Rutgers University
School of Engineering

Fall 2011

14:440:127 - Introduction to Computers for Engineers

Sophocles J. Orfanidis
ECE Department
orfanidi@ece.rutgers.edu

week 2

Weekly Topics

Week 1 - Basics – variables, arrays, matrices, plotting (ch. 2 & 3)
Week 2 - Basics – operators, functions, program flow (ch. 2 & 3)
Week 3 - Matrices (ch. 4)
Week 4 - Plotting – 2D and 3D plots (ch. 5)
Week 5 - User-defined functions (ch. 6)
Week 6 - Input-output formatting – fprintf, sprintf (ch. 7)
Week 7 - Program flow control & relational operators (ch. 8)
Week 8 - Matrix algebra – solving linear equations (ch. 9)
Week 9 - Structures & cell arrays (ch. 10)
Week 10 - Symbolic math (ch. 11)
Week 11 - Numerical methods – data fitting (ch. 12)
Week 12 – Selected topics

Textbook: H. Moore, *MATLAB for Engineers*, 2nd ed., Prentice Hall, 2009

MATLAB Basics

1. MATLAB desktop
2. MATLAB editor
3. Getting help
4. Variables, built-in constants, keywords
5. Numbers and formats
6. Arrays and matrices
7. Operators and expressions
8. Functions
9. Basic plotting
10. Function maxima and minima
11. Relational and logical operators
12. Program flow control
13. Matrix algebra and linear equations

week 1

week 2

These should be enough to get you started. We will explore them further, as well as other topics, in the rest of the course.

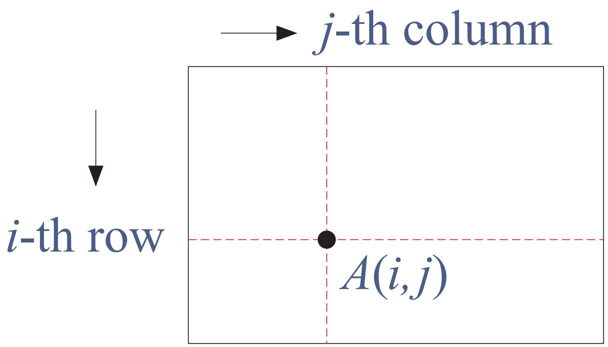
6. Arrays and Matrices

arrays and matrices are the most important data objects in MATLAB

Last week we discussed one-dimensional arrays, i.e., column or row vectors.

Next, we discuss matrices, which are two-dimensional arrays. We will explore them further in Chapters 4 & 9.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$



matrix indexing convention

```
>> A = [1 2 3; 2 0 4; 0 8 5]
A =
     1     2     3
     2     0     4
     0     8     5

>> size(A)          % [N,M] = size(A), NxM matrix
ans =
     3     3
```

accessing matrix elements:

```
>> A(1,1)        % 11 matrix element
ans =
     1
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

```
>> A(2,3)        % 23 matrix element
ans =
     4
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

```
>> A(:,2)        % second column
ans =
     2
     0
     8
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

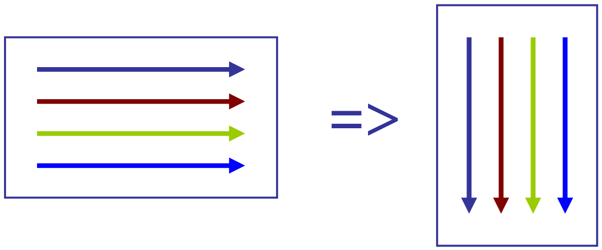
```
>> A(3,:)         % third row
ans =
     0     8     5
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 0 & 8 & 5 \end{bmatrix}$$

transposing a matrix:
rows become columns and vice versa

```
>> A = [1 2 3 4; 2 0 5 6; 0 8 7 9]    % size 3x4
A =
     1     2     3     4
     2     0     5     6
     0     8     7     9

>> A'                                  % size 4x3
ans =
     1     2     0
     2     0     8
     3     5     7
     4     6     9
```



transposition operation

For more information on elementary matrices see:

```
>> help elmat

  Elementary matrices and matrix manipulation.

  Elementary matrices.
    zeros       - Zeros array.
    ones        - Ones array.
    eye         - Identity matrix.
    repmat      - Replicate and tile array.
    linspace    - Linearly spaced vector.
    logspace    - Logarithmically spaced vector.

    etc.
```

7. Operators and Expressions

operation	element-wise	matrix-wise
addition	`+`	`+`
subtraction	`-`	`-`
multiplication	`.*`	`*`
division	`./`	`/`
left division	`.\`	`\`
exponentiation	`.^`	`^`
transpose w/o complex conjugation		`.'`
transpose with complex conjugation		`'`

```
>> help /
>> help precedence
```

```
>> a = [1  2 5];
>> b = [4 -5 1];

>> a+b
ans =
     5    -3      6

>> a.*b
ans =
     4   -10      5

>> a./b
ans =
    0.2500   -0.4000    5.0000

>> a.\b
ans =
    4.0000   -2.5000    0.2000

% note: (a./b).*(a.\b) = [1,1,1]
```

```
>> a = [2 3 4 5];

>> a.^2             % [2^2, 3^2, 4^2, 5^2]
ans =
     4     9    16    25

>> 2.^a             % [2^2, 2^3, 2^4, 2^5]
ans =
     4     8    16    32

>> a+10
ans =
     12    13    14    15
```

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> [A, A.^2; A^2, A*A]                 % form sub-blocks
ans =
     1     2     1     4
     3     4     9    16
     7    10     7    10               % note A^2 = A*A
    15    22    15    22

>> B = 10.^A;
>> [B, log10(B)]
ans =
          10         100           1           2
        1000       10000           3           4
```

$$B = \begin{bmatrix} 10^1 & 10^2 \\ 10^3 & 10^4 \end{bmatrix}$$

8. Functions

```
>> help elfun     % elementary functions list
```

Some typical built-in elementary functions are:

```
 sin(x),  cos(x),  tan(x),  cot(x)
asin(x), acos(x), atan(x), acot(x)

 sinh(x),  cosh(x),  tanh(x),  coth(x)
asinh(x), acosh(x), atanh(x), acoth(x)

exp(x), log(x), log10(x), log2(x)

fix(x), floor(x), ceil(x), round(x)

sqrt(x), sign(x), abs(x)

sum(x), prod(x), cumsum(x), cumprod(x)
```

Some more functions:

```
size(x), length(x), class(x)

sinc(x)                          % sin(pi*x)/(pi*x)

max(x), min(x), sort(x)

mean(x), std(x),                 % statistics
median(x), mode(x)

rand, randn,               % random number generators
randi, rng                 % initialize with rng

filter, conv, fft          % DSP functions

clock, date

factorial(n), nchoose(n,k)     % discrete math
```

for a complete list, see Appendix A of your text

Most functions admit scalar or array and matrix input arguments and operate on each element of the array

$$\mathbf{x} = [x_1, x_2, x_3, \ldots]$$

$$f(\mathbf{x}) = [f(x_1), f(x_2), f(x_3), \ldots]$$

```
>> x = [0, pi/4, pi/3, pi/2, pi];

>> sin(x)
ans =
      0    0.7071    0.8660    1.0000    0.0000

>> sin(sym(x))              % use symbolic toolbox
                            % to see exact expressions
ans =
    [ 0,  2^(1/2)/2, 3^(1/2)/2,  1,  0]
```

```
>> x = [2.1, 2.8, -3.1, -3.5, 4.5];

>> y = exp(x)
y =
    8.1662   16.4446    0.0450    0.0302   90.0171

>> z = log(y)          % note log(exp(x)) = x
z =
    2.1000    2.8000   -3.1000   -3.5000    4.5000

>> [fix(x); floor(x); ceil(x); round(x)]
ans =
     2     2    -3    -3     4
     2     2    -4    -4     4
     3     3    -3    -3     5
     2     3    -3    -4     5
```

Example: verify the following geometric-series identity using the function **sum(x)**,

$$\frac{1}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}+\cdots+\frac{1}{2^N}=1-\frac{1}{2^N}$$

summation notation

$$\sum_{n=1}^{N}\frac{1}{2^n}=1-\frac{1}{2^N}$$

```
>> format long;
>> N = 8; n = 1:N;        % n = [1, 2, ..., 8 ]
                          % [1/2^1, 1/2^2, ..., 1/2^8]
>> sum(1./2.^n)           % note the operations ./ and .^
>> 1 - 2^(-N)
ans =
      0.99609375000000
ans =
      0.99609375000000
```

`y = cumsum(x)` – cumulative sum of the elements of x

$$y(1) = x(1)$$

$$y(2) = x(1) + x(2)$$

$$y(3) = x(1) + x(2) + x(3)$$

$$\cdots$$

$$y(n) = \sum_{i=1}^{n} x(i) = x(1) + x(2) + \cdots + x(n)$$

cumsum example:

```
>> N = 8; n = 1:N;              % n is a row vector

>> y = cumsum(1./2.^n);         % y,z should be equal
>> z = 1 - 1./2.^n;

>> fprintf('%d   %1.8f   %1.8f\n',[n; y; z]);

1    0.50000000   0.50000000
2    0.75000000   0.75000000
3    0.87500000   0.87500000
4    0.93750000   0.93750000
5    0.96875000   0.96875000
6    0.98437500   0.98437500
7    0.99218750   0.99218750
8    0.99609375   0.99609375
```

fprintf operates column-wise on the 3x8 matrix **[n; y; z]**, i.e.,

$$\begin{bmatrix} n_1 & n_2 & n_3 & \cdots \\ y_1 & y_2 & y_3 & \cdots \\ z_1 & z_2 & z_3 & \cdots \end{bmatrix}$$

```
>> seed = 127; rng(seed);
>> x = randn(5,3)
x =
    0.0294   -1.0928    1.6686
   -1.5732   -0.1697   -0.4750
   -1.1899    0.5751   -0.7604
    1.8115    0.6548   -1.1189
    0.0426   -0.0969    0.1698

>> min(x), max(x), mean(x), std(x)
ans =
   -1.5732   -1.0928   -1.1189
ans =
    1.8115    0.6548    1.6686
ans =
   -0.1759   -0.0259   -0.1032
ans =
    1.3248    0.7051    1.0972
```

initialize generator,
5x3 matrix of zero-mean,
unit-variance, gaussian,
random numbers

```
>> help rng
>> help rand
>> help randn
>> help randi
```

computed column-wise

MATLAB is
column-dominant

```
x =
    0.0294   -1.0928    1.6686
   -1.5732   -0.1697   -0.4750
   -1.1899    0.5751   -0.7604
    1.8115    0.6548   -1.1189
    0.0426   -0.0969    0.1698
```

i=2

min,max,sort

act column-wise
on matrix inputs

```
>> [m,i] = min(x), min(min(x))
m =
   -1.5732   -1.0928   -1.1189
i =
     2     1     4
ans =
   -1.5732
```

minimum of each column,
index within each column,
overall minimum

```
>> sort(x)
ans =
   -1.5732   -1.0928   -1.1189
   -1.1899   -0.1697   -0.7604
    0.0294   -0.0969   -0.4750
    0.0426    0.5751    0.1698
    1.8115    0.6548    1.6686
```

sort each column in
ascending order

sort(x,'ascend')
sort(x,'descend')

Make up your own functions using three methods:

1. function-handle, @(x)
2. inline
3. M-file

example: $$f(x) = e^{-0.5x}\sin(5x)$$

```
>> f = @(x) exp(-0.5*x).*sin(5*x);

>> g = inline('exp(-0.5*x).*sin(5*x)');

% edit & save file h.m containing the lines:
function y = h(x)
y = exp(-0.5*x).*sin(5*x);
```

.* allows vector or matrix inputs x

How to include parameters in functions

example: $f(x) = e^{-ax}\sin(bx)$

```
% method 1: define a,b first, then define f

a = 0.5; b = 5;
f = @(x) exp(-a*x).*sin(b*x);


% method 2: pass parameters as arguments to f

f = @(x,a,b) exp(-a*x).*sin(b*x);

% this defines the function f(x,a,b)
% so that f(x, 0.5, 5) would be equivalent to
% the f(x) defined in method 1.
```

9. Basic Plotting

MATLAB has extensive facilities for the plotting of curves and surfaces, and visualization. We will be discussing these in detail later on.

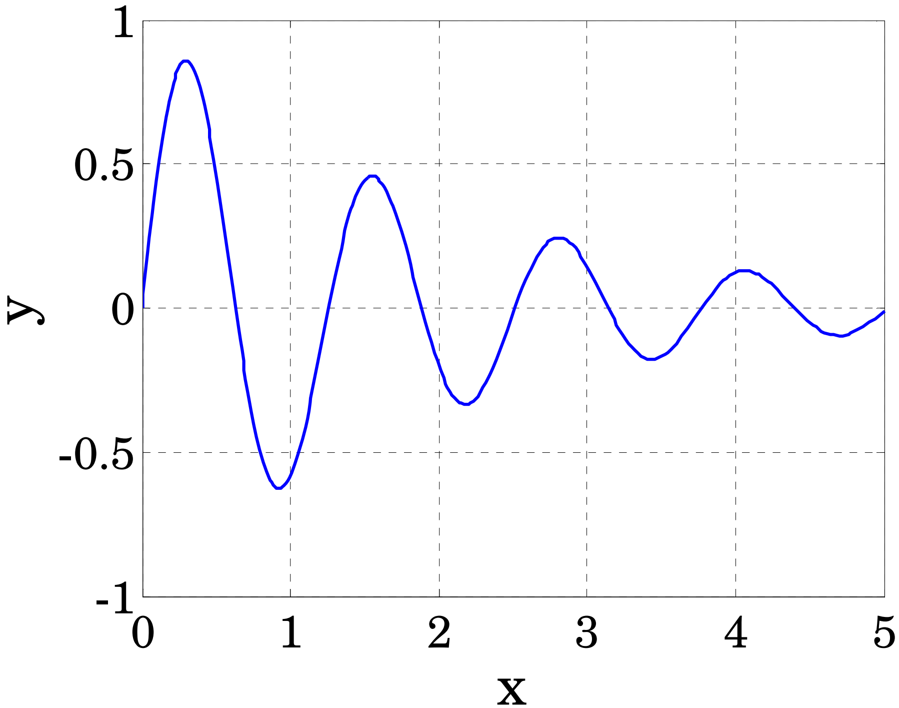
Basic 2D plots of functions and (x,y) pairs can be done with the functions:

`plot,  fplot,  ezplot`

```
>> help plot        % 2-D plotting
>> help fplot       % function plotting
>> help ezplot      % easy function plotting
```

If a function f(x) has already been defined by a function-handle or inline, it can be plotted quickly with **fplot**, **ezplot**, which are very similar. One only needs to specify the plot range. For example:

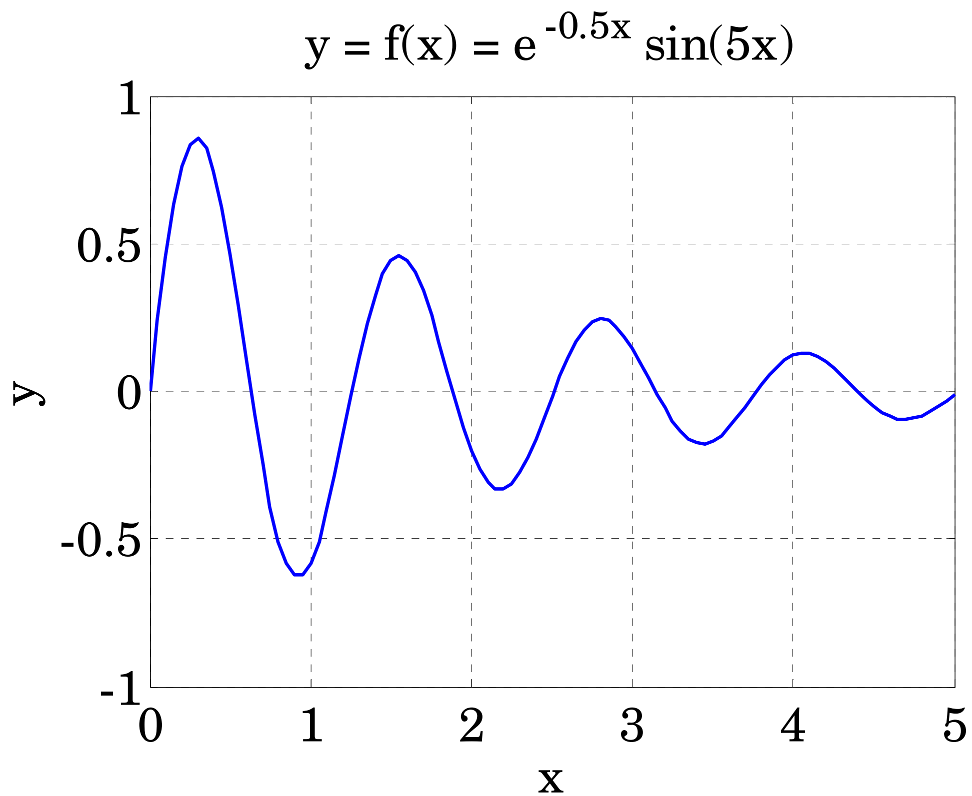
```
>> f = @(x) exp(-0.5*x).*sin(5*x);
>> fplot(f,[0,5]);          % plot over interval [0,5]
```



A figure window opens up, allowing further editing of the graph, e.g., adding x,y axis labels, titles, grid, changing colors, and saving the graph is some format, such as WMF, PNG, or EPS.

using the plot function

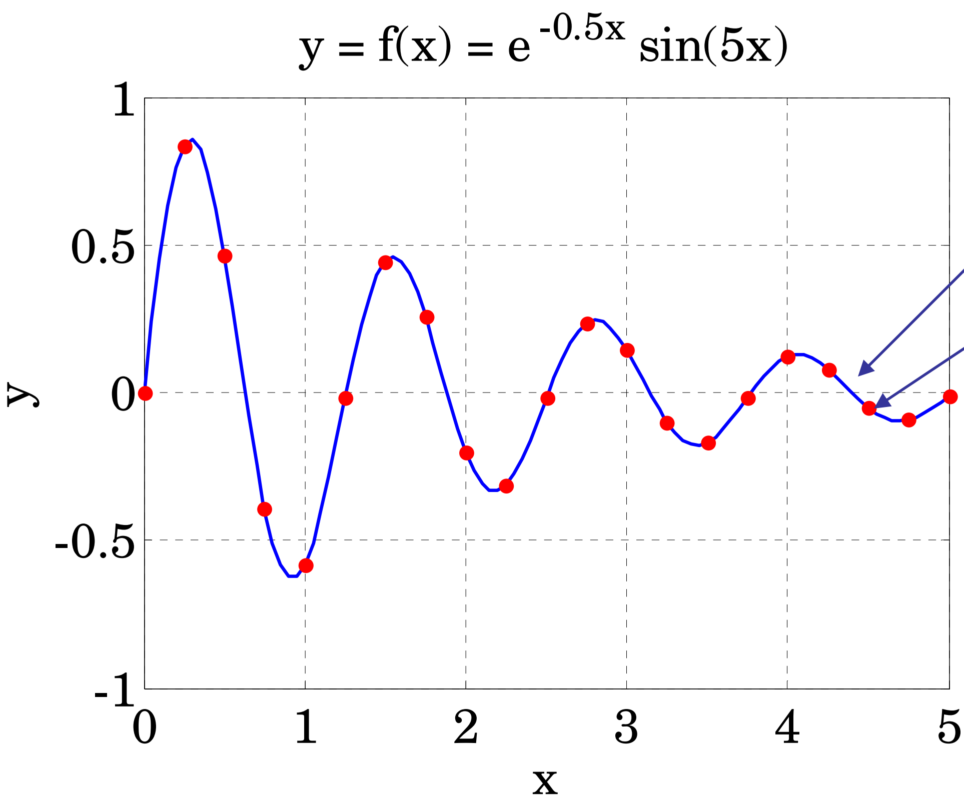
```
>> x = linspace(0,5,101);
>> y = f(x);
>> plot(x,y,'b-');                          % blue-solid line
>> xlabel('x'); ylabel('y'); grid;
>> title('f(x) = e^{-0.5x} sin(5x)');
```



plot annotation can be done by separate commands, as shown above, or from the plot editor in the figure window.

multiple graphs on same plot

```
>> x5 = x(1:5:end);              % plot every 5th data point
>> y5 = y(1:5:end);
>> plot(x,y,'b-', x5,y5, 'r.');    % blue-line, red dots
>> xlabel('x'); ylabel('y'); grid;
>> title('f(x) = e^{-0.5x} sin(5x)');
```



(x,y) plotted as blue-solid line

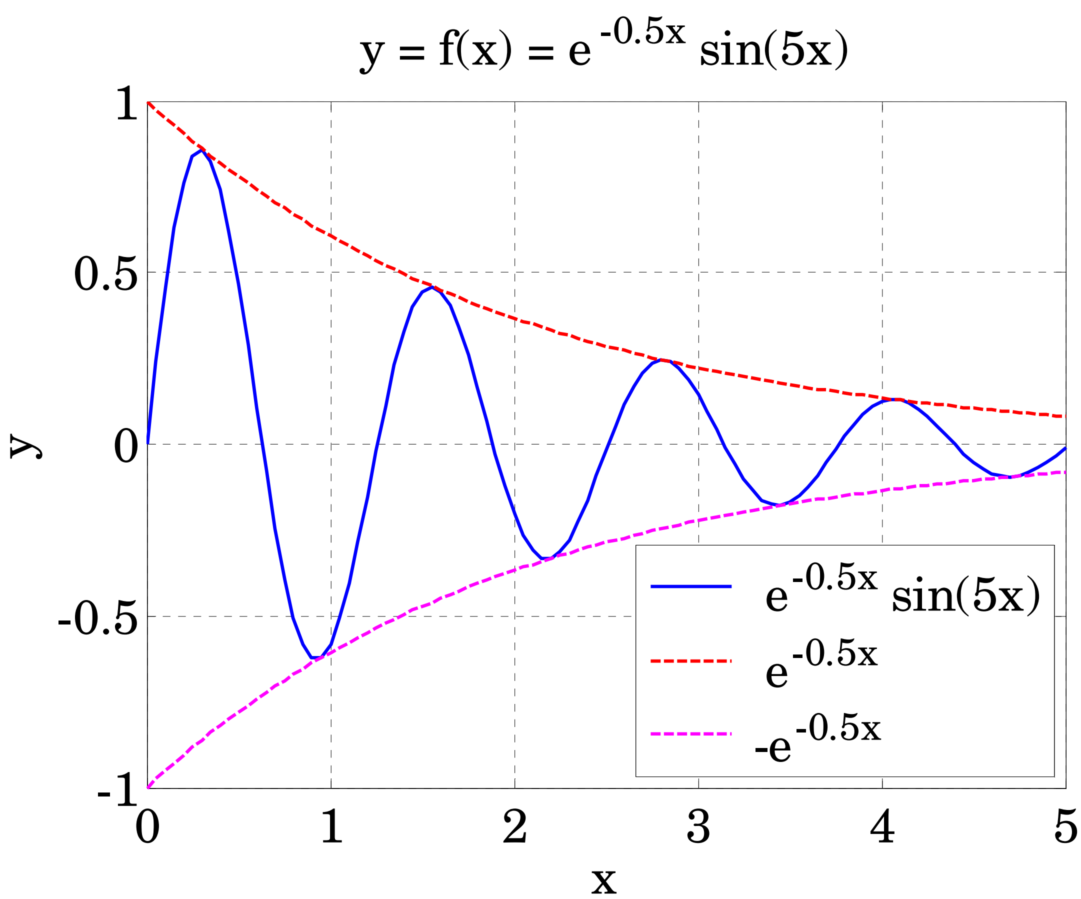
(x5,y5) pairs plotted as red dots

multiple (x,y) pairs---not necessarily of the same size---can be plotted with different line styles.

```
>> e = exp(-0.5*x);                          % envelope of f(x)
>> plot(x,y,'b-', x,e,'r--', x,-e,'m--');
>> xlabel('x'); ylabel('y'); grid;
>> title('f(x) = e^{-0.5x} sin(5x)');
>> legend('e^{-0.5x} sin(5x)', 'e^{-0.5x}', ...
   '-e^{-0.5x}', 'location','SE');
```

south-east

ellipsis continues to next line



plotting multiple curves and adding legends

legends can also be inserted with plot editor

10. Function Maxima and Minima

Engineers always like to optimize their designs by finding the best possible solutions. This usually amounts to minimizing or maximizing some function of the design parameters.

Suppose a function f(x) has a minimum (or maximum) within an interval [a,b], or, $a \le x \le b$. The following three methods can be used to find it:

1. Graphical method using the function **`min`** (or **`max`**)
2. Using the built-in function **`fminbnd`**
3. Using the function **`fzero`**, (requires the derivative of f(x))

(use **`fminsearch`** for multivariable functions)

MATLAB implementation of the three methods

```
f = @(x) ...         % define your function here
                     % f(x) must admit vector inputs
                     % and return vector outputs
```

1.

```
x = linspace(a,b,N);          % larger N works better
[fmin,imin] = min(f(x));      % imin = index at min
xmin = x(imin);               % where the minimum is
plot(x,f(x), xmin,fmin,'o');          % display it
```

2.

```
[xmin,fmin] = fminbnd(f,a,b);   % search in [a,b]
```

```
F = @(x) ...              % define derivative of f(x)
                          % or use symbolic toolbox
```

3.

```
xmin = fzero(F,x0);       % search near x0
fmin = f(xmin);           % minimum value of f(x)
```

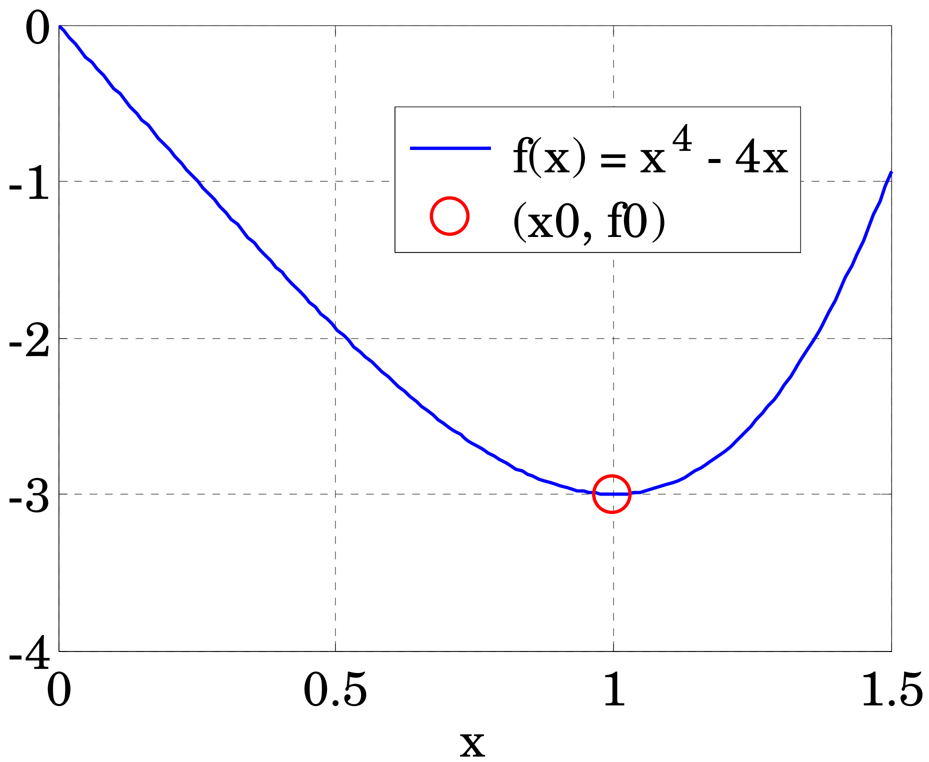
```
f = @(x) x.^4 - 4*x;

x = linspace(0, 1.5, 150);

[f0,i0] = min(f(x)); x0 = x(i0);

plot(x,f(x),'b-', x0,f0,'ro');
xlabel('x'); grid;
legend('f(x)=x^4-4x', '(x0,f0)');
```

Example: finding the minimum of a curve using the function **min**



f0 is minimum of the array **y=f(x)**

i0 is the index of array at its min, i.e., **f0=y(i0)**

x0 is value of **x** at the minimum of **y**

exact values are:

x0 = 1
f0 = -3

finding the minimum of f(x) using the function **fminbnd**

both **fminbnd** and **fzero** admit function handles as inputs

```
f = @(x) x.^4 - 4*x;                % find minimum of f(x)
[x1,f1] = fminbnd(f,0,1.5);         % in the interval[0,1.5]
```

finding the minimum of f(x) using the function **fzero**, requires derivative F(x) = df(x)/dx

```
F = @(x) 4*x.^3 - 4;                   % derivative of f(x)
x2 = fzero(F, 0.5); f2 = f(x2);

[x0,x1,x2; f0,f1,f2]        % compare the three methods

ans =
    0.9966    1.0000    1.0000
   -2.9999   -3.0000   -3.0000
```

11. Relational and Logical Operators

Relational and logical functions

```
find, logical, true, false

ischar, isequal, isfinite, isinf, 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
```

Relational Operators

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

Logical Operators

```
&     logical AND
&&    logical AND for scalars w/ short-circuiting
|     logical OR
||    logical OR for scalars w/ short-circuiting
~     logical NOT
xor  exclusive OR
any  true if any elements are non-zero
all  true if all elements are non-zero
```

```
>> a = [1 2 0 -3 7];
>> b = [3 2 4 -1 7];
>> a == b
ans =
     0     1     0     0     1

>> a == -3
ans =
     0     0     0     1     0

>> find(a==-3)    % otherwise, it returns empty
ans =
     4

>> find(a), find(a>=2), find(a<=0)
ans =
     1     2     4     5
ans =
     2     5
ans =
     3     4
```

```
>> a = [1 2 0 -3 7];
>> b = [3 2 4 -1 7];

>> a>=2, b<=2
ans =
     0     1     0     0     1
ans =
     0     1     0     1     0

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

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

logical indexing

```
>> a = [1 3 4 -3 7];

>> k = (a>=2), m = find(a>=2)
k =
     0     1     1     0     1
m =
     2     3     5

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

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

% but a(logical(i)) works
```

`class(k)` is logical

k is logical index, m is normal

`class(i)` is double, even though `i==k` is `true`

more on logical indexing

```
>> 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

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

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

12. Program Flow Control

Program flow is controlled by the following control structures:

1. for . . . end   `% loops`
2. while . . . end

3. if . . . end   `% conditional`
4. if . . . else . . .end
5. if . . . elseif . . . else . . . end
6. switch . . . case . . . otherwise. . .end

7. break, continue, return

for-loops and conditional ifs are by far the most commonly used control stuctures

for-loops

```
for variable = expression
   statements ...
end
```

```
>> N=1000; S=0;
>> for n=1:N,
     S = S + 1/n^2;     % compute the sum:
   end
```

$$S = \sum_{n=1}^{N} \frac{1}{n^2}$$

```
>> S
S =
    1.6439

>> n = 1:N; S = sum(1./n.^2)    % vectorized version
S =
    1.6439
```

```
while condition
   statements ...
end
```

while-loops

```
>> N=1000; S=0; n=1;
>> while n<=N,
         S = S + 1/n^2;    % compute the sum:
         n = n+1;
   end
```

$$S = \sum_{n=1}^{N} \frac{1}{n^2}$$

```
>> S
S =
    1.6439

>> pi^2/6       % note the limiting sum,
ans =           % first derived by Euler
    1.6449
```

$$\frac{\pi^2}{6} = \sum_{n=1}^{\infty} \frac{1}{n^2}$$

three versions of conditional ifs

```
if condition
   statements ...
end
```

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

```
if condition
   statements ...
elseif condition
   statements ...
elseif condition
   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 expression
   case expression
      statements ...
   case expression
      statements ...
   otherwise
      statements ...
end
```

this expression is evaluated first, and if its value matches any of these, then the corresponding case-statements are executed

several **case** statements may be present

equivalent calculation using the built-in function **norm** :

```
x = [1, -4, 5, 3]; p = inf;
switch p
   case 1
      N = sum(abs(x));               % N = norm(x,1);
   case 2
      N = sqrt(sum(abs(x).^2));      % N = norm(x,2);
   case inf
      N = max(abs(x));               % N = norm(x,inf);
   otherwise
      N = sqrt(sum(abs(x).^2));      % N = norm(x,2);
end
```

L_1, L_2, and L_∞ norms of a vector

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

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

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

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

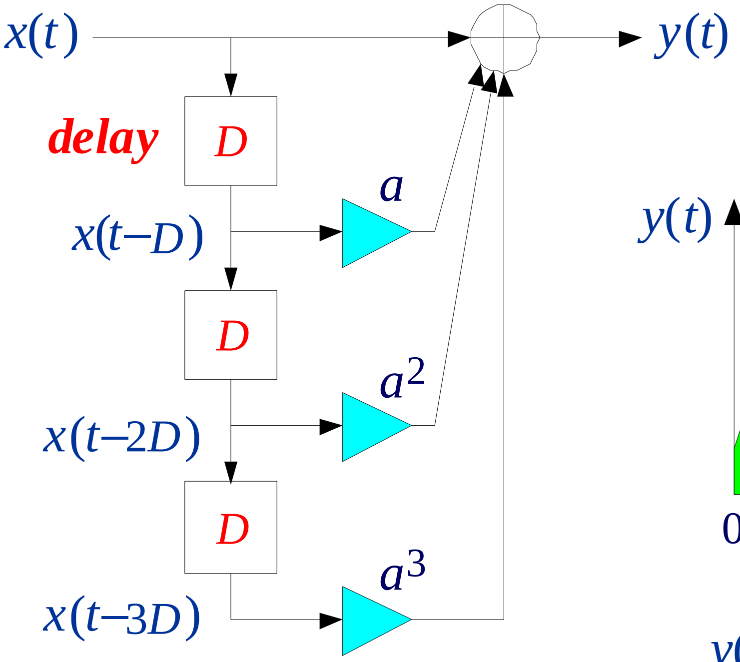
```
>> help norm      % vector and matrix norms
```

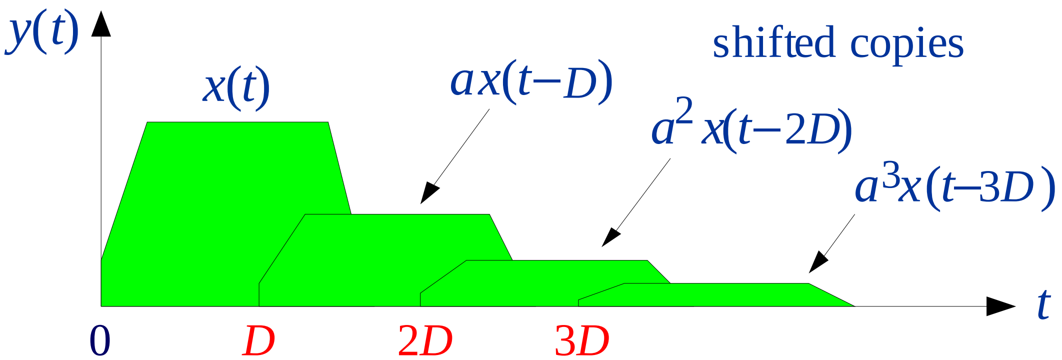
Example: Overlapping Echoes

- DSP application, implementing a Digital Audio Effect
- reads a wave file and plays a 20-second portion of it
- then, adds three overlapping, slightly delayed, copies of itself and plays the result
- illustrates the use of for-loops, if-statements, and pre-allocation to speed up processing

complete program, **`echoes.m`**, and supporting wave files are in the zip file, **`echoes.zip`**.

block-diagram realization





$$y(t) = x(t) + ax(t-D) + a^2x(t-2D) + a^3x(t-3D)$$

```
% echoes.m - listen to overlapping echoes

clear all;

[x,Fs] = wavread('dsummer.wav');    % read wave file and its Fs

N = min(round(20*Fs), length(x));  % play no more than 20 sec
x = x(1:N);                        % truncate x to length N

sound(x,Fs);                       % play x

T = 1/2; D = round(T*Fs);  % echo delay in sec and in samples

Fs, N, D              % here, Fs=44100, N=839242, D=22050

a = 0.5;              % multiplier coefficient

y = zeros(size(x));   % pre-allocation speeds up processing
```

Note: the sampling rate Fs is the number of samples per second, thus,
N = 20*Fs = (20 sec)*(samples/sec) = number of samples in 20 sec

```
tic                        % tic-toc - measures execution time
for n=1:length(x),         % construct overlapped signal y
   if n<=D,
      y(n) = x(n);
   elseif n<=2*D,
      y(n) = x(n) + a * x(n-D);
   elseif n<=3*D,
      y(n) = x(n) + a * x(n-D) + a^2 * x(n-2*D);
   else,
      y(n) = x(n) + a * x(n-D) + a^2 * x(n-2*D) +...
             a^3 * x(n-3*D);
   end
end
toc

pause; sound(y,Fs);           % play y
```

proper indentation improves readability,

try to read this

```
%tic for n=1:length(x),if n<=D,y(n)=x(n);elseif n<=2*D,y(n)=...
%x(n)+a*x(n-D);elseif n<=3*D,y(n)=x(n)+a*x(n-D)+a^2*x(n-2*D);...
%else,y(n)=x(n)+a*x(n-D)+a^2*x(n-2*D)+a^3*x(n-3*D); end end...
%toc pause;sound(y,Fs);
```

pre-allocation results

wave file	Fs	N	with	without
JB.wav	16000	71472	0.02 sec	34.44 sec
nodelay.wav	22050	266758	0.13 sec	702.33 sec
dsummer.wav	44100	839242	0.39 sec	too long

13. Matrix Algebra

- dot product
- matrix-vector multiplication
- matrix-matrix multiplication
- matrix inverse
- solving linear systems

dot product

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

a, **b** must have the same dimension

$$\mathbf{a}^T\mathbf{b} = [a_1, a_2, a_3] \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

$$\mathbf{a}^T\mathbf{b} = \mathbf{a}'\,\mathbf{b} = \mathbf{a} \cdot \mathbf{b} = \mathbf{a}\,.'\ast\,\mathbf{b}$$

math notations

MATLAB notation

dot product
for complex-valued vectors

hermitian conjugate of **a**

$$\mathbf{a}^{\dagger}\,\mathbf{b} = [a_1^*,\, a_2^*,\, a_3^*]\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = a_1^* b_1 + a_2^* b_2 + a_3^* b_3$$

$$\mathbf{a}^{\dagger}\,\mathbf{b} = \mathbf{a}^{H}\mathbf{b} = \mathbf{a}' * \mathbf{b}$$

math notations

MATLAB notation

for real-valued vectors, the operations `'` and `.'` are equivalent

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix}$$

$$[1,\ 2,\ -3]\begin{bmatrix} 4 \\ -5 \\ 2 \end{bmatrix} = 1 \times 4 + 2 \times (-5) + (-3) \times 2 = -12$$

```
>> a = [1; 2; -3];  b = [4; -5; 2];
>> a'*b
ans =
   -12
>> dot(a,b)            % built-in function
ans =
   -12
```

matrix-vector multiplication

$$[4,\ 1,\ 2]\begin{bmatrix} 5 \\ -4 \\ -7 \end{bmatrix} = 2$$

combine three dot product operations into a single matrix-vector multiplication

$$[1,\ -1,\ 1]\begin{bmatrix} 5 \\ -4 \\ -7 \end{bmatrix} = 2 \qquad \Rightarrow \qquad \begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix}\begin{bmatrix} 5 \\ -4 \\ -7 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}$$

$$[2,\ 1,\ 1]\begin{bmatrix} 5 \\ -4 \\ -7 \end{bmatrix} = -1$$

matrix-matrix multiplication

$$\begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ -4 \\ -7 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ 1 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

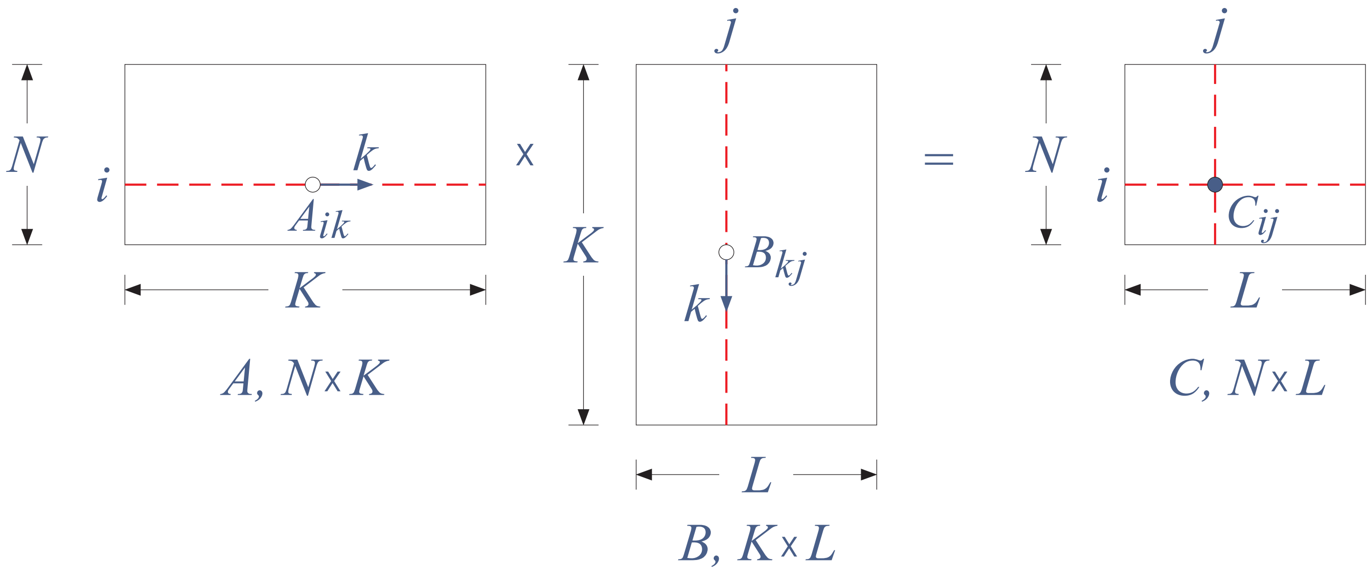
combine three matrix-vector multiplications into a single matrix-matrix multiplication

$$\begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 5 & -1 & -3 \\ -4 & 3 & 1 \\ -7 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 \\ 2 & -2 & 2 \\ -1 & 3 & 1 \end{bmatrix}$$

```
>> A = [4 1 2; 1 -1 1; 2 1 1]
A =
     4     1     2
     1    -1     1
     2     1     1

>> B = [5 -1 -3; -4 3 1; -7 2 6]
B =
     5    -1    -3
    -4     3     1
    -7     2     6

>> C = A*B
C =
     2     3     1
     2    -2     2
    -1     3     1
```



$$C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}, \qquad 1 \le i \le N, \quad 1 \le j \le L$$

C(*i*,*j*) is the dot product of *i*-th row of *A* with *j*-th column of *B*

$$\begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 5 & -1 & -3 \\ -4 & 3 & 1 \\ -7 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 \\ 2 & -2 & 2 \\ -1 & 3 & 1 \end{bmatrix}$$

$$2 \times (-1) + 1 \times 3 + 1 \times 2 = 3$$

solving linear systems

$$\begin{aligned} 4x_1 + x_2 + 2x_3 &= 10 \\ x_1 - x_2 + x_3 &= 20 \\ 2x_1 + x_2 + x_3 &= 10 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 20 \\ 10 \end{bmatrix}$$

$$A\mathbf{x} = \mathbf{b}$$

$$A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \mathbf{x} = A^{-1}\mathbf{b} = A \backslash \mathbf{b}$$

always use the backslash operator to solve a linear system, instead of inv(A)

solving linear systems (using backslash)

$$
\begin{aligned}
4x_1 + x_2 + 2x_3 &= 10 \\
x_1 - x_2 + x_3 &= 20 \\
2x_1 + x_2 + x_3 &= 10
\end{aligned}
\quad \Rightarrow \quad
\begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}
=
\begin{bmatrix} 10 \\ 20 \\ 10 \end{bmatrix}
$$

```
>> A = [4 1 2; 1 -1 1; 2 1 1];
>> b = [10 20 10]';
>> x = A\b
x =
   -30
    10
    60

>> norm(A*x-b)      % test - should be zero
ans =
     0
```

solving linear systems (using inv)

$$\begin{aligned} 4x_1 + x_2 + 2x_3 &= 10 \\ x_1 - x_2 + x_3 &= 20 \\ 2x_1 + x_2 + x_3 &= 10 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} 4 & 1 & 2 \\ 1 & -1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 20 \\ 10 \end{bmatrix}$$

```
>> A = [4 1 2; 1 -1 1; 2 1 1];
>> b = [10 20 10]';
>> inv(A)                        % same as A^(-1)
ans =
     2    -1    -3
    -1     0     2
    -3     2     5

>> x = inv(A) * b                % but prefer backslash
x =
   -30
    10
    60
```