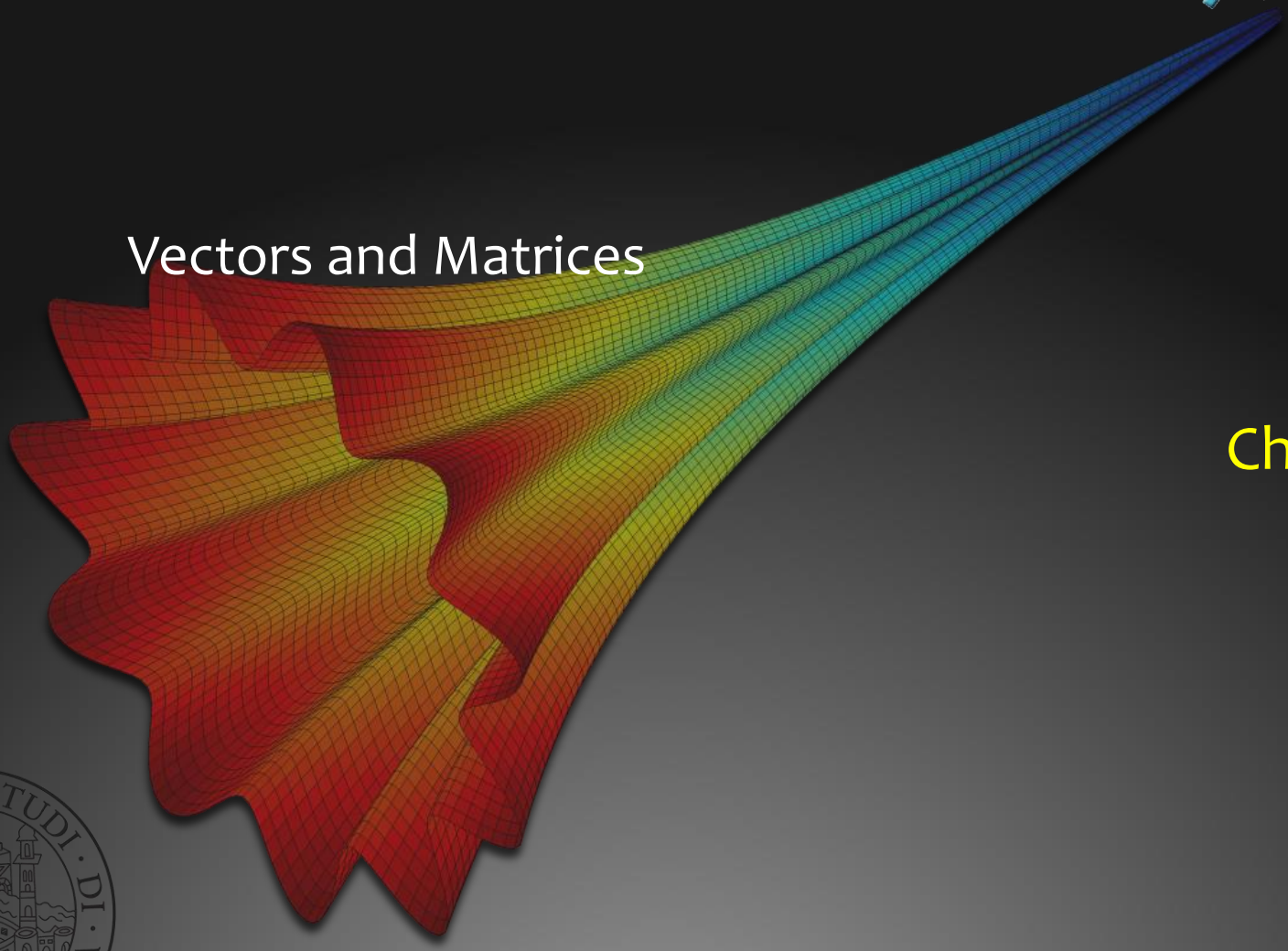
Vectors and Matrices



Chapter 2

Linguaggio Programmazione Matlab-Simulink (2017/2018)

Matrices

- A ***matrix*** is used to store a set of values of the same type; every value is stored in an ***element***
- MATLAB stands for "matrix laboratory"
- A matrix looks like a table; it has both rows and columns
- A matrix with m rows and n columns is called *m x n*; these are called its ***dimensions***; e.g. this is a 2 x 3 matrix:

9	6	3
5	7	2

- The term ***array*** is frequently used in MATLAB to refer generically to a matrix or a vector

Vectors and Scalars

- A ***vector*** is a special case of a matrix in which one of the dimensions is 1
  - a row vector with n elements is *1 x n*, e.g. *1 x 4*:

5	88	3	11

  - a column vector with m elements is *m x 1*, e.g. *3 x 1*:

3
7
4

- A ***scalar*** is an even more special case; it is *1 x 1*, or in other words, just a single value

5

Creating Row Vectors

- Direct method: put the values you want in square brackets, separated by either commas or spaces

```
>> v = [1  2  3  4]
v =
   1  2  3  4
>> v = [1,2,3,4]
v =
   1  2  3  4
```

- Colon operator: iterates through values in the form first:step:last e.g. `5:3:14` returns vector `[5 8 11 14]`
  - If no step is specified, the default is 1 so for example `2:4` creates the vector `[1 2 3 4]`
  - Can go in reverse e.g. `4:-1:1` creates `[4 3 2 1]`

Functions linspace, logspace

- The function **linspace** creates a linearly spaced vector; **linspace(x,y,n)** creates a vector with *n* values in the inclusive range from *x* to *y*
  - e.g. **linspace(4,7,3)** creates a vector with 3 values including the 4 and 7 so it returns `[4  5.5  7]`
  - If n is omitted, the default is 100 points
- The function **logspace** creates a logarithmically spaced vector; **logspace(x,y,n)** creates a vector with *n* values in the inclusive range from 10^x to 10^y
  - e.g. **logspace(2,4,3)** returns [100 1000 10000]
  - If n is omitted, the default is 50 points

Concatenation

- Vectors can be created by joining together existing vectors, or adding elements to existing vectors
- This is called ***concatenation***
- For example:

```
>> v = 2:5;
>> x = [33  11  2];
>> w = [v  x]
w =
     2     3     4     5    33    11     2
>> newv = [v  44]
newv =
     2   3   4   5   44
```

Referring to Elements

- The elements in a vector are numbered sequentially; each element number is called the ***index***, or ***subscript*** and are shown above the elements here:

1	2	3	4	5
5	33	11	-4	2

- Refer to an element using its *index* or *subscript* in parentheses, e.g. vec(4) is the 4th element of a vector *vec* (assuming it has at least 4 elements)
- Can also refer to a subset of a vector by using an *index vector* which is a vector of indices e.g. vec([2 5]) refers to the 2nd and 5th elements of vec; vec([1:4]) refers to the first 4 elements

Modifying Vectors

- Elements in a vector can be changed e.g.

  vec(3) = 11

- A vector can be extended by referring to elements that do not yet exist; if there is a gap between the end of the vector and the new specified element(s), zeros are filled in, e.g.

```
>> vec = [3   9];
>> vec(4:6) = [33 2 7]
vec =
     3     9     0    33     2     7
```

- Extending vectors is not a good idea if it can be avoided, however

Column Vectors

- A column vector is an *m x 1* vector
- Direct method: can create by separating values in square brackets with semicolons e.g. [4; 7; 2]
- You cannot directly create a column vector using methods such as the colon operator, but you can create a row vector and then ***transpose*** it to get a column vector using the transpose operator '
- Referring to elements: same as row vectors; specify indices in parentheses

Creating Matrix Variables

- Separate values within rows with blanks or commas, and separate the rows with semicolons
- Can use any method to get values in each row (any method to create a row vector, including colon operator)

```
>> mat = [1:3;  6 11 -2]
mat =
     1     2     3
     6    11    -2
```

- *There must ALWAYS be the same number of values in every row!!*

Functions that create matrices

- There are many built-in functions to create matrices
  - **rand(n)** creates an *nxn* matrix of random reals
  - **rand(n,m)** create an *nxm* matrix of random reals
  - **randi([range],n,m)** creates an *nxm* matrix of random integers in the specified range
  - **zeros(n)** creates an *nxn* matrix of all zeros
  - **zeros(n,m)** creates an *nxm* matrix of all zeros
  - **ones(n)** creates an *nxn* matrix of all ones
  - **ones(n,m)** creates an *nxm* matrix of all ones

  Note: there is no twos function – or thirteens – just **zeros** and **ones**!

Matrix Elements

- To refer to an element in a matrix, you use the matrix variable name followed by the index of the row, and then the index of the column, in parentheses

```
>> mat = [1:3; 6 11 -2]
mat =
    1    2    3
    6   11   -2
>> mat(2,1)
ans =
    6
```

- ALWAYS refer to the row first, column second
- This is called ***subscripted indexing***
- Can also refer to any subset of a matrix
  - To refer to the entire mth row: mat(m,:)
  - To refer to the entire nth column: mat(:,n)

Matrix Indexing

- To refer to the last row or column use **end**, e.g. mat(end,m) is the mth value in the last row
- Can modify an element or subset of a matrix in an assignment statement
- ***Linear indexing***: only using one index into a matrix (MATLAB will unwind it column-by column)
  - Note, this is not generally recommended

Modifying Matrices

- An individual element in a matrix can be modified by assigning a new value to it
- Entire rows and columns can also be modified
- Any subset of a matrix can be modified, as long as what is being assigned has the same dimensions as the subset being modified
- Exception to this: a scalar can be assigned to any size subset; the same scalar is assigned to every element in the subset

Matrix Dimensions

- There are several functions to determine the dimensions of a vector or matrix:
  - **length**(vec) returns the # of elements in a vector
  - **length**(mat) returns the larger dimension (row or column) for a matrix
  - **size** returns the # of rows and columns for a vector or matrix
  - Important: capture both of these values in an assignment statement

    [r c] = size(mat)
  - **numel** returns the total # of elements in a vector or matrix

- Very important to be general in programming: do not assume that you know the dimensions of a vector or matrix – use **length** or **size** to find out!

Functions that change dimensions

Many function change the dimensions of a matrix:

- **reshape** changes dimensions of a matrix to any matrix with the same number of elements
- **rot90** rotates a matrix 90 degrees counter-clockwise
- **fliplr** flips columns of a matrix from left to right
- **flipud** flips rows of a matrix up to down
- **flip** flips a row vector left to right, column vector or matrix up to down

Replicating matrices

- **repmat** replicates an entire matrix; it creates *m x n* copies of the matrix
- **repelem** replicates each element from a matrix in the dimensions specified

```
>> mymat = [33  11; 4  2]
mymat =
   33   11
    4    2
>> repmat(mymat, 2,3)
ans =
   33   11   33   11   33   11
    4    2    4    2    4    2
   33   11   33   11   33   11
    4    2    4    2    4    2
>> repelem(mymat,2,3)
ans =
   33   33   33   11   11   11
   33   33   33   11   11   11
    4    4    4    2    2    2
    4    4    4    2    2    2
```

Empty Vectors

- An *empty vector* is a vector with no elements; an empty vector can be created using square brackets with nothing inside [ ]
- to delete an element from a vector, assign an empty vector to that element
- delete an entire row or column from a matrix by assigning [ ]
  - Note: cannot delete an individual element from a matrix
- Empty vectors can also be used to "grow" a vector, starting with nothing and then adding to the vector by concatenating values to it (usually in a loop, which will be covered later)
  - This is not efficient, however, and should be avoided if possible

3D Matrices

- A three dimensional matrix has dimensions *m x n x p*
- Can create using built-in functions, e.g. the following creates a *3 x 5 x 2* matrix of random integers; there are 2 layers, each of which is a *3 x 5* matrix

```
>> randi([0 50], 3,5,2)
ans(:,:,1) =
    36    34     6    17    38
    38    33    25    29    13
    14     8    48    11    25
ans(:,:,2) =
    35    27    13    41    17
    45     7    42    12    10
    48     7    12    47    12
```

Arrays as function arguments

- Entire arrays (vectors or matrices) can be passed as arguments to functions; this is very powerful!
- The result will have the same dimensions as the input
- For example:

```
>> vec = randi([-5 5], 1, 4)
vec =
    -3     0     5     1
>> av = abs(vec)
av =
     3     0     5     1
```

Powerful Array Functions

- There are a number of very useful function that are built-in to perform operations on vectors, or on columns of matrices:
  - **min** the minimum value
  - **max** the maximum value
  - **sum** the sum of the elements
  - **prod** the product of the elements
  - **cumprod** cumulative, or running, product
  - **cumsum** cumulative, or running, sum
  - **cummin** cumulative minimum
  - **cummax** cumulative maximum

min, max Examples

```
>> vec = [4  -2  5  11];
>> min(vec)
ans =
   -2
>> mat = randi([1, 10], 2,4)
mat =
    6    5    7    4
    3    7    4   10
>> max(mat)
ans =
    6    7    7   10
```

- Note: the result is a scalar when the argument is a vector; the result is a *1 x n* vector when the argument is an *m x n* matrix

sum, **cumsum** vector Examples

- The **sum** function returns the sum of all elements; the **cumsum** function shows the running sum as it iterates through the elements (4, then 4+-2, then 4-2+5, and finally 4-2+5+11)

```
>> vec = [4  -2  5  11];
>> sum(vec)
ans =
    18
>> cumsum(vec)
ans =
     4     2     7    18
```

sum, **cumsum** matrix Examples

- For matrices, the functions operate column-wise:

```
>> mat = randi([1, 10], 2,4)
mat =
    1   10    1    4
    9    8    3    7
>> sum(mat)
ans =
   10   18    4   11
>> cumsum(mat)
ans =
    1   10    1    4
   10   18    4   11
```

The **sum** is the sum for each column; **cumsum** shows the cumulative sums as it iterates through the rows

prod, cumprod Examples

- These functions have the same format as **sum/cumsum,** but calculate products

```
>> v = [2:4   10]
v =
    2    3    4   10
>> cumprod(v)
ans =
    2    6   24  240
>> mat = randi([1, 10], 2,4)
mat =
    2    2    5    8
    8    7    8   10
>> prod(mat)
ans =
   16   14   40   80
```

Overall functions on matrices

- Since these functions operate column-wise for matrices, it is necessary to nest calls to them in order to get the function for all elements of a matrix, e.g.:

```
>> mat = randi([1, 10], 2,4)
mat =
    9    7    1    6
    4    2    8    5
>> min(mat)
ans =
    4    2    1    5
>> min(min(mat))
ans =
    1
```

diff Function

- The **diff** function returns the differences between consecutive elements in a vector
- For a vector with a length of n, the length of the result will be n-1

```
>> diff([4 7 2 32])
ans =
     3    -5    30
```

- For a matrix, the diff function finds the differences column-wise

Scalar operations

- Numerical operations can be performed on every element in a vector or matrix
- For example, ***Scalar multiplication:*** multiply every element by a scalar

```
>> [4  0  11] *  3
ans =
    12     0    33
```

- Another example: scalar addition; add a scalar to every element

```
>> zeros(1,3) + 5
ans =
     5     5     5
```

Array Operations

- ***Array operations*** on two matrices A and B:
  - these are applied term-by-term, or element-by-element
  - this means the matrices must have the same dimensions
  - In MATLAB:
    - matrix addition: A + B
    - matrix subtraction: A – B or B – A
  - For operations that are based on multiplication (multiplication, division, and exponentiation), a dot must be placed in front of the operator
    - array multiplication: A .* B
    - array division: A ./ B, A .\ B
    - array exponentiation A .^ 2
  - matrix multiplication: NOT an array operation

Logical Vectors

- Using relational operators on a vector or matrix results in a **logical** vector or matrix

```
>> vec = [44  3  2  9  11  6];
>> logv = vec > 6
logv =
1    0    0    1    1    0
```

- can use this to index into a vector or matrix (only if the index vector is the type **logical**)

```
>> vec(logv)
ans =
44    9    11
```

True/False

- **false** equivalent to logical(0)
- **true** equivalent to logical(1)

- **false** and **true** are also functions that create matrices of all **false** or **true** values

- As of R2016a, this can also be done with **ones** and **zeros**, e.g.

  logzer = ones(1,5, 'logical')

Logical Built-in Functions

- **any** returns true if anything in the input argument is true
- **all** returns true only if everything in the input argument is true

- **find** finds locations and returns indices

```
>> vec
vec =
    44     3     2     9    11     6
>> find(vec>6)
ans =
     1     4     5
```

Comparing Arrays

- The **isequal** function compares two arrays, and returns logical **true** if they are equal (all corresponding elements) or **false** if not

```
>> v1 = 1:4;
>> v2 = [1 0 3 4];
>> isequal(v1,v2)
ans =
     0
>> v1 == v2
ans =
     1     0     1     1
>> all(v1 == v2)
ans =
     0
```

Element-wise operators

- | and & are used for matrices; go through element-by-element and return logical 1 or 0
- || and && are used for scalars

Matrix Multiplication: Dimensions

- ***Matrix multiplication*** is NOT an array operation
  - it does NOT mean multiplying term by term
- In MATLAB, the multiplication operator * performs matrix multiplication
- Matrix multiplication has a very specific definition
- In order to be able to multiply a matrix A by a matrix B, the number of columns of A must be the same as the number of rows of B
- If the matrix A has dimensions *m x n*, that means that matrix B must have dimensions *n x something*; we' ll call it *p*
  - In mathematical notation, [A]*m x n* [B]*n x p*
  - We say that the ***inner dimensions*** must be the same
- The resulting matrix C has the same number of rows as A and the same number of columns as B
  - in other words, the ***outer dimensions*** *m x p*
  - In mathematical notation, [A]*m x n* [B]*n x p* = [C]*m x p*.
  - This only defines the size of C; it does not explain how to calculate the values

Matrix Multiplication

- The elements of the matrix C are found as follows:
- the sum of products of corresponding elements in the rows of A and columns of B, e.g.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Matrix Multiplication Example

$$\begin{bmatrix} 3 & 8 & 0 \\ 1 & 2 & 5 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 & 1 \\ 4 & 5 & 1 & 2 \\ 0 & 2 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 35 & 46 & 17 & 19 \\ 9 & 22 & 20 & 5 \end{bmatrix}$$

The 35, for example, is obtained by taking the first row of A and the first column of B, multiplying term by term and adding these together. In other words, 3*1 + 8*4 + 0*0, which is 35.

Vector Operations

- Since vectors are just special cases of matrices, the matrix operations described including addition, subtraction, scalar multiplication, multiplication, and transpose work on vectors as well, as long as the dimensions are correct
- Specific vector operations:
  - The ***dot product*** or ***inner product*** of two vectors a and b is defined as $a_1b_1 + a_2b_2 + a_3b_3 + \ldots + a_nb_n$
    - built-in function **dot** to do this
  - Also, **cross** for cross product

Common Pitfalls

- Attempting to create a matrix that does not have the same number of values in each row
- Confusing matrix multiplication and array multiplication. Array operations, including multiplication, division, and exponentiation, are performed term by term (so the arrays must have the same size); the operators are .*, ./, .\, and .^. For matrix multiplication to be possible, the inner dimensions must agree and the operator is *.
- Attempting to use an array of **double** 1s and 0s to index into an array (must be **logical**, instead)
- Attempting to use || or && with arrays. Always use | and & when working with arrays; || and && are only used with scalars.

Programming Style Guidelines

- If possible, try not to extend vectors or matrices, as it is not very efficient.
- Do not use just a single index when referring to elements in a matrix; instead, use both the row and column subscripts (use subscripted indexing rather than linear indexing)
- To be general, never assume that the dimensions of any array (vector or matrix) are known. Instead, use the function **length** or **numel** to determine the number of elements in a vector, and the function **size** for a matrix:

```
len = length(vec);

[r, c] = size(mat);
```

- Use **true** instead of **logical(1)** and **false** instead of **logical(0)**, especially when creating vectors or matrices.

Exercises

1. Create a vector variable and subtract 3 from every element in it. Create a matrix variable and divide every element by 3. Create a matrix variable and square every element.
2. When two matrices have the same dimensions and are square, both array and matrix multiplication can be performed on them. For the following two matrices, perform A.*B, A*B, and B*A by hand and then verify the results in MATLAB.

Exercises

3. Think about what would be produced by the following sequence of statements and expressions, and then type them in to verify your answers.

```
>>mat = [1:3; 44 9  2; 5:-1:3]
>>mat(3,2)              >>mat(2,:)
>>size(mat)             >>mat(:,4) = [8;11;33]
>>numel(mat)            >>v = mat(3,:)
>>v(v(2))      >>v(1) = []  reshape(mat,2,6)
```