International University
School of Electrical Engineering

Introduction to Computers for Engineers

Dr. Hien Ta

Lecturely Topics

References: H. Moore, *MATLAB for Engineers*, 4/e, Prentice Hall, 2014
G. Recktenwald, *Numerical Methods with MATLAB*, Prentice Hall, 2000
A. Gilat, *MATLAB, An Introduction with Applications*, 4/e, Wiley, 2011

User-Defined Functions

M-files, script files, function files

anonymous & inline functions

function handles

function functions, **`fzero,fminbnd`**

multiple inputs & outputs

subfunctions, nested functions

homework template function

function types

recursive functions, fractals

M-files: script files and function files

Script M-files contain commands to be executed as though they were typed into the command window, i.e., they collect many commands together into a single file.

Function M-files must start with a **`function`** definition line, and may accept input variables and/or return output variables.

The function definition line has syntax:

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

where the function name, **`func`**, is arbitrary and must match the name of the M-file, i.e., **`func.m`**

Variables defined in a script M-file are known to the whole current workspace, i.e., inside and outside the script file.

Script M-files may not have any function definitions in them, unless the functions are defined as inline or anonymous one-line functions, e.g., using the function-handle **@(x)**.

Variables in a function M-file are local to that function only and are not recognized outside the function (unless they are declared as global variables, which is usually not recommended.)

Function files may include the definition of other functions, either as sub-functions, or as nested functions. This helps to collect together all relevant functions into a single file (e.g., this is how you may structure your homework solutions.)

Example:

```
% file rms.m calculates the
% root-mean-square (RMS) value and the
% mean-absolute value of a vector x:

function [r,m] = rms(x)
  r = sqrt(sum(abs(x).^2) / length(x));
  m = sum(abs(x)) / length(x);
```

```
>> x = -4:4;
>> [r,m] = rms(x)
r =
    2.5820
m =
    2.2222
```

```
>> r = rms(x);
```

returns only the first output

returns only the second output

```
>> [~,m] = rms(x);
```

Make up your own functions using three methods:

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

example 1: $f(x) = e^{-0.5x}\sin(5x)$

```
>> 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 2: $f(x) = e^{-ax}\sin(bx)$

```
% 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.
```

example 3: test the convergence of the following series for π , (Madhava, ca.1400):

$$\pi = \lim_{N\to\infty} 2\sqrt{3} \sum_{k=0}^{N} \frac{(-1)^k}{(2k+1)\,3^k}$$

```
g = @(N) 2*sqrt(3) * cumsum(...
         (-1).^(0:N)./(2*(0:N)+1)./3.^(0:N));
```

```
% edit & save file f.m containing the lines:

function y = f(N)

k=0:N;

y = 2*sqrt(3)*cumsum((-1).^k./(2*k+1)./3.^k);
```

convergence results:

```
  N    f(N) or g(N)        digit accuracy
--------------------------------------
  5    3.141                    3
 10    3.14159                  5
 15    3.14159265               8
 20    3.1415926535            10
 25    3.1415926535897         13
Inf    3.1415926535897...
```

Note: the functions f(N) and g(N) give equivalent results,

g(N) is a one-line definition, but much harder to read,

f(N) is easy to read, but requires its own M-file, here, **`f.m`**

example 4: Fourier series approximation of the function,

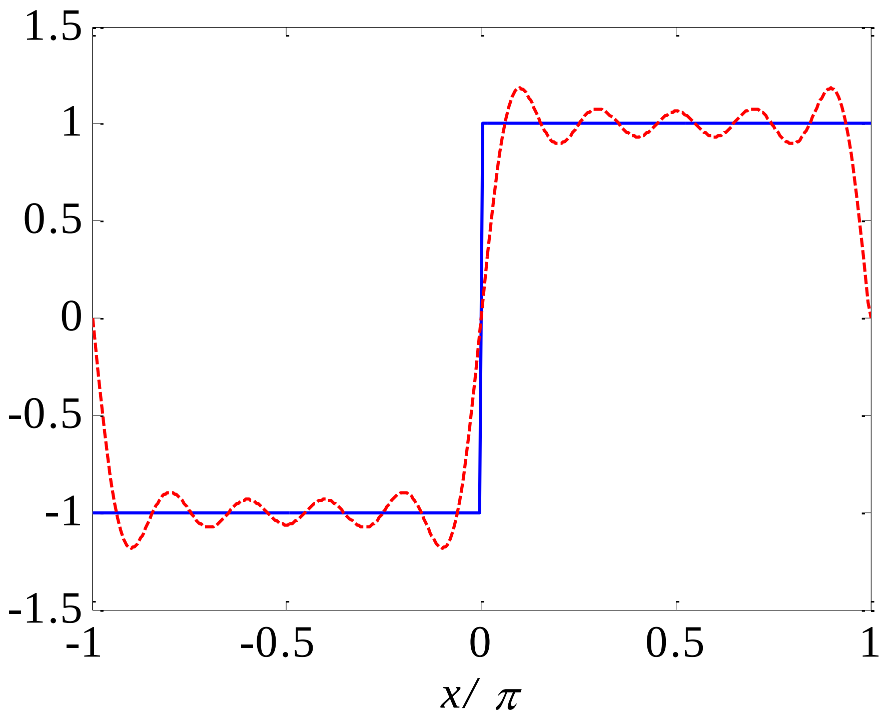
$$f(x) = \begin{cases} +1, & 0 < x \leq \pi \\ 0, & x = 0 \\ -1, & -\pi \leq x < 0 \end{cases}$$

$$f(x) = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{\sin\big((2k+1)x\big)}{2k+1}$$

keep only the k=0:4 terms, define the function F(x), and compute and plot both f(x) and F(x)

$$F(x) = \frac{4}{\pi} \sum_{k=0}^{4} \frac{\sin\big((2k+1)x\big)}{2k+1}$$

```
>> f = @(x) sign(x) .* (abs(x)<=pi);

>> F = @(x) 4/pi*( sin(x) + sin(3*x)/3 + ...
            sin(5*x)/5 + sin(7*x)/7 + sin(9*x)/9 );

>> x = linspace(-pi,pi,501);
>> plot(x/pi,f(x),'b-', x/pi,F(x),'r--');
>> xlabel('\itx/\pi');
```



Note: when **`x`** is a vector, the logical statement

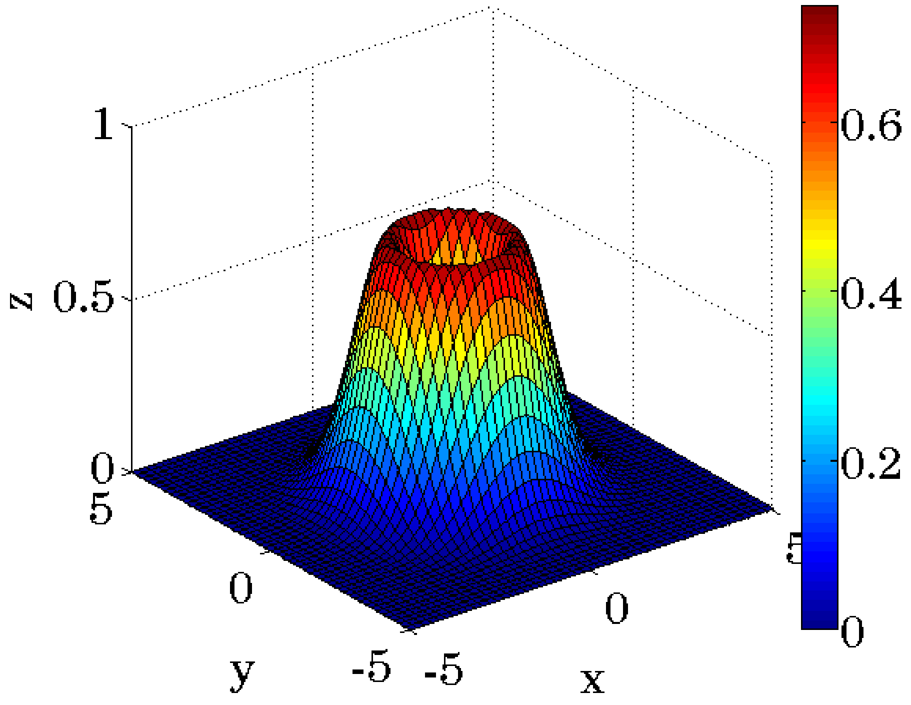
`(abs(x)<=pi)`

results in a vector of 0s or 1s,

see the section on relational and logical operators, in week-7 lecture notes

example 5: Anonymous functions with multiple arguments

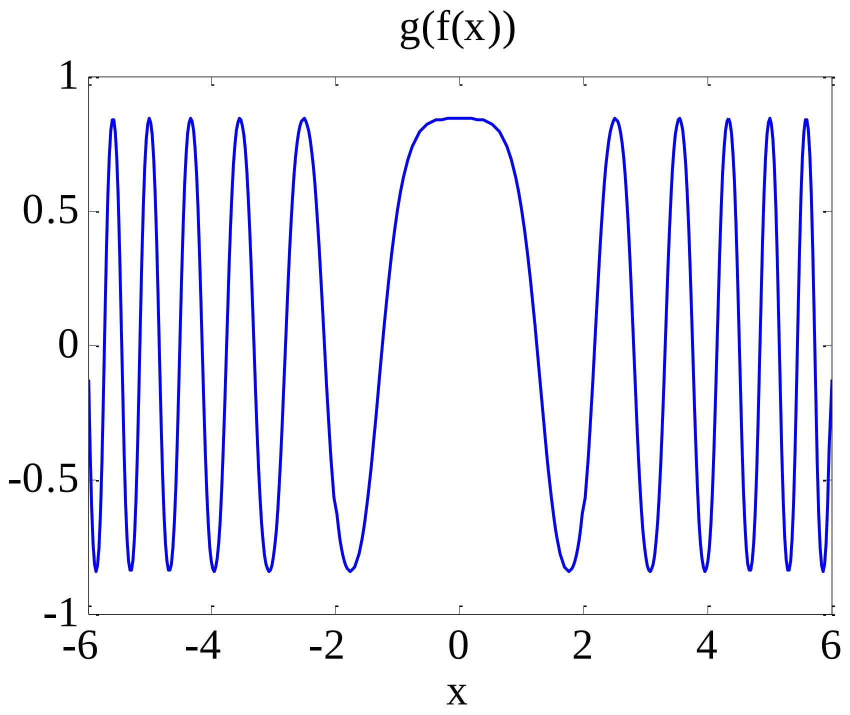
```
f = @(x,y) (x.^2 + y.^2) .* exp(-(x.^2 + y.^2)/2);

x = linspace(-5,5,51);
y = linspace(-5,5,51);
[X,Y] = meshgrid(x,y);

Z = f(X,Y);

surf(X,Y,Z);
colorbar;
```



$$f(x,y) = (x^2 + y^2)\exp\left(-\tfrac{1}{2}(x^2 + y^2)\right)$$

Anonymous functions can be nested

```
>> f = @(x) x.^2;
>> g = @(x) sin(cos(x));
>> h = @(x) g(f(x));          % i.e., sin(cos(x.^2))

>> fplot(h,[-6,6],'b-');
>> xlabel('x'); title('g(f(x))');
```



Function Handles

A function handle is a data type that allows the referencing and evaluation of a function, as well as passing the function as an input to other functions, e.g., to **`fplot, ezplot, fzero, fminbnd, fminsearch`**.

In anonymous functions, e.g., **`f = @(x) (expression)`** the defined quantity **`f`** is already a function handle.

For built-in, or user-defined functions in M-files, the function handle is obtained by prepending the character @ in front of the function name, e.g.,

```
f_handle = @sin;
f_handle = @my_function;
```

Function Functions

>> help funfun

A number of MATLAB functions accept other functions as arguments. Such functions cover the following categories:

1. Function optimization (min/maximization) , root finding, and plotting, and data fitting, e.g., **`fplot, ezplot, fzero, fminbnd, fminsearch, nlinfit, lsqcurvefit`**.

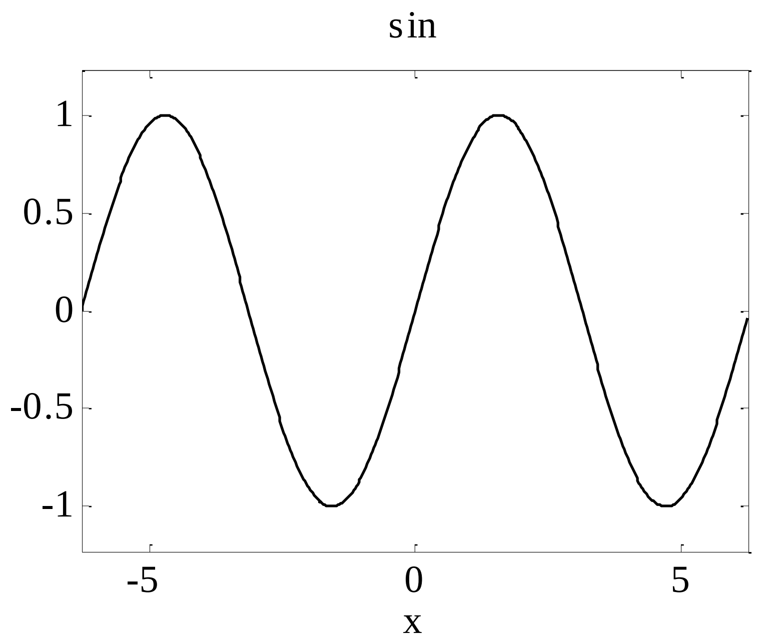
2. Numerical integration (quadrature) , e.g., **`quad`**, and its variants.

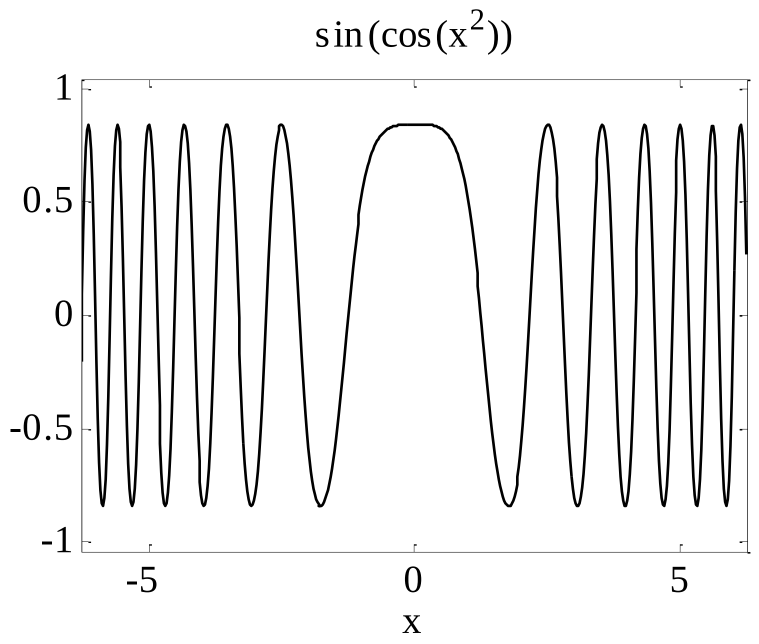
3. Differential equation solvers, e.g., **`ode45`**, and others.

4. Initial value and boundary value problem solvers.

The function argument is passed either as a function handle (new method), or, as a string of the function name (old method)

```
>> ezplot(@sin);          % pass by function handle
>> ezplot('sin');         % older method
>> ezplot('sin(x)');

>> f = @(x) sin(cos(x.^2));
>> ezplot(f);                                 % equivalent
>> ezplot(@(x) sin(cos(x.^2)));               % methods
>> ezplot(@(x) sin(cos(x^2)));
>> ezplot('sin(cos(x^2)');
```





Solution of the Van der Waals equation using **fzero**

$$f(V) = \left(P + \frac{n^2 a}{V^2}\right)(V - nb) - nRT = 0$$

```
P = 220; n = 2;                % values are from
a = 5.536; b = 0.03049;        % Problem 2.7
R = 0.08314472; T = 1000;

V0 = n*R*T/P;    % ideal-gas case, V0 = 0.7559

f = @(V) (P + n^2*a./V.^2).*(V-n*b) - n*R*T;

V = fzero(f, V0)

V =
    0.6825
```

seek a solution of **f(V) = 0**, near **V0**

passing to **fzero** a function that has additional parameters

```
f = @(x,a,b) ...    % define f(x,a,b) here,
                    % or, in a separate M-file

% find the solution of f(x,a,b)=0
% define a,b here

x = fzero(@(x) f(x,a,b), x0);
```

effectively defines a new anonymous function and passes its handle to **fzero**

```
>> doc fzero
```

same method can be used for **fminbnd**, and all other function functions

Example 1

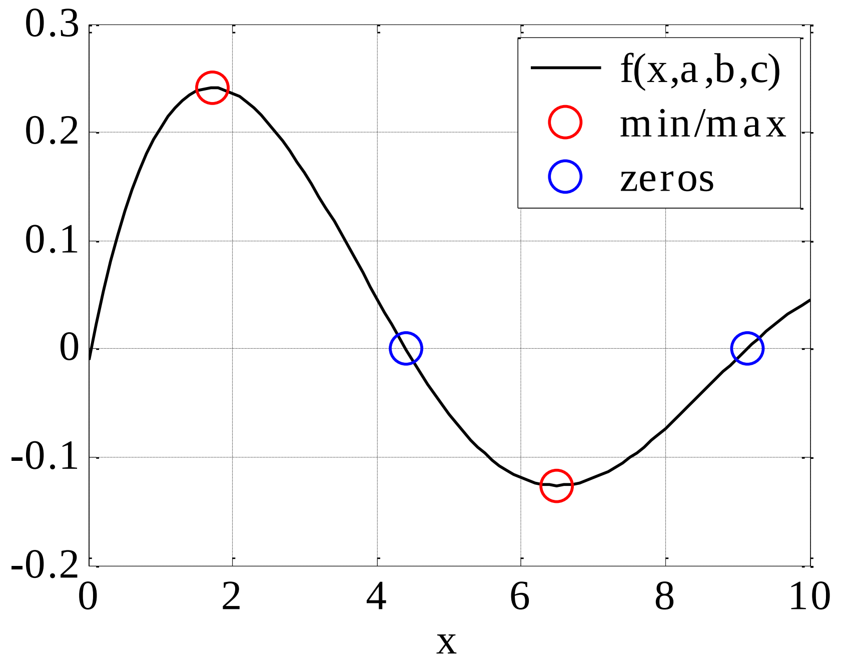
```
P = 220; n = 2;                % values are from
a = 5.536; b = 0.03049;        % Problem 2.7
R = 0.08314472; T = 1000;

V0 = n*R*T/P;     % ideal-gas case, V0=0.7559

f=@(V,a,b) (P + n^2*a./V.^2).*(V-n*b) - n*R*T;

V = fzero(@(V) f(V,a,b), V0)

V =
    0.6825
```

defines a new anonymous function and passes its handle to **`fzero`**

Example 2

```
f = @(x,a,b,c) sin(a*x)./(x + b) + c;

a = 0.7; b = 2; c = -0.01;

[x1,f1] = fminbnd(@(x) -f(x,a,b,c), 0,4); f1 = -f1;
[x2,f2] = fminbnd(@(x) f(x,a,b,c), 6,8);

x3 = fzero(@(x) f(x,a,b,c), 5);
x4 = fzero(@(x) f(x,a,b,c), 9);

x = linspace(0,10,101);

y = f(x,a,b,c);
```

$$\boxed{f(x,a,b,c) = \frac{\sin(ax)}{x+b} + c}$$



```
plot(x,y, [x1,x2],[f1,f2],'ro',[x3,x4],[0,0],'bo');
```

`sinc` functions appear in many engineering applications:

1. Fourier analysis of signals
2. Optical systems (resolving power of microscopes, telescopes)
3. Radar systems
4. DSP applications and digital communications
5. Antenna arrays, sonar and seismic arrays
6. Playback systems of CD and MP3 players (known as sinc interpolation filters or oversampling digital filters)
7. And many others

$$\mathrm{sinc}(x) = \frac{\sin(x)}{x}, \quad \mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

math definition

MATLAB definition

Example 3 - 3-dB width of the sinc function

```
fplot(@sinc, [-4,4], 'b-'); hold on;
f =  @(x) sinc(x)-1/sqrt(2);
x0 = fzero(f,0.5);                  % x0 = 0.443
plot([-x0,x0],[1,1]/sqrt(2),'r-');
```

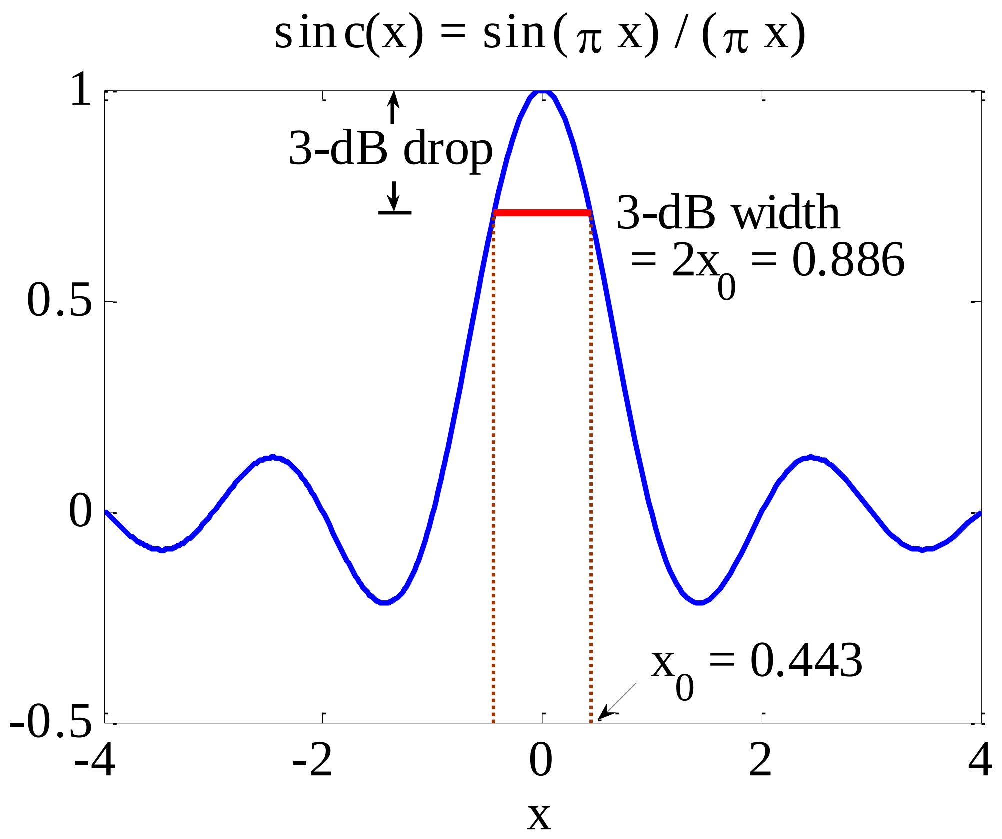
`x0` is the solution of the equation:

$$\left[\frac{\sin(\pi x)}{\pi x}\right]^2 = \frac{1}{2}$$

or,

$$\mathrm{sinc}(x) = \frac{1}{\sqrt{2}}$$

$$10\log_{10}(1/2) = -3\text{ dB}$$



Multi-Input Multi-Output Functions

In general, a function can accept several variables as input arguments and produce several variables as outputs.

The input arguments are separated by commas, and the output variables are listed within brackets, and can have different sizes and types:

```
[out1, out2, ...] = funct(in1, in2,...)
```

The number of input and output variables are counted by the reserved variables: **`nargin, nargout`**

Functions can also have a variable number of inputs and outputs controlled by: **`varargin, varargout`**

Example: calculate the *x*,*y* coordinates and *x*,*y* velocities v_x, v_y of a projectile, at a vector of times *t*, launched from height h_0 with initial velocity v_0, at angle θ_0 (in degrees) from the horizontal, under vertical acceleration of gravity *g*:

```
[x,y,vx,vy] = trajectory(t,v0,th0,h0,g);
```

The equations of motion are:

$$x = v_0 \cos\theta_0\, t$$

$$y = h_0 + v_0 \sin\theta_0\, t - \frac{1}{2} g\, t^2$$

$$v_x = v_0 \cos\theta_0$$

$$v_y = v_0 \sin\theta_0 - gt$$

The function **`trajectory`** should have the following possible ways of calling it:

```
[x,y,vx,vy] = trajectory(t,v0);
[x,y,vx,vy] = trajectory(t,v0,th0);
[x,y,vx,vy] = trajectory(t,v0,th0,h0);
[x,y,vx,vy] = trajectory(t,v0,th0,h0,g);
[~,y,~,vy]  = trajectory(t,v0,th0,h0,g);
```

place-holders for unused output arguments

```
          x = trajectory(t,v0,th0,h0);
      [x,y] = trajectory(t,v0,th0,h0);
[x,y,vx] = trajectory(t,v0,th0,h0);
```

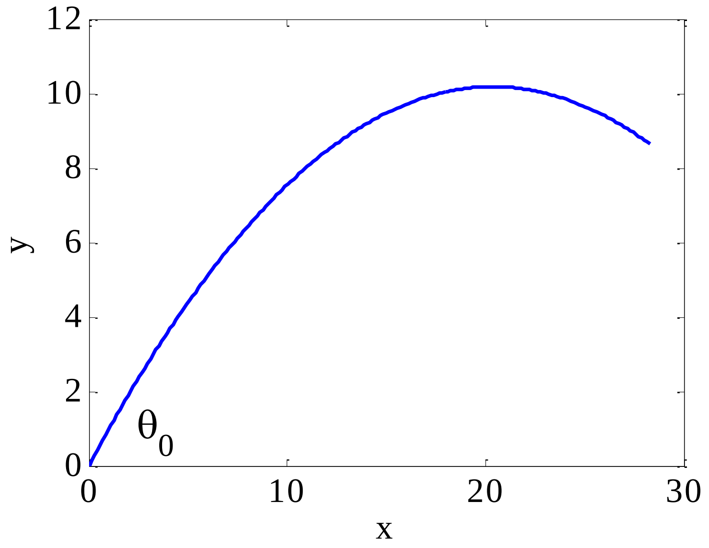
where, if omitted, the default input values should be:

```
th0 = 90;               % vertical launch, degrees
h0 = 0;                 % ground level
g = 9.81;               % m/sec^2
```

only the listed output variables are returned

```
function [x,y,vx,vy] = trajectory(t,v0,th0,h0,g)

if nargin<=4, g = 9.81; end        % default values
if nargin<=3, h0 = 0; end
if nargin==2, th0 = 90; end

th0 = th0 * pi/180;          % convert th0 to radians

x = v0*cos(th0)*t;
y = h0 + v0*sin(th0)*t - 1/2*g*t.^2;
vx = v0*cos(th0);
vy = v0*sin(th0) - g*t;
```

```
t = linspace(0,2,201);
v0 = 20; th0 = 45;
[x,y]=trajectory(t,v0,th0);
plot(x,y, 'b');
xlabel('x'); ylabel('y');
```



Subfunctions and Nested Functions

A function can include, at its end, the definitions of other functions, referred to as subfunctions.

The subfunctions can appear in any order and each can be called by any of the other ones within the primary function.

Each subfunction has its own workspace variables that are not shared by the other subfunctions or the primary one, i.e., each subfunction communicates only through its output variables.

Nested functions share their workspace variables with those of the primary function. They must all end with the keyword **`end`**, i.e., if an end is used in one, it must used also in all the others.

Example:

```
% alternative version of rms.m

function [r,m] = rms(x)
  r = rmsq(x);           % root-mean-square
  m = mav(x);            % mean absolute value

function y = rmsq(x)
   y = sqrt(sum(abs(x).^2) / length(x));

function y = mav(x)
   y = sum(abs(x)) / length(x);
```

the appearance of the keyword **`function`** signals the beginning of each subfunction

Example:

```
% nested version of rms.m

function [r,m] = rms(x)
  N = length(x);
  r = rmsq(x);             % root-mean-square
  m = mav(x);              % mean absolute value

  function y = rmsq(x)
    y = sqrt(sum(abs(x).^2)/N);
  end                             % end of rmsq

  function y = mav(x)
    y = sum(abs(x))/N;
  end                             % end of mav

end                               % end of rms
```

N is known to the nested subfunctions

Example: recommended structure for homework reports

```
function set1

  problem1
  problem2
  problem3

function problem1
   ...
function problem2
   ...
function problem3
   ...

function other1
   ...
function other2
   ...
function other3
   ...
```

primary function

execute problem subfunctions to get the homework results

define the problem subfunctions implementing each problem

define any other subfunctions that may be called by the problem subfunctions

Summary of Function Types

- Primary functions
- Anonymous functions
- Subfunctions
- Nested functions
- Private functions
- Overloaded functions
- Recursive functions

Recursive Functions

Recursive functions call themselves

i.e., they define themselves by calling themselves

Not quite as circular as it sounds
(e.g., a tall person is defined as one who is tall)

Interesting and elegant programming concept,
but tends to be very slow in execution (it exists in other languages like C/C++ and Java)

Nicely suited for repetitive tasks, like generating fractals

Example 1: Fibonacci numbers, $f(n) = f(n-1) + f(n-2)$

```
function y = fib(n,c)

if n==1, y = c(1); end
if n==2, y = c(2); end

if n>=3,
   y = fib(n-1,c) + fib(n-2,c);
end
```

initial values:

f(1) = c(1);
f(2) = c(2);

c = [c(1),c(2)];

```
y = []; c = [0,1];
for n=1:10,
   y = [y, fib(n,c)];
end

y =
     0    1    1    2    3    5    8   13   21   34
```

Example 2: Binomial Coefficients, `nchoosek(n,k)`

```
function C = bincoeff(n,k)

if (k==0)|(k==n),          % assumes n>=0, k>=0
   C = 1;
elseif k>n,
   C = 0;
else
   C = bincoeff(n-1,k) + bincoeff(n-1,k-1);
end
```

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\boxed{(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k}$$

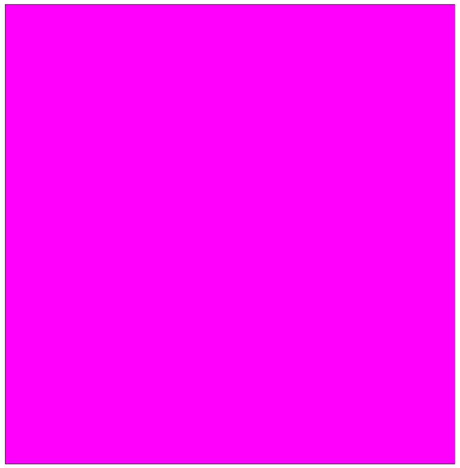
```
for n=0:6,
   C=[];
   for k=0:n,
      C = [C, bincoeff(n,k)];
   end
   disp(C);
end
```

```
 1
 1     1
 1     2     1
 1     3     3     1
 1     4     6     4     1
 1     5    10    10     5     1
 1     6    15    20    15     6     1
```

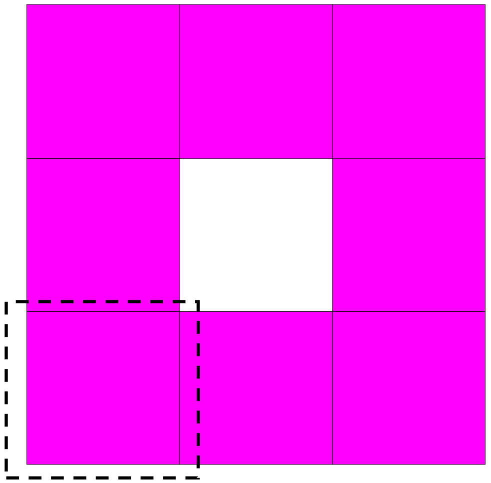
Pascal triangle

Example 3: Sierpinsky Carpet

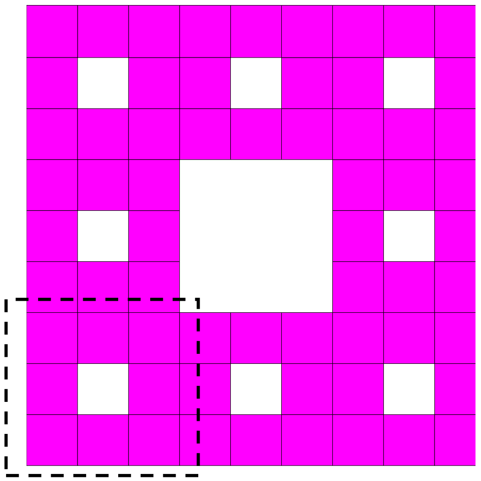
level = 0



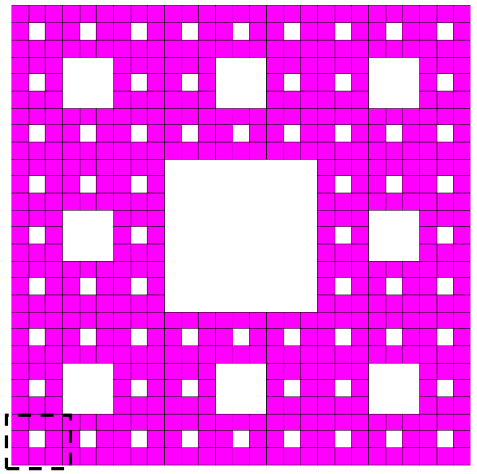
level = 1



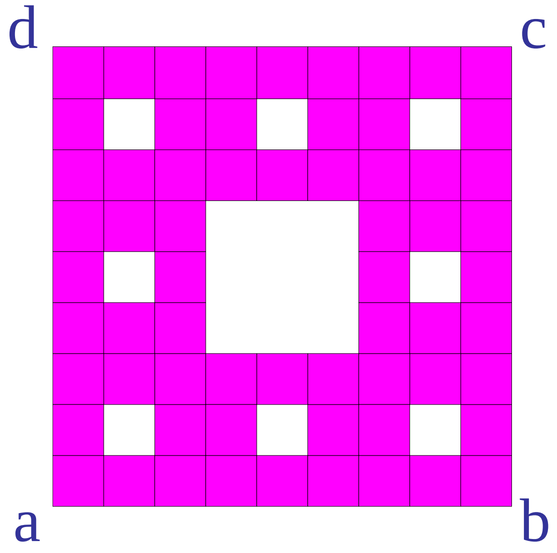
level = 2



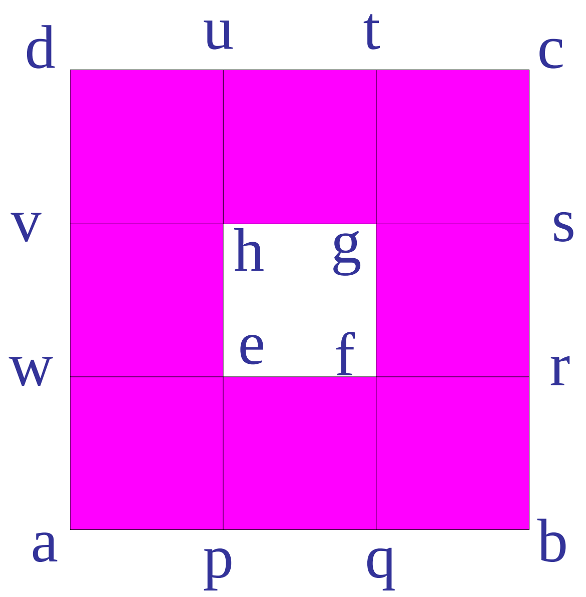
level = 3



```
level=2;
a=[0,0]; b=[1,0]; c=[1,1]; d=[0,1];
carpet(a,b,c,d,level);
axis equal; axis off;
```



```
function carpet(a,b,c,d,level)

p = (2*a+b)/3; q = (a+2*b)/3;
r = (2*b+c)/3; s = (b+2*c)/3;
t = (d+2*c)/3; u = (2*d+c)/3;
v = (2*d+a)/3; w = (d+2*a)/3;

e = (2*w+r)/3; f = (w + 2*r)/3;
g = (2*s+v)/3; h = (s + 2*v)/3;

if level==0,
  fill([a(1),b(1),c(1),d(1)], [a(2),b(2),c(2),d(2)], 'm');
  hold on;
else
   carpet(a,p,e,w, level-1);   % recursive calls
   carpet(p,q,f,e, level-1);
   carpet(q,b,r,f, level-1);
   carpet(f,r,s,g, level-1);
   carpet(g,s,c,t, level-1);
   carpet(h,g,t,u, level-1);
   carpet(v,h,u,d, level-1);
   carpet(w,e,h,v, level-1);
end
```



carpet.m
in course-functions
resources on sakai

see, **`carpet2.m`**