Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Array Operations

Pedro Barahona
DI/FCT/UNL
Introdução aos Computadores e à Programação
2º Semestre 2011/2012

Aggregation Operations on Vectors

- In many applications it is important to obtain features of data structures that require the consideration of all their elements to obtain an aggregated result.
- This is the case of arithmetic operations, such as the sum and the product, that are **commutative** and **associative** and where the aggregated value is the application of such operation to all elements of the data structure.
- In addition to these operations we may also consider operations such as the max and the min, that are also commutative and associative.
- These operations can be provided by user-defined functions as shown below for the sum and the maximum of a vector V, that update an **accumulation variable** with the operation on all elements of the vector, initialised to the **neutral value** of the operation.

```
function s = addition(V);
   s = 0;
   for i = 1:length(V)
      s = s + V(i);
   endfor
endfunction
```

```
function m = maximum(V);
   m = -inf;
   for i = 1:length(V)
      m = max(m, V(i));
   endfor
endfunction
```

Aggregation Operations on Vectors

- In fact, these are so common operations that they are predefined in MATLAB,
  - `sum(V)` : returns the sum of all elements of the vector
  - `prod(V)` : returns the product of all elements of the vector
  - `max(V)` : returns the maximum of all elements of the vector
  - `min(V)` : returns the minimum of all elements of the vector
- In the case of the last two function one may be interested in obtaining the index of the maximum/minimum element of the vector, and these can be obtained by a simple adaptation of the previous function

```
function [m,k] = maximum(V);
   m = -inf;
   for i = 1:length(V)
      m = max(m, V(i));
      k = i;
   endfor
endfunction
```

Aggregation Operations on Vectors

- This is, in fact, how the max/min functions are predefined in MATLAB, as you can check by calling the function with 2 return values

```
function [m,k] = maximum(V);
   m = -inf;
   for i = 1:length(V)
      m = max(m, V(i));
      k = i;
   endfor
endfunction
```

```
>> V = 1:3:10;
V =
    1  4  7 10
>>[a,b] = maximum(V)
a = 10
b = 4
>>[c,d] = maximum(V)
c = 10
d = 4
```

Aggregation Operations on Matrices

- These aggregation operations are also important for **matrices** and arrays of any number of dimensions (**multidimension** arrays).
- The definition of the sum for 2 and 3 dimensions can be done by adapting again the addition function, by considering nested loops to sweep all the elements of the data structures.

```
function s = addition_2(V);
   s = 0;
   for i = 1:size(V,1)
      for j = 1:size(V,2)
         s = s + V(i,j);
      endfor
   endfor
endfunction
```

```
function s = addition_3(V);
   s = 0;
   for i = 1:size(V,1)
      for j = 1:size(V,2)
         for k = 1:size(V,3)
            s = s + V(i,j,k);
         endfor
      endfor
   endfor
endfunction
```

Aggregation Operations on Matrices

- MATLAB does not provide the "expected" predefined aggregation functions for arrays of dimensions *greater than 1*, although it allows these functions to be called with higher dimension arrays as parameters.
- In this case, the operation is only applied to the elements of the first dimension , that is different from 1, returning an array where this dimension becomes 1, and each element is the result of the aggregation function over all elements with the same index in that dimension.
- More precisely, the sum applied to a **m×n** matrix (m>1) returns a vector with **1** row and **n** columns, each element representing the sum over the **m** rows of the corresponding elements of the original matrix. If n = 1 the result is presented as a "scalar".
- The case with dimensional arrays is a bit more "confusing". The sum applied to a **m×n×p** multidimensional array (m>1) returns a **1×n×p** multidimensional array where an element in each of the **n** columns and **p** layers is the sum over all the rows of the original multidimensional array.

Aggregation Operations on Matrices

Examples:

```
>> M = [1,3,5;2,4,6]
M =
     1  3  5
     2  4  6
>> S = sum(M)
S =
     3  7  11
>> U  = sum(S)
U = 21
>> V = sum(sum(M))
U = 21
```

```
>> X = ...
X(:,:,1) =
     3  4  5  6
     4  5  6  7
     5  6  7  8
X(:,:,2) =
     4  5  6  7
     5  6  7  8
     6  7  8  9
>> Y = sum(X)
Y(:,:,1) =
     12  15  18  21
Y(:,:,2) =
     15  18  21  24
>> Z = sum(Y)
Y(:,:,1) = 66
Y(:,:,2) = 78
>> A = sum(Z)
A = 144
>> B = sum(sum(sum(X)))
B = 144
```

Algebraic Operations on Vectors and Matrices

- Algebraic operations on vectors and matrices may be specified by user-defined functions.
- For example, one may consider the **dot product** of 2 vectors (U a row vector and U a column vector), with the same number of elements by simply summing the product of the corresponding elements of U and V.

```
function m = dot_product(U, V);
   m = 0;
   for i = 1:length(V)
      m = m + U(i)*V(i);
   endfor
endfunction
```

- Since MATLAB is specially designed for matrix operations, it “overloads” the * operator for the case of vectors and matrices.
- In case of vectors the dot_product is obtained by multiplying two vectors of the same size.

Algebraic Operations on Vectors and Matrices

- Note that the vectors must have "compatible" dimensions as seen below

```
>> V = [1,3,5]
V =
    1 3 5
>> V = [2,4,6]
U =
    2 4 6
>> p = U*V
error: operator *: nonconformant arguments (op1 is 1x3, op2 is 1x3)

>> q = U*V'
q = 44
```

- In fact our definition of dot product does not check that the U must be a row vector and V a column vector

```
>> p = dot_product(U*V)
p = 44
>> q = dot_product(U*V')
q = 44
```

Algebraic Operations on Vectors and Matrices

- An important characteristic of a (physical) vector is its size, i.e. Its Euclidean norm. Given a vector $\mathbf{A = [a_1,...,a_n]}$ its Euclidean norm is defined as

$$\mathbf{|A| = (a_1^2 + a_1^2 + ... \ a_n^2)^{1/2}}$$

and hence the function euclidean_norm can be defined as

```
function p = euclidean_norm(V);
   p = sqrt( V*V')
endfunction
```

which is actually available as the predefined function norm.

- The dot product of two vectors can be expressed as A • B = | A | | B | cos(α), where α is the angle between the two vectors. Given the previous results we may define the angle of two vector (in degrees) by function

```
function alpha = angle(V1, V2);
   g = (180/pi)* acos(180 * dot(V1,V2)) / (norm(V1) * norm(V2))
endfunction
```

Algebraic Operations on Vectors and Matrices

- An important algebraic operation on matrices is their product. The product of an **m×p** matrix **M** by a **p×n** matrix **N** is an **m×n** matrix **P**, whose elements P(i,j) are the dot product of the **i**th row of matrix M by the jth column of matrix N.
- This suggests a straightforward implementation of the matrix_product function

```
function P = matrix_product(M,N);
   m = size(M,1);
   n = size(N,2);
   P = zeros(m,n);
   for i = 1:m
      for j = 1:n
         M(i,j) = M(i;:) * N(:;j)
      endfor
   endfor
endfunction
```

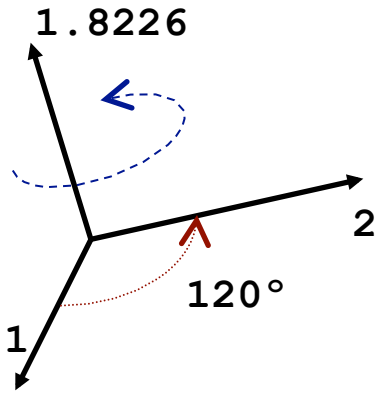
- Again, MATLAB is specially designed for matrix operations and so the operator * is overloaded for the case of matrices allowing the product to be specified “directly” as

```
P = M * N;
```

Cross Product of 2 Vectors

- Another type of product between 2 vectors U and V in 3D space is the cross product, defined as the vector Q, with length equal to the product of the lengths of vectors V1 and V2 and the sine of their angle, and perpendicular to both vectors, with direction defined by the “cork-screw rule”.

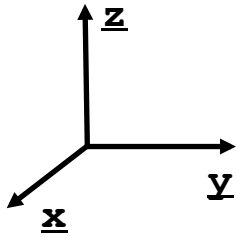
```
1.8226 = abs(1*2*sin(120*pi/180))
```



- To obtain this product, it is convenient to use matrix multiplication as shown next.

Cross Product of 2 Vectors

- Denoting by $\underline{x}$, $\underline{y}$ e $\underline{z}$ the unary vectors defining the 3 orthogonal axes, and according to the definition we have
  - $\underline{\mathbf{x}} \times \underline{\mathbf{x}} = \underline{\mathbf{y}} \times \underline{\mathbf{y}} = \underline{\mathbf{z}} \times \underline{\mathbf{z}} = \mathbf{0}$ (a vector is at an angle of 0º with itself)
  - $\underline{\mathbf{x}} \times \underline{\mathbf{y}} = \underline{\mathbf{z}}$; $\underline{\mathbf{x}} \times \underline{\mathbf{z}} = -\underline{\mathbf{y}}$; $\underline{\mathbf{y}} \times \underline{\mathbf{z}} = \underline{\mathbf{x}}$ (corkscrew rule) ;
  - $\underline{\mathbf{y}} \times \underline{\mathbf{x}} = -\underline{\mathbf{z}}$; $\underline{\mathbf{z}} \times \underline{\mathbf{x}} = \underline{\mathbf{y}}$; $\underline{\mathbf{z}} \times \underline{\mathbf{y}} = -\underline{\mathbf{x}}$ (corkscrew rule);



- Since the cross product is distributive wrt sum, given vectors $\mathbf{A} = a_x \underline{x} + a_y \underline{y} + a_z \underline{z}$ and $\mathbf{B} = b_x \underline{x} + b_y \underline{y} + b_z \underline{z}$ the cross product is given by

$$A \times B = (a_x \underline{x} + a_y \underline{y} + a_z \underline{z}) \times (b_x \underline{x} + b_y \underline{y} + b_z \underline{z}) =$$
$$= (a_y b_z - a_z b_y) \underline{x} + (a_z b_x - a_x b_z) \underline{y} + (a_x b_y - a_y b_x) \underline{z}$$

which can be obtained by the multiplication of vector **A** by a matrix **M** obtained from vector **B** as follows

$$[a_x \; a_y \; a_z] \times \begin{bmatrix} 0 & -b_z & b_y \\ b_z & 0 & -b_x \\ -b_y & b_x & 0 \end{bmatrix} = [a_y b_z - a_z b_y \quad a_z b_x - a_x b_z \quad a_z b_x - a_x b_z]$$

M

Cross Product of 2 Vectors

- Hence the cross product of two vectors can be defined with function

```
function P = cross_product(A,B)
   M = zeros(3,3);
   M(1,2) = -B(3); M(1,3) =  B(2);
   M(2,1) =  B(3); M(2,3) = -B(1);
   M(3,1) = -B(2); M(3,2) =  B(1);
   P = A * M;
endfunction
```

$$\begin{bmatrix} 0 & -b_z & b_y \\ b_z & 0 & -b_x \\ -b_y & b_x & 0 \end{bmatrix}$$

- The cross product is predefined in MATLAB, and can be obtained by function **cross**.

```
>> A = [ 1 -3  5]; B = [ -1 0 4];

>> C = cross(A,B)
C = -12 -9 -3

>> D = cross_product(B,A)
D =  12  9  3          %  A × B = - B × A
```

Matrices and Systems of Equations

- Matrices can of course be used for solving systems of linear equations.

$$\begin{aligned} a_{11}\,x1 + a_{12}\,x2 + \ldots + a_{1n}\,x_n &= b_1 \\ a_{21}\,x1 + a_{22}\,x2 + \ldots + a_{2n}\,x_n &= b_2 \\ \ldots\ldots & \\ a_{n1}\,x1 + a_{n2}\,x2 + \ldots + a_{nn}\,x_n &= b_n \end{aligned} \quad \Leftrightarrow \; AX = B$$

- A system of **n** linear equations on **n** unknowns can be represented in matrix form by a square **n×n** matrix **A** of the coeficients of the unknowns and a column vector **B** of values.

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ & \ldots\ldots & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{bmatrix}$$

which can be solved through the use of the inverse matrix

$$X = A^{-1}B$$

Matrices and Systems of Equations

- There are several ways of inverting a matrix. But since matrices are the basic data types of MATLAB, there are predefined functions and operations that can be used directly, namely matrix inversion and division.
- **Inverted Matrix:** The inversion of a matrix **M** can be obtained by calling the predefined function **inv(M)**, or through the usual algebraic notation $\mathtt{M}^{-1}$.
- In this case the equation system `AX = B` can be solved as

```
X = A^-1*B      or      X = inv(A)*B
```

- **Matrix Division:** The division operator is overloaded for matrix division, so as to equate the division by the multiplication with the inverse. But since matrix multiplication is not commutative there are two different divisions to consider.
  - **Left Division:** $\mathtt{A \backslash B = A^{-1} * B}$
  - **Right Division:** $\mathtt{B / A = B * A^{-1}}$
- Now, the above equation system can be solved with left division as

```
X = A \ B
```

Matrices and Systems of Equations

- **Example:** The system of 3 equations on 3 unknowns

$$\begin{aligned} 2\,x1 + 4\,x2 - x3 &= 7 \\ x1 - 2\,x2 + x3 &= 0 \\ -3\,x1 + 3\,x2 - x3 &= 2 \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} 2 & 4 & -1 \\ 1 & -2 & 1 \\ -3 & 4 & -1 \end{bmatrix} * \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 7 \\ 0 \\ 2 \end{bmatrix}$$

can be solved either by matrix inversion or matrix division

```
>> A = [2 4 -1; 1 -2 1 ; -1 3 -1]
A =
     2   4  -1
     1  -2   1
    -1   3  -1
>> B = [7 ; 0 ; 2]
B =
    7
    0
    2
>> A^-1
ans =
    0.33333   -0.33333   -0.66667
    0.00000    1.00000    1.00000
   -0.33333    3.33333    2.66667
```

```
>> X = A^-1*B
X =
    1.00000
    2.00000
    3.00000
>> X = A\B
X =
    1.00000
    2.00000
    3.00000
```

Filters

- Array operations involving Booleans may be used as filters and allow a very "compact" programming "instructions", as illustrated below.

Example: Obtain the number of elements in a vector **V** that are greater than k.

- This problem can be specified as an aggregated function on vectors, as before
- But we can notice that operation V > k "distributes" the relational operation through all elements of the vector, returning a vector of Booleans.
- Now summing this vector, gives the answer to the intended goal.

```
>> V = [2 5 8 3 0 4 8 7 9];
V = 2   5   8   3   0   4   8   7   9

>> k = 5;

>> B = V > k
B = 0   0   1   0   0   0   1   1   1

>> c = sum(B)
c = 4

>> c = sum(V>k)
c = 4
```