Arrays and Vectors

Victor Eijkhout, Harika Gurram,
Je'aime Powell, Charley Dey

Fall 2018

Fortran dimension

Preferred way of creating arrays through `dimension` keyword:

```
real(8), dimension(100) :: x,y
```

One-dimensional arrays of size 100.

Older mechanism works too:

```
integer :: i(10,20)
```

Two-dimensional array of size 10×20.

These arrays are statically defined, and only live inside their program unit.

1-based Indexing

```
integer,parameter :: N=8
real(4),dimension(N) :: x
do i=1,N
  ... x(i) ...
```

Lower bound

```
real,dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Array initialization

```
real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
/* ... */
real5 = [ (1.01*i,i=1,size(real5,1)) ]
/* ... */
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

Array sections example

Use the colon notation to indicate ranges:

```
real(4),dimension(4) :: y
real(4),dimension(5) :: x
x(1:4) = y
x(2:5) = x(1:4)
```

Use of sections

Code:

```
real(8),dimension(5) :: x = &
     [.1d0, .2d0, .3d0, .4d0, .5d0]
x(2:5) = x(1:4)
print '(f5.3)',x
```

Output
[arrayf] sectionassign:

```
0.100
0.100
0.200
0.300
0.400
```

Exercise 1

Code out the above array assignment with an explicit, indexed loop. Do you get the same output? Why? What conclusion do you draw about internal mechanisms used in array sections?

Strided sections

Code:

```
integer,dimension(5) :: &
     y = [0,0,0,0,0]
integer,dimension(3) :: &
     z = [3,3,3]
y(1:5:2) = z(:)
print '(i3)',y
```

Output
[arrayf] sectionmg:

```
  3
  0
  3
  0
  3
```

Index arrays

```
integer,dimension(4) :: i = [2,4,6,8]
real(4),dimension(10) :: x
print *,x(i)
```

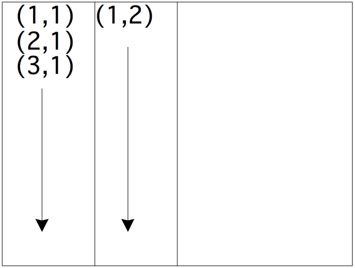
Multi-dimension arrays

```
real(8),dimension(20,30) :: array
array(i,j) = 5./2
```

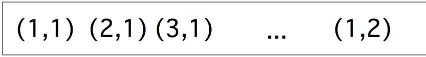
Array layout

Sometimes you have to take into account how a higher rank array is laid out in (linear) memory:

Fortran column major



Physical:



'First index varies quickest'

Array sections in multi-D

```
real(8),dimension(10) :: a,b
a(1:9) = b(2:10)
```

or

```
logical,dimension(25,3) :: a
logical,dimension(25)   :: b
a(:,2) = b
```

You can also use strides.

Query functions

- Bounds: `lbound`, `ubound`
- `size`

```
integer :: x(8), y(5,4)
size(x)
size(y,2)
```

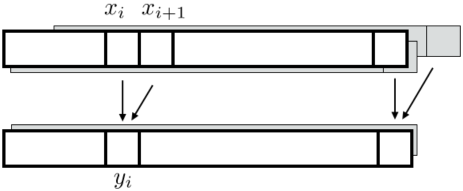
Pass array to subroutine

```
  real(8) function arraysum(x)
    implicit none
    real(8),intent(in),dimension(:) :: x
  /* ... */
    do i=1,size(x)
       tmp = tmp+x(i)
    end do
  /* ... */
Program ArrayComputations1D
    use ArrayFunction
    implicit none

    real(8),dimension(:) :: x(N)
  /* ... */
    print *,"Sum of one-based array:",arraysum(x)
```

(note: the function is in a module)

Exercise 2



Code $\forall_i : y_i = (x_i + x_{i+1})/2$:

- First with a do loop; then
- in a single array assignment statement by using sections.

Initialize the array `x` with values that allow you to check the correctness of your code.

Array allocation

```
real(8), dimension(:), allocatable :: x,y

n = 100
allocate(x(n), y(n))
```

You can `deallocate` the array when you don't need the space anymore.

Array intrinsics

- `MaxVal` finds the maximum value in an array.
- `MinVal` finds the minimum value in an array.
- `Sum` returns the sum of all elements.
- `Product` return the product of all elements.
- `MaxLoc` returns the index of the maximum element.

  ```
  i = MAXLOC( array [, mask ] )
  ```
- `MinLoc` returns the index of the minimum element.
- `MatMul` returns the matrix product of two matrices.
- `Dot_Product` returns the dot product of two arrays.
- `Transpose` returns the transpose of a matrix.
- `Cshift` rotates elements through an array.

Exercise 3

The 1-norm of a matrix is defined as the maximum sum of absolute values in any column:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

while the infinity-norm is defined as the maximum row sum:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Implement these functions using array intrinsics.

Exercise 4

Compare implementations of the matrix-matrix product.

1. Write the regular `i,j,k` implementation, and store it as reference.
2. Use the `DOT_PRODUCT` function, which eliminates the `k` index. How does the timing change? Print the maximum absolute distance between this and the reference result.
3. Use the `MATMUL` function. Same questions.
4. Bonus question: investigate the `j,k,i` and `i,k,j` variants. Write them both with array sections and individual array elements. Is there a difference in timing?

Does the optimization level make a difference in timing?

Timer routines

```
integer :: clockrate,clock_start,clock_end
call system_clock(count_rate=clockrate)
/* ... */
call system_clock(clock_start)
/* ... */
call system_clock(clock_end)
print *,"time:",(clock_end-clock_start)/REAL(clockrate)
```

operate where

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )
  sequence of array statements
ELSEWHERE
  sequence of array statements
END WHERE
```

Do concurrent

The *do concurrent* is a true do-loop. With the `concurrent` keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
do concurrent (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end do
```