Modern Fortran Programming II

Alexander B. Pacheco
LTS Research Computing
April 28, 2015

Outline

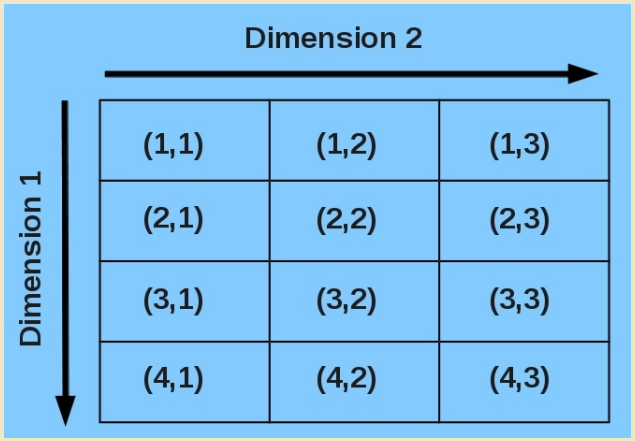
Arrays

Arrays

- Arrays (or matrices) hold a collection of different values at the same time.
- Individual elements are accessed by subscripting the array.
- A 10 element array is visualized as

1	2	3	· · ·	8	9	10

  while a 4x3 array as



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

Array Declarations

- The **dimension** attribute declares arrays.
- Usage: **dimension**(lower_bound:upper_bound)

  Lower bounds of one (1:) can be omitted
- Examples:

```
integer, dimension(1:106) :: atomic_number
real, dimension(3,0:5,-10:10) :: values
character(len=3),dimension(12) :: months
```

- Alternative form for array declaration

```
integer :: days_per_week(7), months_per_year(12)
real :: grid(0:100,-100:0,-50:50)
complex :: psi(100,100)
```

- Another alternative form which can be very confusing for readers

```
integer, dimension(7) :: days_per_week, months_per_year(12)
```

Array Terminology

```
real :: a(0:20), b(3,0:5,-10:10)
```

Rank: Number of dimensions.

`a` has rank 1 and `b` has rank 3

Bounds: upper and lower limits of each dimension of the array.

`a` has bounds 0:20 and `b` has bounds 1:3, 0:5 and -10:10

Extent: Number of element in each dimension

`a` has extent 21 and `b` has extents 3,6 and 21

Size: Total number of elements.

`a` has size 21 and `b` has 30

Shape: The shape of an array is its rank and extent

`a` has shape 21 and `b` has shape (3,6,21)

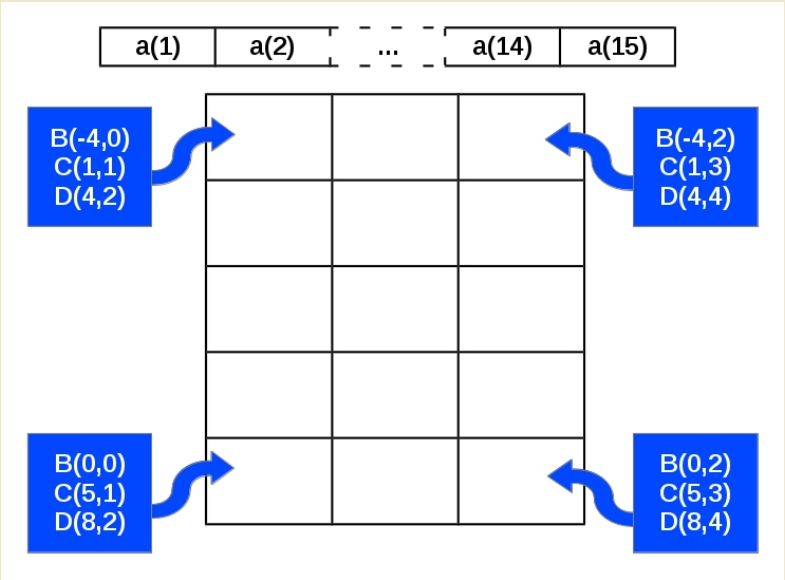
- Arrays are conformable if they share a shape.
- The bounds do not have to be the same

```
c(4:6)= d(1:3)
```

Array Visualization

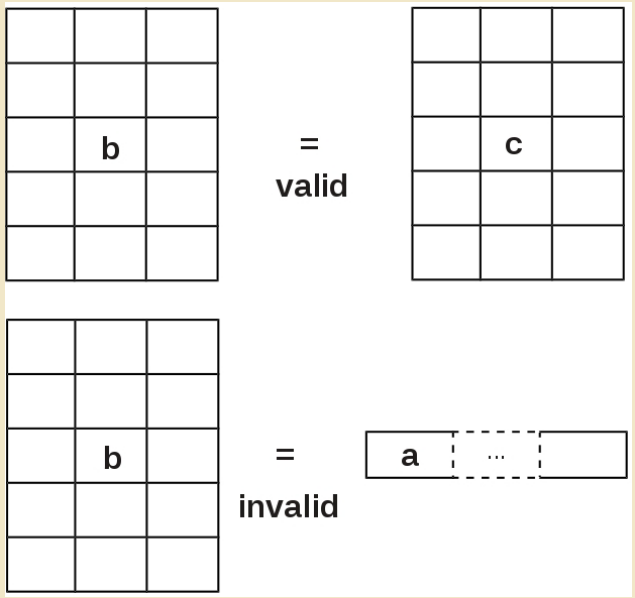
- Define arrays `a`,`b`,`c` and `d` as follows

```
real,dimension(15) :: a
real,dimension(-4:0,0:2) :: b
real,dimension(5,3) :: c
real,dimension(4:8,2:4) :: d
```



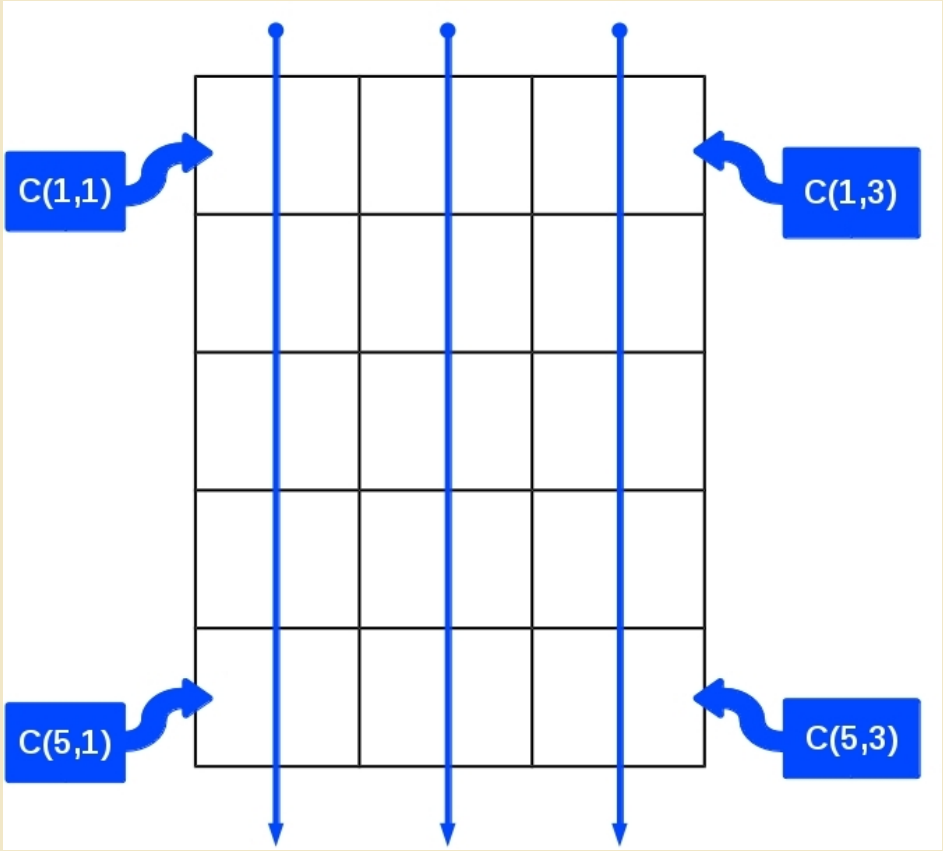
Array Conformance

- Array or sub-arrays must conform with all other objects in an expression
  1. a scalar conforms to an array of any shape with the same value for every element
     `c = 1.0` is the same as `c(:,:) = 1.0`
  2. two array references must conform in their shape.



Array Element Ordering

- Fortran is a column major form i.e. elements are added to the columns seqeuntially. This ordering can be changed using the `reshape` intrinsic.



Array Constructors I

- Used to give arrays or sections of arrays specific values

```
implicit none
integer :: i
integer, dimension(10) :: ints
character(len=5),dimension(3) :: colors
real, dimension(4) :: height
height = (/5.10, 5.4, 6.3, 4.5 /)
colors = (/'red  ', 'green', 'blue ' /)
ints = (/ 30, (i = 1, 8), 40 /)
```

- constructors and array sections must conform.

  `ints = (/ 30, (i = 1, 10), 40/)` is invalid
- strings should be padded so that character variables have correct length.
- use **`reshape`** intrinsic for arrays for higher ranks
- `(i = 1, 8)` is an implied **`do`**.
- You can also specify a stride in the implied **`do`**.

  `ints = (/ 30, (i = 1, 16, 2), 40/)`
- There should be no space between / and (or)

Array Constructors II

- **reshape**(source, **shape**, **pad**, order) constructs an array with a specified shape **shape** starting from the elements in a given array source.
- If **pad** is not included then the size of source has to be at least **product** (**shape**).
- If **pad** is included it has to have the same type as source.
- If order is included, it has to be an **integer** array with the same shape as **shape** and the values must be a permutation of (1,2,3,...,N), where N (max value is 7) is the number of elements in **shape**.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & a & a \\ a & 0 & a \\ a & a & 0 \end{pmatrix}$$

```
rcell = reshape( (/ &
     0.d0, 0.d0, a,    a,    &
     0.d0, a,    0.d0, a,    &
     0.d0, a,    a,    0.d0 &
     /),(/4,3/) )
```

```
rcell = reshape( (/ &
     0.d0, 0.d0, 0.d0 &
     0.d0, a   , a    &
     a,    0.d0, a    &
     a,    a,    0.d0 &
     /),(/4,3/),order=(/2,1/)
          )
```

- In Fortran, for a multidimensional array, the first dimension has the fastest index while the last dimension has the slowest index i.e. memory locations are continuous for the last dimension.

Array Constructors III

- The `order` statement allows the programmer to change this order. The last example above sets the memory location order which is consistent to that in C/C++.
- Arrays can be initialized as follows during variable declaration

```
integer, dimension(4) :: imatrix = (/ 2, 4, 6, 8/)
character(len=*),dimension(3) :: colors = (/'red  ', 'green', 'blue '/)}
! All strings must be the same length}
real, dimension(4) :: height = (/5.10, 5.4, 6.3, 4.5/)
integer, dimension(10) :: ints = (/ 30, (i = 1, 8), 40/)
real, dimension(4,3), parameter :: rcell = reshape( (/0.d0, 0.d0, 0.d0, 0.d0,\&
  a, a, a,0.d0, a, a, a, 0.d0 /),(/4,3/),order=(/2,1/))
```

Array Syntax

- Arrays can be treated as a single variable when performing operations
  1. set whole array to a constant: `a = 0.0`
  2. can use intrinsic operators between conformable arrays (or sections)
     `b = c * d + b**2`
     this is equivalent to

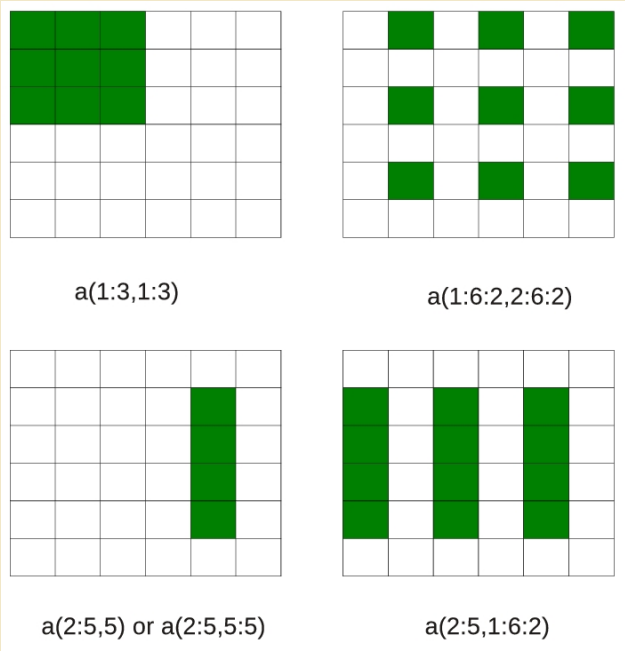
```
b(-4,0) = c(1,1) * d(4,2) + b(-4,0)**2
b(-3,0) = c(2,1) * d(5,2) + b(-3,0)**2
...
b(-4,0) = c(1,1) * d(4,2) + b(-4,0)**2
b(-4,1) = c(1,2) * d(4,3) + b(-4,1)**2
...
b(-3,2) = c(4,3) * d(7,4) + b(-3,2)**2
b(-4,2) = c(5,3) * d(8,4) + b(-4,2)**2
```

  3. elemental intrinsic functions can be used: `b = sin(c)+ cos(d)`
  4. All operations/functions are applied element by element

Array Sections I

```
real, dimension(6:6):: a
```

- `a(1:3,1:3)= a(1:6:2,2:6:2)` and `a(1:3,1:3)= 1.0` are valid
- `a(2:5,5)= a(2:5,1:6:2)` and `a(2:5,1:6:2)= a(1:6:2,2:6:2)` are not
- `a(2:5,5)` is a 1D section while `a(2:5,1:6:2)` is a 2D section



- The general form for specifying sub-arrays or sections 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, by default *stride=1*
- *<bound1>*, *<bound2>*, *<stride>* must all be scalar integer expressions.

Array Sections II

```
real, dimension(1:20) :: a
integer :: m,n,k
```

`a(:)`	the whole array
`a(3:9)`	elements 3 to 9 in increments of 1
`a(3:9:1)`	as above
`a(m:n)`	elements m through n
`a(m:n:k)`	elements m through n in increments of k
`a(15:3:-2)`	elements 15 through 3 in increments of -2
`a(15:3)`	zero size array
`a(m:)`	elements m through 20, default upper bound
`a(:n)`	elements 1, default lower bound through n
`a(::2)`	all elements from lower to upper bound in increments of 2
`a(m:m)`	1 element section
`a(m)`	array element not a section

are valid sections.

```
real,dimension(4,4):: a
```

- Arrays are printed in the order that they appear in memory

  ```
  print *, a
  ```

  would produce on output

  ```
  a(1,1),a(2,1),a(3,1),a(4,1),a(1,2),a(2,2),···,a(3,4),a(4,4)
  ```

  ```
  read *, a
  ```

  would read from input and assign array elements in the same order as above
- The order of array I/O can be changed using intrinsic functions such as **reshape**, **transpose** or **cshift**.

- Example: consider a 3x3 matrix

1	4	7
2	5	8
3	6	9

- The following print statements

```
print *, 'array element   = ',a(3,3)
print *, 'array section   = ',a(:,2)
print *, 'sub-array       = ',a(:3,:2)
print *, 'whole array     = ',a
print *, 'array transpose = ',transpose(a)
```

- would produce the following output

```
array element   = 9
array section   = 4 5 6
sub-array       = 1 2 3 4 5 6
whole array     = 1 2 3 4 5 6 7 8 9
array transpose = 1 4 7 2 5 8 3 6 9
```

Array Intrinsic Functions I

size(x[,n])	The size of x (along the n^{th} dimension, optional)
shape(x)	The shape of x
lbound(x[,n])	The lower bound of x
ubound(x[,n])	The upper bound of x
minval(x)	The minimum of all values of x
maxval(x)	The maximum of all values of x
minloc(x)	The indices of the minimum value of x
maxloc(x)	The indices of the maximum value of x
sum(x[,n])	The sum of all elements of x (along the n^{th} dimension, optional)

$sum(x) = \sum_{i,j,k,\ldots} x_{i,j,k,\ldots}$

Array Intrinsic Functions II

product(x[,n]) The product of all elements of x (along the n^{th} dimension, optional)
$prod(x) = \prod_{i,j,k,\ldots} x_{i,j,k,\ldots}$

transpose(x) Transpose of array x: $x_{i,j} \Rightarrow x_{j,i}$

dot_product(x,y) Dot Product of arrays x and y: $\sum_i x_i * y_i$

matmul(x,y) Matrix Multiplication of arrays x and y which can be 1 or 2 dimensional arrays: $z_{i,j} = \sum_k x_{i,k} * y_{k,j}$

conjg(x) Returns the conjugate of x: $a + \imath b \Rightarrow a - \imath b$

cshift(ARRAY, SHIFT, dim) perform a circular shift by SHIFT positions to the left on array ARRAY along the $\mathrm{dim}^{\mathrm{th}}$ dimension

Allocatable Arrays I

Why?

- At compile time we may not know the size an array needs to be
- We may want to change the problem size without recompiling
- The molecular dynamics code was written for 4000 atoms. If you want to run a simulation for 256 and 1024 atoms, do you need to recompile and create two executables?

- Allocatable arrays allow us to set the size at run time.

```
real, allocatable :: force(:,:)
```

```
real, dimension(:), allocatable :: vel
```

- We set the size of the array using the allocate statement.

```
allocate(force(natoms,3))
```

- We may want to change the lower bound for an array

```
allocate(grid(-100,100))
```

Allocatable Arrays II

- We may want to use an array once somewhere in the program, say during initialization. Using allocatable arrays also us to dynamically create the array when needed and when not in use, free up memory using the **deallocate** statement

```
deallocate(force,grid)
```

- Sometimes, we want to check whether an array is allocated or not at a particular part of the code
- Fortran provides an intrinsic function, **allocated** which returns a scalar logical value reporting the status of an array

```
if ( allocated(grid))deallocate(grid)
```

```
if ( .not. allocated(force))allocate(force(natoms,3))
```

Masked Array Assignment: Where Statement

- Masked array assignment is achieved using the **where** statement

  ```
  where ( c < 2)a = b/c
  ```

  the left hand side of the assignment must be array valued.

  the mask (logical expression) and the right hand side of the assignment must all conform
- Fortran 95/2003 introduced the **where ... elsewhere ... end where** functionality
- **where** statement cannot be nested

```
! Apply PBC to coordinates
where ( coord(i,:) > boxl(:) )
   coord(i,:) = coord(i,:) - boxl(:)
elsewhere ( coord(i,:) < 0d0 )
   coord(i,:) = coord(i,:) + boxl(:)
end where
```

```
! Apply PBC to coordinates
do j = 1, 3
   if ( coord(i,j) > boxl(j) ) then
      coord(i,j) = coord(i,j) - boxl(j)
   else if ( coord(i,j) < 0d0 ) then
      coord(i,j) = coord(i,j) + boxl(j)
   endif
end do
```

Procedures

Program Units I

- Most programs are hundreds or more lines of code.
- Use similar code in several places.
- A single large program is extremely difficult to debug and maintain.
- Solution is to break up code blocks into procedures

  Subroutines: Some out-of-line code that is called exactly where it is coded

  Functions: Purpose is to return a result and is called only when the result is needed

  Modules: A module is a program unit that is not executed directly, but contains data specifications and procedures that may be utilized by other program units via the use statement.

Program Units II

```
program main

  use module1 ! specify which modules to use

  implicit none  ! implicit typing is not recommended
  variable declarations  ! declare all variables used in the program


  .
  .
  . ! executable statements in seqeunce

  call routine1(arg1,arg2,arg3) ! call subroutine routine1 with arguments

  .
  .
  .
  abc = func(arg1,arg2) ! abc is some function of arg1 and arg2

  .
  .
  .

  contains  ! internal procedures are listed below

    subroutine routine1(arg1,arg2)  ! subroutine routine1 contents go here

      .
      .
      .
    end subroutine routine1  ! all program units must have an end statement

    function func(arg1,arg2) ! function func1 contents go here
      ...
    end function func

  end program main
```

Program Units III

```
program md

  ! Molecular Dynamics code for equilibration of Liquid Argon
  ! Author: Alex Pacheco
  ! Date  : Jan 30, 2014

  ! This program simulates the equilibration of Liquid Argon
  ! starting from a FCC crystal structure using Lennard-Jones
  ! potential and velocity verlet algorithm

  ! This program should be the starting point to learn Modern
  ! Fortran.
  ! This program is hard coded for 4000 atoms equilibrated at
  ! 10K with a time step of 0.001 time units and 1000 time steps
  ! Lets assume that time units is femtoseconds, so total simulation
  ! time is 1 femtosecond

  ! Your objective for
  ! Modern Fortran Training:
  ! Modify this code using the Fortran Concepts learned
  !   1. split code into smaller subunits, modules and/or subroutines
  !   2. generalize, so that the following parameters can be read from a input file
  !       a. number of atoms or number of unit cells (you can't do both)
  !       b. equilibration temperature
  !       c. time step
  !       d. number of time steps i.e. how long in fs do you want the simulation to run
  !       e. read input parameters using namelists
  !      You will need to make use allocatable arrays. If you do not know why, review
  !      training slides or ask
  !   3. Can you use Modern Fortran Concepts such as derived types? If yes, program it
  !   4. If you use derived types, can you overload operators? If yes, program it
  ! OpenMP/OpenACC Training
  !   Lets assume that you have completed upto step 2 from Modern Fortran objective
  !   Parallelize the code for OpenMP/OpenACC (can also be done from step 3 or 4)
  !
  ! There is no time limit for completing this exercise. This exercise is for measuring
  ! what have you got from the training.
  ! Solutions are present in the separate directories for comparison.
  ! Hints are provided whereever needed
```

Program Units IV

```
  ! As an additional exercise, use other potentials such as Morse potential and
  ! read from input file which potential you want to use.
  ! All Lennard-Jones Potential parameters are set to 1.

  ! Disclaimer:
  ! This is code can be used as an introduction to molecular dynamics. There are lot more
  ! concepts in MD that are not covered here.

  ! Parameters:
  ! npartdim : number of unit cells, uniform in all directions. change to nonuniform if you desire
  ! natom : number of atoms
  ! nstep : nummber of simulation time steps
  ! tempK : equilibration temperature
  ! dt : simulation time steps
  ! boxl : length of simulation box in all directions
  ! alat : lattice constant for fcc crystal
  ! kb : boltzmann constant, set to 1 for simplicity
  ! mass : mass of Ar atom, set to 1 for simplicity
  ! epsilon, sigma : LJ parameters, set to 1 for simplicity
  ! rcell : FCC unit cell
  ! coord, coord_t0 : nuclear positions for each step, current and initial
  ! vel, vel_t0 : nuclear velocities for each step
  ! acc, acc_t0 : nuclear acceleration for each step
  ! force, pener : force and potential energy at current step
  ! avtemp : average temperature at current time step
  ! scale : scaling factor to set current temperature to desired temperature
  ! gasdev : Returns a normally distributed deviate with zero mean and unit variance from Numerical recipes

    implicit none
    ! Use either kind function or selected_real_kind
    integer,parameter :: npartdim = 10
    integer,parameter :: natom = 4.d0 * npartdim ** 3
    integer,parameter :: nstep = 1000
    real*8, parameter :: tempK = 10, dt = 1d-3
    integer :: istep
    real*8 :: boxl(3), alat
    integer :: n, i, j, k, l

  ! Can you use derived types for coord, vel, acc and force
    real*8 :: coord_t0(natom,3), coord(natom,3)
```

Program Units V

```
real*8 :: vel_t0(natom,3), vel(natom,3)
real*8 :: acc_t0(natom,3), acc(natom,3)
real*8 :: force(natom,3), pener, mass

real*8 :: vcm(3), r(3), rr, r2, r6, f
real*8 :: avtemp, ke, kb, epsilon, sigma, rcell(3,4), scale
real*8 :: gasdev

alat = 2d0 ** (2d0/3d0)
! Hint: Array operations
do i = 1, 3
   boxl(i) = npartdim * alat
end do
kb = 1.d0
mass = 1.d0
epsilon = 1.d0
sigma = 1.d0

! Create FCC unit cell
! Hint: Simplify unit cell creation, maybe in variable declaration
rcell(1,1) = 0d0
rcell(2,1) = 0d0
rcell(3,1) = 0d0
rcell(1,2) = 0.5d0 * alat
rcell(2,2) = 0.5d0 * alat
rcell(3,2) = 0d0
rcell(1,3) = 0d0
rcell(2,3) = 0.5d0 * alat
rcell(3,3) = 0.5d0 * alat
rcell(1,4) = 0.5d0 * alat
rcell(2,4) = 0d0
rcell(3,4) = 0.5d0 * alat

! Set initial coordinates, velocity and acceleration to zero
! Hint: Use Array operations
do i = 1, natom
   do j = 1, 3
      coord_t0(i,j) = 0d0
      vel_t0(i,j) = 0d0
      acc_t0(i,j) =  0d0
```

Program Units VI

```
   end do
end do

!================================================
! Initialize coordinates and random velocities
!================================================

! Put initialization in a seperate subroutine
! call initialize(coord_t0, vel_t0, ...)
! Create a FCC crystal structure
n = 1
do i = 1, npartdim
   do j = 1, npartdim
      do k = 1, npartdim
         do l = 1, 4
            coord_t0(n,1) = alat * dble(i - 1) + rcell(1,l)
            coord_t0(n,2) = alat * dble(j - 1) + rcell(2,l)
            coord_t0(n,3) = alat * dble(k - 1) + rcell(3,l)
            n = n + 1
         end do
      end do
   end do
end do

open(unit=1,file='atom.xyz',status='unknown')
write(1,'(i8)') natom
write(1,*)
do i = 1, natom
   write(1,'(a2,2x,3f12.6)') 'Ar', coord_t0(i,1), coord_t0(i,2), coord_t0(i,3)
end do
close(1)

! Assign initial random velocities
do i = 1, natom
   do j = 1, 3
      vel_t0(i,j) = gasdev()
   end do
end do

! Set Linear Momentum to zero
```

Program Units VII

```
! Hint: This is needed again below so put in a subroutine
! call linearmom(vel_t0, ...)
! First get center of mass velocity
vcm = 0d0
do i = 1, natom
   do j = 1, 3
      vcm(j) = vcm(j) + vel_t0(i,j)/natom
   end do
end do
! Now remove center of mass velocity from all atoms
do i = 1, natom
   do j = 1, 3
      vel_t0(i,j) = vel_t0(i,j) - vcm(j)
   end do
end do

! scale velocity to desired tempearture
! call get_temp( vel_t0, ... ) will be needed again
ke = 0d0
do i = 1, natom
   do j = 1, 3
      ! Hint: Use dot_product function to calculate vel**2
      ! If using derived types, overload dot_product function
      ke = ke + mass * vel_t0(i,j)**2
   end do
end do
avtemp = mass * ke / ( 3d0 * kb * ( natom - 1))

print '(a,2x,1pe15.8)', 'Initial Average Temperature: ', avtemp

! scale initial velocity to desired temperature
scale = sqrt( tempK / avtemp )
ke = 0d0
do i = 1, natom
   do j = 1, 3
      vel_t0(i,j) = vel_t0(i,j) * scale
      ! See Hint above on dot_product and function overloading
      ke = ke + mass * vel_t0(i,j)**2
   end do
end do
```

Program Units VIII

```
avtemp = mass * ke / ( 3d0 * kb * ( natom - 1))
print '(a,2x,1pe15.8)', 'Initial Scaled Average Temperature: ', avtemp


!================================================
! MD Simulation
!================================================

do istep = 1, nstep

   ! Set coordinates, velocity, acceleration and force at next time step to zero
   ! Hint: Use Array properties
   do i = 1, natom
      do j = 1, 3
         coord(i,j) = 0d0
         vel(i,j) = 0d0
         acc(i,j) =  0d0
         force(i,j) =  0d0
      end do
   end do
   pener = 0d0

   ! Get new atom positions from Velocity Verlet Algorithm
   ! Hint: create a subroutine to do velocity verlet
   ! Hint: OpenMP/OpenACC
   do i = 1, natom
      do j = 1, 3
         coord(i,j) = coord_t0(i,j) + vel_t0(i,j) * dt + 0.5d0 * acc_t0(i,j) * dt ** 2
         ! Apply PBC to coordinates
         if ( coord(i,j) > boxl(j) ) then
            coord(i,j) = coord(i,j) - boxl(j)
         else if ( coord(i,j) < 0d0 ) then
            coord(i,j) = coord(i,j) + boxl(j)
         endif
      end do
   end do

   ! Get force at new atom positions
   ! Using Lennard Jones Potential
   ! Hint: you might want to also seperate the potential and force calculation into a separate subroutine
```

Program Units IX

```
! this will be useful if you want to use other potentials

do i = 1, natom - 1
   do j = i + 1, natom
      do k = 1, 3
         r(k) = coord(i,k) - coord(j,k)
         ! minimum image criterion
         ! interaction of an atom with another atom or its image within the unit cell
         r(k) = r(k) - nint( r(k) / boxl(k) ) * boxl(k)
      end do
      ! Hint: Use dot_product
      rr = r(1) ** 2 + r(2) ** 2 + r(3) ** 2
      r2 = 1.d0 / rr
      r6 = r2 ** 3
      ! Lennard Jones Potential
      ! V = 4 * epsilon * [ (sigma/r)**12 - (sigma/r)**6 ]
      !   = 4 * epsilon * (sigma/r)**6 * [ (sigma/r)**2 - 1 ]
      !   = 4 * r**(-6) * [ r**(-2) - 1 ] for epsilon=sigma=1
      ! F_i = 48 * epsilon * (sigma/r)**6 * (1/r**2) * [ ( sigma/r)** 6 - 0.5 ] * i where i = x,y,z
      !     = 48 * r**(-8) * [ r**(-6) - 0.5 ] * i  for epsilon=sigma=1
      pener = pener + 4d0 * r6 * ( r6 - 1.d0 )
      f = 48d0 * r2 * r6 * ( r6 - 0.5d0 )
      do k = 1, 3
         ! use array function to obtain r(k)*f
         force(i,k) = force(i,k) + r(k) * f
         force(j,k) = force(j,k) - r(k) * f
      end do
   end do
end do

! Calculate Acceleration and Velocity  at current time step
do i = 1, natom
   do j = 1, 3
      acc(i,j) = force(i,j) / mass
      vel(i,j) = vel_t0(i,j) + 0.5d0 * (acc(i,j) + acc_t0(i,j)) * dt
   end do
end do

! Set Linear Momentum to zero
! First get center of mass velocity
```

Program Units X

```
! See Hint above on Linear Momentum
vcm = 0d0
do i = 1, natom
   do j = 1, 3
      vcm(j) = vcm(j) + vel(i,j)/natom
   end do
end do
! Now remove center of mass velocity from all atoms
do i = 1, natom
   do j = 1, 3
      vel(i,j) = vel(i,j) - vcm(j)
   end do
end do

! compute average temperature
! See Hint above on calculating average temperature
ke = 0d0
do i = 1, natom
   do j = 1, 3
      ke = ke + vel(i,j) ** 2
   end do
end do
avtemp = mass * ke / ( 3d0 * kb * ( natom - 1))

print '(a,2x,i8,2x,1pe15.8,1x,1pe15.8)', 'Average Temperature: ' , istep, avtemp, pener

scale = sqrt ( tempk/ avtemp )
! Reset for next time step
! Hint:  Use Array properties
do i = 1, natom
   do j = 1, 3
      acc_t0(i,j) = acc(i,j)
      coord_t0(i,j) = coord(i,j)
      ! scale velocity to desired temperature
      vel_t0(i,j) = vel(i,j) * scale
   end do
end do

! Write current coordinates to xyz file for visualization
open(unit=1,file='atom.xyz',position='append')
```

Program Units XI

```
      write(1,'(i8)') natom
      write(1,*)
      do i = 1, natom
         write(1,'(a2,2x,3f12.6)') 'Ar', coord_t0(i,1), coord_t0(i,2), coord_t0(i,3)
      end do
      close(1)
   end do

end program md

double precision function gasdev()
   implicit none
   real*8 :: v1, v2, fac, rsq
   real*8, save :: gset
   logical, save :: available = .false.

   if (available) then
      gasdev = gset
      available = .false.
   else
      do
         call random_number(v1)
         call random_number(v2)
         v1 = 2.d0 * v1 - 1.d0
         v2 = 2.d0 * v2 - 1.d0
         rsq = v1**2 + v2**2
         if ( rsq > 0.d0 .and. rsq < 1.d0 ) exit
      end do
      fac = sqrt(-2.d0 * log(rsq) / rsq)
      gasdev = v1 * fac
      gset = v2 * fac
      available = .true.
   end if
end function gasdev
```

Subroutines I

- Call Statement:
  - The **`call`** statement evaluates its arguments and transfers control to the subroutine
  - Upon return, the next statement is executed.
- SUBROUTINE Statement:
  - The **`subroutine`** statement declares the procedure and its arguments.
  - These are also known as dummy arguments.
- The subroutine's interface is defined by
  - The **`subroutine`** statement itself
  - The declarations of its dummy arguments
  - Anything else that the subroutine uses

Subroutines II

- Statement Order
  1. A **`subroutine`** statement starts a subroutine
  2. Any **`use`** statements come next
  3. **`implicit none`** comes next, followed by
  4. rest of the declarations,
  5. executable statements
  6. End with a **`end subroutine`** statement
- Dummy Arguments
  - Their names exist only in the procedure and are declared as local variables.
  - The dummy arguments are associated with the actual arguments passed to the subroutines.
  - The dummy and actual argument lists must match, i.e. the number of arguments must be the same and each argument must match in type and rank.

Subroutines III

```
subroutine verlet(coord, coord_t0, vel, vel_t0, acc, acc_t0, force, pener)
  use precision
  use potential
  use param, only : natom, mass, dt, boxl, pot
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener
  integer(ip) :: i, j, k
  real(dp) :: epot
  real(dp) :: r(3), f(3)

  ! Set coordinates, velocity, acceleration and force at next time step to zero
  coord = 0d0 ; vel = 0d0 ; acc = 0d0 ; force = 0d0
  pener = 0d0

  ! Get new atom positions from Velocity Verlet Algorithm
  coord = coord_t0 + vel_t0 * dt + 0.5d0 * acc_t0 * dt ** 2
  do i = 1, natom
     ! Apply PBC to coordinates
     where ( coord(i,:) > boxl(:) )
        coord(i,:) = coord(i,:) - boxl(:)
     elsewhere ( coord(i,:) < 0d0 )
        coord(i,:) = coord(i,:) + boxl(:)
     end where
  end do

  ! Get force at new atom positions
  do i = 1, natom - 1
     do j = i + 1, natom
        r(:) = coord(i,:) - coord(j,:)
        ! minimum image criterion
        r = r - nint( r / boxl ) * boxl
        select case(pot)
        case('mp')
           call morse( r, f, epot )
        case default
           call lennard_jones( r, f, epot )
        end select
        pener = pener + epot
        force(i,:) = force(i,:) + f(:)
        force(j,:) = force(j,:) - f(:)
     end do
  end do

  ! Calculate Acceleration and Velocity  at current time step
  acc = force / mass
  vel = vel_t0 + 0.5d0 * ( acc + acc_t0 ) * dt

end subroutine verlet
```

```
program md

. . .

  real(dp), dimension(:,:), allocatable :: coord_t0, coord
  real(dp), dimension(:,:), allocatable :: vel_t0, vel
  real(dp), dimension(:,:), allocatable :: acc_t0, acc, force
  real(dp) :: pener

  interface
     . . .
     subroutine verlet(coord, coord_t0, vel_t0, vel, acc_t0, acc, force, pener)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
       real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
       real(dp), intent(out) :: pener
     end subroutine verlet
     . . .
  end interface

  . . .

  do istep = 1, nstep

     ! Set coordinates, velocity, acceleration and force at next time step to
          zero
     coord = 0d0 ; vel = 0d0 ; acc = 0d0
     force = 0d0 ; pener = 0d0

     ! Get new atom positions from Velocity Verlet Algorithm
     call verlet(coord, coord_t0, vel_t0, vel, acc_t0, acc, force, pener)

     . . .
  end do

  ! Free up memory
  deallocate(coord_t0,vel_t0,acc_t0,coord,vel,acc,force)

end program md
```

Internal Procedures

- Internal procedures appear just before the last `end` statement and are preceeded by the `contains` statement.
- Internal procedures can be either subroutines or functions which can be accessed only by the program, subroutine or module in which it is present
- Internal procedures have declaration of variables passed on from the parent program unit
- If an internal procedure declares a variable which has the same name as a variable from the parent program unit then this supersedes the variable from the outer scope for the length of the procedure.

Functions

- **function**s operate on the same principle as **subroutine**s
- The only difference is that **function** returns a value and does not involve the **call** statement

```
module potential
  use precision
  implicit none
  real(dp) :: r2, r6, d2, d
  real(dp), parameter :: de = 0.176d0, a = 1.4d0, re = 1d0
  real(dp) :: exparre

contains
  subroutine lennard_jones(r,f,p)
    ! Lennard Jones Potential
    ! V = 4 * epsilon * [ (sigma/r)**12 - (sigma/r)**6 ]
    !   = 4 * epsilon * (sigma/r)**6 * [ (sigma/r)**2 - 1 ]
    !   = 4 * r**(-6) * [ r**(-2) - 1 ] for epsilon=sigma=1
    ! F_i = 48 * epsilon * (sigma/r)**6 * (1/r**2) * [ ( sigma/r)** 6 - 0.5 ] *
              i where i = x,y,z
    !     = 48 * r**(-8) * [ r**(-6) - 0.5 ] * i  for epsilon=sigma=1
            implicit none
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    r2 = 1.d0 / dot_product(r,r)
    r6 = r2 ** 3

    f = dvdr_lj(r2, r6) * r
    p = pot_lj(r2, r6)
  end subroutine lennard_jones

  subroutine morse(r,f,p)
    ! Morse Potential
    ! V = D * [ 1 - exp(-a*(r - re)) ]^2
    ! F_i = 2*D * [ 1 - exp(-a*(r - re)) ] * a exp(-a*(r-re)) * i / r
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    d2 = dot_product(r,r)
    d = sqrt(d2)
    exparre = exp( -a * (d - re ))

    f = dvdr_mp(exparre) * r
    p = pot_mp(exparre)
  end subroutine morse

  function pot_lj(r2, r6)
    implicit none
    real(dp), intent(in) :: r2, r6
    real(dp) :: pot_lj
    pot_lj = 4d0 * r6 * ( r6 - 1.d0 )
  end function pot_lj
  function pot_mp(exparre)
    implicit none
    real(dp), intent(in) :: exparre
    real(dp) :: pot_mp
    pot_mp = de * ( 1d0 - exparre )**2
  end function pot_mp

  function dvdr_lj(r2,r6)
    implicit none
    real(dp), intent(in) :: r2, r6
    real(dp) :: dvdr_lj
    dvdr_lj = 48d0 * r2 * r6 * ( r6 - 0.5d0 )
  end function dvdr_lj
  function dvdr_mp(exparre)
    implicit none
    real(dp), intent(in) :: exparre
    real(dp) :: dvdr_mp
    dvdr_mp = 2d0 * de * a * (1d0 - exparre) * exparre
  end function dvdr_mp
end module potential
```

Array-valued Functions

- **`function`** can also return arrays

```
module potential
  use precision
  implicit none
  real(dp) :: r2, r6, d2, d
  real(dp), parameter :: de = 0.176d0, a = 1.4d0, re = 1d0
  real(dp) :: exparre

contains
  subroutine lennard_jones(r,f,p)
    ! Lennard Jones Potential
    ! V = 4 * epsilon * [ (sigma/r)**12 - (sigma/r)**6 ]
    !   = 4 * epsilon * (sigma/r)**6 * [ (sigma/r)**2 - 1 ]
    !   = 4 * r**(-6) * [ r**(-2) - 1 ] for epsilon=sigma=1
    ! F_i = 48 * epsilon * (sigma/r)**6 * (1/r**2) * [ ( sigma/r)** 6
             - 0.5 ] * i where i = x,y,z
    !     = 48 * r**(-8) * [ r**(-6) - 0.5 ] * i  for epsilon=sigma=1
                implicit none
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    r2 = 1.d0 / dot_product(r,r)
    r6 = r2 ** 3

    f = dvdr_lj(r2, r6, r)
    p = pot_lj(r2, r6)
  end subroutine lennard_jones

  subroutine morse(r,f,p)
    ! Morse Potential
    ! V = D * [ 1 - exp(-a*(r - re)) ]^2
    ! F_i = 2*D * [ 1 - exp(-a*(r - re)) ] * a exp(-a*(r-re)) * i / r
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    d2 = dot_product(r,r)
    d = sqrt(d2)
    exparre = exp( -a * (d - re ))

    f = dvdr_mp(exparre,r)
    p = pot_mp(exparre)
  end subroutine morse

  function pot_lj(r2, r6)
    implicit none
    real(dp), intent(in) :: r2, r6
    real(dp) :: pot_lj
    pot_lj = 4d0 * r6 * ( r6 - 1.d0 )
  end function pot_lj
  function pot_mp(exparre)
    implicit none
    real(dp), intent(in) :: exparre
    real(dp) :: pot_mp
    pot_mp = de * ( 1d0 - exparre )**2
  end function pot_mp

  function dvdr_lj(r2,r6,r)
    implicit none
    real(dp), intent(in) :: r2, r6, r
    real(dp), dimension(size(r)) :: dvdr_lj
    dvdr_lj = 48d0 * r2 * r6 * ( r6 - 0.5d0 ) * r
  end function dvdr_lj
  function dvdr_mp(exparre,r)
    implicit none
    real(dp), intent(in) :: exparre, r
    real(dp), dimension(size(r)) :: dvdr_mp
    dvdr_mp = 2d0 * de * a * (1d0 - exparre) * exparre * r
  end function dvdr_mp
end module potential
```

Recursive Procedures

- In Fortran 90, recursion is supported as a feature
  1. **recursive** procedures call themselves
  2. **recursive** procedures must be declared explicitly
  3. **recursive function** declarations must contain a **result** keyword, and
  4. one type of declaration refers to both the function name and the result variable.

```
program fact

  implicit none
  integer :: i
  print *, 'enter integer whose factorial you want to calculate
        '
  read *, i

  print '(i5,a,i20)', i, '! = ', factorial(i)

contains
  recursive function factorial(i) result(i_fact)
    integer, intent(in) :: i
    integer :: i_fact

    if ( i > 0 ) then
       i_fact = i * factorial(i - 1)
    else
       i_fact = 1
    end if
  end function factorial

end program fact
```

```
[apacheco@qb4 Exercise] ./factorial
 enter integer whose factorial you want to calculate
10
   10! =              3628800
[apacheco@qb4 Exercise] ./fact1
 Enter an integer < 15
10
  10!=        3628800
```

Argument Association

- Recall from MD code example the invocation

```
call linearmom(vel_t0)
```

- and the subroutine declaration

```
subroutine linearmom(vel)
```

- `vel_t0` is an actual argument and is associated with the dummy argument `vel`
- In `subroutine linearmom`, the name `vel` is an alias for `vel_t0`
- If the value of a dummy argument changes, then so does the value of the actual argument
- The actual and dummy arguments must correspond in type, kind and rank.

- In **subroutine** linearmom,

  i and vcm are local objects.
- Local Objects
  - are created each time a procedure is invoked
  - are destroyed when the procedure completes
  - do not retain their values between calls
  - do not exist in the programs memory between calls.

Example

```
subroutine linearmom(vel)
  use precision
  use param, only : natom
  implicit none
  real(dp), dimension(:,:), intent(inout) :: vel
  integer(ip) :: i
  real(dp) :: vcm(3)

  ! First get center of mass velocity
  vcm = 0d0
  do i = 1, 3
     vcm(i) = sum(vel(:,i))
  end do
  vcm = vcm / real(natom,dp)

  ! Now remove center of mass velocity from all atoms
  do i = 1, natom
     vel(i,:) = vel(i,:) - vcm(:)
  end do

end subroutine linearmom
```

Optional & Keyword Arguments I

- Optional Arguments
  - allow defaults to be used for missing arguments
  - make some procedures easier to use
- once an argument has been omitted all subsequent arguments must be keyword arguments
- the **present** intrinsic can be used to check for missing arguments
- if used with external procedures then the **interface** must be explicit within the procedure in which it is invoked.

```
subroutine get_temp(vel,boltz)
  use precision
  use param, only : natom, avtemp, mass, kb
  implicit none
  real(dp), dimension(:,:), intent(in) :: vel
  real(dp), optional :: boltz
  integer(ip) :: i
  real(dp) :: ke

  if (present(boltz))kb = boltz
  ke = 0d0
  do i = 1, natom
     ke = ke + dot_product(vel(i,:),vel(i,:))
  end do
  avtemp = mass * ke / ( 3d0 * kb * real( natom - 1, dp))

end subroutine get_temp
```

```
subroutine initialize(coord_t0, vel_t0, acc_t0)
...
  interface
     subroutine linearmom(vel)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(inout) :: vel
     end subroutine linearmom
     subroutine get_temp(vel, boltz)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(in) :: vel
       real(dp), optional :: boltz
     end subroutine get_temp
  end interface
...
  call get_temp(vel_t0)
...
```

Optional & Keyword Arguments II

- Keyword Arguments
  - allow arguments to be specified in any order
  - makes it easy to add an extra argument - no need to modify any calls
  - helps improve readability of the program
  - are used when a procedure has optional arguments
- once a keyword is used, all subsequent arguments must be keyword arguments
- if used with external procedures then the **`interface`** must be explicit within the procedure in which it is invoked.

```
subroutine initialize(coord, vel, acc)

  ...
  real(dp),dimension(:,:), intent(out) :: coord, vel
         , acc
  ...
end subroutine initialize
```

```
program md
        ...
     call initialize(coord_t0, vel_t0, acc_t0)
        ...
end program md
```

Optional & Keyword Arguments III

- **subroutine** initialize can be invoked using
  1. using the positional argument invocation
  2. using keyword arguments

```
program md
...
  interface
     subroutine initialize(coord, vel, acc)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(out) :: coord, vel, acc
     end subroutine initialize
  end interface
...
! All three calls give the same result.
  call initialize(coord_t0, vel_t0, acc_t0)
  call initialize(coord=coord_t0, acc=acc_t0, vel=vel_t0)
  call initialize(coord_t0, acc=acc_t0, vel=vel_t0)
...
```

Dummy Array Arguments

- There are two main types of dummy array argument:
  1. *explicit-shape*: all bounds specified
     ```
     real, dimension(4,4), intent(in):: explicit_shape
     ```
     The actual argument that becomes associated with an explicit shape dummy must conform in size and shape
  2. *assumed-shape*: no bounds specified, all inherited from the actual argument
     ```
     real, dimension(:,:), intent(out):: assumed_shape
     ```
     An explicit interface must be provided
  3. *assumed-size*: final dimension is specified by $*$
     ```
     real :: assumed_size(dim1,dim2,*)
     ```
     Commomly used in FORTRAN, use assumed-shape arrays in Modern Fortran.
- dummy arguments cannot be (unallocated) allocatable arrays.

Explicit-shape Arrays

```
program md
  use precision
  use param
  implicit none
  integer(ip) :: n, i, j, k, l
  real(dp), dimension(:,:), allocatable :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), allocatable :: coord, vel, acc, force
  ...
  ! Allocate arrays
  allocate(coord(natom,3), coord_t0(natom,3))
  allocate(vel(natom,3), vel_t0(natom,3))
  allocate(acc(natom,3), acc_t0(natom,3))
  allocate(force(natom,3))

  !================================================
  ! Initialize coordinates and random velocities
  !================================================

  call initialize(coord_t0, vel_t0, acc_t0)

  ...
end program md

subroutine initialize(coord_t0, vel_t0, acc_t0)
  use precision
  use param, only : natom, npartdim, alat, rcell
  implicit none
  real(dp), dimension(natom,3) :: coord_t0, vel_t0, acc_t0
  integer(ip) :: n, i, j, k, l

  ! Set initial coordinates, velocity and acceleration to zero
  coord_t0 = 0d0 ; vel_t0 = 0d0 ; acc_t0 = 0d0
  ...
end subroutine initialize
```

Assumed-Shape Arrays

```
program md
  use precision
  use param
  implicit none
  integer(ip) :: n, i, j, k, l
  real(dp), dimension(:,:), allocatable :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), allocatable :: coord, vel, acc, force
  ...
  interface
     subroutine initialize(coord_t0, vel_t0, acc_t0)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(out) :: coord_t0, vel_t0, acc_t0
     end subroutine initialize
     ...
  end interface
  ...
  ! Allocate arrays
  allocate(coord(natom,3), coord_t0(natom,3))
  allocate(vel(natom,3), vel_t0(natom,3))
  allocate(acc(natom,3), acc_t0(natom,3))
  allocate(force(natom,3))

  !================================================
  ! Initialize coordinates and random velocities
  !================================================

  call initialize(coord_t0, vel_t0, acc_t0)
  ...
end program md

subroutine initialize(coord_t0, vel_t0, acc_t0)
  use precision
  use param, only : natom, npartdim, alat, rcell
  implicit none
  real(dp), dimension(:,:), intent(out) :: coord_t0, vel_t0, acc_t0
  integer(ip) :: n, i, j, k, l

  ! Set initial coordinates, velocity and acceleration to zero
  coord_t0 = 0d0 ; vel_t0 = 0d0 ; acc_t0 = 0d0
  ...
end subroutine initialize
```

Automatic Arrays

- Automatic Arrays: Arrays which depend on dummy arguments
  1. their size is determined by dummy arguments
  2. they cannot have the `save` attribute or be initialized.
- The **`size`** intrinsic or dummy arguments can be used to declare automatic arrays.

```
program main
  implicit none
  integer :: i,j
  real, dimension(5,6) :: a

  .
  .
  .
  call routine(a,i,j)

  .
  .
  .
  contains
    subroutine routine(c,m,n)
      integer :: m,n
      real, dimension(:,:), intent(inout) :: c    ! assumed shape array
      real :: b1(m,n)                             ! automatic array
      real, dimension(size(c,1),size(c,2)) :: b2 ! automatic array

      .
      .
      .
    end subroutine routine
end program main
```

Save Attribute and Arrays

- Declaring a variable (or array) as `save` gives it a static storage memory.
- i.e information about variables is retained in memory between procedure calls.

```
subroutine something(iarg1)
  implicit none
  integer, intent(in) :: iarg1
  real,dimension(:,:),allocatable,save :: a
  real, dimension(:,:),allocatable :: b

  .
  .
  .
  if (.not.allocated(a))allocate(a(i,j))
  allocate(b(j,i))

  .
  .
  .
  deallocate(b)
end subroutine something
```

- Array `a` is saved when `something` exits.
- Array `b` is not saved and needs to be allocated every time in `something` and deallocated, to free up memory, before `something` exits.

Intent

- `intent` attribute was introduced in Fortran 90 and is recommended as it
  1. allows compilers to check for coding errors
  2. facilitates efficient compilation and optimization
- Declare if a parameter is
  - Input: `intent(in)`
  - Output: `intent(out)`
  - Both: `intent(inout)`

```
subroutine verlet(coord, coord_t0, vel_t0, vel, acc_t0, acc, force, pener)
  use precision
  use param, only : natom, mass, boxl, dt
  implicit none
  real(dp),dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp),dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener

  .
  .
  .
end subroutine verlet
```

- A variable declared as `intent(in)` in a procedure cannot be changed during the execution of the procedure (see point 1 above)

- The `interface` statement is the first statement in an interface block.
- The `interface` block is a powerful structure that was introduced in FORTRAN 90.
- When used, it gives a calling procedure the full knowledge of the types and characteristics of the dummy arguments that are used inside of the procedure that it references.
- This can be a very good thing as it provides a way to execute some safety checks when compiling the program.
- Because the main program knows what argument types should be sent to the referenced procedure, it can check to see whether or not this is the case.
- If not, the compiler will return an error message when you attempt to compile the program.

Interfaces II

```
subroutine verlet(coord, coord_t0, vel, vel_t0, acc, acc_t0, force,
       pener)
  use precision
  use param, only : natom, mass, dt, boxl, pot
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener
  integer(ip) :: i

  interface
     subroutine get_pot_force(coord, force, pener)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(in) :: coord
       real(dp), dimension(:,:), intent(out) :: force
       real(dp), intent(out) :: pener
     end subroutine get_pot_force
  end interface

  ! Set coordinates, velocity, acceleration and force at next time
       step to zero
  coord = 0d0 ; vel = 0d0 ; acc = 0d0

  ! Get new atom positions from Velocity Verlet Algorithm
  coord = coord_t0 + vel_t0 * dt + 0.5d0 * acc_t0 * dt ** 2
  do i = 1, natom
     ! Apply PBC to coordinates
     where ( coord(i,:) > boxl(:) )
        coord(i,:) = coord(i,:) - boxl(:)
     elsewhere ( coord(i,:) < 0d0 )
        coord(i,:) = coord(i,:) + boxl(:)
     end where
  end do

  ! Get Potential and force at new atom positions
  call get_pot_force(coord, force, pener)

  ! Calculate Acceleration and Velocity  at current time step
  acc = force / mass
  vel = vel_t0 + 0.5d0 * ( acc + acc_t0 ) * dt

end subroutine verlet

subroutine get_pot_force(coord, force, pener)
  use precision
  use potential
  use param, only : natom, boxl
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord
  real(dp), dimension(:,:), intent(out) :: force
  real(dp), intent(out) :: pener
  integer(ip) :: i, j
  real(dp) :: epot
  real(dp) :: r(3), f(3)

  pener = 0d0
  force = 0d0
  do i = 1, natom - 1
     do j = i + 1, natom
        r(:) = coord(i,:) - coord(j,:)
        ! minimum image criterion
        r = r - nint( r / boxl ) * boxl
        select case(pot)
        case('mp')
           call morse( r, f, epot )
        case default
           call lennard_jones( r, f, epot )
        end select
        pener = pener + epot
        force(i,:) = force(i,:) + f(:)
        force(j,:) = force(j,:) - f(:)
     end do
  end do

end subroutine get_pot_force
```

Interfaces III

```
subroutine verlet(coord, coord_t0, vel, vel_t0, acc, acc_t0, force,
       pener)
  use precision
  use param, only : natom, mass, dt, boxl, pot
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener
  integer(ip) :: i

  ! Set coordinates, velocity, acceleration and force at next time
         step to zero
  coord = 0d0 ; vel = 0d0 ; acc = 0d0

  ! Get new atom positions from Velocity Verlet Algorithm
  coord = coord_t0 + vel_t0 * dt + 0.5d0 * acc_t0 * dt ** 2
  do i = 1, natom
     ! Apply PBC to coordinates
     where ( coord(i,:) > boxl(:) )
        coord(i,:) = coord(i,:) - boxl(:)
     elsewhere ( coord(i,:) < 0d0 )
        coord(i,:) = coord(i,:) + boxl(:)
     end where
  end do

  ! Get Potential and force at new atom positions
  call get_pot_force(coord, force, pener)

  ! Calculate Acceleration and Velocity  at current time step
  acc = force / mass
  vel = vel_t0 + 0.5d0 * ( acc + acc_t0 ) * dt

contains
```

```
  subroutine get_pot_force(coord, force, pener)
    use potential
    implicit none
    real(dp), dimension(:,:), intent(in) :: coord
    real(dp), dimension(:,:), intent(out) :: force
    real(dp), intent(out) :: pener
    integer(ip) :: i, j
    real(dp) :: epot
    real(dp) :: r(3), f(3)

    pener = 0d0
    force = 0d0
    do i = 1, natom - 1
       do j = i + 1, natom
          r(:) = coord(i,:) - coord(j,:)
          ! minimum image criterion
          r = r - nint( r / boxl ) * boxl
          select case(pot)
          case('mp')
             call morse( r, f, epot )
          case default
             call lennard_jones( r, f, epot )
          end select
          pener = pener + epot
          force(i,:) = force(i,:) + f(:)
          force(j,:) = force(j,:) - f(:)
       end do
    end do

  end subroutine get_pot_force

end subroutine verlet
```

- Here since **subroutine** get_pot_force is an internal procedure, no **interface** is required since it is already implicit and all variable declarations are carried over from **subroutine** verlet

Modules I

- Modules were introduced in Fortran 90 and have a wide range of applications.
- Modules allow the user to write object based code.
- A **`module`** is a program unit whose functionality can be exploited by other programs which attaches to it via the **`use`** statement.
- A **`module`** can contain the following
  1. global object declaration: replaces Fortran 77 **`COMMON`** and **`INCLUDE`** statements
  2. interface declaration: all external procedures using assumed shape arrrays, intent and keyword/optional arguments must have an explicit interface
  3. procedure declaration: include procedures such as subroutines or functions in modules. Since modules already contain explicit interface, an interface statement is not required

Modules II

```
module precision
  implicit none
  save
  integer, parameter :: ip = selected_int_kind(15)
  integer, parameter :: dp = selected_real_kind(15)
end module precision

module param
  use precision
  implicit none
  integer(ip) :: npartdim, natom, nstep, istep
  real(dp) :: tempK, dt, boxl(3), alat, mass
  real(dp) :: avtemp, ke, kb, epsilon, sigma, scale
  real(dp),dimension(3,4) :: rcell = reshape( (/ &
       0.0D+00, 0.0D+00, 0.0D+00, &
       0.5D+00, 0.5D+00, 0.0D+00, &
       0.0D+00, 0.5D+00, 0.5D+00, &
       0.5D+00, 0.0D+00, 0.5D+00 /), (/ 3, 4 /) )
  character(len=2) :: pot
end module param
```

- within a **module**, functions and subroutines are called module procedures.
- **module** procedures can contain internal procedures
- **module** objects that retain their values should be given a `save` attribute
- **module**s can be used by procedures and other modules, see **module precision**.
- **module**s can be compiled separately. They should be compiled before the program unit that uses them.

  Observe that in my examples with all code in single file, the **module**s appear before the main program and subroutines.

Modules III

Visibility of module procedures

- By default, all module procedures are public i.e. they can accessed by program units that use the module using the `use` statement
- To restrict the visibility of the module procedure only to the module, use the `private` statement
- In the `module potential`, all functions which calculate forces can be declared as private as follows

```
module potential
  use precision
  implicit none
  real(dp) :: r2, r6, d2, d
  real(dp), parameter :: de = 0.176d0, a = 1.4d0, re = 1d0
  real(dp) :: exparre
  public :: lennard_jones, morse, pot_lj, pot_mp
  private :: dvdr_lj, dvdr_mp

contains
  ...
end module potential
```

- Program Units in the MD code can directly call `lennard_jones`, `morse`, `pot_lj` and `pot_mp` but cannot access `dvdr_lj` and `dvdr_mp`

Using Modules

- The **use** statement names a module whole public definitions are to be made accessible.

  To use all variables from **module** param in **program** md:

```
program md
  use param
  ...
end program md
```

- **module** entities can be renamed

  To rename pot and dt to more user readable variables:

```
use param, pot => potential, dt => timestep
```

- It's good programming practice to use only those variables from modules that are neccessary to avoid name conflicts and overwrite variables.
- For this, use the **use** <modulename>, **only** statement

```
subroutine verlet(coord,force,pener)
  use param,only : dp,npart,boxl,tstep
  ...
end subroutine verlet
```

Compiling Modules I

- Consider the MD code containing a main program `md.f90`, modules `precision.f90`, `param.f90` and `potential.f90` and subroutines `initialize.f90`, `verlet.f90`, `linearmom.f90` and `get_temp.f90`.
- In general, the code can be compiled as

```
ifort -o md md.f90 precision.f90 param.f90 potential.f90 initialize.f90 \
   verlet.f90 linearmom.f90 get_temp.f90
```

- Most compilers are restrictive in the order of compilation.
- The order in which the sub programs should be compiled is
  1. Modules that do not use any other modules.
  2. Modules that use one or more of the modules already compiled.
  3. Repeat the above step until all modules are compiled and all dependencies are resolved.
  4. Main program followed by all subroutines and functions (if any).
- In the MD code, the module `precision` does not depend on any other modules and should be compiled first
- The modules `param` and `potential` only depend on `precision` and can be compiled in any order

Compiling Modules II

- The main program and subroutines can then be compiled

```
ifort -o md md.f90 precision.f90 param.f90 potential.f90 initialize.f90 \
   verlet.f90 linearmom.f90 get_temp.f90
```

- modules are designed to be compiled independently of the main program and create a `.mod` files which need to be linked to the main executable.

```
ifort -c precision.f90 param.f90 potential.f90
        creates precision.mod param.mod potential.mod
```

- The main program can now be compiled as

```
ifort -o md md.f90 initialize.f90 verlet.f90 linearmom.f90 get_temp.f90 \
   -I{path to directory containing the .mod files}
```

- The Makefile tutorial will cover this aspect in more detail.

Derived Types

Derived Types I

- Defined by user (also called structures)
- Can include different intrinsic types and other derived types
- Components are accessed using the percent operator (%)
- Only assignment operator (=) is defined for derived types
- Can (re)define operators - see operator overloading
- Derived type definitions should be placed in a **`module`**.
- Previously defined type can be used as components of other derived types.

```
type line_type
  real :: x1, y1, x2, y2
end type line_type

type(line_type) :: a, b

type vector_type
  type(line_type) :: line ! defines x1,y1,x2,y2
  integer :: direction ! 0=nodirection, 1=(x1,y1)->(x2,y2)
end type vector_type

type(vector_type) :: c, d
```

Derived Types II

- values can be assigned to derived types in two ways
  1. component by component
     individual component may be selected using the % operator
  2. as an object
     the whole object may be selected and assigned to using a constructor

```
a%x1 = 0.0 ; a%x2 = 0.5 ; a%y1 = 0.0 ; a%y2 = 0.5

c%direction = 0
c%line%x1 = 0.0 ; c%line%x2 = 1.0
c%line%y1 = -1.0 ; c%line%y2 = 0.0

b = line_type(0.0, 0.0, 0.5, 0.5)

d%line = line_type(0.0, -1.0, 1.0, 0.0)}
d = vector_type( d%line, 1 )
! or
d = vector_type( line_type(0.0, -1.0, 1.0, 0.0), 1)
```

Derived Types III

- Assigment between two objects of the same derived type is intrinsically defined

  In the previous example: `a = b` is allowed but `a = c` is not.

```
coord_t0(n)%x = alat * real(i - 1, dp) + rcell(1,1)
coord_t0(n)%y = alat * real(j - 1, dp) + rcell(2,1)
coord_t0(n)%z = alat * real(k - 1, dp) + rcell(3,1)
  OR
x = alat * real(i - 1, dp) + rcell(1,1)
y = alat * real(j - 1, dp) + rcell(2,1)
z = alat * real(k - 1, dp) + rcell(3,1)
coord_t0(n) = dynamics( x, y, z )
```

- I/O on Derived Types
  - Can do normal I/O on derived types

    `print *, a` will produce the result `1.00.51.5`

    `print *, c` will produce the result `2.00.00.00.0`
- Arrays and Derived Types
  - Can define derived type objects which contain non-allocatable arrays and arrays of derived type objects
- Derived Type Valued Functions

Derived Types IV

- Functions can return results of an arbitrary defined type.

- Private Derived Types
  - A derived type can be wholly private or some of its components hidden

```
module data
  type :: position
    real, private :: x, y, z
  end type position
  type, private :: acceleration
    real, private :: x, y, z
  end type acceleration
  contains

  .
  .
  .
end module data
```

  - Program units that use **`data`** have **`position`** exported but not it's components `x, y, z` and the derived type `acceleration`

- In Fortran, most intrinsic functions are generic in that their type is determined by their argument(s)
- For example, the `abs(x)` intrinsic function comprises of
  1. `cabs` : called when `x` is **`complex`**
  2. **`abs`** : called when `x` is **`real`**
  3. `iabs` : called when `x` is **`integer`**
- These sets of functions are called *overload sets*
- Fortran users may define their own *overload sets* in an **`interface`** block

```
interface clear
   module procedure clear_real, clear_type, clear_type1D
end interface
```

- The generic name `clear` is associated with specific names `clear_real`, `clear_type`, `clear_type1D`

Generic Procedures II

```
module dynamic_data
  ...
  type dynamics
     real(dp) :: x,y,z
  end type dynamics
  interface dot_product
     module procedure dprod
  end interface dot_product
  interface clear
     module procedure clear_real, clear_type,
           clear_type1D
  end interface
contains
  function dprod(a,b) result(c)
    type(dynamics),intent(in) :: a,b
    real(dp) :: c
    c = a%x * b%x + a%y * b%y + a%z * b%z
  end function dprod
  subroutine clear_real(a)
    real(dp),dimension(:,:),intent(out) :: a
    a = 0d0
  end subroutine clear_real

  subroutine clear_type(a)
    type(dynamics),dimension(:),intent(out) ::
          a
    a%x = 0d0 ; a%y = 0d0 ; a%z = 0d0
  end subroutine clear_type

  subroutine clear_type1D(a)
    type(dynamics),intent(out) :: a
    a%x = 0d0 ; a%y = 0d0 ; a%z = 0d0
  end subroutine clear_type1D
end module dynamic_data
```

```
program md
  use dynamic_data
  ...
  type(dynamics),dimension(:),allocatable :: coord,coord
       0,vel,force
  ...
  allocate(coord(npart),coord0(npart),vel(npart),force(
       npart))
  ...
     do i=1,npart
        v2t = v2t + dot_product(vel(i),vel(i))
     enddo
  ...
end program md

subroutine setup(coord,vel,coord0)
  ...
  type(dynamics) :: vt
  ...
  call clear(coord)
  call clear(coord0)
  call clear(vel)
  ...
  call clear(vt)
  ...
end subroutine setup
```

Generic Procedures III

- The `dot_product` intrinsic function is overloaded to inlcude derived types
- The procedure `clear` is overloaded to set all components of derived types and all elements of 2D real arrays to zero.

- Intrinsic operators such as +, -, * and / can be overloaded to apply to all types of data
- Recall, for derived types only the assignment (=) operator is defined
- In the MD code, `coord_t(i)= coord_t0(i)` is well defined, but `vel_t(i)= vel_t(i)* scalef` is not
- Operator overloading as follows
  1. specify the generic operator symbol in an **`interface operator`** statement
  2. specify the overload set in a generic interface
  3. declare the **`module`** `procedure`s (**`function`**s) which define how the operations are implemented.
  4. these functions must have one or two non-optional arguments with **`intent`**`(`**`in`**`)` which correspond to monadic or dyadic operators

Operator Overloading II

```
module dynamic_data
  ...
  type dynamics
     real(dp) :: x,y,z
  end type dynamics

  interface operator (*)
     module procedure scale_tr, scale_rt
  end interface operator (*)
  interface operator (+)
     module procedure add
  end interface operator (+)
contains
  type(dynamics) function scale_tr(a,b) result(c)
    type(dynamics),intent(in)::a
    real(dp),intent(in) :: b
    type(dynamics) :: c
    c%x = a%x * b
    c%y = a%y * b
    c%z = a%z * b
  end function scale_tr
  type(dynamics) function scale_rt(b,a) result(c)
    type(dynamics),intent(in)::a
    real(dp),intent(in) :: b
    type(dynamics) :: c
    c%x = b * a%x
    c%y = b * a%y
    c%z = b * a%z
  end function scale_rt
  type(dynamics) function add(a,b) result(c)
    type(dynamics),intent(in) :: a,b
    type(dynamics) :: c
    c%x = a%x + b%x
    c%y = a%y + b%y
    c%z = a%z + b%z
  end function add
end module dynamic_data
```

- The following operations are now defined for derived types `a,b,c` and scalar `r`

```
c = a * r
c = r * a
c = a + b
```

- If operator overloading is not defined, the above operations would have to be executed as follows whereever needed

```
c%x = a%x * r
c%y = a%y * r
c%z = a%z * r

c%x = r * a%x
c%y = r * a%y
c%z = r * a%z

c%x = a%x + b%x
c%y = a%y + b%y
c%z = a%z + b%z
```

Object Based Programming

- Fortran 90 has some Object Oriented facilites such as
  1. data abstraction: user defined types (covered)
  2. data hiding - private and public attributes (covered)
  3. encapsulation - modules and data hiding facilities (covered)
  4. inheritance and extensibility - super-types, operator overloading and generic procedures (covered)
  5. polymorphism - user can program his/her own polymorphism by generic overloading
  6. resuability - modules

- In Fortran, a `pointer` variable or simply a `pointer` is best thought of as a "free-floating" name that may be associated with or "aliased to" some object.
- The object may already have one or more other names or it may be an unnamed object.
- The object represent data (a variable, for example) or be a procedure.
- A `pointer` is any variable that has been given the `pointer` attribute.
- A variable with the `pointer` attribute may be used like any ordinary variable.

- Each pointer is in one of the following three states:

undefined	condition of each **pointer** at the beginning of a **program**, unless it has been initialized
null	not an alias of any data object
associated	it is an alias of some target data object

- **pointer** objects must be declared with the **pointer** attribute

```
real, pointer :: p
```

- Any variable aliased or "pointed to" by a **pointer** must be given the **target** attribute

```
real, target :: r
```

- To make `p` an alias to `r`, use the **pointer assignment** statement

```
p => r
```

- The variable declared as a **pointer** may be a simple variable as above, an array or a structure

```
real, dimension(:), pointer :: v
```

- **pointer** v declared above can now be aliased to a 1D array of reals or a row or column of a multi-dimensional array

```
real, dimension(100,100), target :: a
```

```
v => a(5,:)
```

- **pointer** variables can be used as any other variables

  For example, **print** *, v and **print** *, a(5,:) are equivalent

  v = 0.0 is the same as a(5,:)= 0.0

- **pointer** variables can also be an alias to another **pointer** variable

- Consider the following example

```
real, target :: r
real, pointer :: p1, p2
r = 4.7
p1 => r
p2 => r
print *, r, p1, p2
r = 7.4
print *, r, p1, p2
```

- The output on the screen will be

```
4.7   4.7   4.7
7.4   7.4   7.4
```

- Changing the value of `r` to 7.4 causes the value of both `p1` and `p2` to change to 7.4

- Consider the following example

```
real, target :: r1, r2
real, pointer :: p1, p2
r1 = 4.7 ; r2 = 7.4
p1 => r1 ; p2 => r2
print *, r1, r2, p1, p2
p1 = p2
print *, r1, r2, p1, p2
```

- The output on the screen will be

```
4.7   7.4   4.7   7.4
4.7   4.7   4.7   4.7
```

- The assignment statement `p2 = p1` has the same effect of `r2 = r1` since `p1` is an alias to `r1` and `p2` is an alias to `r2`

- The **`allocate`** statement can be used to create space for a value and cause a pointer to refer to that space.

`allocate(p1)` creates a space for one real number and makes `p1` an alias to that space.

- No real value is stored in that space so it is neccessary to assign a value to `p1`
- `p1 = 4.7` assigns a value 4.7 to that allocated space
- Before a value is assigned to `p1`, it must either be associated with an unnamed target using the **`allocate`** statement or be aliased with a target using the pointer assignment statement.
- **`deallocate`** statement dissociates the pointer from any target and nullifies it

```
deallocate(p1)
```

- null intrinsic
  - **pointer** variables are undefined unless they are initialized
  - **pointer** variable must not be reference to produce a value when it is undefined.
  - It is sometime desirable to have a **pointer** variable in a state of not pointing to anything
  - The **null** intrinsic function nullifies a pointer assignment so that it is in a state of not pointing to anything
    `p1 => null()`
  - If the target of `p1` and `p2` are the same, then nullifying `p1` does not nullify `p2`
  - If `p1` is null and `p2` is pointing to `p1`, then `p2` is also nullified.
- associated intrinsic
  - The **associated** intrinsic function queries whether a pointer varibale is pointing to, or is an alias for another object.
    associated`(p1,r1)` and **associated**`(p2,r2)` are true, but
    associated`(p1,r2)` and **associated**`(p2,r1)` are false

- Recall the derived type example which has as a component another derived type

```
type, public :: line_type
  real :: x1, y1, x2, y2
end type line_type
type, public :: vector_type
  type(line_type) :: line !position of center of sphere
  integer :: direction ! 0=no direction, 1=(x1,y1)->(x2,y2)
end type vector_type
```

- An object, `c`, of type `vector_type` is referenced as `c%line%x1`, `c%line%y1`, `c%line%x2`, `c%line%y2` and `c%direction` which can be cumbersome.

- In Fortran, it is possible to extend the base type `line_type` to other types such as `vector_type` and `painted_line_type` as follows

```
type, public, extends(line_type) :: vector_type
  integer :: direction
end type vector_type
type, public, extends(line_type) :: painted_line_type
  integer :: r, g, b ! rgb values
end type painted_line_type
```

- An object,`c` of type `vector_type` inherits the components of the type `line_type` and has components `x1,y1,x2,y2` and `direction` and is referenced as `c%x1, c%y1, c%x1, c%y2` and `c%direction`
- Similarly, an object, `d` of type `painted_line_type` is referenced as `d%x1, d%y2, d%x2, d%y2, d%r, d%g` and `d%b`
- The three derived types constitute a `class`; the name of the class is the name of the base type `line_type`

- Fortran 95/2003 Explained, Michael Metcalf
- Modern Fortran Explaned, Michael Metcalf
- Guide to Fortran 2003 Programming, Walter S. Brainerd
- Introduction to Programming with Fortran: with coverage of Fortran 90, 95, 2003 and 77, I. D. Chivers
- Fortran 90 course at University of Liverpool, `http://www.liv.ac.uk/HPC/F90page.html`
- Introduction to Modern Fortran, University of Cambridge, `http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/Fortran`
- Scientific Programming in Fortran 2003: A tutorial Including Object-Oriented Programming, Katherine Holcomb, University of Virginia.

Exercise

- Molecular Dynamics code for melting of solid Argon using Lennard-Jones Potential.
- Your goal is to rewrite the code using Modern Fortran concepts that you have grasped.
- This exercise is more of a "What concepts have I learned of Modern Fortran?", so there are multiple correct solutions
- Code can be obtained from `http://www.hpc.lsu.edu/training/archive/tutorials.php`:
- md-orig.f90 is the original code that you should begin working on (this is the same code that was shown in todays slides)
- There is no "correct solution", however there are multiple solutions md-v{1-5}.f90 based on various concepts presented.
- It's entirely up to you to decide which solution you want to arrive at.
- Compare the results of your edited code with that of md-v0.out. If the results are not the same, debug your code.

Calculate pi by Numerical Integration I

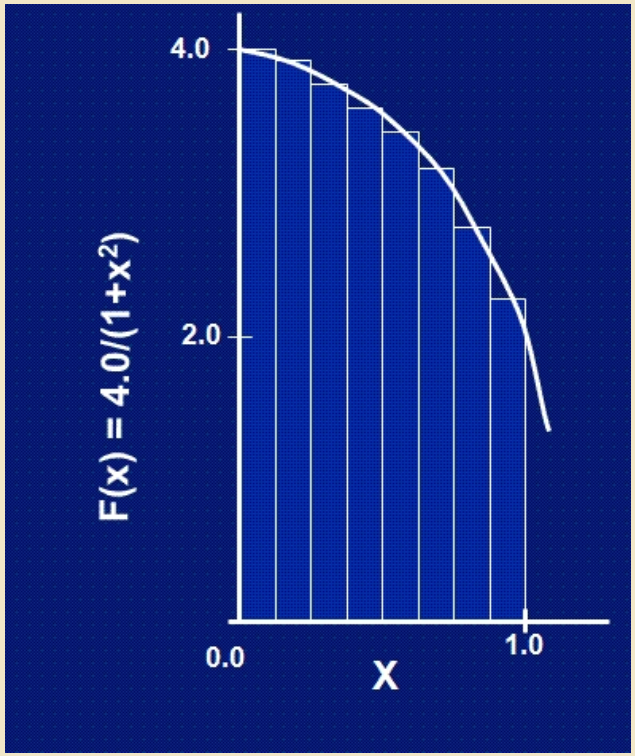
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A “hands-on” introduction to OpenMP, SC09



Calculate pi by Numerical Integration II

Algorithm 1 Pseudo Code for Calculating Pi

program CALCULATE_PI
  $step \leftarrow 1/n$
  $sum \leftarrow 0$
  do $i \leftarrow 0 \cdots n$
    $x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$
  end do
  $pi \leftarrow sum * step$
end program

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$
- Write a SAXPY code to multiply a vector with a scalar.

Algorithm 2 Pseudo Code for SAXPY

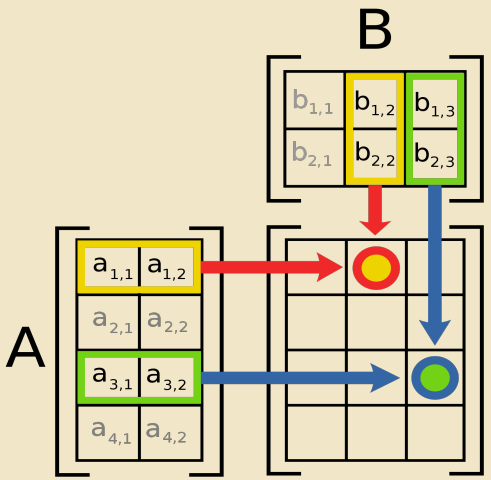
---

program SAXPY
  $n \leftarrow$ some large number
  $x(1:n) \leftarrow$ some number say, 1
  $y(1:n) \leftarrow$ some other number say, 2
  $a \leftarrow$ some other number ,say, 3
  do $i \leftarrow 1 \cdots n$
    $y_i \leftarrow y_i + a * x_i$
  end do
end program SAXPY

Matrix Multiplication I

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Matrix Multiplication II

Algorithm 3 Pseudo Code for MATMUL

program MATMUL
  $m, n \leftarrow$ some large number ≤ 1000
  Define a_{mn}, b_{nm}, c_{mm}
  $a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$
  do $i \leftarrow 1 \cdots m$
    do $j \leftarrow 1 \cdots m$
      $c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$
    end do
  end do
end program MATMUL