6 Processamento de Matrizes

A definição e o processamento de matrizes e vetores sempre foi um recurso presente em todas as linguagens de programação, inclusive no Fortran 77. Uma novidade importante introduzida no Fortran 90 é a capacidade estendida de processamento das mesmas. A partir de então é possível trabalhar diretamente com a matriz completa, ou com seções da mesma, sem ser necessário o uso de laços `DO`. Novas funções intrínsecas agora atuam de forma *elemental* (em todos os elementos) em matrizes e funções podem retornar valores na forma de matrizes. Também estão disponíveis as possibilidades de matrizes alocáveis, matrizes de forma assumida e matrizes dinâmicas.

Estes e outros recursos serão abordados neste e nos próximos capítulos.

6.1 Terminologia e especificações de matrizes

Uma *matriz* ou *vetor*[1] é um outro tipo de objeto composto suportado pelo Fortran. Uma matriz consiste em um conjunto retangular de elementos, todos do mesmo tipo e espécie do tipo. Uma outra definição equivalente seria: uma matriz é um grupo de posições na memória do computador as quais são acessadas por intermédio de um único nome, fazendo-se uso dos **subscritos** da matriz. Este tipo de objeto é útil quando for necessário se fazer referência a um número grande, porém a princípio desconhecido, de variáveis do tipo intrínseco ou outras estruturas, sem que seja necessário definir um nome para cada variável.

O Fortran permite que uma matriz tenha até 15 (quinze) subscritos, cada um relacionado com uma dimensão da matriz. Os índices de cada subscrito da matriz são constantes ou variáveis inteiras e, por convenção, eles começam em 1, exceto quando um intervalo distinto de valores é especificado, através do fornecimento de um limite inferior e um limite superior.

Uma matriz pode ser declarada de diversas maneiras distintas. Neste capítulo, serão consideradas somente matrizes com um número fixo de elementos. O Fortran suporta também *matrizes alocáveis*, as quais são matrizes cujo número total de elementos pode ser definido dinamicamente durante a execução do programa. Este tipo de objeto dinamico de dados será abordado na seção 7.1.

Suponha que seja necessário declarar o vetor `A`, composto por 5 (cinco) elementos reais. Até o padrão Fortran 77, este vetor poderia ser declarado diretamente na declaração de tipo,

```
REAL A(5)
```

ou por meio da declaração `DIMENSION`,

```
REAL A
DIMENSION A(5)
```

Ambas as maneiras continuam sendo válidas e ambas criam um total de 10 variáveis reais com um nome em comum. Quando o nome `A` aparece sem os parênteses, este identifica a **matriz completa** (*whole array*), ao passo que as variáveis individuais armazenadas no vetor `A` são denominadas os **elementos da matriz** (*array elements*) e são acessadas por meio dos subscritos `A(1)`, `A(2)`, ..., `A(5)`, podendo aparecer tanto em expressões (lado direito da igualdade) quanto em atribuições (lado esquerdo). Na memória do computador, os elementos de uma matriz de tamanho fixo são armazenados em espaços contíguos de memória, sendo assim rapidamente acessados. Isto é ilustrado na figura 6.1.

[1] Nome usualmente empregado para uma matriz de uma dimensão.

Uma terceira forma de declaração do vetor A faz uso de uma constante inteira:

```
INTEGER, PARAMETER :: NMAX= 5
REAL A(NMAX)
```

Outros exemplos que fazem uso destas formas de declarações de matrizes são:

```
INTEGER NMAX
INTEGER POINTS(NMAX),MAT_I(50)
REAL R_POINTS(0:50), B
DIMENSION B(NMAX,50)
CHARACTER COLUMN(5)*25, ROW(10)*30
```

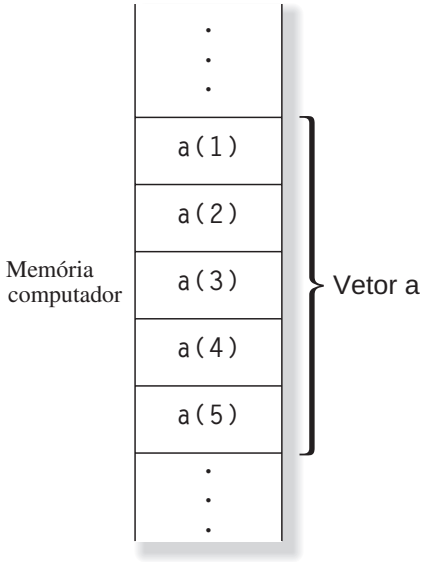

Figura 6.1: Os elementos do vetor *a* ocupam posições sucessivas na memória do computador.

No último exemplo acima, o vetor COLUMN possui 5 elementos, COLUMN(1), COLUMN(2), ..., COLUMN(5), cada um destes sendo uma variável de caractere de comprimento 25. Já o vetor ROW possui 10 elementos, cada um sendo uma variável de caractere de comprimento 30. A matriz real B possui 2 dimensões, sendo NMAX linhas e 50 colunas. Todas as matrizes neste exemplo têm seus índices iniciando em 1, exceto pela matriz real R_POINTS, a qual inicia em 0: R_POINTS(0), R_POINTS(1), ..., R_POINTS(50). Ou seja, este vetor possui 51 componentes.

A partir do Fortran 90, as dimensões de uma matriz podem ser especificadas também empregando-se o *atributo* DIMENSION. Por exemplo,

```
REAL, DIMENSION(5) :: A
REAL, DIMENSION(5:54) :: X ! Elementos de X: X(5), X(6), ..., X(54)
CHARACTER(LEN= 25), DIMENSION(5) :: COLUMN
CHARACTER(LEN= 30), DIMENSION(10) :: ROW
```

As formas de declarações de matrizes em Fortran 77 são aceitas no Fortran Moderno. Porém, é recomendável que estas sejam declaradas na forma de *atributos* de tipos de variáveis.

Antes de prosseguir, será introduzida a terminologia usada com relação a matrizes.

Posto. O *posto* (*rank*) de uma matriz é o número de dimensões da mesma. Assim, um escalar tem posto 0, um vetor tem posto 1 e uma matriz tem posto maior ou igual a 2.

Limites. Os *limites* (*bounds*) de uma matriz em uma dada dimensão são o menor e o maior valores dos índices naquela dimensão.

Extensão. A *extensão* (*extent*) de uma matriz também se refere a uma dimensão em particular e é o número de componentes naquela dimensão.

Forma. A *forma* (*shape*) de uma matriz é um vetor cujos componentes são a extensão de cada dimensão da matriz.

Tamanho. O *tamanho* (*size*) de um matriz é o número total de elementos que compõe a mesma. Este número pode ser zero, em cujo caso esta se denomina **matriz de tamanho zero** (*zero-sized array*).

Duas matrizes são ditas serem **conformáveis** (*conformable*) se elas têm a mesma forma. Todas as matrizes são conformáveis com um escalar, uma vez que o escalar é expandido em uma matriz com a mesma forma (ver seção 6.2).

Por exemplo, dadas as seguintes matrizes:

```
REAL, DIMENSION(-3:4,7) :: A
REAL, DIMENSION(8,2:8) :: B
REAL, DIMENSION(8,0:8) :: D
INTEGER :: C
```

A matriz A possui:

- posto 2;
- extensões 8 e 7;
- forma `[8,7]`, onde os símbolos "`[`" e "`]`" são os *construtores de matrizes*, isto é, eles definem ou inicializam os valores de um vetor ou matriz. Estes construtores de matrizes serão descritos com mais detalhes na seção 6.6.
- Tamanho $8 \times 7 = 56$.

Além disso, `A` é conformável com `B` e `C`, uma vez que a forma de `B` também é `[8,7]` e `C` é um escalar. Contudo, `A` não é conformável com `D`, uma vez que esta última tem forma `[8,9]`.

A forma geral da declaração de uma ou mais matrizes é como se segue:

```
<tipo-esp>[[, DIMENSION(<lista-exten>)] [, <atributos>] ::] <lista-nomes>
```

Entretanto, a forma recomendável da declaração é a seguinte:

```
<tipo-esp>, DIMENSION(<lista-exten>) [, <outros atributos>] :: <lista-nomes>
```

Onde `<tipo-esp>` pode ser um tipo e espécie intrínsecos de variável ou um tipo derivado (desde que a definição do tipo derivado esteja acessível). A `<lista-exten>` fornece as dimensões e extensões da matriz, através de:

- constantes inteiras;
- expressões inteiras usando variáveis **mudas** (*dummy*) ou constantes;
- somente o caractere "`:`" para indicar que a matriz é alocável (seção 7.1) ou de forma assumida (seção 9.2.11.2).

Os outros `<atributos>` podem ser quaisquer da seguinte lista:

```
PARAMETER   ALLOCATABLE   INTENT(INOUT)   OPTIONAL
SAVE        EXTERNAL      INTRINSIC       PUBLIC
PRIVATE     POINTER       TARGET
```

Os atributos contidos na lista acima serão abordados ao longo deste e dos próximos capítulos.

Finalmente, segue a `<lista-nomes>` válidos no Fortran, onde os mesmos são atribuídos às matrizes. Os seguintes exemplos mostram a forma da declaração de diversos tipos diferentes de matrizes.

1. Inicialização de vetores contendo 3 elementos:

```
INTEGER :: I
INTEGER, DIMENSION(3) :: IA= [1,2,3], IB= [(I, I=1,3)]
```

onde `IA` foi inicializada com um construtor de matrizes e `IB` foi inicializada via um *`DO` implícito*. Ambos os recursos serão discutidos na seção 6.6.

2. Declaração da matriz automática `LOGB`:

```
LOGICAL, DIMENSION(SIZE(LOGA)) :: LOGB
```

Aqui, a matriz `LOGA` é um argumento mudo de uma rotina (capítulo 9) e `SIZE` é uma função intrínseca que retorna um escalar inteiro correspondente ao tamanho do seu argumento.

3. Declaração das matrizes dinâmicas, ou alocáveis, de duas dimensões (posto 2) `A` e `B`:

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A,B
```

A forma das matrizes será definida *a posteriori* por um comando `ALLOCATE`, discutido na seção 7.1.

4. Declaração das matrizes de *forma assumida* de três dimensões `A` e `B`:

```
REAL, DIMENSION(:,:,:) :: A,B
```

A forma das matrizes será assumida a partir das informações transferidas pela rotina que aciona o sub-programa onde esta declaração é feita. Este recurso também será discutido no capítulo 9.

MATRIZES DE TIPOS DERIVADOS. A capacidade de se misturar matrizes com definições de tipos derivados possibilita a construção de objetos de complexidade crescente. Alguns exemplos ilustram estas possibilidades.

Um tipo derivado pode conter um ou mais componentes que são matrizes:

```
TYPE :: TRIPLETO
   REAL :: U
   REAL, DIMENSION(3) :: DU
   REAL, DIMENSION(3,3) :: D2U
END TYPE TRIPLETO
TYPE(TRIPLETO) :: T
```

Este exemplo serve para definir, em uma única estrutura, um tipo de variável denominado `TRIPLETO`, cujos componentes correspondem ao valor de uma função de 3 variáveis (componente `U`), suas 3 derivadas parciais de primeira ordem (componente `DU`) e suas 9 derivadas parciais de segunda ordem (componente `D2U`). Se a variável `T` é do tipo `TRIPLETO`, `T%U` é um escalar real, mas `T%DU` e `T%D2U` são matrizes do tipo real.

É possível agora realizar-se combinações entre matrizes e o tipo derivado `TRIPLETO` para se obter objetos mais complexos. No exemplo abaixo, declara-se um vetor cujos elementos são `TRIPLETO`s de diversas funções distintas:

```
TYPE(TRIPLETO), DIMENSION(10) :: V
```

Assim, a referência ao objeto `V(2)%U` fornece o valor da função correspondente ao segundo elemento do vetor `V`; já a referência `V(5)%D2U(1,1)` fornece o valor da derivada segunda em relação à primeira variável da função correspondente ao elemento 5 do vetor `V`, e assim por diante.

O primeiro programa a seguir exemplifica um uso simples de matrizes:

```
!  Declara vetores, atribui valores aos elementos dos mesmos e imprime
!   estes valores na tela.
program ex1_array
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
integer :: i
real, dimension(10) :: vr
real(kind= dp), dimension(10) :: vd

do i= 1,10
   vr(i)= sqrt(real(i))
   vd(i)= sqrt(real(i))
end do
print *, "Raiz quadrada dos 10 primeiros inteiros, em precisão simples:"
print *, vr      ! Imprime todos os componentes do vetor.
print *, " "
print *, "Raiz quadrada dos 10 primeiros inteiros, em precisão dupla:"
print *, vd
end program ex1_array
```

O segundo programa-exemplo, a seguir, está baseado no programa 3.6. Agora, será criado um vetor para armazenar os dados de um número fixo de alunos e os resultados somente serão impressos após a aquisição de todos os dados.

```
!Dados acadêmicos de alunos usando tipo derivado.
program alunos_vet
implicit none
integer :: i,ndisc= 5 !Mude este valor, caso seja maior.
type:: aluno
   character(len= 20):: nome
   integer:: codigo
   real:: n1,n2,n3,mf
end type aluno
```

```
type(aluno), dimension(5):: disc

do i= 1,ndisc
   print*, "Nome:"
   read "(a)", disc(i)%nome
   print*, "código:"
   read*,disc(i)%codigo
   print*, "Notas: N1,N2,N3:"
   read*,disc(i)%n1,disc(i)%n2,disc(i)%n3
   disc(i)%mf= (disc(i)%n1 + disc(i)%n2 + disc(i)%n3)/3.0
end do
do i= 1,ndisc
   print*, " "
   print*, "-------------> ",disc(i)%nome," (",disc(i)%codigo,") <-------------"
   print*, "          Média final: ",disc(i)%mf
end do
end program alunos_vet
```

6.2 EXPRESSÕES E ATRIBUIÇÕES ENVOLVENDO MATRIZES

Até o Fortran 77 não era possível desenvolver expressões envolvendo o conjunto de todos os elementos de uma matriz simultaneamente. Ao invés disso, cada elemento da matriz deveria ser envolvido na expressão separadamente, em um processo que com frequência demandava o uso de diversos laços `DO` encadeados. Quando a operação envolvia matrizes grandes, com 100×100 elementos ou mais, tais processos podiam ser extremamente dispendiosos do ponto de vista do tempo necessário para a realização de todas as operações desejadas, pois os elementos da(s) matriz(es) deveriam ser manipulado de forma sequencial. Além disso, o código tornava-se gradualmente mais complexo para ser lido e interpretado, à medida que o número de operações envolvidas aumentava. Este tipo de procedimento trabalhoso e suscetível a erros continua sendo necessário em algumas linguagens de programação modernas.

Um desenvolvimento novo, introduzido no Fortran 90, é a habilidade de realizar operações envolvendo a matriz na sua totalidade, possibilitando o tratamento de uma matriz como um objeto único, o que, no mínimo, facilita enormemente a construção, leitura e interpretação do código. Uma outra vantagem ainda mais importante que resulta dessa nova estratégia de manipulação de matrizes se aplica aos sistemas de *processamento distribuído* ou *processamento paralelo*. As normas definidas pelos comitês J3/WG5 para o padrão da linguagem Fortran supõe que compiladores usados em sistemas distribuídos devem se encarregar de distribuir automaticamente os processos numéricos envolvidos nas expressões com matrizes de forma equilibrada entre os diversos processadores que compõe a arquitetura. A evidente vantagem nesta estratégia consiste no fato de que as mesmas operações numéricas são realizadas de forma simultânea em diversos componentes distintos das matrizes, acelerando substancialmente a eficiência do processamento. Com a filosofia das operações sobre matrizes inteiras, a tarefa de implantar a paralelização do código numérico fica, essencialmente, a cargo do compilador e não do programador. Uma outra vantagem deste enfoque consiste na manutenção da portabilidade dos códigos numéricos.

Para que as operações envolvendo matrizes inteiras sejam possíveis, é necessário que as matrizes consideradas sejam *conformáveis*, ou seja, elas devem todas ter a mesma forma. Operações entre duas matrizes conformáveis são realizadas na maneira **elemental**, isto é, elemento a elemento e distribuindo as operações entre os diversos processadores, se existirem, e todos os operadores numéricos definidos para operações entre escalares também são definidos para operações entre matrizes.

Por exemplo, sejam `A` e `B` duas matrizes 2×3:

$$\mathtt{A} = \begin{pmatrix} 3 & 4 & 8 \\ 5 & 6 & 6 \end{pmatrix}, \; \mathtt{B} = \begin{pmatrix} 5 & 2 & 1 \\ 3 & 3 & 1 \end{pmatrix},$$

o resultado da adição de A por B é:

$$\texttt{A} + \texttt{B} = \begin{pmatrix} 8 & 6 & 9 \\ 8 & 9 & 7 \end{pmatrix},$$

o resultado da multiplicação é:

$$\texttt{A*B} = \begin{pmatrix} 15 & 8 & 8 \\ 15 & 18 & 6 \end{pmatrix}$$

e o resultado da divisão é:

$$\texttt{A/B} = \begin{pmatrix} 3/5 & 2 & 8 \\ 5/3 & 2 & 6 \end{pmatrix}.$$

Se um dos operandos é um escalar, então este é distribuído (*broadcast*) em uma matriz conformável com o outro operando. Assim, o resultado de adicionar 5 a A é:

$$\texttt{A} + 5 = \begin{pmatrix} 3 & 4 & 8 \\ 5 & 6 & 6 \end{pmatrix} + \begin{pmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{pmatrix} = \begin{pmatrix} 8 & 9 & 13 \\ 10 & 11 & 11 \end{pmatrix}.$$

Esta distribuição de um escalar em uma matriz conformável também é empregada no momento da inicialização dos elementos de uma matriz.

Da mesma forma como acontece com expressões e atribuições escalares, em uma linha de programação como a seguir,

```
A= A + B
```

sendo A e B matrizes, a expressão do lado direito é desenvolvida antes da atribuição do resultado da expressão à matriz A. Este ordenamento é importante quando uma matriz aparece em ambos os lados de uma atribuição, como no caso do exemplo acima.

Além dos recursos algébricos para manipulações de matrizes, o padrão também estabelece uma abundância de rotinas intrínsecas (discutidas no capítulo 8) destinadas a realizar outras operações. As vantagens desta estratégia ficam evidentes nos exemplos a seguir:

1. Considere três vetores, A, B e C, todos do mesmo comprimento. Inicialize todos os elementos de A a zero e realize as atribuições `A(I)= A(I)/3.1 + B(I)*SQRT(C(I))` para todos os valores de $0 \leqslant \texttt{I} \leqslant 20$:

   ```
   REAL, DIMENSION(20) :: A= 0.0, B, C
   A= A/3.1 + B*SQRT(C)
   ```

   Note como a função intrínseca SQRT opera de forma elemental sobre cada elemento do vetor C.

2. Considere uma matriz tri-dimensional. Encontre o maior valor menor que 1000 nesta matriz:

   ```
   REAL, DIMENSION(5,5,5) :: A
   REAL :: VAL_MAX
   VAL_MAX= MAXVAL(A, MASK=(A<1000.0))
   ```

   A função intrínseca MAXVAL devolve o maior valor entre os elementos de uma matriz. O argumento opcional MASK=(...) estabelece uma *máscara,* isto é, uma expressão lógica envolvendo a(s) matriz(es). Em MASK=(A<1000.0), somente aqueles elementos de A que satisfazem a condição de ser menores que 1000 são levados em consideração.

3. Encontre o valor médio dos elementos maiores que 3000 na matriz A do exemplo anterior:

   ```
   REAL, DIMENSION(5,5,5) :: A
   REAL :: MEDIA
   MEDIA= SUM(A,MASK=(A>3000.0))/COUNT(MASK=(A>3000.0))
   ```

   A função intrínseca SUM realiza a soma dos elementos da matriz que satisfazem a condição estabelecida pela máscara MASK, enquanto que a função COUNT conta quantos elementos da matriz satisfazem a mesma condição.

4. Dadas as matrizes `A(100,200)` e `B(200,300)`, obtenha a matriz `C(100,300)` a partir do produto matricial entre `A` e `B`:

```
REAL A(100,200), B(200,300), C(100,300)
C= MATMUL(A,B)
```

   Como o nome implica, a função intrínseca `MATMUL` realiza o produto matricial entre duas matrizes.

Todos as operações contidas nestes exemplos, as quais foram realizadas em apenas uma linha de código usando o Fortran, necessitariam de diversas linhas de programação em outras linguagens. Adicionalmente, o padrão exige que as operações realizadas por essas funções sejam sempre realizadas da maneira mais otimizada possível, tendo em vista o hardware disponível e as opções de compilação empregadas.

Uma matriz também pode ser uma constante nomeada e, neste caso, não é necessário declarar a sua forma, a qual será tomada diretamente do valor da matriz, empregando-se um asterisco no lugar do limite superior em uma dada dimensão da matriz. Isto caracteriza **matrizes de forma implícita** (*implied-shape array*s), exemplificadas em

```
REAL, DIMENSION(*), PARAMETER :: FIELD= [ 0.0, 10.0, 20.0 ]
CHARACTER, PARAMETER :: VOGAIS(*)= [ 'A', 'E', 'I', 'O', 'U' ]
```

6.3 SEÇÕES DE MATRIZES

Uma **submatriz**, também chamada **seção de matriz**, pode ser acessada através da especificação de um intervalo de valores de subscritos da matriz. Uma seção de matriz pode ser acessada e operada da mesma forma que a matriz completa, mas não é possível fazer-se referência direta a elementos individuais ou a subseções da seção. Neste caso, deve-se realizar a atribuição desta submatriz a uma outra matriz e então elementos individuais poderão ser acessados.

Seções de matrizes podem ser extraídas usando-se um dos seguintes artifícios:

- Um subscrito simples.
- Um tripleto de subscritos.
- Um vetor de subscritos.

Estes recursos serão descritos a seguir.

6.3.1 SUBSCRITOS SIMPLES

Um subscrito simples seleciona um único elemento da matriz. Considere a seguinte matriz 5×5, denominada `RA`. Então o elemento `X` pode ser selecionado através de `RA(2,2)`:

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA}(2,2) = \texttt{X}.$$

6.3.2 TRIPLETO DE SUBSCRITOS

A forma de um tripleto de subscritos é a seguinte, sendo esta forma geral válida para todas as dimensões definidas para a matriz:

```
[<limite inferior>] : [<limite superior>] : [<passo>]
```

Se um dos limites, inferior ou superior (ou ambos) for omitido, então o limite ausente é assumido como o limite inferior ou superior, respectivamente, da correspondente dimensão da matriz da qual a seção está sendo extraída; se o `<passo>` for omitido, então assume-se `<passo>=1`.

Os exemplos a seguir ilustram várias seções de matrizes usando-se tripletos. Os elementos das matrizes marcados por `X` denotam a seção a ser extraída. Novamente, no exemplo será utilizada a matriz 5×5 denominada `RA`.

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA(2:2,2:2) = RA(2,2) = X;}$$

Elemento simples, escalar.
Forma: $[1]$

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \texttt{X} & \texttt{X} & \texttt{X} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA(3,3:5);}$$

Seção de linha da matriz.
Forma: $[3]$

$$\texttt{RA} = \begin{bmatrix} 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \\ 0 & 0 & \texttt{X} & 0 & 0 \end{bmatrix} \Longrightarrow \texttt{RA(:,3);}$$

3ª Coluna inteira.
Forma: $[5]$

$$\texttt{RA} = \begin{bmatrix} 0 & \texttt{X} & \texttt{X} & \texttt{X} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & \texttt{X} & \texttt{X} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \texttt{X} & \texttt{X} & \texttt{X} & 0 \end{bmatrix} \Longrightarrow \texttt{RA(1::2,2:4);}$$

Seções de linhas com passo 2.
Forma: $[3, 3]$

6.3.3 Vetores de subscritos

Um vetor de subscritos é uma expressão inteira de posto 1, isto é, um vetor. Cada elemento desta expressão deve ser definido com valores que se encontrem dentro dos limites dos subscritos da matriz-mãe. Os elementos de um vetor de subscritos podem estar em qualquer ordem.

Um exemplo ilustrando o uso de um vetor de subscritos, denominado `IV`, é dado a seguir:

```
REAL, DIMENSION(6) :: RA= [ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 ]
REAL, DIMENSION(3) :: RB
INTEGER, DIMENSION(3) :: IV= [ 1, 3, 5 ] ! Expressão inteira de posto 1.
RB= RA(IV)                                ! IV é o vetor de subscritos.
! Resulta:
!RB= [ RA(1), RA(3), RA(5) ], ou
!RB= [ 1.2, 3.0, 1.0 ].
```

Um vetor de subscritos pode também estar do lado esquerdo de uma atribuição:

```
RA= [ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 ]
IV= [ 1, 3, 5 ]
RA(IV)= [ -1.2, 7.8, 5.6 ] !Atribuições dos elementos 1, 3 e 5 de RA.
! Resulta:
! RA(1)= -1.2; RA(3)= 7.8; RA(5)= 5.6
! Agora: RA= [ -1.2, 3.4, 7.8, 11.2, 5.6, 3.7 ]
```

Pode-se também repetir elementos em um vetor de subscritos, caso este seja empregado em expressões (*i.e.*, no lado direito). Por exemplo,

```
RA= [ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 ]
IV= [1, 3, 1]
RB= RA(IV)
! Resulta: RB= [ 1.2, 3.0, 1.2 ]
```

Neste caso, o que resulta é uma **seção de matriz muitos-de-um** (*many-one section*). De fato, com este recurso uma seção de matriz pode até ser mais extensa que a matriz original. Contudo, uma seção muitos-de-um não pode ser empregada em atribuições (*i.e.*, no lado esquerdo):

```
RA([ 1, 3, 1])= [ -1.2, 7.8, 5.6 ] ! Inválido.
```

Uma vez que o padrão não estabelece a ordem de atribuições neste caso.

6.4 ATRIBUIÇÕES DE MATRIZES E SUBMATRIZES

Tanto matrizes inteiras quanto seções de matrizes podem ser usadas como operandos (isto é, podem estar tanto no lado esquerdo quanto do lado direito de uma atribuição) desde que todos os operandos sejam conformáveis (página 64). Por exemplo,

```
REAL, DIMENSION(5,5) :: RA, RB, RC
INTEGER :: ID
...
RA= RB + RC*ID !Forma: [5,5].
...
RA(3:5,3:4)= RB(1::2,3:5:2) + RC(1:3,1:2)
!Forma [3,2]:
!RA(3,3)= RB(1,3) + RC(1,1)
!RA(4,3)= RB(3,3) + RC(2,1)
!RA(5,3)= RB(5,3) + RC(3,1)
!RA(3,4)= RB(1,5) + RC(1,2)
!etc.
...
RA(:,1)= RB(:,1) + RB(:,2) + RC(:3)
!Forma [5].
```

Um outro exemplo, acompanhado de figura, torna a operação com submatrizes conformáveis mais clara.

```
REAL, DIMENSION(10,20) :: A,B
REAL, DIMENSION(8,6) :: C
...
C= A(2:9,5:10) + B(1:8,15:20) !Forma: [8,6].
...
```

A figura 6.2 ilustra como as duas submatrizes são conformáveis neste caso e o resultado da soma das duas seções será atribuído à matriz `C`, a qual possui a mesma forma.

O programa-exemplo a seguir (programa 6.1) mostra algumas operações e atribuições básicas de matrizes. Nota-se que já se faz uso de instruções saída formatada de matrizes (seção 10.1) e de laços `DO` implícitos (seção 6.6).

Listagem 6.1: Primeiro programa envolvendo matrizes.

```
! Testa atribuições e seções de matrizes.
program testa_atr_matr
implicit none
real, dimension(3,3) :: a
real, dimension(2,2) :: b
integer :: i, j

do i= 1, 3
   do j= 1, 3
      a(i,j)= sin(real(i)) + cos(real(j))
   end do
end do
b= a(1:2, 1:3:2)
print*, "Matriz A:"
print"(3(f12.5))", ((a(i,j), j= 1,3), i= 1,3)
print*, "Matriz B:"
print"(2(f12.5))", ((b(i,j), j= 1,2), i= 1,2)
end program testa_atr_matr
```

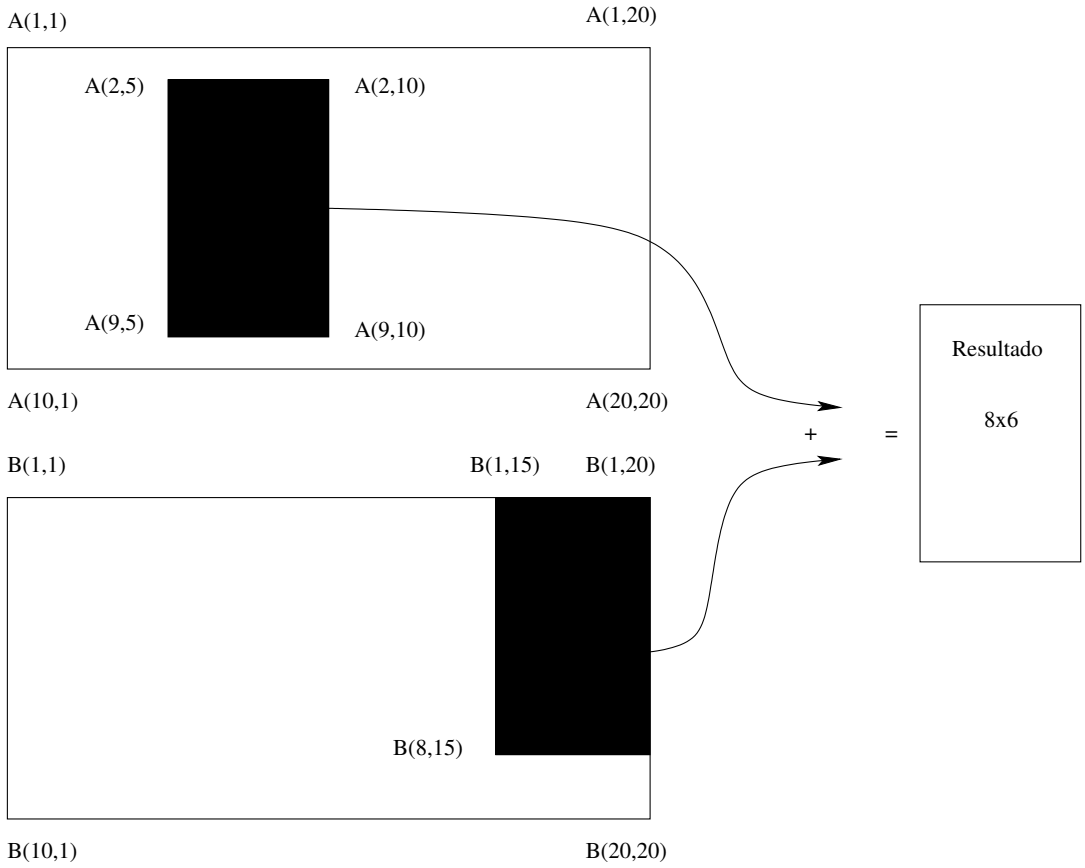

Figura 6.2: A soma de duas seções de matrizes conformáveis.

6.5 MATRIZES DE TAMANHO ZERO

Matrizes de tamanho zero (*zero-sized arrays*), ou *matrizes nulas* também são permitidas em Fortran. A noção de uma matriz nula é útil quando se quer contar com a possibilidade de existência de uma matriz sem nenhum elemento, o que pode simplificar a programação do código em certas situações. Uma matriz é nula quando o limite inferior de uma ou mais de suas dimensões é maior que o limite superior.

Por exemplo, o código abaixo resolve um sistema linear de equações que já estão na forma triangular:

```
DO I= 1, N
   X(I)= B(I)/A(I,I)
   B(I+1:N)= B(I+1:N) - A(I+1:N,I)*X(I)
END DO
```

Quando `I` assume o valor `N`, as submatrizes `B(N+1:N)` e `A(N+1:N,N)` se tornam nulas. Se esta possibilidade não existisse, seria necessária a inclusão de linhas adicionais de programação.

As matrizes nulas seguem as mesmas regras de operação e atribuição que as matrizes usuais, porém elas ainda devem ser conformáveis com as matrizes restantes. Por exemplo, duas matrizes nulas podem ter o mesmo posto mas formas distintas; as formas podem ser `[2,0]` e `[0,2]`. Neste caso, estas matrizes não são conformáveis. Contudo, uma matriz é sempre conformável com um escalar, assim a atribuição

```
<matriz nula>= <escalar>
```

é sempre válida.

6.6 CONSTRUTORES DE MATRIZES

Um *construtor de matrizes* (*array constructor*) cria um vetor (matriz de posto 1) contendo valores constantes. Estes construtores servem, por exemplo, para inicializar os elementos de um vetor, como foi exemplificado na página 65. Outro exemplo de uso dos construtores de matrizes está na definição de um vetor de subscritos, como foi abordado nas seções 6.3 e 6.4.

A forma geral de um construtor de matrizes é a seguinte:

```
(/ [<type-spec> :: ] <lista de valores do construtor> /)
```

ou

```
[ [<type-spec> :: ] <lista de valores do construtor> ]
```

sendo que a última forma (com os colchetes) é a preferida no padrão mais recente, pois diminui a quantidade de parênteses.

Nas definições acima, `<lista de valores do construtor>` pode ser tanto uma lista de constantes, como no exemplo

```
IV= [ 1, 3, 5 ]
```

como pode ser um conjunto de expressões numéricas e/ou com rotinas:

```
A= [ I+J, 2*I, 2*J, I**2, J**2, SIN(REAL(I)), COS(REAL(J)) ]
```

ou ainda uma instrução de **DO implícito** (*implied-do*) no construtor, cuja forma geral é a seguinte:

```
(<lista de valores do construtor>, <variável>= <exp1>, <exp2>, [<exp3>])
```

onde `<variável>` é uma variável escalar inteira e `<exp1>`, `<exp2>` e `<exp3>` são expressões escalares inteiras. A interpretação dada a este DO implícito é que a `<lista de valores dos construtor>` é escrita um número de vezes igual a

```
MAX((<exp2> - <exp1> + <exp3>)/<exp3>, 0)
```

sendo a `<variável>` substituída por `<exp1>`, `<exp1> + <exp3>`, ..., `<exp2>`, como acontece no construto DO (seção 5.2). Também é possível encadear-se DO's implícitos.

A seguir, alguns exemplos destes recursos são apresentados:

```
[ (i, i= 1,6) ] ! Resulta: [ 1, 2, 3, 4, 5, 6 ]


[ 7, (i, i= 1,4), 9 ] ! Resulta: [ 7, 1, 2, 3, 4, 9 ]


[ (1.0/REAL(I), I= 1,6) ]
! Resulta: [ 1.0/1.0, 1.0/2.0, 1.0/3.0, 1.0/4.0, 1.0/5.0, 1.0/6.0 ]


! DO's implícitos encadeados:
[ ((I, I= 1,3), J= 1,3) ]
! Resulta: [ 1, 2, 3, 1, 2, 3, 1, 2, 3 ]


[ ((I+J, I= 1,3), J= 1,2) ]
! = [ ((1+J, 2+J, 3+J), J= 1,2) ]
! Resulta: [ 2, 3, 4, 3, 4, 5 ]
```

Um construtor de matrizes pode também ser criado usando-se tripletos de subscritos:

```
[ A(I,2:4), A(1:5:2,I+3) ]
! Resulta: [ A(I,2), A(I,3), A(I,4), A(1,I+3), A(3,I+3), A(5,I+3) ]
```

Retornando à forma geral de um construtor de matrizes, <type-spec> se refere ao nome do tipo de objeto de dados (intrínseco ou derivado), seguido pelos parâmetros do tipo em parênteses. Se o construtor de matrizes inicia por <type-spec> ::, esta instrução determina o tipo e parâmetros do tipo das constantes que aparecem na <lista de valores do construtor> e os valores nesta lista são convertidos ao tipo e parâmetros do tipo adequados, dependendo do contexto em que o construtor é empregado. Um exemplo seria:

```
[ character(len= 33) :: 'the good', 'the bad', 'and', 'the appearance-challenged' ]
```

Todas as constantes de caracteres nesta lista têm comprimento 33 e serão submetidas às regras de conversão adequadas no momento da atribuição.

6.6.1 A FUNÇÃO INTRÍNSECA RESHAPE

Uma matriz de posto maior que um pode ser construída a partir de um construtor de matrizes através do uso da função intrínseca RESHAPE. Por exemplo,

```
RESHAPE( SOURCE= [ 1, 2, 3, 4, 5, 6 ], SHAPE= [ 2, 3 ] )
```

toma o vetor [1, 2, 3, 4, 5, 6] e gera com o mesmo a matriz de posto 2:

$$\begin{matrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{matrix}$$

a qual é uma matriz de forma [2, 3], isto é, 2 linhas e 3 colunas.

Um outro exemplo seria:

```
REAL, DIMENSION(3,2) :: RA
RA= RESHAPE( SOURCE= [ ((I+J, I= 1,3), J= 1,2) ], SHAPE= [ 3,2 ] )
```

O construtor em SOURCE gera o vetor [2, 3, 4, 3, 4, 5] e a instrução SHAPE organiza o vetor na forma [3,2], resultando em

$$\mathtt{RA} = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$$

A função RESHAPE também deve ser empregada para uma matriz de posto maior que um caso esta seja uma constante nomeada. Neste caso, novamente pode-se empregar asteriscos para os limites superiores em cada dimensão, como em

```
INTEGER, DIMENSION(0:*, *), PARAMETER :: POWERS= &
   RESHAPE( [ 0, 1, 2, 3, 0, 1, 4, 9, 0, 1, 8, 27 ], [ 4, 3 ])
```

A definição completa da função RESHAPE será apresentada na seção 8.14.3. Com a mesma, o programa-exemplo 6.1 apresenta uma forma mais concisa:

```
! Emprega a função intrínseca RESHAPE no programa 6.1
!   para definir a matriz a.
program testa_atr_matr
implicit none
real, dimension(3,3) :: a
real, dimension(2,2) :: b
integer :: i, j

a= reshape(source= [((sin(real(i))+cos(real(j)), i= 1,3), j= 1,3)], &
           shape= [3,3])
b= a(1:2,1:3:2)
print*, "Matriz A:"
print"(3(f12.5))", ((a(i,j), j= 1,3), i= 1,3)
Print*, "Matriz B:"
print"(2(f12.5))", ((b(i,j), j= 1,2), i= 1,2)
end program testa_atr_matr
```

6.6.2 A ORDEM DOS ELEMENTOS DE MATRIZES

A maneira como a função RESHAPE organizou os elementos das matrizes na seção 6.6.1 seguiu a denominada *ordem dos elementos de matriz*, a qual é a maneira como os compiladores de Fortran armazenam os elementos de matrizes em espaços contíguos de memória. Este ordenamento é obtido variando-se inicialmente o índice da primeira dimensão da matriz, depois variando-se o índice da segunda dimensão e assim por diante. Em uma matriz com 2 dimensões isto é obtido variando-se inicialmente as linhas e depois as colunas. A figura 6.3 ilustra este procedimento com a matriz `B(5,4)`.

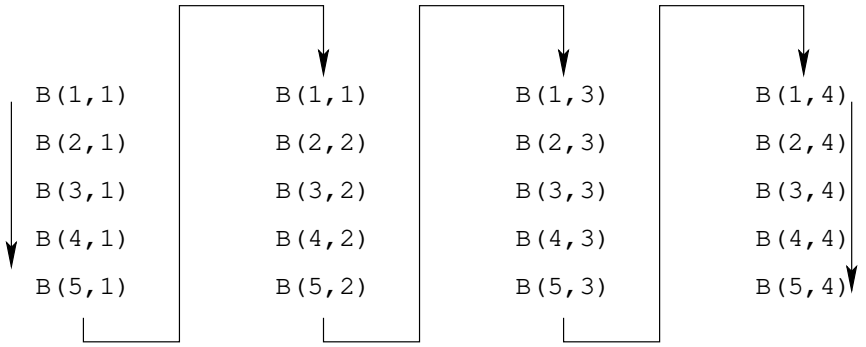

Figura 6.3: O ordenamento dos elementos da matriz *B(5,4)*.

Em uma matriz com mais de 2 dimensões, o ordenamento é realizado da mesma maneira. Assim, dada a matriz de posto 7

```
REAL, DIMENSION(-10:5, -20:1, 0:1, -1:0, 2, 2, 2) :: G
```

cujo tamanho é $16 \times 22 \times 2 \times 2 \times 2 \times 2 \times 2 = 11264$, o ordenamento segue a ordem:

$$
\begin{array}{c}
\mathtt{G}(-10,-20,0,-1,1,1,1) \to \mathtt{G}(-9,-20,0,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-20,0,-1,1,1,1) \to \\
\mathtt{G}(-10,-19,0,-1,1,1,1) \to \mathtt{G}(-9,-19,0,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-19,0,-1,1,1,1) \to \\
\mathtt{G}(-10,-18,0,-1,1,1,1) \to \mathtt{G}(-9,-18,0,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-18,0,-1,1,1,1) \to \\
\vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\
\mathtt{G}(-10,-20,1,-1,1,1,1) \to \mathtt{G}(-9,-20,1,-1,1,1,1) \to \ldots \to \mathtt{G}(5,-20,1,-1,1,1,1) \to \\
\vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\
\mathtt{G}(-10,1,1,0,2,2,2) \to \mathtt{G}(-9,1,1,0,2,2,2) \quad \to \ldots \to \mathtt{G}(5,1,1,0,2,2,2).
\end{array}
$$

Em muitas situações, o programa executável é mais eficiente se o código-fonte foi escrito levando em conta o ordenamento dos elementos de matrizes. As rotinas intrínsecas do Fortran que manipulam matrizes inteiras foram concebidas de forma a levar em conta este fato.

6.7 ROTINAS INTRÍNSECAS PARA MATRIZES

O padrão fornece uma biblioteca extensa, com diversas rotinas intrínsecas relacionadas com matrizes. Essas rotinas dividem-se em três categorias: *funções elementais*, *funções inquiridoras* ou *inquisidoras* e *funções transformacionais*. Nesta seção será apresentada somente uma descrição curta dessas rotinas; as definições completas serão apresentadas no capítulo 8.

6.7.1 ROTINAS INTRÍNSECAS ELEMENTAIS APLICÁVEIS A MATRIZES

O Fortran permite a existência de rotinas (funções ou subrotinas) intrínsecas *elementais*, isto é, rotinas que se aplicam a cada elemento de uma matriz. Matrizes podem, então, ser usadas como argumentos de rotinas intrínsecas, da mesma forma que escalares. As operações definidas

na rotina intrínseca serão aplicadas a cada elemento da matriz separadamente. Novamente, caso mais de uma matriz apareça no argumento da rotina, estas devem ser conformáveis.

A seguir, alguns exemplos de rotinas intrínsecas elementais. A lista completa destas rotinas pode ser obtida no capítulo 8.

1. Calcule as raízes quadradas de todos os elementos da matriz `A`. O resultado será atribuído à matriz `B`, a qual é conformável com `A`.

```
REAL, DIMENSION(10,10) :: A, B
B= SQRT(A)
```

2. Calcule a exponencial de todos os argumentos da matriz `A`. Novamente, o resultado será atribuído a `B`.

```
COMPLEX, DIMENSION(5,-5:15, 25:125) :: A, B
B= EXP(A)
```

3. Encontre o comprimento da string (variável de caractere) excluindo brancos no final da variável para todos os elementos da matriz `CH`.

```
CHARACTER(LEN= 80), DIMENSION(10) :: CH
INTEGER :: COMP
COMP= LEN_TRIM(CH)
```

6.7.2 Funções inquiridoras

São funções cujo valor depende das propriedades da matriz sendo investigada. A tabela 6.1 apresenta algumas funções inquisidoras de matrizes.

Tabela 6.1: Algumas funções inquisidoras de matrizes.

Nome da função e invocação	Propósito
`ALLOCATED(ARRAY)`	Determina o status de alocação de uma matriz alocável (seção 7.1)
`LBOUND(ARRAY, [DIM])`	Retorna os limites inferiores das extensões da matriz `ARRAY` em um vetor
`UBOUND(ARRAY, [DIM])`	Retorna os limites superioes das extensões da matriz `ARRAY` em um vetor
`SHAPE(ARRAY)`	Retorna a forma da matriz `ARRAY` em um vetor
`SIZE(ARRAY, [DIM])`	Retorna o tamanho da matriz ou a extensão da mesma ao longo da dimensão `DIM`

As definições das funções acima e de outras funções inquiridoras são dadas nas seções 8.2 e 8.13.

6.7.3 Funções transformacionais

São funções que têm como argumento uma ou mais matrizes. Ao contrário das funções elementais, as funções transformacionais atuam sobre a matriz como um todo. A tabela 6.2 apresenta algumas funções transformacionais de matrizes.

Várias das funções acima possuem opções adicionais. Suas definições e de muitas outras rotinas são dadas nas seções 8.12 e 8.14.

6.8 Comando e construto WHERE

Em muitas situações, é desejável realizar-se operações somente para alguns elementos de uma matriz, mediante a aplicação de alguma condição lógica aos seus elementos ou aos elementos de uma outra matriz conformável. O comando/construto `WHERE` oferece uma implementação para este tipo de problema.

Tabela 6.2: Algumas funções transformacionais de matrizes.

Nome da função e invocação	Propósito
ALL(MASK)	Função lógica que retorna .TRUE. se *todos* os elementos em MASK forem verdadeiros
ANY(MASK)	Função lógica que retorna .TRUE. se *algum* elemento em MASK for verdadeiro
COUNT(MASK)	Conta o número de elementos verdadeiros em MASK
MAXLOC(ARRAY)	Retorna em um vetor a *posição* do *maior* elemento de ARRAY
MAXVAL(ARRAY)	Retorna o *valor* do *maior* elemento de ARRAY
MINLOC(ARRAY)	Retorna em um vetor a *posição* do *menor* elemento de ARRAY
MINVAL(ARRAY)	Retorna o *valor* do *menor* elemento de ARRAY
FINDLOC(ARRAY, VALUE)	Retorna em um vetor as *posições* dos elementos de ARRAY que são iguais a VALUE
DOT_PRODUCT(VECTORA, VECTORB)	Retorna o produto escalar de dois vetores conformáveis
MATMUL(ARRAYA, ARRAYB)	Retorna o produto matricial de duas matrizes
PRODUCT(ARRAY)	Calcula o produto dos elementos de ARRAY
SUM(ARRAY)	Calcula a soma dos elementos de ARRAY
RESHAPE(ARRAY, SHAPE)	Muda a forma de ARRAY para a forma dada por SHAPE
TRANSPOSE(ARRAY)	Retorna a transporta de uma matriz de posto 2

6.8.1 COMANDO WHERE

O comando WHERE fornece esta possibilidade em uma única linha de instruções. A forma geral do comando é

```
WHERE (<expressão lógica matriz>) <variável matriz>= <expressão matriz>
```

A <expressão lógica matriz> deve ter a mesma forma que a <variável matriz>. A <expressão lógica matriz> é desenvolvida inicialmente, elemento a elemento. Em seguida, a <expressão matriz> será desenvolvida e os resultados atribuídos à <variável matriz>, novamente elemento a elemento, mas somente para aqueles subscritos nos quais a <expressão lógica matriz> teve resultado verdadeiro. Os elementos restantes permanecem inalterados.

Um exemplo simples é:

```
REAL, DIMENSION(10,10) :: A
WHERE (A > 0.0) A= 1.0/A
```

o qual fornece a recíproca (inversa) de todos os elementos positivos de A, deixando os demais inalterados.

6.8.2 CONSTRUTO WHERE

Uma única expressão lógica de matriz pode ser usada para determinar uma sequência de operações e atribuições em matrizes, todas com a mesma forma. A sintaxe deste construto é:

```
WHERE (<expressão matriz lógica>)
   <operações atribuições matrizes>
END WHERE
```

Inicialmente, a <expressão matriz lógica> é desenvolvida em cada elemento da matriz, resultando em uma matriz lógica temporária, cujos elementos são os resultados da <expressão matriz lógica>. Então, cada operação e atribuição de matriz no bloco do construto é executada sob o controle da *máscara* determinada pela matriz lógica temporária; isto é, as <operações atribuições matrizes> serão realizadas em cada elemento das matrizes do bloco que corresponda ao valor .TRUE. da matriz lógica temporária.

Existe também uma forma mais geral do construto WHERE que permite a execução de atribuições a elementos de matrizes que não satisfazem o teste lógico no cabeçalho:

```
WHERE (<expressão matriz lógica>)
   <operações atribuições matrizes 1>
ELSEWHERE
   <operações atribuições matrizes 2>
END WHERE
```

O bloco <operações atribuições matrizes 1> é executado novamente sob o controle da máscara definida na <expressão matriz lógica> e somente elementos que satisfazem esta máscara são afetados. Em seguida, as <operações atribuições matrizes 2> são executadas sob o controle da máscara definida por .NOT. <expressão matriz lógica>, isto é, novas atribuições são realizadas sobre elementos que não satisfazem o teste lógico definido no cabeçalho.

Um exemplo simples do construto WHERE é:

```
WHERE (PRESSURE <= 1.0)
   PRESSURE= PRESSURE + INC_PRESSURE
   TEMP= TEMP + 5.0
ELSEWHERE
   RAINING= .TRUE.
END WHERE
```

Neste exemplo, PRESSURE, INC_PRESSURE, TEMP e RAINING são todas matrizes com a mesma forma, embora não do mesmo tipo.

O programa-exemplo 6.2 mostra outra aplicação do construto WHERE.

Listagem 6.2: Exemplo do construto *WHERE*.

```
! Testa o construto WHERE
program testa_where
implicit none
real, dimension(3,3) :: a
integer :: i, j

a= reshape(source= [ ((sin(real(i)) + cos(real(j)), i= 1, 3), j= 1, 3) ], &
           shape= [ 3, 3 ])
print*, "Matriz A original:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)

where (a >= 0.0)
   a= sqrt(a)
elsewhere
   a= a**2
end where
print*, "Matriz A modificada:"
print*, a(1,:)
print*, a(2,:)
print*, a(3,:)
end program testa_where
```

É possível também realizar-se o mascaramento na instrução ELSEWHERE, juntamente com a possibilidade de existir um número qualquer de instruções ELSEWHERE mascaradas mas com uma única instrução ELSEWHERE sem máscara, a qual deve ser a última. Todas as expressões lógicas que definem as máscaras devem ter a mesma forma. Adicionalmente, os construtos WHERE podem ser encadeados e nomeados; neste caso, as condições lógicas de todas as máscaras devem ter a mesma forma. O seguinte exemplo ilustra este recurso:

```
ATRIB1: WHERE (<cond 1>)
```

```
            <corpo 1>          !Mascarado por <cond 1>
         ELSEWHERE (<cond 2>) ATRIB1
            <corpo 2>          !Masc. por <cond 2> .AND. .NOT. <cond 1>
ATRIB2:     WHERE (<cond 4>)
               <corpo 4>       !Masc. por <cond 2> .AND. .NOT. <cond 1> .AND. <cond 4>
            ELSEWHERE ATRIB2
               <corpo 5>       !Masc. por <cond 2> .AND. .NOT. <cond 1>
               ...             !    .AND. .NOT. <cond 4>
            END WHERE ATRIB2
            ...
         ELSEWHERE (<cond 3>) ATRIB1
            <corpo 3>          !Masc. por <cond 3> .AND. .NOT. <cond 1>
            ...                !    .AND. .NOT. <cond 2>
         ELSEWHERE ATRIB1
            <corpo 6>          !Masc. por .NOT. <cond 1> .AND. .NOT. <cond 2>
            ...                !    .AND. .NOT. <cond 3>
         END WHERE ATRIB1
```

6.9 CONSTRUTO DO CONCURRENT

Uma forma aperfeiçoada do construto DO discutido na seção 5.2, o construto DO CONCURRENT, foi criada para possibilitar a execução em paralelo de iterações em laços. Para que este aperfeiçoamento possa ser empregado, o programador deve se certificar de que não existam interdependências entre as iterações do laço.

A lista de exigências para o uso do construto pode ser dividida em "limitações" a respeito do que pode ser colocado no bloco do construto e "garantias do programador" de que o código possui as propriedades que possibilitam a paralelização. Deve-se observar que se essas as exigências forem satisfeitas, não é necessário que o computador realmente disponha de múltiplos processadores ou que, caso os possua, que eles venham a ser realmente empregados. A implementação do construto também prevê que outros recursos existentes de optimização possam ser aplicados, tais como vetorização ou encadeamento de instruções (*pipelining*).

A forma geral do construto é:

```
DO [,] CONCURRENT ([<int-type-spec> ::] <index-spec-list> [, <scalar-mask-exp>])
   <do-conc-instrucs>
END DO
```

onde <int-type-spec> (caso presente) determina a espécie das variáveis inteiras dos índices, enquanto que <index-spec-list> é uma lista de especificações de índices, sendo que cada elemento da lista tem a forma

```
<index-var-name>= <valor-inicial> : <valor-final> [: <valor-passo>]
```

É importante ressaltar que cada <index-var-name> é local ao laço; isto é, terá sua ação limitada ao bloco do construto e qualquer variável com o mesmo nome que exista fora do construto não terá seu valor modificado. Se <int-type-spec> for omitido, a variável deve necessariamente ser do tipo inteiro padrão. Os campos <valor-inicial> , <valor-final> e <valor-passo> são expressões inteiras escalares e se <valor-passo> for omitido, é adotado <valor-passo>= 1.

Finalmente, a máscara opcional <scalar-mask-exp> é uma expressão lógica tal que somente aquelas iterações para as quais o resultado é verdadeiro serão realmente executadas.

Um exemplo simples do construto é:

```
DO CONCURRENT (I= 1:N)
   A(I,J)= A(I,J) + ALPHA*B(I,J)
END DO
```

durante cuja execução o índice J permanece constante.

Um outro exemplo usando uma máscara é :

```
DO CONCURRENT (I= 1:N, J= 1:M, I /= J)
   A(I,J)= A(I,J) + ALPHA*B(I,J)
END DO
```

Uma maneira de se escrever as manipulações realizadas neste DO CONCURRENT em termos de laço DO usuais é a seguinte:

```
DO I= 1, N
   DO J= 1, M
      IF(I /= J)A(I,J)= A(I,J) + ALPHA*B(I,J)
   END DO
END DO
```

Entretanto, existem diferenças fundamentais entre as duas estratégias:

1. Com os construtos DO usuais, os índices I e J não são locais; se estas variáveis forem usadas fora dos laços, elas terão seus valores alterados na saída.
2. Os laços DO usuais não são otimizados; para tanto, o programador deverá usar diretivas especiais de compilação e alguma estratégia de otimização.
3. O laço em J é previamente estabelecido como o laço interno, enquanto que o laço em I é o externo. Com o construto DO CONCURRENT o oposto também pode ocorrer; na verdade, os laços são executados em uma ordem qualquer, determinada pelo compilador.

As exigências impostas para o uso do construto são descritas a seguir. Os primeiros itens são os casos proibidos em um construto DO CONCURRENT:

- Alguma instrução que determina o encerramento não usual do construto ou de alguma iteração: um comando RETURN (seção 9.2.4), uma instrução EXIT ou uma instrução CYCLE com o nome de um construto DO externo.
- Uma instrução de controle de imagem.[2]
- Uma referência a uma rotina que não é *pura* (seção 9.2.17).
- Uma referência a uma das rotinas intrínsecas destinadas à manipulação de exceções de ponto flutuante.[3]
- Uma instrução de entrada/saída com um especificador ADVANCE= (seções 10.6 e 10.7).

O padrão exige que o compilador detecte estes casos, exceto o caso com o especificador ADVANCE.

Além das exigências acima, é necessário também que o programador escreva o código de tal forma que os resultados obtidos em uma iteração não afetem as outras iterações. Desta forma, as iterações podem ser realizadas em qualquer ordem, a qual será determinada pelo compilador. Em particular, é necessário que:

- qualquer variável referenciada em uma dada iteração, ou foi definida na mesma iteração ou seu valor não é afetado por qualquer outra iteração, a qual pode estar ocorrente simultaneamente;
- qualquer ponteiro referenciado em uma dada iteração, ou foi associado a um *alvo* na mesma iteração ou não tem o seu status de associação alterado por uma outra iteração;
- qualquer objeto alocável (seção 7.3) que é alocado ou dealocado em uma dada iteração não será usado por qualquer outra iteração, exceto se em cada iteração ocorrerem a alocação e a dealocação deste objeto;
- registros (ou posições) em um arquivo (capítulo 10) não serão ambos escritos por uma iteração e lidos por outra.

Registros podem ser escritos em um arquivos sequencial (seção 10.5) por mais de uma iteração, mas a ordem em que os registros são gravados no arquivo é indeterminada.

Se as condições acima forem satisfeitas e o contruto for executado, ao final da execução do mesmo,

[2] Aplica-se a coarrays, não discutidos nesta Apostila.
[3] Este conteúdo também não será abordado nesta Apostila.

- qualquer variável cujo valor é afetado por mais do que uma iteração se torna indefinida ao final do laço;
- qualquer ponteiro cujo status de associação (seção 7.2) é alterado por mais do que uma iteração passa a ter o status indefinido.

Também é importante enfatizar que mesmo que todos os requisitos sejam cumpridos e os laços sejam efetivamente paralelizados, isto não garante que o código executável será executado mais rapidamente que um código criado a partir de laços DO usuais ou por meio de alguma outra estratégia de otimização. Isto é particularmente verdade quando o número de iterações for pequeno, pois qualquer ganho em otimização será compensado com prejuízo devido ao custo adicional (*overhead*) em tempo de execução imposto pela inicialização de linhas (*threads*) paralelas de execução. Independente disso, o compilador pode ser esperto o suficiente para detectar que mesmo o laço DO é vetorizável/paralelizável e gerar o código incluindo as mesmas estratégias de otimização que o DO CONCURRENT. Em resumo, para verificar se existe uma real vantagem no uso do DO CONCURRENT, é recomendável que sejam realizados alguns testes comparativos. O programa abaixo implementa um destes testes.

Listagem 6.3: Compara desempenho de álgebra matricial com três abordagens distintas.

```
program compare_loop_times
use, intrinsic :: iso_fortran_env, only: dp => real64
implicit none
integer, parameter :: n= 10000000 ! n= 10^7
integer, parameter :: loop_count= 20
integer :: i, j
real :: time_begin, time_end, time, fconv= 1.0e6
real(kind= dp), dimension(n) :: x, y, z

! Atribui valores aos vetores x e y
call random_number(x) ! A rotina random_number() atribui valores
call random_number(y) !  aleatórios às matrizes (seção 8.17.3)

! Testa sintaxe de matriz inteira
call cpu_time(time_begin)
do j= 1, loop_count
   z= x + y
end do
call cpu_time(time_end)
time= fconv*(time_end - time_begin)
print'(a,g0)', 'Tempo para operação de matriz inteira (us): ', time
print'(a,g0)', 'Tempo médio (s):                             ',  time/loop_count

! Testa construto DO usual
call cpu_time(time_begin)
do j= 1, loop_count
   do i= 1, n
      z(i)= x(i) + y(i)
   end do
end do
call cpu_time(time_end)
time= fconv*(time_end - time_begin)
print'(/,a,g0)', 'Tempo para construto DO (us):               ', time
print'(a,g0)', 'Tempo médio (us):                            ',  time/loop_count

! Testa construto DO CONCURRENT
call cpu_time(time_begin)
do j= 1, loop_count
   do concurrent (i= 1:n)
      z(i)= x(i) + y(i)
   end do
end do
call cpu_time(time_end)
time= fconv*(time_end - time_begin)
print'(/,a,g0)', 'Tempo para construto DO CONCURRENT(s):      ', time
```

```
print '(a,g0)', 'Tempo médio (s):                                    ', time/loop_count

end program compare_loop_times
```

O programa `compare_loop_times.f90` foi testado com os compiladores Intel® Fortran e GFortran com diferentes opções de compilação em uma arquitetura contendo um núcleo com 8 processadores Intel® Core™ i7-4710HQ @ 2.50GHz. Os tempos médios de execução em cada caso são reportados na tabela 6.3. As opções de compilação foram as seguintes:

GFortran (S0)	`gfortran compare_loop_times.f90`	(opções-padrão)
GFortran (O1)	`gfortran -O3 compare_loop_times.f90`	(maior otimização)
Intel (S0)	`ifort compare_loop_times.f90`	(opções-padrão)
Intel (O1)	`ifort -fast -parallel compare_loop_times.f90`	(maior otimização)

Os resultados reportados na tabela 6.3 foram obtidos após repetidas execuções do programa (para cada compilação), uma vez que os tempos medidos podem variar substancialmente entre diferentes execuções. A tabela mostra que houve um ganho substancial de desempenho com o compilador GFortran quando foram empregadas as opções de máxima otimização. Já com o compilador da Intel® (`ifort`) não houve variação significativa entre as compilações com opções-padrão ou com otimização máxima. Observou-se também que para este teste em particular, ambos os compiladores geraram executáveis com performances equivalentes.

Tabela 6.3: Tempos médios de execução (em μs) das operações algébricas no programa 6.3 com diferentes compiladores e opções de compilação.

	GFortran (S0)	GFortran (O1)	Intel (S0)	Intel (O1)
Matriz inteira	$2,36 \times 10^4$	$0,150$	$0,100$	$0,200$
Laço `DO`	$3,19 \times 10^4$	$0,100$	$0,051$	$0,100$
Laço `DO CONCURRENT`	$2,39 \times 10^4$	$0,050$	$0,051$	$0,049$