Lecture 15
MATLAB II: Conditional Statements and Arrays

Conditional Statements

Recall boolean Expressions

- The boolean operators in MATLAB are:
  - > greater than
  - < less than
  - >= greater than or equals
  - <= less than or equals
  - == equality
  - ~= inequality
- The resulting type is **logical** 1 for true or 0 for false
- The logical operators are:
  - || or for scalars
  - && and for scalars
  - ~ not
- Also, **xor** function which returns logical true if only one of the arguments is true

If Statement

- The **if** statement is used to determine whether or not a statement or group of statements is to be executed
- General form:

```
if condition
    action
end
```

- the *condition* is any boolean expression
- the *action* is any number of valid statements (including, possibly, just one)
- if the condition is true, the action is executed – otherwise, it is skipped entirely

If-else Statements

- The **if-else** statement chooses between two actions
- General form:

```
if condition
    action1
else
    action2
end
```

- One and only one action is executed; which one depends on the value of the condition (action1 if it is logical true or action2 if it is false)

Nested if-else statements are ugly :-(

```
if cond1
    action1
else
    if cond2
        action2
    else
        if cond3
            % cond1 and cond2 False, cond3 True
            action3
            …
        else
            actionN
        end
    end
end
```

if-elseif statements are better :-|

MATLAB has an **elseif** clause which shortens nested if-else

```
if cond1
    action1
elseif cond2
    action2
elseif cond3
    % cond1 and cond2 False, cond3 True
    action3
…
else
    % if no other conditions met
    default_action
end
```

switch statements are (*sometimes*) best :-)

MATLAB also has a **switch** statement!

```
switch var
    case case1    % var == case1
        action1;
    case case2    % var == case2
        action2;
    case {case3,case4}
        % var == case3 || var == case4
        action3;
    …
    otherwise
        % var doesn't match any case
        default_action;
end
```

Example: branching.m

```
%%
x=-5;
%%
% Implements x = abs(x);
if x<0
    x = -x;
end
%%
% Forces x into the interval [a,b]
if x>b
    x = b;
elseif x<a
    x = a;
end
x
%%
% Forces x into the interval [a,b], and changes it's value to x^2.
x = 2; a=3; b=7;
if x>b
    x = b^2;
elseif x<a
    x = a^2;
else
    x = x^2;
end
x
% Is there a better way?
%%
g = 3; x=25; thresh = .1
% One step of Heron's squareroot
if abs(g^2-x)>thresh
    g = (g+x/g)/2
end
```

iClicker Question: What is the value of x?

```
x = 3; a=2; b=7;
if x>b
    x = b;
elseif x<a
    x = a^2;
else
    x = x^3;
end
```

A) x = 3

B) x = 27

C) x = 9

D) undefined

iClicker Question: What is the value of x?

```
x = 3; a=2; b=7;
if x>b
    x = b;
elseif x<a
    x = a^2;
else
    x = x^3;
end
```

A) x = 3

B) x = 27

C) x = 9

D) undefined

Common Pitfalls

- Using = instead of == for equality in conditions
- Putting a space in the keyword **elseif**
- Not using quotes when comparing a **char** variable to character,

        letter == y

  instead of

        letter == 'y'

- Writing conditions that are more complicated than necessary, such as

  if `(x < 5) == 1` instead of just `if (x < 5)`

Example: myQuadMin.m

```
function xmin = myQuadMin(a,b,c,L,R)
% xmin = quadMinizer(a,b,c,L,R)
% Returns x in the interval [L,R] that minimizes the quadratic function
% ax^2+bx+c.  Assumes a>=0, and L<R.
```

Example: myQuadMin.m

```
if a>0 % Parabola
    x0 = -b/(2*a); % argmin ax^2+bx+c for a>0
    if R<x0
        % [L,R] is to left of x0
        xmin = R;
    elseif L<=x0 && x0<=R
        % [L,R] contains x0
        xmin = x0;
    else
        % [L,R] is to right of x0
        xmin = L;
    end

elseif a==0 % Straight line
    if b>0
        % bx+c is sloping up
        xmin = L;
    elseif b<0
        % bx+c is sloping down
        xmin = R;
    else
        % bx+c is flat
        xmin = L;
    end
end
```

Programming Style Guidelines

- Use indentation to show the structure of a script or function. In particular, the actions in an **if** statement should be indented.
- When the **else** clause isn't needed, use an **if** statement rather than an **if-else** statement

Arrays

Arrays and Matrices

- Array_Basics.mlx

Arrays and Matrices

- An **array** is used to store sets of values of same type; each value is stored in an element of the array
  - A **matrix** is a two-dimensional array
  - A **vector** is a one-dimensional array
- Other programming languages mostly work with numbers one at a time, MATLAB® was designed from the ground up to operate primarily on whole matrices and arrays
- Most MATLAB classes come with multidimensional array support

Examples

1-Dimensional Arrays (Vectors)

- Point in R^n, Polynomial Coefficients
- Time Series – temp(t), annual snow falls, music, v(t), price(t)
- Strings, texts, webpages, DNA sequences

2-Dimensional Arrays (Matrices)

- System of equations, Linear Transforms, Covariance
- Images (m by n black and white image)
- Digital elevation data, Collections of points
- Stock market prices

3-Dimensional Arrays (3-D Matrix)

- Black and White Video
- Color Images

Matrices

- A **matrix** (2-D array) looks like a table; it has both rows and columns
- A matrix with m rows and n columns is said to be "m by n". Write this "*m x n*". Its first **dimension** is m; the second is n.

- This is a 2 x 3 matrix:

9	6	3
5	7	2

- The first row of is [9 6 3], the second row is [5 7 2]
- The first column is [9 5]', the last column is [3 2]'

Vectors and Scalars

- A ***vector*** (1-D array) 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: Use square brackets, with elements 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

>> x = [-10 v]
x = -10  1  2  3  4
```

Colon Operator

The colon operator creates evenly spaced row vectors;

start:step:max

produces a vector whose first element is **start** and whose subsequent elements are **step** apart, the last element is <= **max**.

```
>> 5:3:14
ans = [5 8 11 14]
>> 2:4 % default step size is 1
ans = [2 3 4]
>> 4:-1:1 % can go in reverse
ans = [4 3 2 1]
>> 0:.3:1 % fractional step sizes OK
ans = [0 .3 .6 .9]
```

linspace

linspace(a,b,n) creates a linearly (evenly) spaced row vector with ***n*** values starting at ***a*** and ending at ***b***.

```
>> linspace(4,7,3)
ans =  [4  5.5  7]
```

If n is omitted, the default is 100 points

colon vs. linspace

- Use `first:step:max` when you need to specify the `first` element and the `step` size. Last element returned is <= `max`.
- Use `linspace(a, b, n)` when you need to specify the first element `a` and last element `b`. Step size calculated base on number points `n`.

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] % concatenate v and x
w = 2     3     4     5    33    11     2

>> v = [v  44] % append 44 to v
v = 2   3   4   5   44
```

Referring to Elements

- The elements in a vector are indexed sequentially; an example ***index*** is 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,

  vec(4) is the 4^{th} element of a vector

- 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 2^{nd} and 5^{th} elements of vec;
  vec([1:4]) refers to the first 4 elements

Modifying Vectors

Elements in a vector can be changed via the assignment

```
>> vec(3) = 11;
>> vec(1:4) = [3 6 3 1];
>> vec(5:10) = 7;
```

Assignment to elements that do not yet exist is allowed (but not good style); 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
```

Column Vectors

A column vector is an *m x 1* vector; can create in square brackets with semicolons e.g.

```
>> x=[4; 7; 2]
x =
 4
 7
 2
```

- The colon operator only creates row vectors, but you can ***transpose*** row vectors to get a column vectors (and vice-versa) using the transpose operator ’

```
>> x=[4 7 2]'
x =
 4
 7
 2
```

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 uniform random numbers (real)
  - **rand(m,n)** create an *mxn* matrix of uniform random numbers (real)
  - **randi([range],m,n)** creates an *mxn* matrix of random integers in the specified range
  - **zeros(n)** creates an *nxn* matrix of all zeros
  - **zeros(m,n)** creates an *mxn* matrix of all zeros
  - **ones(n)** creates an *nxn* matrix of all ones
  - **ones(m,n)** creates an *mxn* 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

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 largest dimension (row or column) for a matrix - **:o(DO NOT USE length on arrays that are not vectors!**
  - **size** returns the # elements in each dimension of an array
  - Important: can capture multiple values in an assignment statement

    [r c] = size(mat)
  - **numel** returns the total # of elements in an array

- Very important to be general in programming: do not assume fixed dimensions of a vector or matrix – use **numel** or **size** to find out or avoid knowing via use of **end** and **:** inside the paranthesis!!

Functions that change dimensions

Many functions change the dimensions of a matrix:

- **reshape** changes dimensions of a matrix to any matrix with the same number of elements, linear order does not change
- **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
- **repmat** replicates a matrix; creates *m x n* copies of the matrix

Advanced Indexing

- See Array_Indexing.mlx

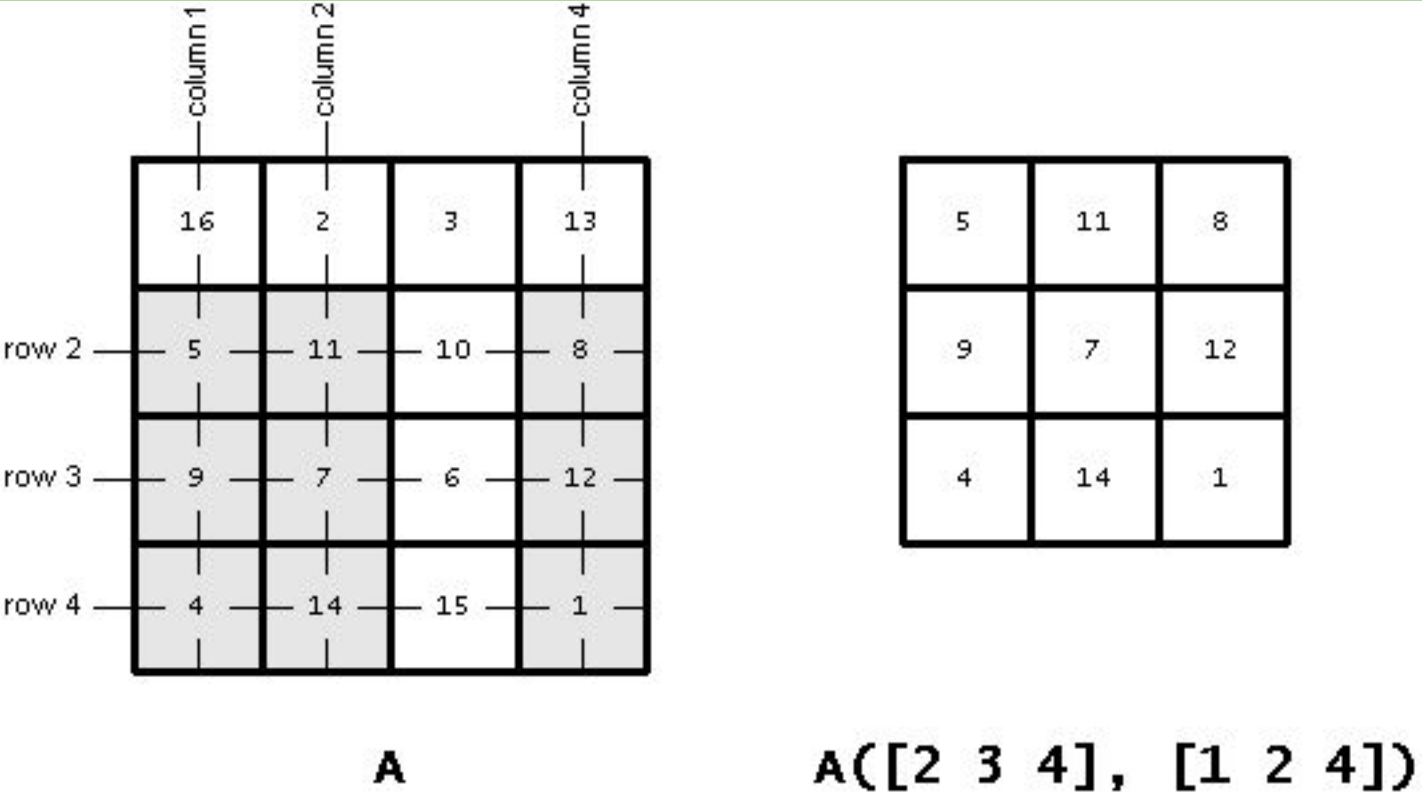
Advanced Indexing

- Isolated colon : refers to entire dimension
  `mat(i,:)` – the ith row of mat
  this is equivalent to `mat(i, 1:size(mat,2))`
- To refer to the last row or column use **end**
  `mat(end,k)` - the kth value in the last row

- Value of **end** and isolated **colon** : is determined by <u>context</u> within subscript.
  `mat(end, end)` – value of `mat(size(mat,1), size(mat,2))`

- Use of index vectors is also allowed
  m([2 4], [1 5]) returns the matrix
  [m(2,1) m(2,5) ; m(4,1) m(4,5)]

Advanced Indexing



Linear Array Indexing

The following works on all arrays (1-D, 2-D, etc.)

- `A(:)` forces A into a column vector containing all elements of `A`
- `A(k)` is the kth element of `A(:)`
- `A(M)` is a array with the same dimensions as `M`. For matrix `M`, the result would have elements `A(M(i,j))`

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

>> a([1 2; 3 4])
ans =

    16     5
     9     4
```

Removing Elements

- An *empty vector* is a vector with no elements; an empty vector can be created using square brackets with nothing inside [ ]

- Delete element(s) from a vector by assigning []

```
>> vec(1)=[];  % remove first element
>> vec[end-2:end]=[];  % remove last 3 elements
```

- Delete row(s) or column(s) from a matrix by assigning []

```
>> mat([1 end],:)=[]; % remove first and last row
```

Note: cannot delete an individual element from a matrix. *Can you see why?*

iClicker Question: Which vehicle is for Prof. G?

A

2012 Honda Pilot
90,000 miles
$0 / month

B

2019 Chevy Silverado
0 miles
$800 / month

C

2015 Jeep Wrangler
50,000 miles
$500 / month

D

2019 Honda Ridgeline
0 miles
$800 / month

E

2019 Jeep Wrangler
0 miles
$800 / month