Vectors and Matrices

Chul Min Yeum

Assistant Professor

Civil and Environmental Engineering

University of Waterloo, Canada

AE121: Computational Method

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Last updated: 2019-05-06

Remind: Matrix, Row Vector, and Column Vector

1.4 Algebraic Procedures to Solve Linear Systems

1.4.1 Matrices and Systems of Equations

A matrix is an ordered rectangular array of numbers. A general matrix with *m* rows and *n* columns has the following structure:

$$\begin{array}{c} \\ \text{row, } i \end{array} \begin{array}{c} \text{column, } j \\ \begin{array}{c} 1 \\ 2 \\ \vdots \\ m \end{array} \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & \cdots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{bmatrix}_{m \times n} \end{array}$$

where a_{ij} is each number in the array indexed by its row position i and column position j. Given their rectangular structure, matrices are ideal for storing systems of equations.

AE115 lecture note (Dr. Brush)

Matrix, Row Vector, and Column Vector

- 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***;
- A ***vector*** is a special case of a matrix in which one of the dimensions is 1
- The term ***array*** is frequently used in MATLAB to refer generically to a matrix or a vector
  - a row vector with n elements is 1 x n, e.g. 1 x 4:
  - a column vector with m elements is m x 1, e.g. 3 x 1:
- A **scalar** is an even more special case; it is 1 x 1, or in other words, just a single value

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 [2 3 4]
  - Can go in reverse e.g. 4:-1:1 creates[4 3 2 1]
  - Will not go beyond last e.g., 1:2:6 creates [1 3 5]

Example: Creating Row Vectors

```
% row vector
row_vec1 = [1 2 3 4]; % sparated by space
row_vec2 = [1   2   3   4]; % separated by multiple spaces
row_vec3 = [1, 2, 3, 4]; % separated by commas
row_vec4 = 1:4; % use a colon operator
row_vec5 = [1:4]; % okay with putting square brackets

% identical
row_vec1
row_vec2
row_vec3
row_vec4
row_vec5
```

```
row_vec1 = 1×4
     1     2     3     4

row_vec2 = 1×4
     1     2     3     4

row_vec3 = 1×4
     1     2     3     4

row_vec4 = 1×4
     1     2     3     4

row_vec5 = 1×4
     1     2     3     4
```

Remind: Transpose

The **transpose** of a matrix is the operation of flipping the rows to columns and vice versa across the diagonal of the matrix. For example,

$$A = \left[\begin{array}{c|c} 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \end{array}\right]_{3\times 2} \quad \text{has transpose} \quad A^T = \left[\begin{array}{c:c:c} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \end{array}\right]_{2\times 3}$$

where the power of *T* indicates the transpose of a matrix. Note that in the example, the dimensions of the matrix changes from 3x2 to 2x3 (with the same total number of elements) and diagonal elements remain the same.

By AE115 lecture note (Dr. Brush)

Creating 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 ' e.g., `[4 7 2]'`

Example: Creating Column Vectors

```
col_vec1 = [5;4;3];
col_vec2 = 5:-1:3; % not a column vector
col_vec3 = transpose(5:-1:3);
col_vec4 = col_vec2';

% col_vec1 is the same with col_vec3 and col_vec4
% col_vec1 is NOT the same with col_vec2
col_vec1
col_vec2
col_vec3
col_vec4
```

```
col_vec1 = 3×1
     5
     4
     3

col_vec2 = 1×3
     5     4     3

col_vec3 = 3×1
     5
     4
     3

col_vec4 = 3×1
     5
     4
     3
```

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

The index in MATLAB starts from 1

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 recommended

Refereeing: Error

Assigning : Okay

```
vec = [1 2 37];
vec
vec(1) = 5;
vec
vec(1:3) = [6 7 8];
vec
% vec(6) % error due to refer unassigned value
```

```
vec = 1×3
     1     2    37

vec = 1×3
     5     2    37

vec = 1×3
     6     7     8
```

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
```

Example: Concatenation

```
vec = [1 2 3 7];
vec1 = [vec 8 7]; % adding elements
vec2 = [vec [8 7]]; % adding a row vector
vec3 = [vec [8;7]']; % adding a column vector after transposing
vec4 = [vec';8;7]';
% adding a column vector to the column vector (vec1')
% and transpose an entire column vector
vec5 = vec;
vec5(5:6) = [8 7]; % valid code but not recomended

vec6 = cat(2, vec, [8 7]); % doc cat
% 1: column, 2: row, 3: depth

% identical
vec1
vec2
vec3
vec4
vec5
```

```
vec1 = 1×6
     1     2     3     7     8     7

vec2 = 1×6
     1     2     3     7     8     7

vec3 = 1×6
     1     2     3     7     8     7

vec4 = 1×6
     1     2     3     7     8     7

vec5 = 1×6
     1     2     3     7     8     7
```

Creating a Matrix

- 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!!*

Example: Creating Matrices

```
mat1 = [1 2 3;4 5 6;7 8 9]; % direct assignment (concatenate rows)
mat2 = [1:3;4:6;7:9]; % use a colon operator
mat3 = [[1;4;7] [2;5;8] [3;6;9]]; % concatenate columns
mat4 = reshape(1:9, 3, 3)'; % will learn

% identical
mat1
mat2
mat3
mat4
```

```
mat1 = 3×3
     1     2     3
     4     5     6
     7     8     9

mat2 = 3×3
     1     2     3
     4     5     6
     7     8     9

mat3 = 3×3
     1     2     3
     4     5     6
     7     8     9

mat4 = 3×3
     1     2     3
     4     5     6
     7     8     9
```

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**!

Example: Functions to Create Matrices

```
mat1_0 = [0 0 0; 0 0 0; 0 0 0];
mat2_0 = zeros(3,3);
mat1_0
mat2_0

mat1_1 = [1 1 1; 1 1 1; 1 1 1];
mat2_1 = ones(3,3);
mat1_1
mat2_1

rand(3,3) % random matrix where its elements are real number wit

mat1_random = randi([1 10], 3, 3); % random integer
mat1_random
```

```
mat1_0 = 3×3
     0     0     0
     0     0     0
     0     0     0

mat2_0 = 3×3
     0     0     0
     0     0     0
     0     0     0

mat1_1 = 3×3
     1     1     1
     1     1     1
     1     1     1

mat2_1 = 3×3
     1     1     1
     1     1     1
     1     1     1

ans = 3×3
    0.7922    0.0357    0.6787
    0.9595    0.8491    0.7577
    0.6557    0.9340    0.7431

mat1_random = 3×3
     4     8     1
     7     1     1
     2     3     9
```

Matrix Dimension

- There are several functions to determine the dimensions of a vector or 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
  - **length**(vec) returns the # of elements in a vector
  - **length**(mat) returns the larger dimension (row or column) for a 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!

Example: Matrix Dimension

```
mat1 = rand(3,5);
[sc1, sr1] = size(mat1); % use 'size' function
n_elem1 = sc1 * sr1;

sc2 = size(mat1,1); % doc size
sr2 = size(mat1,2); % 1 is a column, 2 is a row direction
n_elem2 = numel(mat1);

sc3 = numel(mat1(:,1)); % return a size of a column in mat1
sr3 = numel(mat1(1,:)); % return a size of a row in mat1
n_elem3 = numel(mat1); % return the total # of elements in mat1

% they are identical
[sc1 sc2 sc3]
[sr1 sr2 sr3]
[n_elem1 n_elem2 n_elem3]
```

```
ans = 1×3
     3     3     3

ans = 1×3
     5     5     5

ans = 1×3
    15    15    15
```

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
- 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) 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

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

Example: Modifying Matrix

```
mat0 = zeros(5,5);
row_vec = 1:5;
col_vec = 6:10;
mat_sub = ones(2,3);

% initialize all mat# with mat0 (a 5x5 zero matrix)
mat01 = mat0; mat02 = mat0; mat03 = mat0;
mat04 = mat0; mat05 = mat0; mat06 = mat0;
mat07 = mat0; mat08 = mat0; mat09 = mat0;
mat10 = mat0; clearvars mat0;

% evaluate your resulting matrix
mat01(2,:) = row_vec;
mat02(:,3) = col_vec;
mat03(1:2,1:3) = mat_sub;
mat04(2:4,1:2) = mat_sub';
mat05(1:4,1) = col_vec(1:4);
mat06(1:4,1) = row_vec(1:4)';
mat07(2:end, 1) = col_vec(2:end);
mat08(2:end, end) = row_vec(2:end)';
mat09(1:4, 1:2) = 3;
mat10(1,:) = 5;
```

mat01 = 5×5

0	0	0	0	0
1	2	3	4	5
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

mat02 = 5×5

0	0	6	0	0
0	0	7	0	0
0	0	8	0	0
0	0	9	0	0
0	0	10	0	0

mat03 = 5×5

1	1	1	0	0
1	1	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

mat04 = 5×5

0	0	0	0	0
1	1	0	0	0
1	1	0	0	0
1	1	0	0	0
0	0	0	0	0

mat05 = 5×5

6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
0	0	0	0	0

mat06 = 5×5

1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
0	0	0	0	0

mat07 = 5×5

0	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0

mat08 = 5×5

0	0	0	0	0
0	0	0	0	2
0	0	0	0	3
0	0	0	0	4
0	0	0	0	5

mat09 = 5×5

3	3	0	0	0
3	3	0	0	0
3	3	0	0	0
3	3	0	0	0
0	0	0	0	0

mat10 = 5×5

5	5	5	5	5
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Linear Indexing

Linear indexing: only using one index into a matrix (MATLAB will unwind it column-by column)

Very important

reshape

```
% introduce 'reshape': doc reshape
vec0 = 1:10;
mat0 = reshape(vec0, 2, 5);
mat1 = reshape(vec0, 2, []); % same with mat0.
% The second dimention is determined based on the # of element

% mat_error = reshape(vec0, 3,5); % error: dimension mismatch

mat0
mat1
```

```
mat0 = 2×5
     1     3     5     7     9
     2     4     6     8    10

mat1 = 2×5
     1     3     5     7     9
     2     4     6     8    10
```

Example: Linear Indexing

```
% linear indexing
mat00 = zeros(4,4);
row_vec = 1:4;
col_vec = transpose(5:8);
mat_sub = [1 2 3;4 5 6];

% initialize all mat# with mat0 (a 5x5 zero m
mat01 = mat00; mat02 = mat00; mat03 = mat00;
mat04 = mat00; mat05 = mat00; mat06 = mat00;
mat07 = mat00; mat08 = mat00;

% You should understand how they work
mat01(1:4) = row_vec;
mat02(1:4) = col_vec;
mat03(1:5) = [col_vec; 10];
mat04(1:6) = mat_sub;
mat05(1:6) = mat_sub';
% mat_error(1:7) =  mat_sub; % because of mismatching # of elements
mat06(8:11) = row_vec;
mat07([1 3 5 6]) = col_vec;
mat08([6 5 3 1]) = col_vec;

mat09 = reshape(1:16, 4, 4);
mat09(16:-1:1) = 1:16;

mat10 = reshape(1:16, 4, 4);
mat10(16:-1:1) = mat10(:); % this's cool!
```

```
mat00 = 4×4
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

row_vec = 1×4
     1     2     3     4

col_vec = 4×1
     5
     6
     7
     8

mat_sub = 2×3
     1     2     3
     4     5     6
```

```
mat01 = 4×4
     1     0     0     0
     2     0     0     0
     3     0     0     0
     4     0     0     0

mat02 = 4×4
     5     0     0     0
     6     0     0     0
     7     0     0     0
     8     0     0     0

mat03 = 4×4
     5    10     0     0
     6     0     0     0
     7     0     0     0
     8     0     0     0

mat04 = 4×4
     1     3     0     0
     4     6     0     0
     2     0     0     0
     5     0     0     0

mat05 = 4×4
     1     5     0     0
     2     6     0     0
     3     0     0     0
     4     0     0     0
```

```
mat06 = 4×4
     0     0     2     0
     0     0     3     0
     0     0     4     0
     0     1     0     0

mat07 = 4×4
     5     7     0     0
     0     8     0     0
     6     0     0     0
     0     0     0     0

mat08 = 4×4
     8     6     0     0
     0     5     0     0
     7     0     0     0
     0     0     0     0

mat09 = 4×4
    16    12     8     4
    15    11     7     3
    14    10     6     2
    13     9     5     1

mat10 = 4×4
    16    12     8     4
    15    11     7     3
    14    10     6     2
    13     9     5     1
```

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

```
vec1 = 1:5;
vec1(end) = []; % remove the last element (become 1x4)
vec1(2) = []; % remove the second element (become 1x3)

% vec1(5) % errors

mat1 = reshape(1:16, 4, 4);
mat1(:,3) = []; % remove the 3rd column in
mat1(2, :) = [];
% mat1(2, 1) = [] % error
```

```
vec1 = 1x3
     1     3     4

mat1 = 3x3
     1     5    13
     3     7    15
     4     8    16
```

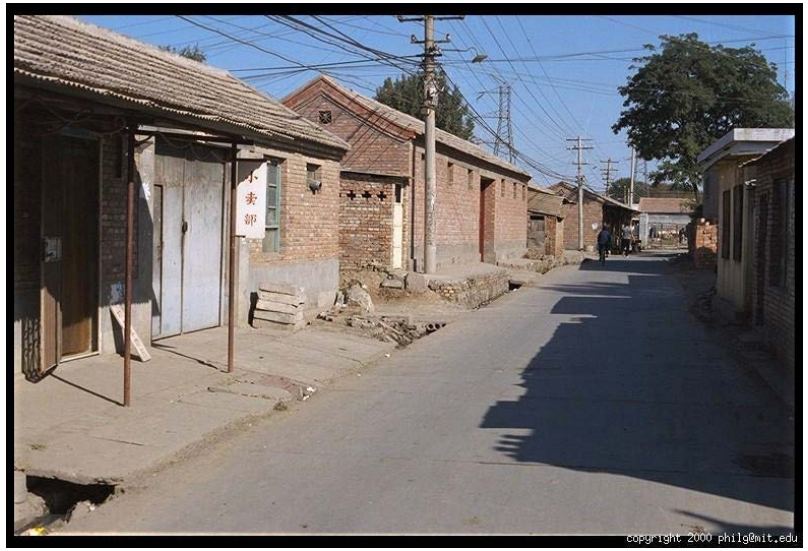
3D Matrices

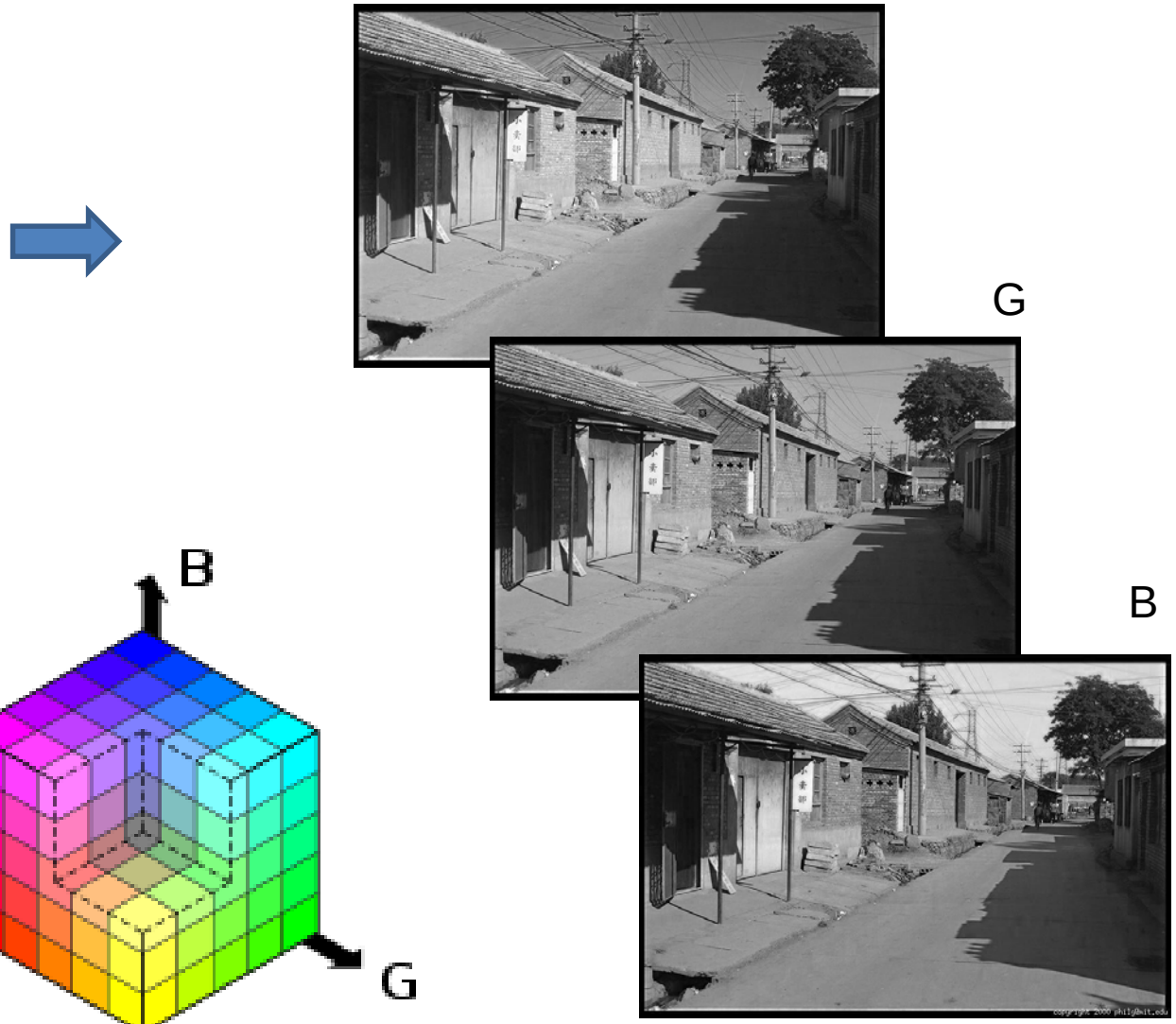
- A three dimensional matrix has dimensions *m x n x p*
- Can create using built-in functions, e.g. the following creates a *3 x 5 x 2* matrix of random integers; there are 2 layers, each of which is a *3 x 5* matrix

```
>> 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
```

Challenging ! But very useful !!

(Off-Topic) Color Image





0	3	2	5	4	7	6	9	8
3	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
5	2	3	0	1	2	3	4	5
4	3	2	1	0	3	2	5	4
7	4	5	2	3	0	1	2	3
6	5	4	3	2	1	0	3	2
9	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Example: 3D Matrix

```
mat3D = reshape(1:120, 8, 5, 3);

mat3D(:,:,1)
mat3D(:,:,2)
mat3D(:,:,3)
```

mat3D(:,:,1)

```
ans = 8×5
     1     9    17    25    33
     2    10    18    26    34
     3    11    19    27    35
     4    12    20    28    36
     5    13    21    29    37
     6    14    22    30    38
     7    15    23    31    39
     8    16    24    32    40
```

mat3D(:,:,2)

```
ans = 8×5
    41    49    57    65    73
    42    50    58    66    74
    43    51    59    67    75
    44    52    60    68    76
    45    53    61    69    77
    46    54    62    70    78
    47    55    63    71    79
    48    56    64    72    80
```

mat3D(:,:,3)

```
ans = 8×5
    81    89    97   105   113
    82    90    98   106   114
    83    91    99   107   115
    84    92   100   108   116
    85    93   101   109   117
    86    94   102   110   118
    87    95   103   111   119
    88    96   104   112   120
```

Example: 3D Matrix (Continue)

```
ans = 8x5
    81    89    97   105   113
    82    90    98   106   114
    83    91    99   107   115
    84    92   100   108   116
    85    93   101   109   117
    86    94   102   110   118
    87    95   103   111   119
    88    96   104   112   120
```

mat3D(:,:,3)

```
mat1 = mat3D(:,1,1);
mat2 = mat3D(1,1,:);
mat3 = mat3D(1,:,1);
```

Column vector

```
mat1 = 8x1
     1
     2
     3
     4
     5
     6
     7
     8
```

8 x 1

```
ans = 8x5
    41    49    57    65    73
    42    50    58    66    74
    43    51    59    67    75
    44    52    60    68    76
    45    53    61    69    77
    46    54    62    70    78
    47    55    63    71    79
    48    56    64    72    80
```

mat3D(:,:,2)

3D vector

```
mat2 =
mat2(:,:,1) =

     1

mat2(:,:,2) =

    41

mat2(:,:,3) =

    81
```

1 x 1 x 3

```
ans = 8x5
     1     9    17    25    33
     2    10    18    26    34
     3    11    19    27    35
     4    12    20    28    36
     5    13    21    29    37
     6    14    22    30    38
     7    15    23    31    39
     8    16    24    32    40
```

mat3D(:,:,1)

Row vector

```
mat3 = 1x5
     1     9    17    25    33
```

1 x 5

Example: 3D Matrix (Continue)

```
ans = 8x5
    81    89    97   105   113
    82    90    98   106   114
    83    91    99   107   115
    84    92   100   108   116
    85    93   101   109   117
    86    94   102   110   118
    87    95   103   111   119
    88    96   104   112   120
```

mat3D(:,:,3)

```
ans = 8x5
    41    49    57    65    73
    42    50    58    66    74
    43    51    59    67    75
    44    52    60    68    76
    45    53    61    69    77
    46    54    62    70    78
    47    55    63    71    79
    48    56    64    72    80
```

mat3D(:,:,2)

```
ans = 8x5
     1     9    17    25    33
     2    10    18    26    34
     3    11    19    27    35
     4    12    20    28    36
     5    13    21    29    37
     6    14    22    30    38
     7    15    23    31    39
     8    16    24    32    40
```

mat3D(:,:,1)

```
mat4 = mat3D(2,:,:);
mat5 = mat3D(:,2,:);
```

```
mat4 =
mat4(:,:,1) =

     2    10    18    26    34

mat4(:,:,2) =

    42    50    58    66    74

mat4(:,:,3) =

    82    90    98   106   114
```

1 x 5 x 3

```
mat5 =
mat5(:,:,1) =

     9
    10
    11
    12
    13
    14
    15
    16

mat5(:,:,2) =

    49
    50
    51
    52
    53
    54
    55
    56

mat5(:,:,3) =

    89
    90
    91
    92
    93
    94
    95
    96
```

8 x 1 x 3

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
```

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
```

```
mat1 = ones(3,3);
mat2 = mat1 + 3; % add 3 to all elements in mat1
mat3 = mat1 + ones(3,3)*3; % it is the same with the above

mat4 = zeros(size(mat1));
mat4(:) = 4; % modify all elements to 4
```

```
mat2 = 3x3
     4     4     4
     4     4     4
     4     4     4

mat3 = 3x3
     4     4     4
     4     4     4
     4     4     4

mat4 = 3x3
     4     4     4
     4     4     4
     4     4     4
```

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

They are totally different!

Example: Array Operations

```
matA = ones(3,3)*6;
matB = ones(3,3)*2;

mat1 = matA + matB;
mat2 = matB*4;
mat3 = matA - matB;
mat4 = matA.^2;

matC = ones(4,4);
matC(1:2,1:2) = 2;

matD = ones(4,4)*12;

mat5 = matC + matD;
mat6 = matC - matD;
mat7 = matC.*matD;
mat8 = matD./matC;
```

```
matA = 3x3
     6     6     6
     6     6     6
     6     6     6

matB = 3x3
     2     2     2
     2     2     2
     2     2     2

matC = 4x4
     2     2     1     1
     2     2     1     1
     1     1     1     1
     1     1     1     1

matD = 4x4
    12    12    12    12
    12    12    12    12
    12    12    12    12
    12    12    12    12
```

```
mat1 = 3x3
     8     8     8
     8     8     8
     8     8     8

mat2 = 3x3
     8     8     8
     8     8     8
     8     8     8

mat3 = 3x3
     4     4     4
     4     4     4
     4     4     4

mat4 = 3x3
    36    36    36
    36    36    36
    36    36    36
```

```
mat5 = 4x4
    14    14    13    13
    14    14    13    13
    13    13    13    13
    13    13    13    13

mat6 = 4x4
   -10   -10   -11   -11
   -10   -10   -11   -11
   -11   -11   -11   -11
   -11   -11   -11   -11

mat7 = 4x4
    24    24    12    12
    24    24    12    12
    12    12    12    12
    12    12    12    12

mat8 = 4x4
     6     6    12    12
     6     6    12    12
    12    12    12    12
    12    12    12    12
```

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
```

- **Extremely important and useful !!**
- Loop structure => Logical operations

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')

Example: Logical Vectors

```
vec1 = 2:11;

logi_vec = ones(1,10, 'logical'); % all true
logi_vec([3 5 7]) = false; % assign false logical values

vec2 = vec1(logi_vec); % access all values except values at 3, 5, and 7
vec3 = vec1(~logi_vec); % access values at 3, 5, and 7 locations

logi_equal_5 = (vec1==5);
vec4 = vec1(logi_equal_5); % values of 5

logi_larger_5 = (vec1>5);
vec5 = vec1(logi_larger_5); % values more than 5

logi_smaller_5 = (vec1<5);
vec6 = vec1(logi_smaller_5); % values less than 5

mat1 = reshape(vec1, 2, 5);
vec7 = mat1(logi_equal_5);
vec8 = mat1(logi_larger_5);
vec9 = mat1(logi_smaller_5);
% comment: use same logical vectors to access values in a matrix
```

```
vec1 =
     2     3     4     5     6     7     8     9    10    11

vec2 =
     2     3     5     7     9    10    11

vec3 =
     4     6     8

vec4 = 5
vec5 =
     6     7     8     9    10    11

vec6 =
     2     3     4

mat1 =
     2     4     6     8    10
     3     5     7     9    11

vec7 = 5
vec8 =
     6     7     8     9    10    11

vec9 =
     2     3     4
```

Row vector

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
```

Example: Logical Vectors

```
vec1 = 2:11;
mat1 = reshape(vec1, 2, 5);

tmp1 = (vec1 == 5);
is_there_five_v1 = logical(sum(tmp1));

is_there_five_v2 = any(vec1==5);
is_there_five_v3 = ~all(vec1~=5);
is_there_five_v4 = any(ismember(vec1, 5)); % doc ismember
% comment: 'ismember' supports various input array forms.

tmp4 = find(vec1==5);
is_there_five_v5 = (numel(tmp4)~=0);

tmp6 = (mat1 == 5);
is_there_five_v6 = logical(sum(tmp6, 'all'));
is_there_five_v7 = logical(sum(tmp6(:)));

tmp8 = find(mat1 ==5);
is_there_five_v8 = (numel(tmp8)~=0);
```

```
is_there_five_v1 = logical
   1
is_there_five_v2 = logical
   1
is_there_five_v3 = logical
   1
is_there_five_v4 = logical
   1
is_there_five_v5 = logical
   1
is_there_five_v6 = logical
   1
is_there_five_v7 = logical
   1
```

Element-wise operators

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

```
mat1 = reshape(0:11, 3, 4);
mat2 = mat1;
mat2(1,3) = 0;

logi_mat1 = (mat1 | mat2)
logi_mat2 = logical(logical(mat1) + logical(mat2))

logi_mat3 = (mat1 & mat2)
logi_mat4 = logical(mat1 .* mat2)
```

```
logi_mat1 = 3×4 logical array
     0   1   1   1
     1   1   1   1
     1   1   1   1

logi_mat2 = 3×4 logical array
     0   1   1   1
     1   1   1   1
     1   1   1   1

logi_mat3 = 3×4 logical array
     0   1   0   1
     1   1   1   1
     1   1   1   1

logi_mat4 = 3×4 logical array
     0   1   0   1
     1   1   1   1
     1   1   1   1
```

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
- 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 Times a Vector

2.3.4 Matrix Multiplication

2.3.4.1 Matrix times a Vector

A linear system of equations with coefficient matrix A, variable vector $\vec{x}$ and constant term vector $\vec{b}$, can be expressed as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$\overset{\text{matrix}}{(m\times n)}\,\overset{\text{vector}}{(n\times 1)} = \overset{\text{vector}}{(m\times 1)}$$

Compatibility – the number of columns in the matrix **must** equal the number of rows in the vector.

AE115 lecture note (Dr. Brush)

Example: Matrix Times a Vector

```
matA = reshape(1:9,3,3);
vecA = [1;2;3];

vec1 = matA*vecA; % matrix times a vector

vec2 = zeros(3,1);
vec2(1) = matA(1,:)*vecA; % row vector x column vector
vec2(2) = matA(2,:)*vecA;
vec2(3) = matA(3,:)*vecA;

vec3 = zeros(3,1);
vec3(1) = sum(matA(1,:)'.*vecA); % element-wise operation (vector, vector)
vec3(2) = sum(matA(2,:)'.*vecA);
vec3(3) = sum(matA(3,:)'.*vecA);

vec4 = zeros(3,1);
vec4(1) = dot(matA(1,:)', vecA); % dot product
vec4(2) = dot(matA(2,:)', vecA);
vec4(3) = dot(matA(3,:)', vecA);

vec5 = sum(matA.*[vecA';vecA';vecA'],2); % element-wise operation (matrix, matrix)
```

```
vec1 =
      30
      36
      42

vec2 =
      30
      36
      42

vec3 =
      30
      36
      42

vec4 =
      30
      36
      42

vec5 =
      30
      36
      42
```

Example: Matrix Times a Vector (Symbolic Expression)

```
syms a11 a12 a13 a21 a22 a23 a31 a32 a33
syms x1 x2 x3
A = [a11 a12 a13; a21 a22 a23; a31 a32 a33];
x = [x1;x2;x3];

vec1_syms = A*x;

vec2_syms(1,1) = A(1,:)*x;
vec2_syms(2,1) = A(2,:)*x;
vec2_syms(3,1) = A(3,:)*x;

vec3_syms(1,1) = sum(transpose(A(1,:)).*x);
vec3_syms(2,1) = sum(transpose(A(2,:)).*x);
vec3_syms(3,1) = sum(transpose(A(3,:)).*x);

vec5_syms = sum(A .* transpose([x x x]), 2);
```

vec1_syms =

$$\begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{pmatrix}$$

vec2_syms =

$$\begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{pmatrix}$$

vec3_syms =

$$\begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{pmatrix}$$

vec5_syms =

$$\begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{pmatrix}$$

Matrix Times a Matrix

2.3.4.2 Matrix times a Matrix

Matrix multiplication is an extension of matrix and vector multiplication.

Consider the product of an $m \times n$ matrix A, and an $n \times p$ matrix B:

$$AB = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ b_{31} & b_{32} & \cdots & b_{3p} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

$$\overset{\text{matrix}}{(m \times n)} \overset{\text{matrix}}{(n \times p)} = \overset{\text{matrix}}{(m \times p)}$$

Compatibility – the number of columns in the first matrix **must** equal the number of rows in the second matrix.

AE115 lecture note (Dr. Brush)

Example: Matrix Times a Matrix

```
matA = reshape(1:9,3,3);
matB = reshape(10:18,3,3);

mat1 = matA*matB; % matrix times a matrix

mat2 = zeros(3,3);
mat2(:,1) = matA*matB(:,1); % matrix x column vector
mat2(:,2) = matA*matB(:,2);
mat2(:,3) = matA*matB(:,3);

mat3 = zeros(3,3);
mat3(1,1) = matA(1,:)*matB(:,1); % row vector x column vector
mat3(1,2) = matA(1,:)*matB(:,2);
mat3(1,3) = matA(1,:)*matB(:,3);

mat3(2,1) = matA(2,:)*matB(:,1);
mat3(2,2) = matA(2,:)*matB(:,2);
mat3(2,3) = matA(2,:)*matB(:,3);

mat3(3,1) = matA(3,:)*matB(:,1);
mat3(3,2) = matA(3,:)*matB(:,2);
mat3(3,3) = matA(3,:)*matB(:,3);

mat4 = transpose(matB'* matA'); % AB = (B^T A^T)^T
```

```
mat1 = 3×3
     138   174   210
     171   216   261
     204   258   312

mat2 = 3×3
     138   174   210
     171   216   261
     204   258   312

mat3 = 3×3
     138   174   210
     171   216   261
     204   258   312

mat4 = 3×3
     138   174   210
     171   216   261
     204   258   312
```

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 **numel** to determine the number of elements in a vector, and the function **size** for a matrix:

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

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

Slide Credits and References

- Stormy Attaway, 2018, Matlab: A Practical Introduction to Programming and Problem Solving, 5th edition
- Lecture slides for “Matlab: A Practical Introduction to Programming and Problem Solving”
- Holly Moore, 2018, MATLAB for Engineers, 5th edition