Introduction to MATLAB

- getting started: basics
- arrays, vectors, matrices
- plotting
- M-files
- control flow

Outline

- getting started: basics
- arrays, vectors, matrices
- plotting
- M-files
- control flow

Basic commands

MATLAB is a computer program 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*3+4^2
ans =
24
```

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

```
>> ans/2
ans =
12
```

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

```
>> a=-3.15
a =
-3.1500
```

  creates a variable named “a” with value equal to -3.15 and displays result

Basic commands

- the priority order can be overridden with parentheses
- for example, because exponentiation has higher priority than negation:

  ```
  >> y = -4^2
  y =
  -16
  ```

  thus, 4 is first squared and then negated
- parentheses can be used to override the priorities as in

  ```
  >> y = (-4)^2
  y =
  16
  ```

- within each precedence level, operators have equal precedence and are evaluated from left to right; as an example,

  ```
  >> 4^2^3
  >> 4^(2^3)
  >> (4^2)^3
  ```

  - in the first case $4^2 = 16$ is evaluated first, which is then cubed to give 4096
  - second case $2^3 = 8$ is evaluated first and then $4^8 = 65,536$
  - third case is the same as the first, but uses parentheses to be clearer

Basic commands

- adding a semicolon (`;`) at end of command suppresses displaying result
- text after (%) on same line are treated as comments and ignored
```
>>  a=4; % create variable 'a' equal 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
```
- MATLAB is case-sensitive manner; variable `a` is not the same as `A`
- an integer after `1e` is used for powers of ten, *e.g.*, 10^2 can be found using
```
1e2 % same as 10^2
ans =
100
```
- MATLAB predefines the variables: `pi`$=\pi$; `Inf` $=\infty$; `NaN` means not a number
- MATLAB displays four decimal points; for additional precision, use `format long`; we can switch back using `format short`
- `format rational` switch to rational format of numbers; `rat(a)` return fraction approximation of `a`

Basic commands

- `clear` command deletes all objects (variables) 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
- 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
- long lines can be continued by placing an ellipsis (three consecutive periods):

  ```
  >> a=1,b=2;...
  c=3;

  a =
  1
  ```

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)`
$\mathrm{sign}(x)$	`sign(x)`

(for list of functions type `help elfun`)

Complex numbers

- unit imaginary number $\mathrm{j} = \mathrm{i} = \sqrt{-1}$ is preassigned to the variable `i` or `j`
- for example, we can create a complex number

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

Complex numbers built-in functions

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)` (or `x'`)	complex conjugate of `x`

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)
`rem(x,y)`	remainder after division
`mod(x,y)`	modulus (signed 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

- strings can be represented by using 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
```

- if you want to display strings in multiple lines, use `sprint` function and `\n`

```
>> disp(sprintf('Hello\nWorld!'))
```

- some useful commands

command	**meaning**
`str2num(s)`	converts string `s` to a number
`num2str(n)`	converts number `n` to a string
`b=strcmp(s1,s2)`	compares two strings, `s1` and `s2`; if equal returns true (`b = 1`); if not equal, returns false (`b = 0`)
`i=strfind(s1,s2)`	returns the starting indices of any occurrences of the string `s2` in the string `s1`

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**
& or &&	logical AND
| or ||	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))
  ```

- & and | always evaluate both sides
- short-circuit behavior:
  - && stops if the first condition is `false` and returns `false`
  - || stops if the first condition is `true` and returns `true`
  - useful for avoiding errors (*e.g.*, division by zero) or extra computations

Logical operator priority

- MATLAB evaluates logical operations according to a fixed priority order
- the priority from highest to lowest is `~` > `&` > `|`
- operators with the same priority are evaluated from left to right
- parentheses can always be used to change or override the natural priority

Example

```
-1 * 2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
```

the first thing that MATLAB does is to evaluate any mathematical expressions; in this example, there is only one: `-1 * 2`

```
-2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
```

next, evaluate all the relational expressions:

```
-2 > 0 & 2 == 2 & 1 > 7 | ~('b' > 'd')
F & T & F | ~F
```

at this point, MATLAB evaluates logical operators in priority order

since has the highest priority, the negation `F` is evaluated first:

```
F & T & F | T
```

next, the & operator is evaluated from left to right:

```
F & T   ->   F
F & F | T
```

the & again has higher priority than `|`:

```
F | T
```

finally, the `|` is evaluated as true

Anonymous function

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

```
fhandle = @(arglist) expression
```

- `fhandle`: function handle used to call the function
- `arglist`: comma-separated list of input arguments
- `expression`: any single valid MATLAB expression

Example 1: we can define $f(t) = e^{-t}\cos(2\pi t)$ as

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

$f(t)$ can be evaluated simply by passing the input values of interest

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

Example 2

```
>> f1 = @(x,y) x^2 + y^2;
>> f1(3,4)
ans =
25
```

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));
```

The `input` function

- `input` function allows you to prompt the user for values directly from the command window
- its syntax is

  ```
  n = input('promptstring')
  ```

- the function displays the promptstring, waits for keyboard input, and then returns the value from the keyboard
- for example,

  ```
  m = input('Mass (kg): ')
  ```

  when this line is executed, the user is prompted with the message

  ```
  Mass (kg):
  ```

  if the user enters a value, it would then be assigned to the variable `m`
- to input string, an `'s'` is appended to the function's argument list; for example,

  ```
  name = input('Enter your name: ','s')
  ```

The `disp` function

- the `disp` function displays a value or message to the command window
- syntax

```
disp(value)
```

- the argument can be a number variable or text string
- `disp` does not print the variable name only the value

Example

```
>> x = 5;
>> y = x^2;
>> disp('the value of y is:')
>> disp(y)
```

output

```
the value of y is:
25
```

The `fprintf` function

- `fprintf` provides detailed control over how information is displayed
- general syntax

```
fprintf('format', x, ...)
```

- the format string specifies how each variable is printed (integer decimal scientific etc)
- MATLAB scans the string from left to right printing characters until it encounters a percent sign % which indicates a format code
- after printing the numeric value MATLAB continues printing normal text until it encounters a backslash \ which introduces a control code such as \n (newline)

Example

```
velocity = 50.6175;
fprintf('The velocity is %8.4f m/s\n', velocity)
```

output

```
The velocity is 50.6175 m/s
```

Common `fprintf` format and control codes

format codes

`%d`	integer format
`%e`	scientific format with lowercase e
`%E`	scientific format with uppercase E
`%f`	decimal format
`%g`	compact form of `%e` or `%f`

control codes

`\n`	start new line
`\t`	tab

Example

```
>> fprintf('%5d %10.3f %8.5e\n',100,2*pi,pi);
100 6.283 3.14159e + 000
```

The `save` and `load` commands

- the `save` command stores variables in a mat–file
- syntax

```
save filename var1 var2 ...  varn
```

- this creates a file `filename.mat` containing the listed variables
- if no variables are specified the entire workspace is saved
- the `load` command retrieves variables from a mat–file

```
load filename var1 var2 ...  varn
```

- only listed variables are loaded; if none are listed all variables in file are loaded

Outline

- getting started: basics
- arrays, vectors, matrices
- plotting
- M-files
- control flow

Vectors

vector arrays x with entries $x_1, \ldots, x_n$ are created using square brackets

- we can create the row vector $x = [1\ \ 2\ \ 3]$ via spaces between entries:

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

- we can create the column vector $y = \begin{bmatrix} -1 + \mathrm{j}2 \\ 3.2 \\ 5 \end{bmatrix}$ via semicolon between entries

```
>> y=[-1+2i;3.2;5]
y =
-1.0000 + 2.0000i
3.2000 + 0.0000i
5.0000 + 0.0000i
```

  – Enter key (carriage return) can be used to separate entries

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

```
>> y'
ans =
-1.0000 - 2.0000i   3.2000 + 0.0000i   5.0000 + 0.0000i
```

Vector indexing

- the lth element of `x` can be extracted using `x(l)`, for example,

  ```
  x=[1 -1 2 4.5];
  x(3)
  ans =
  2
  ```

- we can use `x(l:k)` to extract a slice of a vector for element `l` to element `k`
- for example,

  ```
  x(2:3)
  ans =
  -1     2
  ```

- `end` command automatically references the final index of an array; for example,

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

- we can concatenate vectors to create a larger vector; for example,

  ```
  >> 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 `+,-`
- for example,

```
>> x=[1 2 3]+[4 5 6]
x =
5     7     9
```

- multiplying scalar by a vector is done using `*` command, *e.g.*,

```
>>-3*[1 2 3]
ans =
-3    -6    -9
```

- element-by-element operations are computed using (`.*, ./,.^`); example:

```
>> u=[1 2 3]; v=[-1 -2 -3];
>> w=u.*v
w =  -1    -4    -9
>> w.^2
ans =
1    16    81
```

Functions of vectors

- common built in functions operates elementwise on vectors
- 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 $0 \le t \le 2$ using

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

- 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)`

Useful vector commands

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

Example

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ -2 \\ 3 - \mathrm{j}2 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} -3 \\ 5 + \mathrm{j}7 \\ 6 \\ 2 \end{bmatrix}$$

use MATLAB to compute

(a) $x + y$

(b) inner product: $\langle x, y \rangle = x^* y = \sum_{k=1}^{4} x_k^* y_k$

(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
```

The `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
```

Matrices

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

- row entries are separated by spaces
- a semicolon is used to move to seperate rows
  - or the Enter key (carriage return) can be used to separate the rows
- for example, 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
```

- colon in place of a specific subscript represents the entire row or column

  for example, `A(2,:)` extracts 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
```

Matrix operations

- matrix addition and subtraction are carried out using the commands `+,-`
- matrix-vector and matrix-matrix multiplication is done using `*`, *e.g.*,

```
>> A=[1 2;3 4];
>> b=[1;2];
>> A*b
ans =
5
11
```

- 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`, if a solution exists, using

- backlash operator (left division): `x=A\b` (A can be square singular, tall, or wide)
- or `inv(A)*b` if A is nonsingular, which is less efficient than `x=A\b`

Useful matrix commands

command	**meaning**
`size(A)`	the size of the array `A`
`length(A)`	the length of the largest array dimension in `A`
`numel(A)`	number of elements in `A` (same as `prod(size(A))`)
`sum(A)`	a row vector containing the sum of each column
`sum(A,2)`	a column vector containing the sum of each row
`sum(A,"all")`	the sum of all elements of `A`
`prod(A)`	a row vector containing the sum of each column
`max(A)`/`min(A)`	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`
`flipud(A)`	reverses the order of rows of `A`
`fliplr(A)`	reverses the order of columns of `A`
`rand(dims...)`	array with random, iid, uniformly distributed over $[0, 1)$
`randn(dims...)`	array with random, iid, standard normally distributed
`randi(imax,dims...)`	returns random array of integers drawn from the discrete uniform distribution on the interval `[1,imax]`

Useful matrix commands

command	**meaning**
`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")`	the Frobenius norm of `A`
`sqrtm(A)`	square root of matrix $A^{1/2}A^{1/2} = A$

The `repmat` command

`repmat(A,m,n)` replicate objects; it returns an array containing `n` copies of `A` in the row and column dimensions

for example

```
>> A = diag([1 ;2])
A =
1     0
0     2

>> repmat(A,3,2)
ans =

1     0     1     0
0     2     0     2
1     0     1     0
0     2     0     2
1     0     1     0
0     2     0     2
```

creates a 3×2 block matrix with blocks `A`

The `reshape` command

- `B=reshape(A,sz)` reshapes A using the size vector, `sz`
- for example, `reshape(A,[2,3])` reshapes `A` into a 2-by-3 matrix
- `sz` must contain at least 2 elements, and `prod(sz)` must be same as `numel(A)`

Example

```
A = 1:10;
B = reshape(A,[5,2])
B =
1     6
2     7
3     8
4     9
5    10
```

Multidimensional array

- you can create a multidimensional array by creating matrix, then extend it
- for example, first define a 3-by-3 matrix as the first page in a 3-D array:

```
A = [1 2 3; 4 5 6; 7 8 9]
A =
1     2     3
4     5     6
7     8     9
```

- now add a second page; to do this, assign another 3-by-3 matrix to the index value 2 in the third dimension:

```
A(:,:,2) = [10 11 12; 13 14 15; 16 17 18]
A =
A(:,:,1) =
1     2     3
4     5     6
7     8     9

A(:,:,2) =
10    11    12
13    14    15
16    17    18
```

Elementwise logical operations

- use `&` and `|` when performing *elementwise* logical comparisons on arrays:

```
>> x = -3:3;
>> mask = (x>0) & (x<3);   % element-wise AND
mask =
1×7 logical array
0   0   0   0   1   1   0
```

- use `&&` and `||` when combining *scalar logical expressions*:

```
>> t = 2;
>> condition = (t>0) && (t<5);   % scalar short-circuit AND
condition =
1
```

- `&` and `|` work on arrays (element-by-element)
- `&&` and `||` only work for single (scalar) logical values

Outline

- getting started: basics
- arrays, vectors, matrices
- plotting
- M-files
- control flow

Plotting commands

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

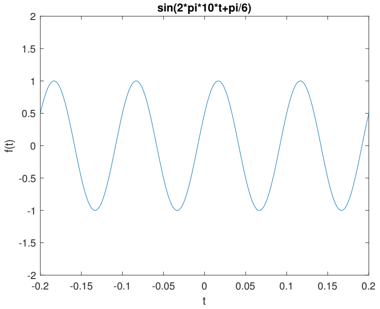
- there are also several other 2D graphical functions in MATLAB including

  `stem`, `bar`, `hist`, `polar`, `stairs`, `plot3`...

- `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

Example: `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
```



Specifiers for colors, symbols, and line types

Colors		Symbols		Line Types	
Blue	`b`	Point	`.`	Solid	`-`
Green	`g`	Circle	`o`	Dotted	`:`
Red	`r`	X-mark	`x`	Dashdot	`-.`
Cyan	`c`	Plus	`+`	Dashed	`--`
Magenta	`m`	Star	`*`		
Yellow	`y`	Square	`s`		
Black	`k`	Diamond	`d`		
White	`w`	Triangle (down)	`v`		
		Triangle (up)	∧		
		Triangle (left)	<		
		Triangle (right)	>		
		Pentagram	`p`		
		Hexagram	`h`		

for example

```
>> plot(t, f, ’s--g’)% green square markers connected by dashed lines
```

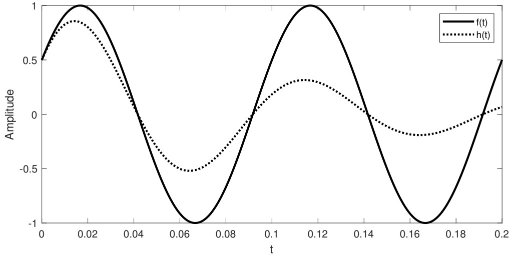
Multiple curves

MATLAB allows you to display more than one data set on same plot; *e.g.*,

```
>> plot(x, y1, x, y2)
```

Example

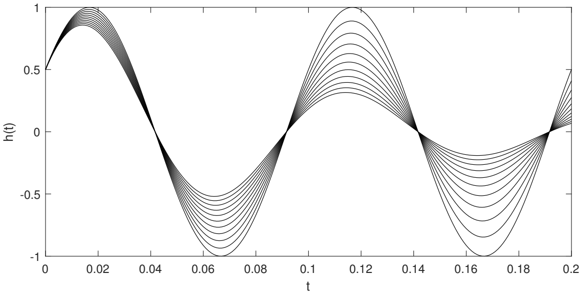
```
>> t = linspace(0,0.2,500);
>> f = sin(2*pi*10*t+pi/6);
>> g = exp(-10*t);
>> h = f.*g;
>> plot(t,f,'-k',t,h,':k','linewidth',2);
>> xlabel('t'); ylabel('Amplitude');
>> legend('f(t)','h(t)');
```



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)');
```

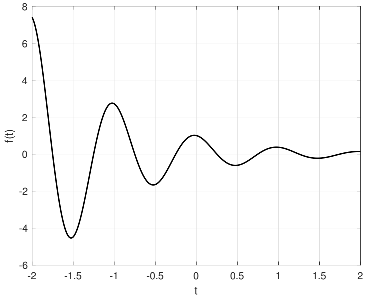


$$h_\alpha(t) = e^{-\alpha t} \sin(2\pi 10t + \pi/6) \text{ for } \alpha = [0, 1, \ldots, 10]$$

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
```



The `hold on` and `subplot` commands

- by default, previous plots are erased when new `plot` command is implemented
- `hold on` command holds the current plot and all axis properties so that additional graphing commands can be added to the existing plot
- for example, the following commands would result in both lines and symbols being displayed:

```
>> plot(t, f1)
>> hold on % keep plot
>> plot(t, f2, ':') %plot f2 on same figure
>> hold off
```

- `subplot` allows you to split the graph window into subwindows; it has syntax

```
subplot(m, n, p)
```

  this command breaks the graph window into an m-by-n matrix of small axes, and selects the p-th axes for the current plot
  - MATLAB numbers subplot positions by row
  - first subplot is the first column of the first row, the second subplot is the second column of the first row, and so on

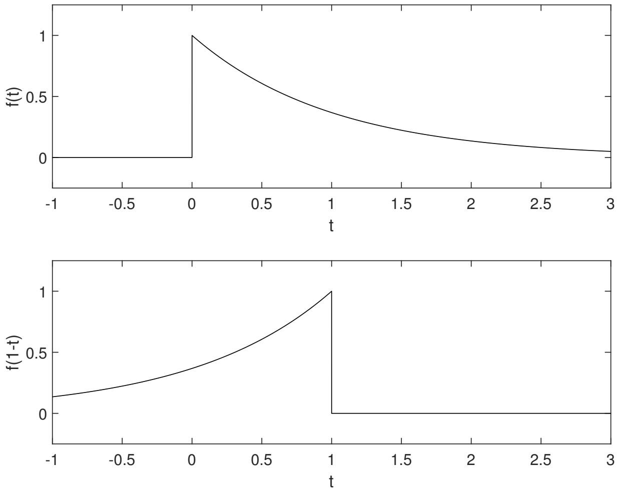
Example

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

```
>> 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 $f(1-t)$:

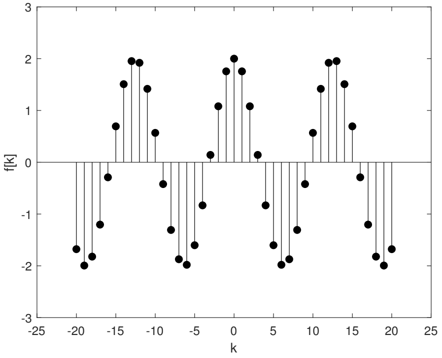
```
>> subplot(2,1,2)
>> plot(t,f(1-t),'k')
>> axis([-1 3 -0.25 1.25]);
>> xlabel('t'); ylabel('f(1-t)');
```


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

the `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])
```



Outline

- getting started: basics
- arrays, vectors, matrices
- plotting
- M-files
- control flow

M-files

Script files

- *script M-files* file is a series of commands saved on a file that can be run at once
- script can be executed by typing the file name in the command window or by pressing the Run button

Function files

- *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 syntax is

```
function outvar = funcname(arglist)
% helpcomments
statements
outvar = value;
```

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

Example

a bungee jumper velocity with mass m and drag coefficient c_d can be described as:

$$\frac{dv}{dt} = g - \frac{c_d}{m}\, v^2 \quad \Longrightarrow \quad v(t) = \sqrt{\frac{gm}{c_d}}\, \tanh\left(\sqrt{\frac{gc_d}{m}}\, t\right)$$

where $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is the hyperbolic tangent and g gravity acceleration

M-file content

```
function v = freefall(t, m, cd)
% freefall: bungee velocity with second-order drag
% v=freefall(t,m,cd) computes the free-fall velocity
% of an object with second-order drag
% input:
% t = time (s)
% m = mass (kg)
% cd = second-order drag coefficient (kg/m)
% output:
% v = downward velocity (m/s)
g = 9.81; % acceleration of gravity
v = sqrt(g * m / cd)*tanh(sqrt(g * cd / m) * t);
```

Example

Execute function M-file

to invoke the function, return to the command window and type in

```
>> freefall(12,68.1,0.25)
```

the result will be displayed as

```
ans =
50.6175
```

- can be invoked repeatedly for different argument values
- note that, at the end of the previous example, if we had typed

  ```
  >> g
  ```

  the following message would have been displayed:

  ```
  ??? Undefined function or variable 'g'.
  ```

- variables within function are *local* and are erased after function is executed

Example

- function M-files can return more than one result
- in such cases, the variables containing the results are comma-delimited and enclosed in brackets

Example: `stats.m`, computes mean and standard deviation of a vector:

```
function [mean, stdev] = stats(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/(n - 1)));
```

here is an example of how it can be applied:

```
>> y = [8 5 10 12 6 7.5 4];
>> [m,s] = stats(y)
m =
7.5000
s =
2.8137
```

Variable scope

- variable's scope is limited either to the MATLAB workspace or within a function
- this principle prevents errors when a programmer unintentionally gives the same name to variables in different contexts
- any variables defined through the command line are within the MATLAB workspace
- however, workspace variables are not directly accessible to functions but rather are passed to functions via their arguments

Example

```
function c = adder(a,b)
x = 88
a
c = a + b
```

if we type

```
>> x = 1; y = 4; c = 8;
>> d = adder(x,y)
x =
88
a =
1
c =
5
d =
5
```

but, if you then type

```
>> c, x, a
c =
8
x =
1
Undefined function or variable ’a’.
Error in ScopeScript (line 6)
c, x, a
```

Global variables

global variables can be defined as in `global X Y Z` and can be accessed in several contexts without passing it as an argument

- if several functions (or workspace), all declare a particular name as global
- then they all share a single value of that variable
- any change to that variable, in any function, is then made to all the other functions that declare it global

Example

```
function C=add(B)
global A
C=A+B;
end
>> global A
>> A=2.4;
>> B=1.2;
>> C=add(B)
C =
3.6000
```

Subfunctions

- functions can call other functions;
- such functions can exist as separate M-files or may also be contained in a single M-file

Example: consider writing the function in example 49 in an M-file is given as

```
function v = freefallsubfunc(t, m, cd)
v = vel(t, m, cd);
end
function v = vel(t, m, cd)
g = 9.81;
v = sqrt(g * m / cd)*tanh(sqrt(g * cd / m) * t);
end
```

then

```
>> freefallsubfunc(12,68.1,0.25)
ans =
50.6175
```

however, if we attempt to run the subfunction vel, an error message occurs:

```
>> vel(12,68.1,.25)
??? Undefined function or method 'vel' for input arguments of type 'double'.
```

Passing functions to M-files

anonymous functions can be passed into function M-files

example:

```
function favg = funcavg (f,a,b,n)
% funcavg: average function height
% favg = funcavg(f,a,b,n): computes average value
% of function over a range
% input:
% f = function to be evaluated
% a = lower bound of range
% b = upper bound of range
% n = number of intervals
% output:
% favg = average value of function
x = linspace(a,b,n);
y = f(x);
favg = mean(y);
```

```
>> vel = @(t) sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t);
>> funcavg(vel,0,12,60)
ans =
36.0127
```

Outline

- getting started: basics
- arrays, vectors, matrices
- plotting
- M-files
- control flow

If statements

if statements execute commands if a certain condition is met

`if` structure

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

`if...else` structure

```
if condition
statements1
else
statements2
end
```

`if...elseif` structure

```
if condition1
statements1
elseif condition2
statements2
elseif condition3
statements3
.
.
.
end
```

Example: set $x = 5$ if $a > 0$ and $x = 100$ otherwise

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

Example

construct sign function:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
% -1 if x is less than zero.
% 0 if x is equal to zero.
if x > 0
sgn = 1;
elseif x < 0
sgn = -1;
else
sgn = 0;
end
```

```
>> mysign(0)
ans =
0
```

this does the same as built-in function `sign` in MATLAB

The `switch` structure

General syntax

```
switch testexpression
case value1
statements1
case value2
statements2
.
.
.
otherwise
statementsotherwise
end
```

Example

```
grade = 'B';
switch grade
case 'A'
disp('Excellent')
case 'B'
disp('Good')
case 'C'
disp('Mediocre')
case 'D'
disp('Whoops')
case 'F'
disp('Fail')
otherwise
disp('Huh!')
end
```

Variable argument lists and `nargin`

- MATLAB functions can receive a variable number of input arguments
- the function `nargin` returns how many arguments the user supplied
- typical pattern (useful for creating functions with default values when inputs are omitted)

```
function y = myfun(a,b,c)
if nargin < 3
c = 10;       % default value
end
if nargin < 2
b = 5;        % default value
end
if nargin == 0
a = 2;        % default value
end
y = a + b + c;
end
```

- this allows flexible function calls: `myfun(2)`, `myfun(2,7)`, or `myfun(2,7,9)`
  - in the command window, `nargin` behaves a little differently
  - it must include a string argument specifying function and it returns no. of arguments
  - for example, `>> nargin('myfun')` return `ans = 3`

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
```

Preallocation of memory

- MATLAB automatically resizes arrays when new elements are added
- resizing inside loops is slow and inefficient
- example of inefficient code that grows y one element at a time

```
t = 0:.01:5;
for i = 1:length(t)
if t(i) > 1
y(i) = 1/t(i);
else
y(i) = 1;
end
end
```

- better approach: preallocate memory before the loop

```
t = 0:.01:5;
y = ones(size(t));          % preallocate
for i = 1:length(t)
if t(i) > 1
y(i) = 1/t(i);
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
```

The `while...break` structure

- sometimes it is useful to terminate a loop in middle based on true condition
- MATLAB does not have a built-in mid-loop test structure, but the behavior can be mimicked using `while(1)` with `break`

```
while (1)
statements
if condition, break, end
statements
end
```

- `break` immediately terminates the loop when the condition becomes true
- placing `break` in the middle creates a midtest loop
- pretest version example

```
while (1)
if x < 0, break, end
x = x - 5;
end
```

- here 5 is subtracted from `x` each iteration so the loop eventually terminates
- every loop must have a termination mechanism, otherwise an infinite loop results

The `continue` commands

`continue` commands jumps to the loop's `end` statement, but then starts the next iteration of the loop

Example: using `continue` to display multiples of 17

```
for i = 1:100
if mod(i,17) ~= 0
continue
end
disp([num2str(i) ' is evenly divisible by 17'])
end
```

- `mod(x,n)` returns the remainder when `x` is divided by `n`
- if the remainder is not zero, `continue` skips to the next iteration
- if the remainder is zero, the number is displayed
- output is

```
17 is evenly divisible by 17
34 is evenly divisible by 17
51 is evenly divisible by 17
68 is evenly divisible by 17
85 is evenly divisible by 17
```

The `pause` command

- the `pause` command temporarily halts program execution
- execution resumes when the user presses any key
- useful when displaying a sequence of plots that user should view one at a time

Example: viewing a sequence of plots

```
for n = 3:10
mesh(magic(n))
pause
end
```

- each `mesh` plot is displayed
- `pause` stops the program and waits for a key press
- after the user presses a key, the loop continues to the next plot

Other useful MATLAB functions

- `beep` causes the computer to emit a sound
- `tic` and `toc` measure elapsed time
- `tic` stores the current time; `toc` displays the time since the most recent `tic`

Example: testing `pause(n)` with sound and timing

```
tic
beep
pause(5)
beep
toc
```

- 1st `beep` sounds; pauses for 5 seconds; 2nd `beep` sounds after the pause
- `toc` prints elapsed time, for example:

```
Elapsed time is 5.006306 seconds
```

- using `pause(inf)` creates an infinite pause
- return to the command prompt with `Ctrl + C` or `Ctrl + Break`