEIF((num >= 10 .AND. num <= 65) .OR. ...
        PRINT*, "Bow wow"
      CASE DEFAULT
!     ELSE
        PRINT*, "Meeeoow"
     END SELECT
!    ENDIF
```

Intrinsic Procedures

Fortran 90 has 113 in-built or *intrinsic* procedures to perform common tasks efficiently, they belong to a number of classes:

- elemental such as:
  - mathematical, for example, `SIN` or `LOG`.
  - numeric, for example, `SUM` or `CEILING`;
  - character, for example, `INDEX` and `TRIM`;
  - bit, for example, `IAND` and `IOR`;
- inquiry, for example, `ALLOCATED` and `SIZE`;
- transformational, for example, `REAL` and `TRANSPOSE`;
- miscellaneous (non-elemental `SUBROUTINE`s), for example, `SYSTEM_CLOCK` and `DATE_AND_TIME`.

Note all intrinsics which take `REAL` valued arguments also accept `DOUBLE PRECISION` arguments.

Type Conversion Functions

It is easy to transform the type of an entity,

- □ `REAL(i)` converts `i` to a real approximation,
- □ `INT(x)` truncates `x` to the integer equivalent,
- □ `DBLE(a)` converts `a` to `DOUBLE PRECISION`,
- □ `IACHAR(c)` returns the position of `CHARACTER c` in the ASCII collating sequence,
- □ `ACHAR(i)` returns the i^{th} character in the ASCII collating sequence.

All above are intrinsic functions. For example,

```
PRINT*, REAL(1), INT(1.7), INT(-0.9999)
PRINT*, IACHAR('C'), ACHAR(67)
```

are equal to

```
1.000000 1 0
67 C
```

Mathematical Intrinsic Functions

Summary,

`ACOS(x)`	arccosine
`ASIN(x)`	arcsine
`ATAN(x)`	arctangent
`ATAN2(y,x)`	arctangent of complex number (x, y)
`COS(x)`	cosine where x is in radians
`COSH(x)`	hyperbolic cosine where x is in radians
`EXP(x)`	e raised to the power x
`LOG(x)`	natural logarithm of x
`LOG10(x)`	logarithm base 10 of x
`SIN(x)`	sine where x is in radians
`SINH(x)`	hyperbolic sine where x is in radians
`SQRT(x)`	the square root of x
`TAN(x)`	tangent where x is in radians
`TANH(x)`	tangent where x is in radians

Numeric Intrinsic Functions

Summary,

ABS(a)	absolute value
AINT(a)	truncates a to whole REAL number
ANINT(a)	nearest whole REAL number
CEILING(a)	smallest INTEGER greater than or equal to REAL number
CMPLX(x,y)	convert to COMPLEX
DBLE(x)	convert to DOUBLE PRECISION
DIM(x,y)	positive difference
FLOOR(a)	biggest INTEGER less than or equal to real number
INT(a)	truncates a into an INTEGER
MAX(a1,a2,a3,...)	the maximum value of the arguments
MIN(a1,a2,a3,...)	the minimum value of the arguments
MOD(a,p)	remainder function
MODULO(a,p)	modulo function
NINT(x)	nearest INTEGER to a REAL number
REAL(a)	converts to the equivalent REAL value
SIGN(a,b)	transfer of sign — ABS(a)*(b/ABS(b))

PRINT Statement

This is the simplest form of directing unformatted data to the standard output channel, for example,

```
PROGRAM Owt
 IMPLICIT NONE
  CHARACTER(LEN=*), PARAMETER :: &
     long_name = "Llanfair...gogogoch"
  REAL :: x, y, z
  LOGICAL :: lacigol
   x = 1; y = 2; z = 3
   lacigol = (y .eq. x)
   PRINT*, long_name
   PRINT*, "Spock says ""illogical&
            &Captain"" "
   PRINT*, "X = ",x," Y = ",y," Z = ",z
   PRINT*, "Logical val: ",lacigol
END PROGRAM Owt
```

produces on the screen,

```
Llanfair...gogogoch
Spock says "illogical Captain"
X =  1.000  Y =  2.000  Z =  3.000
Logical val:  F
```

READ Statement

READ accepts unformatted data from the standard input channel, for example, if the type declarations are the same as for the PRINT example,

```
READ*, long_name
READ*, x, y, z
READ*, lacigol
```

accepts

```
Llanphairphwyll...gogogoch
0.4 5. 1.0e12
T
```

Note,

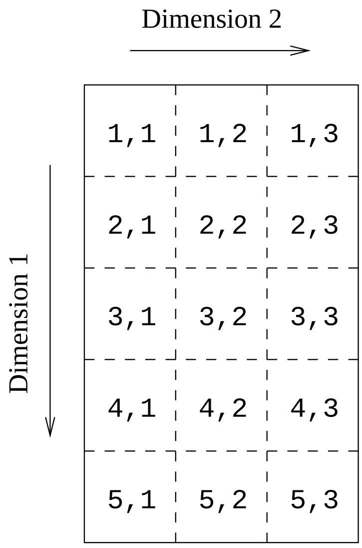
- each READ statement reads from a newline;
- the READ statement can transfer any object of intrinsic type from the standard input;

Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by **subscripting** the array.

A 15 element array can be visualised as:

And a 5×3 array as:



Every array has a type and each element holds a value of that type.

Array Terminology

Examples of declarations:

```
REAL, DIMENSION(15)       :: X
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

The above are *explicit-shape* arrays.

Terminology:

- **rank** — number of dimensions.

  Rank of `X` is 1; rank of `Y` and `Z` is 2.

- **bounds** — upper and lower limits of indices.

  Bounds of `X` are 1 and 15; Bound of `Y` and `Z` are 1 and 5 and 1 and 3.

- **extent** — number of elements in dimension;

  Extent of `X` is 15; extents of `Y` and `Z` are 5 and 3.

- **size** — total number of elements.

  Size of `X`, `Y` and `Z` is 15.

- **shape** — rank and extents;

  Shape of `X` is 15; shape of `Y` and `Z` is 5,3.

- **conformable** — same shape.

  `Y` and `Z` are conformable.

Declarations

Literals and constants can be used in array declarations,

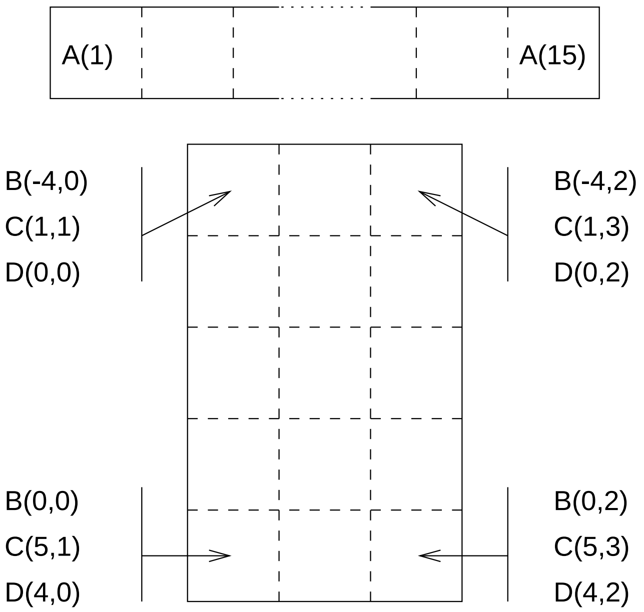
```
REAL, DIMENSION(100)       :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL                       :: T(10,10)
REAL, DIMENSION(-10:-1)    :: X
INTEGER, PARAMETER           :: lda = 5
REAL, DIMENSION(0:lda-1)     :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- default lower bound is 1,
- bounds can begin and end anywhere,
- arrays can be zero-sized (if `lda = 0`),

Visualisation of Arrays

```
REAL, DIMENSION(15)       :: A
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)      :: C
REAL, DIMENSION(0:4,0:2)  :: D
```

Individual array elements are denoted by *subscripting* the array name by an `INTEGER`, for example, `A(7)` 7^{th} element of `A`, or `C(3,2)`, 3 elements down, 2 across.



Array Conformance

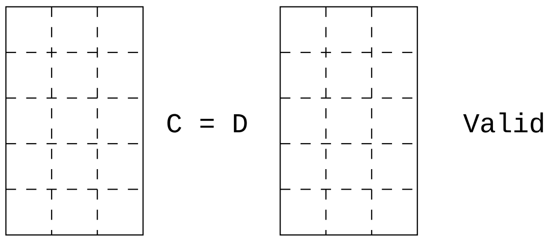
Arrays or sub-arrays must conform with all other objects in an expression:

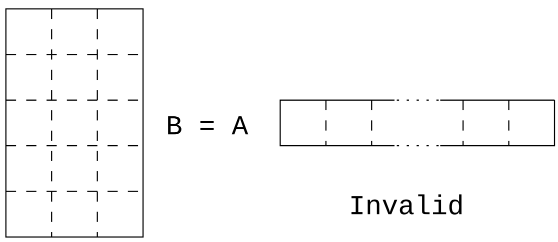
- a scalar conforms to an array of any shape with the same value for every element:

```
C = 1.0   ! is valid
```

- two array references must conform in their shape.

  Using the declarations from before:





`A` and `B` have the same size but have different shapes so cannot be directly equated.

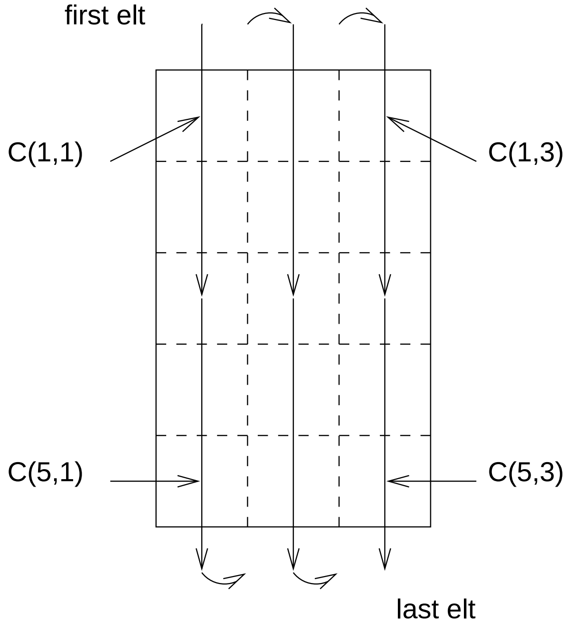
Array Element Ordering

Organisation in memory:

- Fortran 90 does not specify anything about how arrays should be located in memory. **It has no storage association.**

- Fortran 90 does define an array element ordering for certain situations which is of column major form,

The array is conceptually ordered as:



`C(1,1),C(2,1),..,C(5,1),C(1,2),C(2,2),..,C(5,3)`

Array Syntax

Can reference:

- whole arrays
  - `A = 0.0`
    sets whole array `A` to zero.
  - `B = C + D`
    adds `C` and `D` then assigns result to `B`.
- elements
  - `A(1) = 0.0`
    sets one element to zero,
  - `B(0,0) = A(3) + C(5,1)`
    sets an element of `B` to the sum of two other elements.
- array sections
  - `A(2:4) = 0.0`
    sets `A(2)`, `A(3)` and `A(4)` to zero,
  - `B(-1:0,1:2) = C(1:2,2:3) + 1.0`
    adds one to the subsection of `C` and assigns to the subsection of `B`.

Whole Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections),

  ```
  B = C * D - B**2
  ```

  this is equivalent to concurrent execution of:

  ```
  B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
  B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
   ...
  B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
   ...
  B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
  ```

- elemental intrinsic functions can be used,

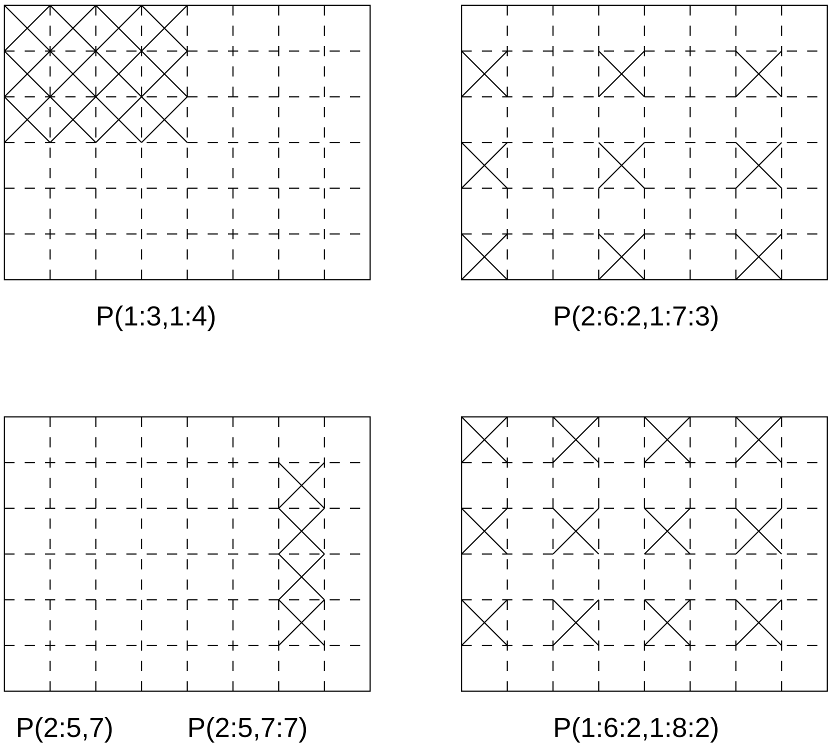
  ```
  B = SIN(C)+COS(D)
  ```

  the function is applied element by element.

Array Sections — Visualisation

Given,

```
REAL, DIMENSION(1:6,1:8) :: P
```



Consider the following assignments,

- `P(1:3,1:4) = P(1:6:2,1:8:2)` and `P(1:3,1:4) = 1.0` are valid.

- `P(2:8:2,1:7:3) = P(1:3,1:4)` and `P(2:6:2,1:7:3) = P(2:5,7)` are not.

- `P(2:5,7)` is a 1D section (scalar in dimension 2) whereas `P(2:5,7:7)` is a 2D section.

Array Sections

subscript-triplets specify sub-arrays. The general form is:

[< *bound1* >]:[< *bound2* >][:< *stride* >]

The section starts at < *bound1* > and ends at or before < *bound2* >. < *stride* > is the increment by which the locations are selected.

< *bound1* >, < *bound2* > and < *stride* > must all be scalar integer expressions. Thus

```
A(:)          ! the whole array
A(3:9)        ! A(m) to A(n) in steps of 1
A(3:9:1)      ! as above
A(m:n)        ! A(m) to A(n)
A(m:n:k)      ! A(m) to A(n) in steps of k
A(8:3:-1)     ! A(8) to A(3) in steps of -1
A(8:3)        ! A(8) to A(3) step 1 => Zero size
A(m:)         ! from A(m) to default UPB
A(:n)         ! from default LWB to A(n)
A(::2)        ! from default LWB to UPB step 2
A(m:m)        ! 1 element section
A(m)          ! scalar element - not a section
```

are all valid sections.

Array I/O

The conceptual ordering of array elements is useful for defining the order in which array elements are output. If `A` is a 2D array then:

```
PRINT*, A
```

would produce output in the order:

```
A(1,1),A(2,1),A(3,1),..,A(1,2),A(2,2),..
```

```
READ*, A
```

would assign to the elements in the above order.

This order could be changed by using intrinsic functions such as `RESHAPE`, `TRANSPOSE` or `CSHIFT`.

Array I/O Example

Consider the matrix `A`:

1	4	7
2	5	8
3	6	9

The following `PRINT` statements

```
   ...
    PRINT*, 'Array element   =',a(3,2)
    PRINT*, 'Array section   =',a(:,1)
    PRINT*, 'Sub-array       =',a(:2,:2)
    PRINT*, 'Whole Array     =',a
    PRINT*, 'Array Transp''d =',TRANSPOSE(a)
  END PROGRAM Owt
```

produce on the screen,

```
Array element    = 6
Array section    = 1 2 3
Sub-array        = 1 2 4 5
Whole Array      = 1 2 3 4 5 6 7 8 9
Array Transposed = 1 4 7 2 5 8 3 6 9
```

Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays:

- Declaration:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: ages   ! 1D
REAL, DIMENSION(:,:), ALLOCATABLE :: speed   ! 2D
```

  Note ALLOCATABLE attribute and fixed rank.

- Allocation:

```
READ*, isize
ALLOCATE(ages(isize), STAT=ierr)
IF (ierr /= 0) PRINT*, "ages : Allocation failed"

ALLOCATE(speed(0:isize-1,10),STAT=ierr)
IF (ierr /= 0) PRINT*, "speed : Allocation failed"
```

- the optional STAT= field reports on the success of the storage request. If the INTEGER variable ierr is zero the request was successful otherwise it failed.

Deallocating Arrays

Heap storage can be reclaimed using the `DEALLOCATE` statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- □ it is an error to deallocate an array without the `ALLOCATE` attribute or one that has not been previously allocated space,
- □ there is an intrinsic function, `ALLOCATED`, which returns a scalar `LOGICAL` values reporting on the status of an array,
- □ the `STAT=` field is optional but its use is recommended,
- □ if a procedure containing an allocatable array which does not have the `SAVE` attribute is exited without the array being `DEALLOCATE`d then this storage becomes inaccessible.

PART II

BARNES-HUT ALGORITHM

PART III

BARNES-HUT IN PARALLEL

PART IV

BIBLIOGRAPHY, INDEX, LIST OF ACRONYMS, APPENDICES, ETC.

Chapter 3

Bibliography

[1] Schlitt, Wayne. *The XStar N-body Solver. Theory of Operation.* `http://www.schlitt.net/xstar/n-body.pdf`.