Tabelas

Problema:

Dado um vector $\vec{x}$, dado por uma sequência de elementos do tipo inteiro, normalize-o, isto é calcule $\vec{x}'_i = x_i / \|\vec{x}\|$.

¿ Como calcular as várias componentes do vector $\vec{x}'$ se é necessário em primeiro calcular a norma de $\vec{x}$, $\|\vec{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$ **?**

¡ Não é possível utilizar a estratégia de: ler, calcular, e escrever componente a componente**!**

¡ É necessário ler todos as componentes para poder calcular a norma e de seguida dividir cada uma das componentes pela norma, isto é, é necessário ter algo aonde guardar todas as n componentes do vector**!**

Problema:

Dados duas matrizes calcular o seu produto.

Especificação:

$\longrightarrow A, B$

$\longleftarrow C = A \times B$, se possível

¿ Como ?

Com $A(l, m)$ e $B(m, n)$ então $C(l, n)$ é tal que:

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}, \qquad i = 1, \ldots, l, \quad j = 1, \ldots, n$$

¡ É necessário possuir uma estrutura de dados que nos permita:

guardar um conjunto de valores !

aceder a cada uma das suas componentes individualmente !

Tabelas ("Arrays")

- A possibilidade de denotar um grupo de elementos, todos do mesmo tipo, com um só identificador.
- A possibilidade de aceder a cada uma das suas componentes individualizáveis através de um índice.

$$a = \begin{bmatrix} 1 & 3 & 4 \\ 7 & 8 & -2 \\ 5 & 4 & 3 \end{bmatrix}$$

1 identificador $\longmapsto$ 9 posições de memória

$$a_{1,3} = 4$$

O identificador de tabela com os índices que permitem identificar uma posição de memória individual.

Tipo de Dados Tabela

Conjunto de Valores

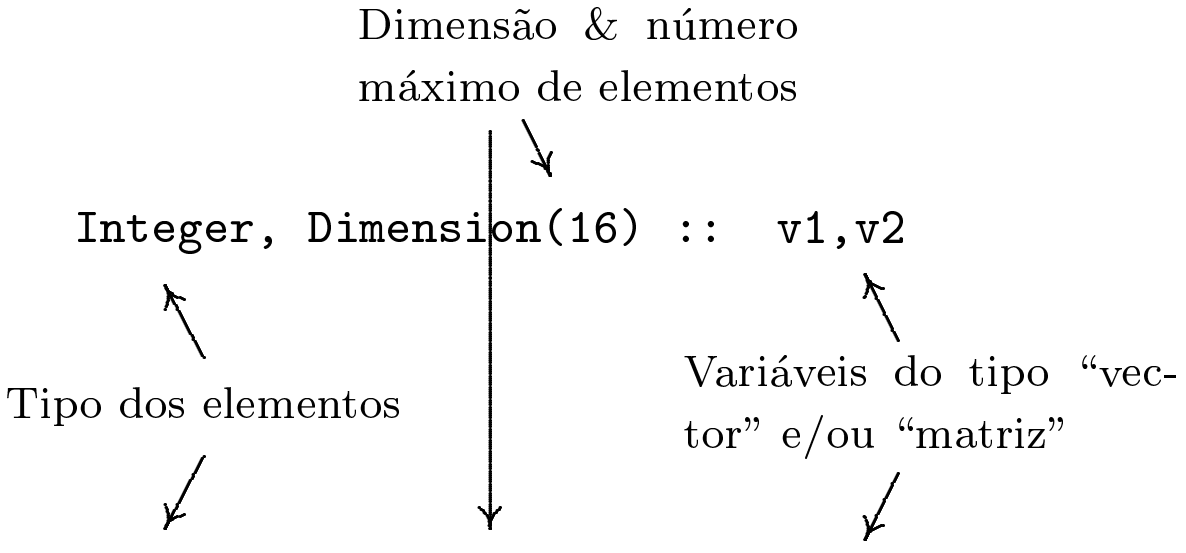
Tabelas n-dimensionais com os elementos pertencentes a um dado tipo de base (para cada n e para cada tipo de elementos temos um conjunto de elementos diferente).

Conjunto de Operações

Depende do conjunto de valores definido:

- atribuição global - todos os tipos
- soma, multiplicação - alguns tipos

Declaração - Exemplos



Declaração (estática)

<*tipo_de_base*>, `DIMENSION (`<*lista_de_índices*>`) ::` <*lista_de_identificadores*>

- <*tipo_de_base*> $\longrightarrow$ qualquer tipo FORTRAN
- <*lista_de_índices*> =
  [<*inf*>`:`] <*sup*>`,`<*lista_de_índices*>
  ou
  <*lista_vazia*>
  com
  - <*inf*> - limite inferior (inteiro)
  - <*sup*> - limite superior (inteiro)

Outro exemplo

```
Integer, Dimension(5:8) ::  v
```

Temos:

- Elementos do tipo `Integer`.
- lista de índices com um só elemento, isto é, estamos a declarar um vector de inteiros.
- `5:8`, $inf = 5$, $sup = 8$, 8-5+1=4, vector de inteiros com, no máximo 4 elementos.
- os índices do vector vão de 5 a 8 ($v_5, \ldots, v_8$).

Por omissão os índices das tabelas em Fortran começam em 1 e vão até ao valor máximo especificado.

Representação

```
Integer, Dimension(5:8) ::  v
```

- `v` $\longrightarrow$ variável tabela (o objecto que contém os quatro elementos).
- `v(6)` $\longrightarrow$ o segundo elemento da tabela (do tipo `Integer`).

  No caso geral tem-se

  <*identificador_de_tabela*>`(`<*índice*>`)`

  com o índice a ser um inteiro n tal que:

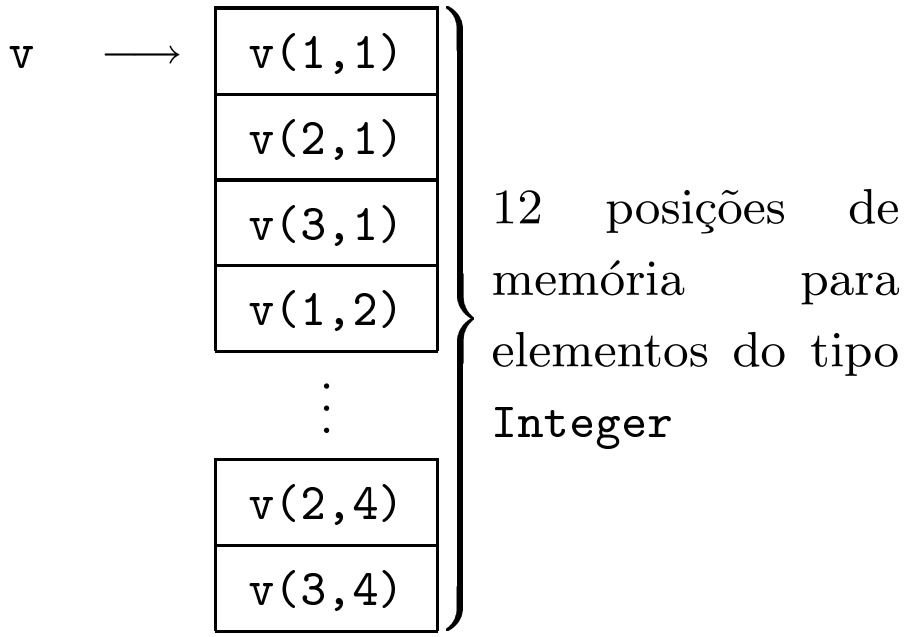
  $<inf> \leq n \leq <sup>$
- `(/ 15,5,-8,9 /)` $\longrightarrow$ um valor constante do tipo declarado acima. Os símbolos `(/`, e `/)`, são os símbolos reservados pelo FORTRAN para construir elementos do tipo tabela.

As Tabelas e a Memória

Uma declaração do tipo

```
Integer, Dimension(3:4) ::  m
```

representa uma reserva de 12 (3×4) posições de memória para elementos do tipo `Integer`.



- Após a declaração os valores dos vários elementos da tabela são indefinidos.
- A especificação dos índices estabelece limites que não podem ser ultrapassados, por exemplo `m(1,5)`, não é válido.
- A reserva de memória estabelece um espaço de utilização máximo, não diz nada sobre mínimos.
- A reserva de memória é estática, isto é, não pode ser redefinida de forma a se ajustar a diferentes utilizações do programa.

Vejamos um exemplo simples.

Problema:

Dado um vector x, calcular a sua norma.

Especificação:

$\longrightarrow x$

$\longleftarrow \|x\|$

¿ Qual é a dimensão de x ?

¡ No caso de um programa em FORTRAN é necessário saber qual é a dimensão (máxima) de uma tabela !

¿ Qual deve ser então a dimensão de x ?

¡ Um valor suficientemente grande para considerar todas (ou quase todas) as utilizações previsíveis do programa !

¡ Um valor não demasiado grande para não levantar problemas (por falta de memória) de execução !

¿¡ Então ?!

¡ 1000 posições !

– Em termos de memória de um computador não é muito significativo (mil inteiros).

– Em termos de utilização já é um valor razoável.

O valor final está sempre dependente das utilizações previsíveis do programa.

Reformulação da especificação

Especificação:

$\longrightarrow x$, vector com, no máximo, 1000 elementos.

$\longleftarrow \|x\|$

Tipo de Dados

vector de inteiros, e valor real (norma).

```
alg
  ler(nelementos)  ! número de elementos
  ler(x)                        ! (1)
  norma ← 0
  para i ← 1 até nelementos
       norma ← norma + x ^2_i
  fimpara
  norma ← √norma
  escreve(norma)
fimalg
```

Não é possível ler o valor de uma tabela como um todo, é necessário ler elemento a elemento.

```
alg  ! (1)
  para i ← 1 até nelementos
       ler(x _i)
  fimpara
fimalg
```

O programa

```
Program norma_vector

 Implicit None

 Integer, Dimension(1000) :: x
 Integer :: i,nelementos
 Real :: norma

 Write (*,*) "Numero de elementos"
 Read (*,*) nelementos

 Write (*,*) "Os elementos do vector"
 Do i=1,nelementos
   Read (*,*) x(i)
 End Do
 norma = 0
 Do i=1,nelementos
   norma = norma + x(i)**2
 End Do
 norma = Sqrt(norma)

 Write (*,*) norma
End Program norma_vector
```

O Fortran permite "compactar" a leitura e a escrita de tabelas através da utilização dos *ciclos* `DO` *implícitos*.

Leitura

Em FORTRAN a instrução

```
Read (*,*) ...
```

tem como efeito a leitura dos valores que se encontram numa dada linha, mudando de seguida de linha. Isto tem como consequência que para uma instrução como:

```
Do i=1,nelementos
  Read (*,*) x(i)
End Do
```

Os valores do vectores têm de ser apresentados um por linha

```
1
4
...
```

Utilizando um `DO` implícito:

```
Read (*,*) (x(i),i=1,nelementos)
```

podemos apresentar os valores todos na mesma linha.

```
1 4 ...
```

Escrita

Em FORTRAN a instrução:

```
Write (*,*) ...
```

tem como efeito a escrita dos valores todos na mesma linha, mudando de seguida de linha. Isto tem como consequência que na escrita de matrizes utilizando `DO`s explícitos:

```
Do i=1,m                          a(1,1)
  Do j=1,n                        a(1,2)
    Write (*,*) a(i,j) ⟼          ...
  End Do                          a(1,n)
End Do                            a(2,1)
                                  ...
```

temos a escrita de um elemento por linha.

Utilizando um `Do` implícito podemos obter a forma habitual de escrita de matrizes:

```
Do i=1,m                                a(1,1) ... a(1,n)
  Write (*,*) (a(i,j),j=1,n) ⟼          a(2,1) ... a(2,n)
End Do                                  ...
```

`DO`s implícitos.

Os `DO`s implícitos têm a mesma operacionalidade que os `DO`s explícitos, no entanto só podem ser usados em instruções de leitura e/ou de escrita e a sua sintaxe é mais próxima da sintaxe de uma expressão do que da sintaxe de uma instrução.

(<*expressão*>, <*variável*>, <*v.inicial*>, <*v.final*>, <*incremento*>)

O especificar do incremento (a exemplo do que acontece nos `DO`s explícitos) é opcional.

Tal como para os `DO`s explícitos é possível combinar vários níveis de `DO`s implícitos.

Por exemplo:

```
write (*,*) ((a(i,j),i=1,m),j=1,n)
```

tem como efeito a escrita dos elementos da matriz todos na mesma linha.

```
a(1,1) ... a(1,n) a(2,1) ...
```

Outros exemplos:

- Produto escalar,
  $mx = (mx_1, mx_2, \ldots, mx_n)$
- Traço de a, tr$a = \Sigma_{i=1}^{n} a_{i,i}$
- Soma de matrizes,
  $S(m,n) = A(m,n) + B(m,n)$

  $$s_{i,j} = a_{i,j} + b_{i,j}$$

  com $i = 1, \ldots, m,\ j = 1, \ldots n$
- Produto de matrizes,
  $P(l,n) = A(l,m) + B(m,n)$

  $$p_{i,j} = \sum_{k=1}^{m} a_{i,k} \times b_{k,j}$$

  com $i = 1, \ldots, l,\ j = 1, \ldots n$

Declaração Dinâmica

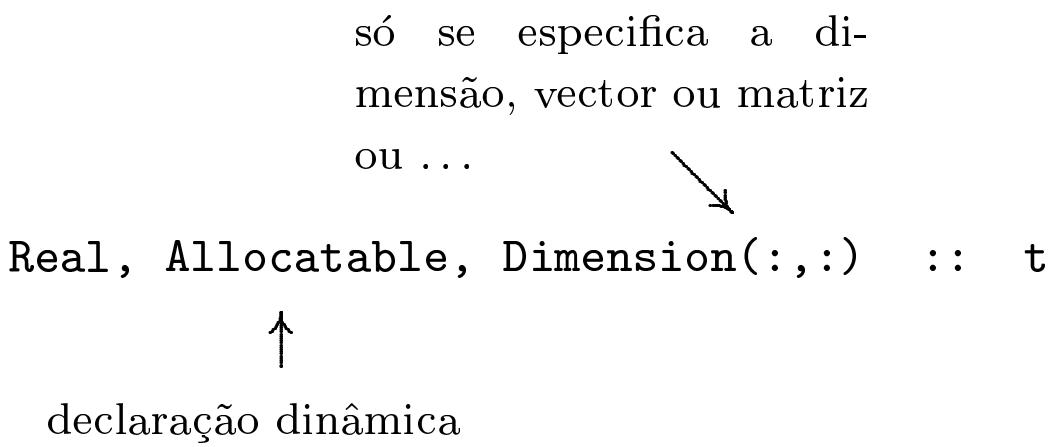
A declaração "normal" (estática) das tabelas tem o defeito de não se adaptar às necessidades de cada utilização:

- dimensão máxima não ultrapassável mesmo que haja recursos disponíveis.
- espaço de memória desperdiçado em muitas das utilizações.

Seria preferível uma declaração dinâmica em que o espaço afectado pela declaração não fosse fixo adaptando-se às necessidades:

- dimensão máxima só dependente dos recursos disponíveis.
- espaço de memória ajustado às necessidades.

O atributo `Allocatable` (alocável).



Uso

Depois da declaração e antes de se poder usar é necessário atribuir um espaço de memória à tabela. Por exemplo:

```
...
Read (*,*) nlinhas,ncolunas
Allocate(t(nlinhas,ncolunas),stat=estado)
```

A cláusula (opcional) `stat=estado` (com `estado` uma variável do tipo inteiro) tem o seguinte significado:

- se a instrução for executada com sucesso (existem recursos de memória suficientes) então `estado=0`
- senão `estado`$\neq$`0`. Os diferentes valores e o seu significado estão dependentes do compilador usado.
- Se se omitir a cláusula `stat` e houver problemas o programa interrompe a sua execução.

Verificar o estado da tabela

Para verificar se uma dada tabela já tem espaço atribuído existe a função (predicado) `Allocated(<`*nome*`>)`. Por exemplo:

```
if (.Not.(Allocated(t))) then
   Read (*,*) nlinhas,ncolunas
   Allocate(t(nlinhas,ncolunas),stat=estado)
End if
```

Libertação do Espaço

Depois da utilização é possível libertar o espaço ocupado pela tabela, isso é muito importante quando se utilizam sub-programas, essa libertação de espaço é feito recorrendo à instrução `Deallocate`.

```
Deallocate(t,stat=estado)
```

Não é necessário usar esta instrução quando se quer usar uma dada tabela até ao fim de um dado programa. O terminar da execução liberta todo o espaço ocupado pelo programa, inclusive o espaço ocupado pelas tabelas declaradas dinamicamente.