Introduction to Applied Scientific Computing using MATLAB

Mohsen Jenadeleh

In this lecture, slides from MIT, Rutgers and Waterloo University are used to form the lecture slides.
Topics

Relational and logical operators
Precedence rules
Logical indexing

**find** function

Program flow control

**if** – statements
**switch** – statements

Examples:
piece-wise functions, unit-step function, indicator functions, sinc function
## Choose Symbolic or Numeric Arithmetic

<table>
<thead>
<tr>
<th>sin(π)</th>
<th>Symbolic</th>
<th>Variable Precision</th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a = sym(pi) sin(a)</td>
<td>b = vpa(pi) %vpa(pi,d) sin(b)</td>
<td>pi sin(pi)</td>
</tr>
<tr>
<td></td>
<td>a = pi ans = 0</td>
<td>b = 3.1415926535897932384</td>
<td>ans = 3.1416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>626433832795</td>
<td>ans = 1.2246e-16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ans = -3.2101083013100396069</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>547145883568e-40</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Round-Off Errors</th>
<th>No, finds exact results</th>
<th>Yes, magnitude depends on precision used (32 default)</th>
<th>Yes, has 16 digits of precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>slowest</td>
<td>Faster, depends on precision used</td>
<td>faster</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>Greatest</td>
<td>Adjustable, depends on precision used</td>
<td>Least</td>
</tr>
</tbody>
</table>
Relational and Logical Operators

Relational and logical functions

find, logical, true, false, any, all
ischar, isequal, isfinite, isnan, isinteger
islogical, isnan, isreal

>> doc is*   % list of all 'is' functions
>> help logical  % convert to logical
>> help true    % logical 1
>> help false   % logical 0
>> help relop   % relational operators (&, |, ...,)
>> help ops     % same as help /
>> help find    % indices of non-zero elements

>> help precedence % Operator Precedence in MATLAB.
Relational Operators

```plaintext
==    equal
~==   not equal
<     less than
>     greater than
<=    less than or equal
>=    greater than or equal
```

Logical Operators

```plaintext
&     logical AND,   e.g., A&B, A,B=expressions
|     logical OR,    e.g., A|B
~     logical NOT,   e.g., ~A
&&    logical AND for scalars w/ short-circuiting
||    logical OR for scalars w/ short-circuiting
xor   exclusive OR,  e.g., xor(A,B)
any   true if any elements are non-zero
all   true if all elements are non-zero
```

>> help relop
Operator Precedence in MATLAB (from highest to lowest):

1. transpose ( . ' ), power ( . ^ ), conjugate transpose ( ' ), matrix power ( ^ )
2. unary plus ( + ), unary minus ( - ), logical negation ( ~ )
3. multiplication ( . * ), right division ( . / ), left division ( . \ ), matrix multiplication ( * ), matrix right division ( / ), matrix left division ( \ )
4. addition ( + ), subtraction ( - )
5. colon operator ( : )
6. less than ( < ), less than or equal to ( <= ), greater than ( > ), greater than or equal to ( >= ), equal to ( == ), not equal to ( ~= )
7. element-wise logical AND ( & )
8. element-wise logical OR ( | )
9. short-circuit logical AND ( && )
10. short-circuit logical OR ( || )
>> a = [1, 0, 2, -3, 7];
>> b = [3, 4, 2, -1, 7];

>> a == b
ans =
     0    0    1    0    1

>> k = a == b    % clearer notation, k = (a==b)
ans =
     0    0    1    0    1

>> class(k)
ans =
    logical

>> a(k)        logical indexing
ans =
     2    7
>> a = [1, 0, 2, -3, 7];
>> b = [3, 4, 2, -1, 7];

>> a == b
ans =
     0     0     1     0     1

>> k = a == b  \% clearer notation, k = (a==b)
ans =
     0     0     1     0     1     1

>> i = find(a==b)  \text{using find}
i =
     3     5

>> a(i)
ans =
     2     7  \rightarrow a(a==b), a(find(a==b))
>> a = [1, 0, 2, -3, 7];
>> b = [3, 4, 2, -1, 7];

>> a == b
ans =
     0     0     1     0     1

>> a ~= b
ans =
     1     1     0     1     0

>> i = find(a~=b)
i =
     1     2     4

>> a(i),b(i)
ans =
     1     0     -3
ans =
      3     4     -1
>> a = [1, 0, 2, -3, 7];

>> ~a
ans =
   0   1   0   0   0   0

>> a==0
ans =
   0   1   0   0   0   0

>> i = find(~a)
i =
   2
>> a = [1, 0, 2, -3, 7];

>> a~=0
ans =
   1     0     1     1     1

>> ~~a
ans =
   1     0     1     1     1

>> logical(a)
ans =
   1     0     1     1     1

>> i = find(a)
i =
   1     3     4     5

>> a(find(a))
an =
   1     2    -3     7
```matlab
>> a = [1, 0, 2, -3, 7];
>> b = [3, 4, 2, -1, 7];

>> a<b, a>=b
ans =
     1     1     0     1     0
ans =
     0     0     1     0     1

>> i = find(a<b)
i =
     1     2     4

>> a(a<b), a(find(a<b))
an =
     1     0     -3
an =
     1     0     -3
```

*case 1: both a, b are vectors*
```matlab
>> a = [1, 0, 2, -3, 7];
>> b = 1;

>> a>=b
ans =
     1     0     1     0     1

>> i = find(a>=b)
i =
     1     3     5

>> a(a>=b), a(find(a>=b)), a(a<b)
ans =
     1     2     7
ans =
     1     2     7
ans =
     0    -3

case 2: a, b are vector, scalar
```
a = [1, 0, 2, -3, 7];
b = [3, 4, 2, -1, 7];

a>=1
ans =
  1  0  1  0  1

b<=2
ans =
  0  0  1  1  0

a>=1 & b<=2  % logical AND
ans =
  0  0  1  0  0

a>=1 | b<=2  % logical OR
ans =
  1  0  1  1  1

logical operations
```matlab
>> a = [1, 3, 4, -3, 7];
>> k = (a>=2), i = find(a>=2)
k =
  0 1 1 0 1
i =
  2 3 5

>> a(i), a(k)
ans =
  3 4 7
ans =
  3 4 7

>> n = [0 1 1 0 1]
>> a(n)
??? Subscript indices must either be real positive integers or logicals.

% but note, a(logical(n)) works
```

**logical indexing**

```
class(k) is logical
```

```
class(n) is double, but n==k is true
```

**class(k) is logical**

```
logical indexing
```

```
logical indexing
```
```matlab
>> A = [3 4 nan; -5 inf 2]
A =
    3     4   NaN
   -5    Inf    2

>> k = isfinite(A)
k =
    1     1     0
    1     0     1

>> A(k)           % listed column-wise
ans =
     3
    -5
     4
     2

>> A(~k) = 0      % set non-finite
A =           % entries to zero
    3     4     0
   -5     0     2
```

More on logical indexing:

```matlab
>> find(k)
ans =
     1
     2
     3
     6

>> [i, j] = find(k)
[i, j] =
     1     1
     2     1
     1     2
     2     3
```
A = [3 4 0; -5 5 2]
A =
    3   4   0
    -5   5   2

>> A > 2
ans =
    1   1   0
    0   1   0

>> k = find(A > 2)
k =
    1
    3
    4

>> [i, j] = find(A > 2);
[i, j] =
    1   1
    1   2
    2   2

>> A(find(A > 2))
ans =
    3
    4
    5
find can also be applied to a matrix of characters, e.g., the keypad matrix from week-3

```matlab
K = ['1' '2' '3'
     '4' '5' '6'
     '7' '8' '9'
     '*' '0' '#'];

K == '8'
ans =
     0   0   0
     0   0   0
     0   1   0
     0   0   0

[i, j] = find(K == '8')
i =
     3
j =
     2

q = find(K == '8')
q =
     7
```

compares every element of K with '8'

find the location of the correct element of K

i, j matrix indices of the location of '8'

q is the column-wise index of '8' in K
A = [9 9 2  
     2 5 4  
     9 8 9];  
B = [7 1 7  
     3 4 8  
     9 4 2];

>> A < B
ans =
0 0 1  
1 0 1  
0 0 0

>> find(A < B)
ans =
2
7
8

[i, j] = find(A < B)
i =
2
1
2
j =
1
3
3

>> A == 9
ans =
1 1 0  
0 0 0  
1 0 1

>> find(A == 9)
ans =
1
3
4
9

>> A(A == 9) = -9
A =
-9 -9 2  
2 5 4  
-9 8 -9
A = [9 9 2
     2 5 4
     9 8 9];
B = [7 1 7
     3 4 8
     9 4 2];

any(A==2)
ans =
     1 0 1

any(A==2,2)
ans =
     1
     1
     0

all(A>B)
ans =
     0 1 0

all(A>B,2)
ans =
     0
     0
     0

A==B
ans =
     0 0 0
     0 0 0
     1 0 0

any(A==B)
ans =
     1 0 0

any(any(A==B))
ans =
     1

all(all(A==B));
\[ A = \begin{bmatrix} 36 & -4 & 9 \\ 16 & 9 & -25 \end{bmatrix}, \quad B = A; \]

\[
\begin{array}{c}
A = \\
\begin{bmatrix}
36 & -4 & 9 \\
16 & 9 & -25 \\
\end{bmatrix}
\end{array}
\]

\[
\begin{array}{c}
>> k = (B>=0) \\
k = \\
\begin{bmatrix}
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{bmatrix}
\end{array}
\]

\[
\begin{array}{c}
>> B(k) = \sqrt{B(k)}; \\
>> B(~k) = -\sqrt{-B(~k)} \\
B = \\
\begin{bmatrix}
6 & -2 & 3 \\
4 & 3 & -5 \\
\end{bmatrix}
\end{array}
\]

Example:
take square-roots of the absolute values, but preserve the signs
Comparing Strings

Strings are arrays of characters, so the condition `s1==s2` requires both `s1` and `s2` to have the same length.

```
>> s1 = 'short'; s2 = 'shore';
>> s1==s1
ans =
    1     1     1     1     1
>> s1==s2
ans =
    1     1     1     1     0
>> s1 = 'short'; s2 = 'long';
>> s1==s2
??? Error using ==> eq
Matrix dimensions must agree.
```
Comparing Strings

Use `strcmp` to compare strings of unequal length, and get a binary decision.

```matlab
>> s1 = 'short'; s2 = 'shore';
>> strcmp(s1,s1)
ans =
    1
>> strcmp(s1,s2)
ans =
    0
```

Use `isequal` to compare the contents of matrices or arrays and get a binary decision.

```matlab
>> s1 = 'short'; s2 = 'long';
>> strcmp(s1,s2)
ans =
    0
```
Functions

• Functions describe subprograms
  –Take inputs, generate outputs
  –Have local variables (invisible in global workspace)

• Core MATLAB (Built-in) Functions
  –sin, abs, exp, ...
  Can’t be displayed on screen

• MATLAB-supplied M-file Functions
  –mean, linspace, ...
  Can be displayed on screen

• User-created M-file Functions

K>> type sin
sin is a built-in function.

K>> type linspace

function y = linspace(d1, d2, n)
%LINESPACE Linearly spaced vector.
% LINESPACE(X1, X2) generates a row vector of 100 linearly
% equally spaced points between X1 and X2.
% LINESPACE(X1, X2, N) generates N points between X1 and X2.
% For N < 2, LINESPACE returns X2.
% Class support for inputs X1,X2;
% float, double, single
% See also LOGSPACE,.

% Copyright 1984-2004 The MathWorks, Inc.
% $Revision: 5.12.4.1 $ $Date: 2004/07/05 17:01:20 $

if nargin == 2
    n = 100;
end

n = double(n);
y = (d1 + (0:n-2)*(d2-d1))/(floor(n-1)/d2);
Core MATLAB (Built-in) Functions

k>> type sin
• Elementary sin is a built-in function.
• >> help elfun % a list of these functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>Sine.</td>
</tr>
<tr>
<td>exp</td>
<td>Exponential.</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute value.</td>
</tr>
<tr>
<td>round</td>
<td>Round towards nearest integer.</td>
</tr>
</tbody>
</table>

• >> help specfun

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lcm</td>
<td>Least common multiple.</td>
</tr>
<tr>
<td>cart2sph</td>
<td>Transform Cartesian to spherical coordinates.</td>
</tr>
</tbody>
</table>

• Special functions - toolboxes
• Each toolbox has a list of special functions that you can use
function y = mean(x)

% MEAN Average or mean value.
% For vectors, MEAN(x) returns the mean value.
% For matrices, MEAN(x) is a row vector containing the mean value of each column.

[m,n] = size(x);
if m == 1
    m = n;
end
y = sum(x)/m;

»output_value = mean(input_value)
Multiple Input & Output Arguments

```matlab
function r = sum(X,y)
% OURRANK Rank of a matrix
if (nargin == 1)
    disp(' error message');
end
r = x+y
```

```matlab
function [mean,stdev] = ourstat(x)
% OURSTAT Mean & std. deviation
[m,n] = size(x);
if m == 1
    m = n;
end
mean = sum(x)/m;
stdev = sqrt(sum(x.^2)/m - mean.^2);

[M,LN] = ourstat(1:99);
```
\textbf{nargin, nargout, nargchk}

- \textbf{nargin} – number of input arguments
  - Many of Matlab functions can be run with different number of input variables.

\begin{verbatim}
if nargin==0
    disp('you have to send input arguments!');
end
\end{verbatim}

- \textbf{nargout} – number of output arguments
  - efficiency

\begin{verbatim}
IsOk=nargchk(1,5,nargin)
\end{verbatim}

- \textbf{nargchk} – check if number of input arguments is between some ‘low’ and ‘high’ values
Program flow is controlled by the following control structures:

1. for . . . end
2. while . . . end
3. break, continue
4. if . . . end
5. if . . . else . . . end
6. if . . . elseif . . . else . . . end
7. switch . . . case . . . otherwise . . . end
8. return

for-loops and conditional ifs are by far the most commonly used control structures
three forms of if statements

```
if condition
    statements ...
end

if condition
    statements ...
else
    statements ...
end

if condition1
    statements ...
elseif condition2
    statements ...
elseif condition3
    statements ...
else
    statements ...
end
```

several elseif statements may be present,

elseif does not need a matching end
>> x = 1;
>> % x = 0/0
>> % x = 1/0

if isinf(x),
    disp('x is infinite');
elseif isnan(x),
    disp('x is not-a-number');
else
    disp('x is finite number');
end

x is finite number
% x is not-a-number
% x is infinite
switch expression0
  case expression1
    statements ...
  case expression2
    statements ...
  otherwise
    statements ...
end

expression0 is evaluated first, and if its value matches any of the cases expression1, expression2, … , then the corresponding case statements are executed.

several case statements may be present

expression comparison rules:

numbers: isequal(expression0, expression1)
strings: strcmp(expression0, expression1)
x = [1, 4, -5, 3];

p = inf;
% p = 1;
% p = 2;

switch p
    case 1
        N = sum(abs(x));  \text{\% N = \text{norm}(x,1);}
    case 2
        N = sqrt(sum(abs(x).^2));  \text{\% N = \text{norm}(x,2);}
    case inf
        N = max(abs(x));  \text{\% N = \text{norm}(x,\text{inf});}
    otherwise
        N = sqrt(sum(abs(x).^2));  \text{\% N = \text{norm}(x,2);}
end

>> N
N =
    5
Example: $L_1$, $L_2$, and $L_\infty$ norms of a vector

\[ \mathbf{x} = [x_1, x_2, \ldots, x_N] \]

\[ \| \mathbf{x} \|_1 = \sum_{n=1}^{N} |x_n| \]

\[ \| \mathbf{x} \|_2 = \sqrt{\sum_{n=1}^{N} |x_n|^2} \]

\[ \| \mathbf{x} \|_\infty = \max(|x_1|, |x_2|, \ldots, |x_N|) \]

\[ d(\mathbf{x}, \mathbf{y}) = \| \mathbf{x} - \mathbf{y} \| \]

used as distance measure between two vectors or matrices

>> help norm  % vector and matrix norms
Example: unit-step function

\[ u(x) = \begin{cases} 
1, & x \geq 0 \\ 
0, & \text{otherwise} 
\end{cases} \]

\[ u = @(x) (x\geq0); \quad \% \text{unit-step function} \]

e.g., \( x = -3, -2, -1, 0, 1, 2, 3 \)
\( u(x) = 0, 0, 0, 1, 1, 1, 1, 1 \)

Example: indicator function

\[ v(x, a, b) = u(x - a) - u(x - b) \]

\[ v = @(x,a,b) u(x-a)-u(x-b); \quad \% \text{indicator} \]
\% \( v = @(x,a,b) (x\geq a \& x<b); \quad \% \text{alternative} \]
Example: Defining piece-wise functions (method 1)

\[ f(x) = \begin{cases} 
2x, & 0 \leq x \leq 0.5 \\
1, & 0.5 \leq x \leq 1.5 \\
4 - 2x, & 1.5 \leq x \leq 2 
\end{cases} \]

\[ v(x, a, b) = \begin{cases} 
1, & a \leq x < b \\
0, & \text{otherwise} 
\end{cases} = \text{(indicator function)} 
\]

\[ f(x) = 2x \cdot v(x, 0, 0.5) + v(x, 0.5, 1.5) + (4 - 2x) \cdot v(x, 1.5, 2) \]
Anonymous Function
- is a function that is not stored in a program file
- can accept inputs and return outputs
- they can contain only a single executable statement.
x = [-0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2 ...  
      0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0 ... 
      1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8 ... 
      1.9  2.0  2.1  2.2  2.3  2.4  2.5];

(x>=0 & x<0.5)
ans =
     0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(x>=0.5 & x<1.5)
ans =
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(x>=1.5 & x<2)
ans =
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
\[ g(c) = \int_{0}^{1} (x^2 + cx + 1) \, dx \]

\[ g = @(c) \left( \text{integral}(@(x) (x.^2 + c*x + 1),0,1)) \right); \]

Write the integrand as an anonymous function,
\[ @(x) (x.^2 + c*x + 1) \]

Evaluate the function from zero to one by passing the function handle to integral,
\[ \text{integral}(@(x) (x.^2 + c*x + 1),0,1) \]

Supply the value for \( c \) by constructing an anonymous function for the entire equation
\[ g = @(c) \left( \text{integral}(@(x) (x.^2 + c*x + 1),0,1)) \right); \]

The final function allows you to solve the equation for any value of \( c \).
\[ g(2) \]
\[ \text{ans} = 2.3333 \]
Using the indicator function

\[ v = @(x,a,b) ((x>=a) \ & \ (x<b)); \]

\[ f = @(x) 2x \ .* \ v(x, 0, 0.5) + \ldots \]

\[ v(x, 0.5, 1.5) + \ldots \]

\[ (4-2x) \ .* \ v(x, 1.5, 2); \]
Example: Defining piece-wise functions (method 2)

\[ x \text{ is a vector} \]

function \( y = f(x) \)

\[ y = \text{zeros(size}(x)); \]

\[ i1 = \text{find}(x \geq 0 \& x < 0.5); \]
\[ y(i1) = 2 \times x(i1); \]

\[ i2 = \text{find}(x \geq 0.5 \& x < 1.5); \]
\[ y(i2) = 1; \]

\[ i3 = \text{find}(x \geq 1.5 \& x < 2); \]
\[ y(i3) = 4 - 2 \times x(i3); \]
x = linspace(-0.5, 2.5, 301);
y = f(x);

figure; plot(x, y, 'b-');

axis([-0.5 2.5 0 1.2]);
xlabel('x-axis')
ylabel('y-axis')
xlim([-0.5 1])
ylim([0 2])
Example: Defining piece-wise functions (method 3)

function \( y = f(x) \)

\[
\begin{align*}
\text{if } x &\geq 0 \text{ and } x < 0.5 \\
y &\equiv 2x; \\
\text{elseif } x &\geq 0.5 \text{ and } x < 1.5 \\
y &\equiv 1; \\
\text{elseif } x &\geq 1.5 \text{ and } x < 2 \\
y &\equiv 4-2x; \\
\text{else} \\
y &\equiv 0; \\
\end{align*}
\]

**pitfall:** function produces wrong results if applied to a vector \( \mathbf{x} \), why?
\( x = \text{linspace}(-0.5,2.5,301); \)

\[
\text{for } n=1:\text{length}(x) \\
\quad y(n) = f(x(n)); \\
\text{end}
\]

\[
\text{figure; plot}(x,y, 'b-'); \\
\text{yaxis}(0,1.2, 0:0.5:1) \\
\text{xaxis}(-0.5,2.5, -0.5:0.5:2.5); \\
\text{xlabel}('\textit{x}');
\]

apply function separately to each element of \( x \), instead of the whole \( x \)
\texttt{x = linspace(-0.5, 2.5, 301);}

\texttt{for n=1:length(x)}
\texttt{    if x(n)>=0 & x(n)<0.5}
\texttt{        y(n) = 2*x(n);}
\texttt{    elseif x(n)>=0.5 & x(n)<1.5}
\texttt{        y(n) = 1;}
\texttt{    elseif x(n)>=1.5 & x(n)<2}
\texttt{        y(n) = 4-2*x(n);}
\texttt{    else}
\texttt{        y(n) = 0;}
\texttt{    end}
\texttt{end}

\texttt{figure; plot(x,y, 'b-');}
\texttt{yaxis(0, 1.2, 0:0.5:1)}
\texttt{xaxis(-0.5, 2.5, -0.5:0.5:2.5);}
\texttt{xlabel('\textbackslash itx');}
\[ f = @(x) 2 \times x \times (x \geq 0 \text{ and } x < 0.5) + \ldots \\
    (x \geq 0.5 \text{ and } x < 1.5) + \ldots \\
    (4 - 2 \times x) \times (x \geq 1.5 \text{ and } x < 2); \]

\[ x = \text{linspace}(0, 10, 501); \]

\textbf{figure; plot}(x, f(x) + f(x-3) + f(x-5), 'b-');

\textbf{replicating } f(x)
Example: Evaluating the sinc function

```matlab
function y = my_sinc(x)
    y = sin(pi*x)./(pi*x);
    y(isinf(x)) = 0;
    y(x==0) = 1;
```

Note: built-in `sinc` function returns NaN when \( x=\text{inf} \)
x = [0, 0, inf, 0, nan];
y = sin(pi*x)./(pi*x)
y =
     NaN   NaN   NaN   NaN   NaN

isinf(x)
ans =
     0     0     1     0     0

y(isinf(x)) = 0
y =
     NaN   NaN   0   NaN   NaN

x==0
ans =
     1     1     0     1     0     0

y(x==0) = 1
y =
     1     1     0     1     NaN