Introduction to MATLAB

- Basics
- Vectors
- Matrices
- Plotting

Outline

- Basics
- Vectors
- Matrices
- Plotting

Basic commands

MATLAB is a computer program that provides the user with a convenient environment for numerical computations and programming

- When MATLAB is ready to accept instructions, a command prompt (>>) is displayed in the command window
- scalar addition, subtraction, multiplication, division, and exponentiation can be computed using the symbols `+,-,*,/` and `^`, for example:

  ```
  >> 2+2
  ans = 4
  ```

- MATLAB automatically assigns the answer to a variable, `ans`; example:

  ```
  >> ans*2
  ans = 8
  ```

- assignment of values to variables can be done using equal sign; example:

  ```
  >> a=2.23
  a = 2.2300
  ```

  creates a variable named “a” with value equal to 2.23 and displays result

- result can be suppressed by terminating the command line with semicolon (`;`)
- text after (%) on same line are treated as comments and ignored

```
>>  a=4; create variable 'a' with value 4 without displaying result
```

- you can type several commands on same line by separating them with commas or semicolons; if you separate them with commas, they will be displayed

```
>> a = 4,A = 6;x = 1;
a = 4
```

- `e` is used for powers of ten (*e.g.*, 10^2 can be found using `1e2` or `10^2`)
- MATLAB predefines the variables
  - `pi`$= \pi$
  - `i=j` $= \sqrt{-1}$ (imaginary number)
  - `Inf` $= \infty$ and `NaN` means not a number

  for example, we can create a complex number

```
>> x=pi+2i
x = 3.1416 + 2i
```

- MATLAB displays four decimal points; for additional precision, use `format long`; we can switch back using `format short`

- `clear` command deletes all objects from the workspace
- `clear` followed by the names of the variables removes specific variables; *e.g.*,
  ```
  >> clear a %removes the variable 'a' from the workspace
  ```
- `clc` command clears the command window
- `save` command, followed by the desired filename, saves the workspace to a file, which has the .mat extension
- `load` command followed by the filename is used to load the data and objects contained in a MATLAB data file (.mat file)
- in the command window, pressing the up or down arrow key scrolls through previous commands and redisplays them at the command prompt
  - typing the first few characters and then pressing the arrow keys scrolls through the previous commands that start with the same characters
  - the arrow keys allow command sequences to be repeated without retyping

Built-in functions

function	command
$\sqrt{x}$	`sqrt(x)`
e^x	`exp(x)`
$\sin(x)$	`sin(x)`
$\cos(x)$	`cos(x)`
$\tan(x)$	`tan(x)`
$\tan^{-1}(x)$	`atan(x)`
$\log_{10} x$	`log10(x)`
$\ln x$	`log(x)`

Complex numbers

command	meaning
`real(x)`	real part of `x`
`imag(x)`	imaginary part of `x`
`abs(x)`	absolute value of `x`
`angle(x)`	phase of `x` in rad/s
`conj(x)`	complex conjugate of `x`

(for list of functions type `help elfun`)

Rounding and remainder

command	meaning
`round(x)`	rounds to nearest integer
`fix(x)`	rounds to nearest integer towards zero
`floor(x)`	rounds down (towards negative infinity)
`ceil(x)`	rounds up (towards positive infinity)
`mod(x,y)`	modulus (signed remainder after division)
`rem(x,y)`	remainder after division

Example

```
>> x = 2.3 - 4.7*i;
>> round(x);   % results in (2 - 5i)
>> fix(x);     % results in (2 - 4i)
>> floor(x);   % results in (2 - 5i)
>> ceil(x);    % results in (3 - 4i)
```

Strings

- character strings can be represented by enclosing the strings within single quotation marks; for example

```
>> f = 'Miles';
>> s = 'Davis';
```

- we can concatenate (*i.e.*, paste together) strings as in

```
>> x = [f s]
x =
Miles Davis
```

- `str2num(s)` converts string `s` to a number

- `num2str(n)` converts number `n` to a string

Relational operations

a relational operator compares two items and indicates whether a condition is true

relational operator	**meaning**
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
~=	not equal to

- if true, a logical true (1) is returned; else, a logical false (0) is returned
- for example

```
>> 1>2
ans =
logical
0
```

Logical operations

logical operator	meaning
&	logical AND
|	logical OR
~	logical negation

- relational operators can be combined using logical operators
- for example, we can test the condition $0 < t < 1$ using

```
>>(t>0)&(t<1)
```

  or

```
>>~((t<=0)|(t>=1))
```

If statements

- *if* statements execute commands if a certain condition is met

```
if condition
statements
else
statments
end
```

- **example:** set $x = 5$ if $a > 0$ and $x = 100$ otherwise

```
>> a = 15;
>> if a > 0,
   x = 5;
   else
  x = 100
 end
```

For loop

- *for* statements loop a specific number of times, and keep track of iteration index

```
for index = values
statements
end
```

- **example:** determine the product of all prime numbers between 1 and 20

```
>> result = 1;
>> for n = 1:20 % iterate over 'n' from 1 to 20
if isprime(n) % built-in function
result = result*n;
end
end
```

While loop

while statements loop as long as a condition remains true

```
while expression
statements
end
```

example: find the first integer n for which factorial(n) is a 100-digit number

```
n = 1;
nFact = 1;
while nFact < 1e100
n = n + 1;
nFact = nFact * n;
end
```

Anonymous function

an *anonymous function* provides a symbolic representation of a function defined in terms of MATLAB operators, functions, or other anonymous functions

Example: we can define $f(t) = e^{-t}\cos(2\pi t)$ as

```
>> f = @(t) exp(-t)*cos(2*pi*t);
```

- symbol `@` identifies the expression as an anonymous function
- parentheses following `@` symbol are used to identify variables (input arguments)
- $f(t)$ can be evaluated simply by passing the input values of interest

  ```
  >> t = 0; f(t)
  ans = 1
  ```

Example: piecewise functions

- the unit step function

$$u(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases}$$

can be created using the command

```
>> u = @(t) 1.0*(t>=0);
```

- the function

$$f(t) = \begin{cases} 1 & 0 \leq t \leq 2 \\ -t & -1 \leq t < 0 \end{cases}$$

can be created using the command

```
>> f = @(t) 1.0*((t>=0)&(t <=2))-t*((t<0)&(t>=-1));
```

Functions M-files

- script M-files file is a series of commands saved on a file that can be run at once
- function M-files can accept input arguments as well as return outputs
- a function M-file is identical to a script M-file except for the first line
- the general form of the first line is

  ```
  function [outputs] = filename(inputs)
  ```

- an M-file is executed by simply typing the filename (without the .m extension)

Example

M-file content

```
function [f1] = myfirsfunx(x)
% input x, output f1
f1 = sin(pi*x); % Calculate function f1
end
```

Execute function M-file

```
>> x = 2; % Define the input argument
>> [y] = myfirstfunc(x); % Output value is returned to y
```

Outline

- Basics
- Vectors
- Matrices
- Plotting

Vectors

vector arrays are created using square brackets and semicolon

- we can create the row vector $x = [1 \ \ 2 \ \ 3]$ via the command

```
>> x = [1 2 3]
x = 2 0 3
```

- we can create the column vector $y = \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix}$ via the command

```
>> y=[-1;-2;-3]
y =
-1
-2
-3
```

- (conjugate) transpose of a vector can be found using apostrophe

```
>> y'
ans =
-1     -2     -3
```

Vector indexing and operations

Vector indexing

- the nth element of `x` can be extracted using `x(n)`
- we can use indexing to get a slice of a vector; for example, `x(98:100)` gives a vector (`x(98)`,`x(99)`,`x(100)`)
- `end` command automatically references the final index of an array; for example,

```
>> x(end-9:end) % extract final 10 values of vector x
```

- we can concatenate vectors to create a larger vector

```
>> a=[1;2];b=[1;1];c=[-1;-1]
>> d=[a;b;c]; % create concatenated vector
```

Vector operations

- vector addition and subtraction are carried out using the commands `+`,`-`
- element-by-element operations are computed using (`.*, ./,.^`); example:

```
>> u=[1 2 3]; v=[-1 -2 -3];
>> w=u.*v
w =  -1    -5    -9
```

Basic vector commands

command	meaning
`a:b:c`	vector with elements between `a` and `c` with increments `b` (command `a:c` assumes increment of 1)
`linspace(x1,x2,n)`	n-vector from `x1` to `x2` with equal spacing `(x2-x1)/(n-1)`
`logspace(x1,x2,n)`	n-vector from $10^{\mathtt{x1}}$ to $10^{\mathtt{x2}}$ logarithmic spacing
`sum(x)`	sums the elements of `x`
`prod(x)`	return products of elements of `x`
`max(x)`	return max value in `x`
`min(x)`	return min value in `x`
`sort(x)`	sorts elements in ascending order
`ones(1,n)`/`zeros(1,n)`	row n-vector of all ones/zeros
`ones(n,1)`/`zeros(n,1)`	column n-vector of all ones/zeros
`length(x)`	returns the length of the vector `x`
`x'y` or `dot(x,y)`	return inner (dot) product between vectors `x` and `y`
`norm(x)`	return the 2-norm of vector `x`
`flipud(x)`	reverses the order of elements in a column vector `x`
`fliplr(x)`	reverses the order of elements in a row vector `x`

Functions of vectors

- common built in functions operates elementwise on vectors and matrices
- example: to compute $\sqrt{x}$ for all values $(1, 2, \ldots, 100)$, we can use

```
>> x = 1:100; y=sqrt(x);
```

- vectors can be used to represent points of a function f over some interval
- for example, we can represent $f = \sin(2\pi 10t + \pi/6)$ over $0 \leq t \leq 2$ using

```
>> t = linspace(0,2,500); %500 points between 0 and 2
>> f = sin(2*pi*10*t+pi/6)
```

- indexing in Matlab starts from 1
- for example, the value of $f(t)$ at $t = 0$ is the first element of the vector `f(1)`

Example

$$x = \begin{bmatrix} 1 \\ 4 \\ -2 \\ (3-j2) \end{bmatrix}, \quad y = \begin{bmatrix} -3 \\ (5+j7) \\ 6 \\ 2 \end{bmatrix}$$

use Matlab to compute

(a) $x + y$

(b) inner product $x^* y = \sum_{k=1}^{4} x(k)^* y(k)$

(c) mean or average $\mathrm{avg}(x) = (1/4) \sum_{k=1}^{4} x(k)$

(d) average energy $E_x = (1/4) \sum_{k=1}^{4} |x(k)|^2$

(e) variance $\mathrm{var}(x) = (1/4) \sum_{k=1}^{4} |x(k) - \mathrm{avg}(x)|^2$

Solution:

(a)
```
>> x = [1;4;-2;3-2*i];
>> y = [-3;5+7*i;6;2];
>> sum_xy = x + y;
```

(b)
```
>> dot_xy = dot(x,y);
>> dot_xy = x'*y; % alternative computation
```

(c)
```
>> mean_x = sum(x)/length(y);
>> mean_x = mean(x); % alternative computation
```

(d)
```
>> avg_x = sum(x.*conj(x))/length(x);
>> avg_x = sum(x'*x)/length(x); % alternative computation
>> avg_x = norm(x)^2/length(x); % alternative computation
>> avg_x = mean(abs(x).^2); % alternative computation
```

(e)
```
>> z=x-mean(x);
>> var_x = sum(z.*conj(z))/length(x);
>> var_x = sum(z'*z))/length(x); % alternative computation
>> var_x = mean(|z|.^2); % alternative computation
```

Find function

the `find` allows us to find indices satisfying certain conditions

Example: find indices of vector x bigger than 1

```
>> x=[1;-1;3;4]
>> find(x>1)
ans =
3
4
```

Outline

- Basics
- Vectors
- Matrices
- Plotting

Matrices

matrices can be created similar to vectors using square brackets and semicolon

- we can create the 3×4 matrix $A = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 0 & 6 \end{bmatrix}$ by the command

```
>> A = [2 3;4 5;0 6]
A = 2 3
    4 5
    0 6
```

- (conjugate) transpose of a matrix can be found using apostrophe

```
>> A'
ans =
2     4     0
3     5     6
```

Matrix indexing

- element (k, l) of matrix `A` can be extracted using `A(k,l)`
- subblocks of `A` can be extracted using indexing; for example

```
>> A = [1 2 3;
         0 4 5;
         0 0 6];
>>A(1:2,2:3)
ans = 2 3
4 5
```

  `A(2,:)` selects all column elements along the second row

```
>> A(2,:)
ans = 0 4 5
```

- we can concatenate arrays to create larger arrays; for example

```
>> a = [1;0;0]; B = [2 3;4 5;0 6]
>> C = [a B]
C = 1 2 3
0 4 5
0 0 6
```

- `repmat` command replicate objects; for example

```
>> u=[1 2]; repmat(u,1,3)
ans = 1     2     1     2     1     2
>> u=[1 2]; repmat(u,2,1)
ans = 1     2
   1     2
```

Matrix operations

- matrix addition and multiplications are carried out using the commands `+,-,*`
- matrix power can be found using `^` (*e.g.*, `A^3`)
- element-by-element operations are computed using `.*, ./,.^`
- passing a matrix into a function computes the function elementwise

Linear equation: we can solve `Ax=b` using backlash operator (left division): `x=A\b`, which is more computationally efficient than `inv(A)*b`

Basic matrix commands

command	**meaning**
`sum(A)`	returns a row vector containing the sum of each column
`sum(A,2)`	returns a column vector containing the sum of each row
`sum(A,"all")`	returns the sum of all elements of A
`prod(A)`	returns a row vector containing the sum of each column
`max(A)/min(A)`	return max/min value in `A`
`eye(m)`	$m \times m$ identity matrix
`ones(m,n)/zeros(m,n)`	$m \times n$ matrix of all ones/zeros
`diag(x)`	creates diagonal matrix with diagonal elements x
`length(A)`	returns the length of the largest array dimension in `A`
`size(A)`	returns the size of the array `A`
`det(A)`	determinant of a square matrix `A`
`inv(A)`	inverse of a square matrix `A`
`eig(A)`	computes eigenvalues and eigenvectors of `A`
`rank(A)`	computes rank of `A`
`norm(A)`	return the 2-norm of A
`norm(A,"fro")`	return the Frobenius norm of A
`flipud(A)`	reverses the order of rows of `A`
`fliplr(A)`	reverses the order of columns of `A`

Outline

- Basics
- Vectors
- Matrices
- Plotting

Plot commands

command	**meaning**
`plot(x,y)`	Plots the vector x versus the vector y
`semilogx(x,y)`	The x-axis is log10; the y-axis is linear
`semilogy(x,y)`	The x-axis is linear; the y-axis is log10.
`loglog(x,y)`	Creates a plot with log10 scales on both axes

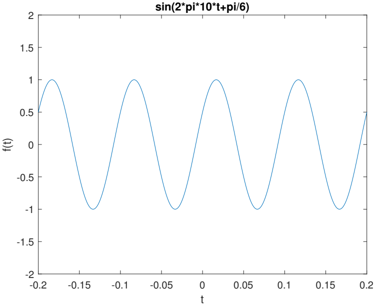
- there are also several other 2D graphical functions in MATLAB including

  `stem`, `bar`, `hist`, `polar`, `stairs`, ...

- `clf` command clears the current figure window
- `axis equal` command ensures that the scale used for the horizontal axis is equal to the scale used for the vertical axis
- we can add labels, change axis range, plot color,...etc

Plot command

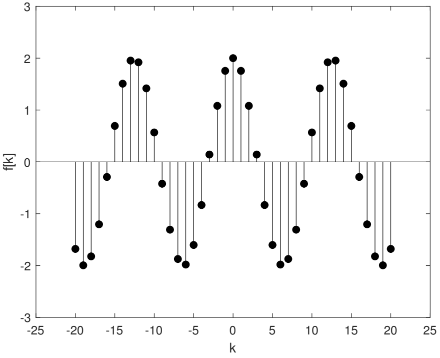
```
>> t = linspace(-0.2,0.2,500);
>> f = sin(2*pi*10*t+pi/6);
>> plot(t,f);
>> axis([-0.2 0.2 -2 2]) % plot range
>> xlabel('t'); ylabel('f(t)'); % label the x and y axis
>> title('sin(2*pi*10*t+pi/6)'); %label the title
```



Stem command

the stem command can be used to plot $f[k]$ against discrete k

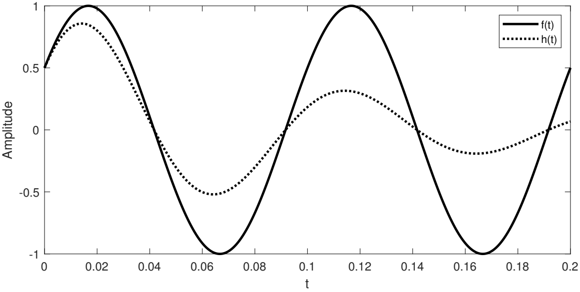
```
>> k = -20:20;
>> f = 2*cos(0.5*k);
>> stem(k,f,,k,'filled'); %'k' for black and filled circle
>> xlabel('k'); ylabel('f[k]');
>> axis([-25 25 -3 3])
```



Multiple curves

plot command can accommodates multiple curves

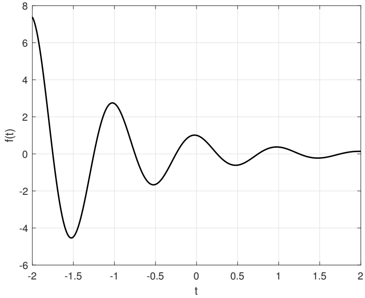
```
>> t = linspace(0,0.2,500);
>> f = sin(2*pi*10*t+pi/6);
>> g = exp(-10*t);
>> h = f.*g;
>> plot(t,f,'-k',t,h,':k','linewidth',2);
>> xlabel('t'); ylabel('Amplitude');
>> legend('f(t)','h(t)');
```



Plotting using anonymous functions

we can use anonymous functions for potting

```
>> f = @(t) exp(-t).*cos(2*pi*t);
>> t = (-2:0.01:2);
>> plot(t,f(t),'k','linewidth',1.4);
>> xlabel('t'); ylabel('f(t)');
>> grid; % adds grid lines
```



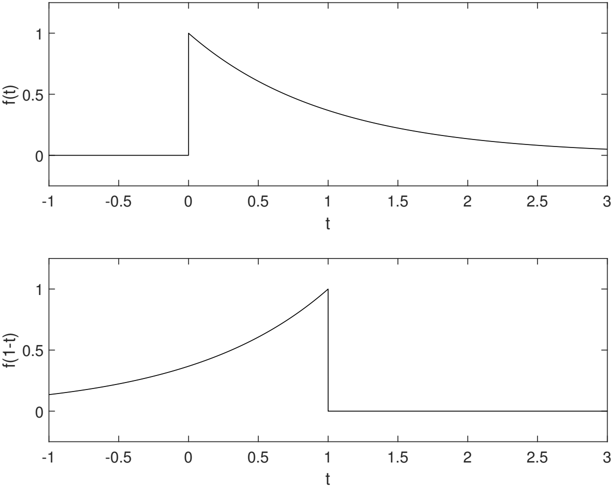
Example

plot $f(t) = e^{-t}u(t)$ over $-1 \le t \le 3$:

```
>> u = @(t) 1.0*(t>0);
>> f = @(t) exp(-t).*u(t);
>> t = -1:0.0001:3;
>> subplot(2,1,1) % create multiple graphs in one figure
>> plot(t,f(t),'k')
>> axis([-1 3 -0.25 1.25]);
>> xlabel('t'); ylabel('f(t)');
```

we can also evaluate a function by passing it an expression; this makes it very convenient to evaluate expressions such as $f(1-t)$, for example:

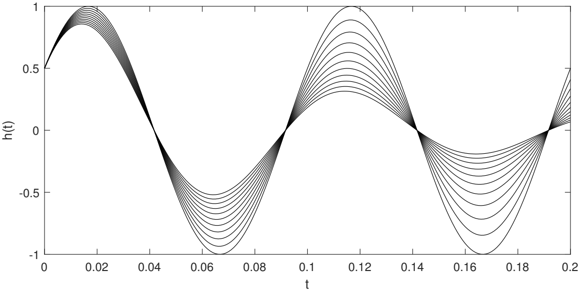
```
>> subplot(2,1,2)
>> plot(t,f(1-t),'k')
>> axis([-1 3 -0.25 1.25]);
>> xlabel('t'); ylabel('f(1-t)');
```


f(t)
1
0.5
0
-1
-0.5
0
0.5
1
1.5
2
2.5
3
t
f(1-t)
1
0.5
0
-1
-0.5
0
0.5
1
1.5
2
2.5
3
t

Family of curves

matrices can be used to create a family of curves

```
>> alpha = (0:10);
>> t = (0:0.001:0.2)'; %defined as column vector
>> T = repmat(t,1,11); %matrix T, columns t repeated 11 times
>> H = exp(-T*diag(alpha)).*sin(2*pi*10*T+pi/6);
>> plot(t,H,'k'); xlabel('t'); ylabel('h(t)');
```



$h_\alpha(t) = e^{-\alpha t}\sin(2\pi 10t + \pi/6)$ for $\alpha = [0, 1, \ldots, 10]$