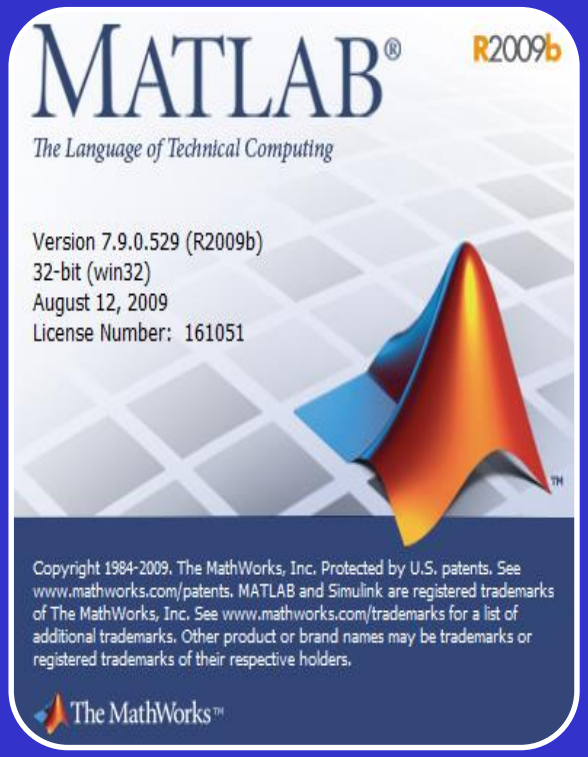
MATLAB Programming (Lecture 4)

Dr. Sun Bing
School of EIE
Beihang University

Contents

User-Defined Functions



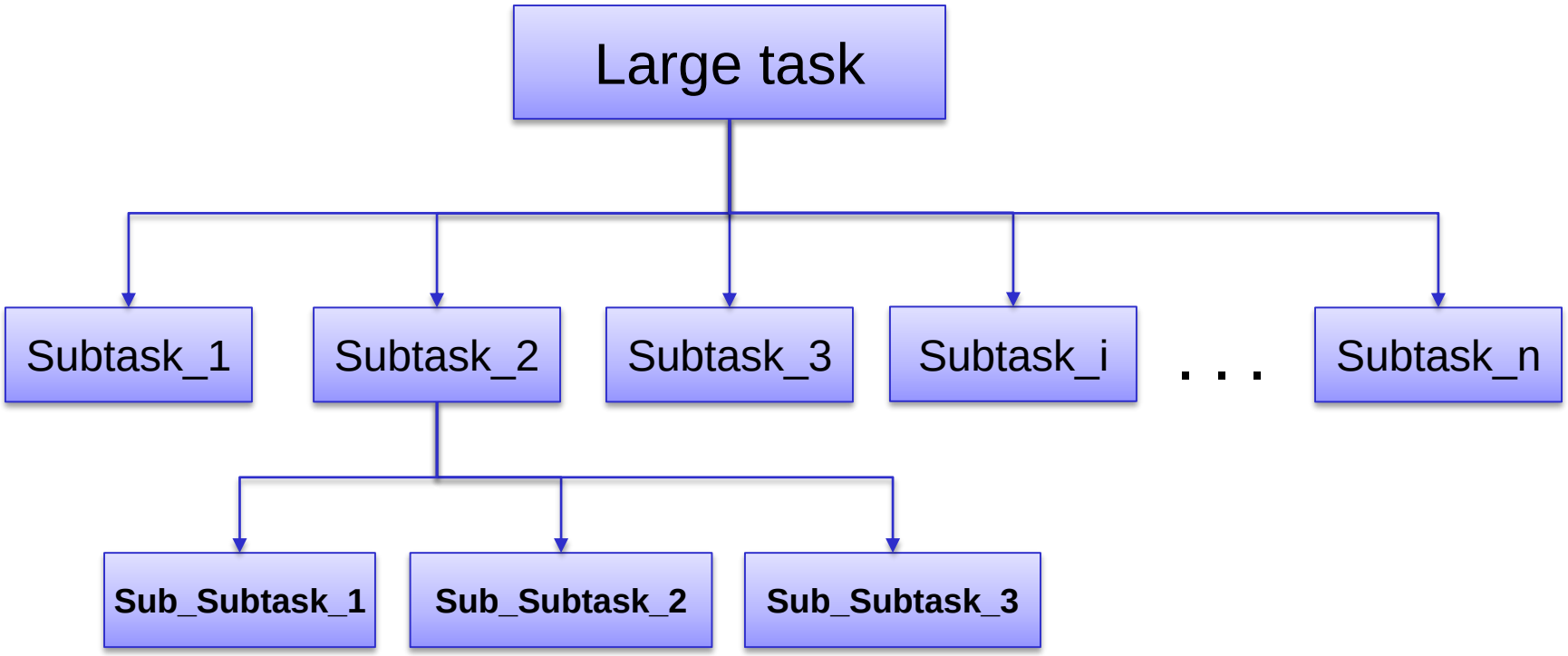
- User defined function
- Function type
- Call-by-value schema
- Optional arguments
- The Recursion
- The global variables
- Preserving data between calls to a function
- Function Functions
- Object Oriented Programming
- Some interesting functions

Review: Introduction to M-file

- Two types of M-file: script **and** function
- *All script files have the following features:*

1. Script files share the command window's workspace. Any variables created by script files remain in the workspace after the script file finishes executing.
2. A script file has no input arguments and does not return output arguments,
3. Script files can communicate with other script files through the data left behind in the workspace.

Review: Top-Down Design Techniques



- Task Division
- Interface Design

4.1 User defined function

- MATLAB user defined function is a special type of M-file called M-file Function.
- The names of the M-file and of the main function should be the same.
- Functions operate on variables within their own independent workspace, separate from the workspace you access at the MATLAB command prompt.
- Functions receive the input data through an input argument list, and return results to the caller through an output argument list.

4.1 User defined function

- It is possible to code each subtask as a separate function. Well-designed functions enormously reduce the effort required on a large programming project.
- Their benefits are :

  (1) Independent Testing of Subtasks

  (2) Reusable Code (for example, sorting function)

  (3) Isolation from Unintended Side Effects (The accidental programming mistakes within a function can only affect the variables within the function in which the mistakes occurred)

4.1.1 The General form of User defined function

```
function  [outarg1,outarg2,…]  =  function_name
   ( inarg1,inarg2,…)
% H1 comment line.
% Other comment lines
……
( executable code )
end
```

- The initial comment lines in the function serve special purpose. The first line after function statement is called H1 comment line, which is searched and displayed by the `lookfor` command. The comment lines from H1 line until the first blank line or executable statement are displayed by `help` command.
- The M-file name must be same as function name.

Basic Parts of an M-File

M-File Element	Description
Function definition line (functions only)	Defines the function name, and the number and order of input and output arguments
H1 line	A one line summary description of the program, displayed when you request help on an entire directory, or when you use lookfor
Help text	A more detailed description of the program, displayed together with the H1 line when you request help on a specific function
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

4.1.2 Important Notes

- **It** is important to note that when a MATLAB function is invoked, Matlab creates a local workspace. The commands in the function cannot refer to variables from the interactive workspace in command window unless they are passed as inputs.

- The variables created as the function executes are erased when the execution of the function ends, unless they are passed back as outputs.

- The local workspace is destroyed when the function execution ends.

4.1.3 MATALB function examples

- The following commands should be stored in the file fcn.m

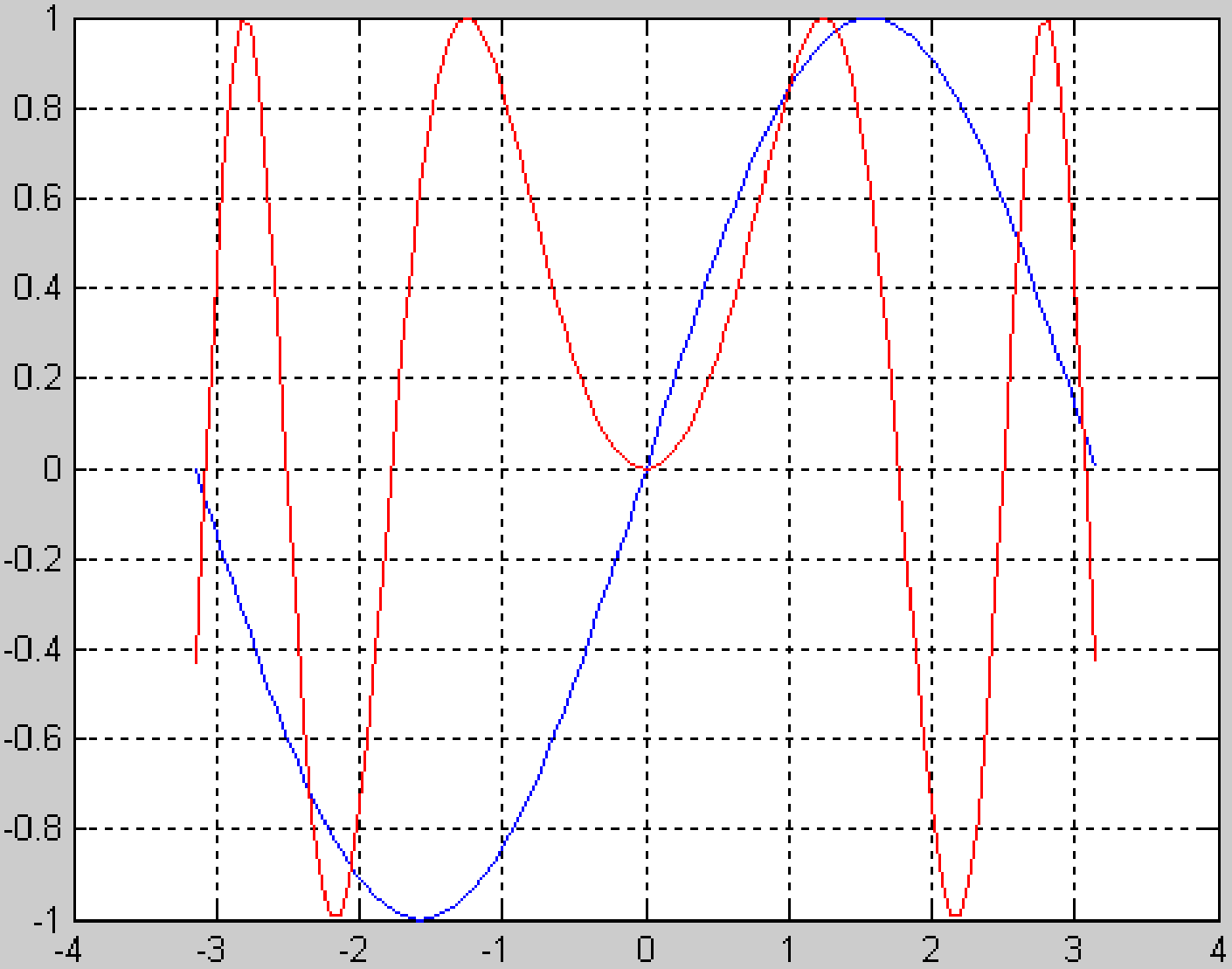
```
function y = fcn(x)
% sample fuction which evaluates the value of
  sin(x^2)
   y = sin(x.^2);
```

Note: the vectorized operator **.^** is used ,so that the function fcn is also vectorized.

4.1.4 Testing the function fnc()

```
% compare the built-in function sin() with
  user defined function fcn()
% M_file name testfcn.m
x1 = (-pi:2*pi/100:pi);
y = sin(x1);
z = fcn(x1);
plot(x1,y,'-b',x1,z,'-r');
grid on;
```

The result plotting



4.2 Function type

- MATLAB offers several different types of functions to use in program.
  - Anonymous Functions
  - Primary and Sub-Functions
  - Private Functions
  - Nested Functions

4.2.1 Anonymous Functions

- An anonymous function is a simple form of MATLAB function that does not require an M-file. It consists of a single MATLAB expression and any number of input and output arguments.

- The syntax for creating an anonymous function from an expression is

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

4.2.1 Anonymous Functions

- **For example,**

```
>>mysquare = @(x) x.^2;

% finds the square of a number.
```

- **To execute the mysqr function defined above, type**

```
>> a = [1:4];

>>s=mysquare(a)

s=

     1     4     9    16
```

4.2.1 Anonymous Functions

- **Note:** You can define an anonymous function right at the MATLAB command line, or within an M-file function or script.
- Function handles not only provide access to anonymous functions. You can create a function handle to any MATLAB function.
- Use the following syntax to construct a function handle, **fhandle = @function_name (e.g., fhandle = @sin).**
- A function handle is a MATLAB value that provides a means of calling a function indirectly.

4.2.2 Primary and sub-functions

- All functions that are not anonymous must be defined within an M-file.
- Each M-file has a required primary function that appears first in the M-file, and any number of sub-functions that follow the primary function.
- Primary functions have a wider scope than sub-functions. That is, primary functions can be invoked from outside of their M-file (from the MATLAB command line or from functions in other M-files) while sub-functions cannot.
- sub-functions are visible only to the primary function and other sub-functions within their own M-file.

4.2.2 Primary and sub-functions Example

```
function [avg,med] = mystats(u)
% mystats() finds the mean and median with subfunction.

n = length(u);
avg = mean(u,n);
med = median(u,n);

% subfunction
function a = mean(v,n)
a = sum(v)/n;

function m = median(v,n)
w = sort(v);
if rem(n,2) == 1
    m = w((n+1)/2);
else
    m = (w(n/2)+w(n/2+1))/2;
end
```

Execute the mystats()

```
>> A = 1:10

A =

     1     2     3     4     5     6     7     8

     9    10

>> [avg, med]= mystats(A)

avg =

    5.5000

med =

    5.5000
```

4.2.3 Private Functions

- A private function is a type of primary M-file function. Its unique characteristic is that it is visible only to a limited group of other functions.
- Private functions reside in subdirectories with the special name *private*. They are visible only to functions in the parent directory.
- The private functions are invisible outside the parent directory, so they can use the same names as functions in other directories. Because MATLAB looks for private functions before standard M-file functions.

4.2.4 Nested Functions

- Simply define one function within the body of another function in an M-file.
- Like any M-file function, a nested function contains any or all of the components described in Basic Parts of an M-File. (e.g. *taxDemo.m*)
- In addition, you must always terminate a nested function with an `end` statement.

```
function x = A(p1, p2)
...
        function y = B(p3)
        ...
        end
 ...
end
```

4.3 Call-by-value schema

- Call-by-values: Same as C language, the MATLAB program communicates with their functions using a call-by-value schema.
- The call-by-value means that each actual argument is evaluated, and its value is copied to the corresponding formal argument. The value stored in the variable of calling environment will no be changed.
- See the Example program ***cbv.m*** and ***exc.m***

4.4 Optional arguments

1. The concept of optional arguments
2. The special functions
3. The examples of function with optional arguments.

4.4.1. The concept of optional arguments

MATLAB functions support optional input arguments and output arguments. For example:

`plot()` supports as few as 1 argument or as many as 7 arguments.

`max()` function supports one or two output arguments. For example:

① `C = max(A)` If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column.

② `[C,I] = max(A)` finds the indices of the maximum values of `A`, and returns them in output vector `I`.

Example : Optional output argument

```
>> a = rand(4)
a =
    0.1389    0.2722    0.4451    0.8462
    0.2028    0.1988    0.9318    0.5252
    0.1987    0.0153    0.4660    0.2026
    0.6038    0.7468    0.4186    0.6721
>> [b1,b2] = max(a)
b1 =
    0.6038    0.7468    0.9318    0.8462
b2 =
     4     4     2     1
>> b = max(a)
b =
    0.6038    0.7468    0.9318    0.8462
```

4.4.2. The special functions

- *nargin* returns the number of input arguments specified for a function.
- *nargout* returns the number of output arguments specified for a function.
- nargchk Check number of input arguments. The syntax is :

```
msg = nargchk(low,high,number)
```

low, high : The minimum and maximum number of input arguments that should be passed.

number : The number of arguments actually passed, as determined by the nargin function.

4.4.2. The special functions

error ： Display error messages.

```
error('message')
```

warning : Display warning message

```
warning('message')
```

(see example polar_value.m)

4.4.3. Examples of function with Optional arguments.

Example: polar_value.m

```
function [mag,angle] = polar_value(x,y)
% Polar_value converts (x,y) to (r,theta).
% Example of function with Optional Arguments

%Check for a legal number of input argument.
msg = nargchk(1,2,nargin);
error(msg);
if nargin<2
    y = 0;
end;
if x==0 & y==0
    msg = 'Both x and y are zero: angle is meaningless!';
    warning(msg);
end
% calculate the first output argument.
mag = sqrt(x.^2+y.^2);
% if the second output argument is present,
% calculate the angle in degrees.
if nargout == 2
    angle = atan2(y,x) * 180/pi;
end;
```

Execute the function polar_value()

```
>> [mag,angle]= polar_value                % zero input
   argument.
??? Error using ==> polar_value
Not enough input arguments.
>> [mag,angle]= polar_value(1,2,3)     % 3 arguments
??? Error using ==> polar_value
Too many input arguments.
>> [mag,angle]= polar_value(2)             % 1 argument
mag =
     2
angle =
     0
```

Execute the function polar_value()

```
>> [mag,angle]= polar_value(1,-1)

mag =

    1.4142

angle =

   -45

>> mag= polar_value(1,-1)

mag =

    1.4142
```

4.5 The Recursion

- The recursion is a powerful tool which can provide elegant solution to some complex programming problem.
- MATLAB function can be recursively invoked.
- *Factorial.m* and *hanoi.m* are two good examples.

```
function nfct=factorial(n)
% Recursion function
% M-file function name factorial.m
if n == 0
    nfct = 1
else
    nfct = n*factorial(n-1);
end
end
```

4.5 The Recursion

- Maximum recursion limit of 500 reached (default).
- Use set(0,'RecursionLimit',N) to change the limit.
- Speed is slower!

```
nfct = n*factorial(n-1);
```

```
nfct = 1;
for i=1:n
    nfct = i*nfct;
end
```

4.6 The global variables

- If you want more than one function can share a single copy of a variable with each other and with the base workspace, simply declare the variable as global in all the functions.
- The form of global variables declare statement is :

```
global var1  var2  var3,…..
```

Note:

1. The global declaration must occur before the variable is actually used in a function.
2. declare the global variables in all capital letters to make them easy to distinguish. (see *seed.m random0.m*)
3. The global variables are stored in the Global Memory. The Global Memory provides a way to share the data between functions.

4.7 Preserving data between calls to a function

- As we know, when the function finishes executing, the local workspace created for that function is destroyed, so all the contents of all local variables within that function will disappear.
- Next time the function is called, a new local workspace will be created again, and all local variables will be set to their default value.
- Some time it is useful to preserve some local values within a function between calls to that function.

4.7 Preserving data between calls to a function

- MATLAB provides Persistent Memory ,This special mechanism to allow local variables to be preserved between calls to the function.
- Persistent Memory is a special type of memory that can only be access from within the function, and preserved unchanged between calls to the function.
- The persistent statement declares the persistent variables. The general form is :

persistent var1 var2 var3 ……

4.7 Preserving data between calls to a function

- MATLAB provides built-in function `mean()` and `std()` to calculate the average and standard deviation of a set of data.
- The formulas are as follow.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} X_i \qquad s = \sqrt{\frac{N\sum_{i=1}^{N} x_i^2 - (\sum_{i=1}^{N} x_i)^2}{N(N-1)}}$$

4.7 Preserving data between calls to a function

- We want to write such a function that must be able to accept input values one at a time and keep running sums of N, $\sum x$, $\sum x^2$, which will be used to calculate the current average and standard deviation. In this case we must use the persistent memory to store the running sums.
- See the examples(See example runstate.m)

4.8 Function Functions

- A class of functions called "function functions" works with nonlinear functions of a scalar variable. That is, one function works on another function.
- Function functions are functions with input argument that include the names of other functions.
- In numerical methods : Zero finding, Optimization, Quadrate, Ordinary differential equations will use function functions.

4.8.1 eval() and feval()

- `eval()` and `feval()` are the two special built-in functions.
- `eval()` evaluates a character string expression as though it has been typed in the command window.

```
eval('expression')
```

- `feval()` evaluates a named function defined by an M-file at a specified input value. The general form of the feval function is

```
feval(fun,value)
```

Where *fun* is either a function handle or function name in string form(a quoted string containing the name of a function).

Example of eval() and feval()

```
>>
>> format long
>> eval('cos(pi/4)')

ans =

   0.70710678118655

>> feval('cos',pi/4)

ans =

   0.70710678118655

>>
```

Example of eval() and feval()

Using eval() for batch processing

```
data100 = randi(100,[100,10]);

%save each row to a txt file
row_1 = data100(1,:);
save('row1.txt','row_1','-ascii');
row_2 = data100(2,:);
save('row2.txt','row_2','-ascii');

% %save 100 rows to 100 txt file
for i=1:100
    filename  = ['row' num2str(i) '.txt'];
    row_i = data100(i,:);
    evalstr = ['save(''' filename ''',''row_i'',''-ascii'');'];
    eval(evalstr)
end
```

4.8.2 The bisection method with Function Functions

A function implementing the method of bisection illustrates :

1. the first important technique is the ability to pass the name of one function to another function. In this case, bisect needs to know the name of the function whose root to be found. This name can be passed as a string. Without this technique a useful bisection routine cannot be written,

2. Must use of the built-in function `feval()` within the function functions

See example bisect.m

Example 1 of function functions

```
1  function c = bisect(fn,a,b,tol)
2  %   Usage : c = bisect('fn',a,b,tol)
3  fa = feval(fn,a);
4  fb = feval(fn,b);
5  if fa*fb >= 0
6    error('The function must have opposite signs at a and b')
7  end
8  done = 0;
9  c = (a+b)/2;
10 while abs(a-b) > 2*tol & ~done
11   fc = feval(fn,c);
12   if fa*fc < 0       % The root is to the left of c
13     b = c;
14     fb = fc;
15     c = (a+b)/2;
16   elseif fc*fb < 0   % The root is to the right of c
17     a = c;
18     fa = fc;
19     c = (a+b)/2;
20   else               % We landed on the root
21     done = 1;
22   end
23 end
```

Example 1 of function functions

Problem

Find the zero point of
`sin(x.^2) =0 at [1,2].`

M-file function name **fcn.m**

function `y=fcn(x)`

```
y=sin(x.^2);
```

```
>> x = bisect('fcn',1,2,1e-6)

x =

    1.7725

>> sqrt(pi)

ans =

    1.7725
```

Example 2 of function functions

- Function myplot(fun,a,b,n) to plotting a function fun at [a,b].

```
function myplot(fun,a,b,npoints)
% Function functions Example
% Plotting a curve f(x) at [a,b]
%fun : function handles or function name in string form.
% a,b : the lower and upper value of interval.
% npoints : number of points equal spaced between [a b]
x=linspace(a,b,npoints);
y=feval(fun,x);
plot(x,y,'linewidth',2);
axis([a,b,-0.6,1.1]);
grid on;
end
```

E.g.

```
sx = @(x) sin(x)./(x+eps);

myplot(sx,-20,20,100);
```

4.8.3 Function Handles

- Construct a function handle in MATLAB using the at sign @, before the function name. For example,

```
shndl = @sin;
```

- creates a function handle for the `sin` function and assigns it to the variable `shndl`.
- You can call a function by means of its handle in the same way that you would call the function using its name. The syntax is

```
shndl(arg1, arg2, ...);
```

4.8.3 Function Handles

- Use function handle to pass function

```
>> bisect(@fcn,1,2,1.0e-6)

ans =

    1.7725
```

- Using function name in string name form.

```
>> bisect('fcn',1,2,1.0e-6)

ans =

    1.7725
```

4.8.4 Using Anonymous Functions

```
>> fs=@(x)sin(x.^2)
fs =
      @(x)sin(x.^2)
>> bisect(fs,1,2,1.0e-6)
ans =
    1.7725
>>fhnd=@(x,y)(x.^2+y.^2);
>> z =fhnd(2,3)
z =
    13
```

4.8.5 Common MATLAB function functions

- `x = fminbnd(fun,x1,x2)` finds the minimum of a function of one variable within a fixed interval.
- `x = fzero(fun,x0)` finds zero of a function of one variable.
- `q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of 1e-6.
- `fplot(fun,limits)` plots fun between the limits specified by limits. limits is a vector specifying the x-axis `limits ([xmin xmax]),` or the x- and y-axes limits, `([xmin xmax ymin ymax]).`
- `ezplot(fun)` plots the expression fun(x) over the default domain `-2pi < x < 2pi.`

4.8.5 Common MATLAB function functions

- `x = fminbnd(fun,x1,x2)` finds the minimum of a function of one variable within a fixed interval.
- `x = fzero(fun,x0)` finds zero of a function of one variable.
- `q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of 1e-6.
- `fplot(fun,limits)` plots fun between the limits specified by limits. limits is a vector specifying the x-axis `limits ([xmin xmax]),` or the x- and y-axes limits, `([xmin xmax ymin ymax]).`
- `ezplot(fun)` plots the expression fun(x) over the default domain `-2pi < x < 2pi.`

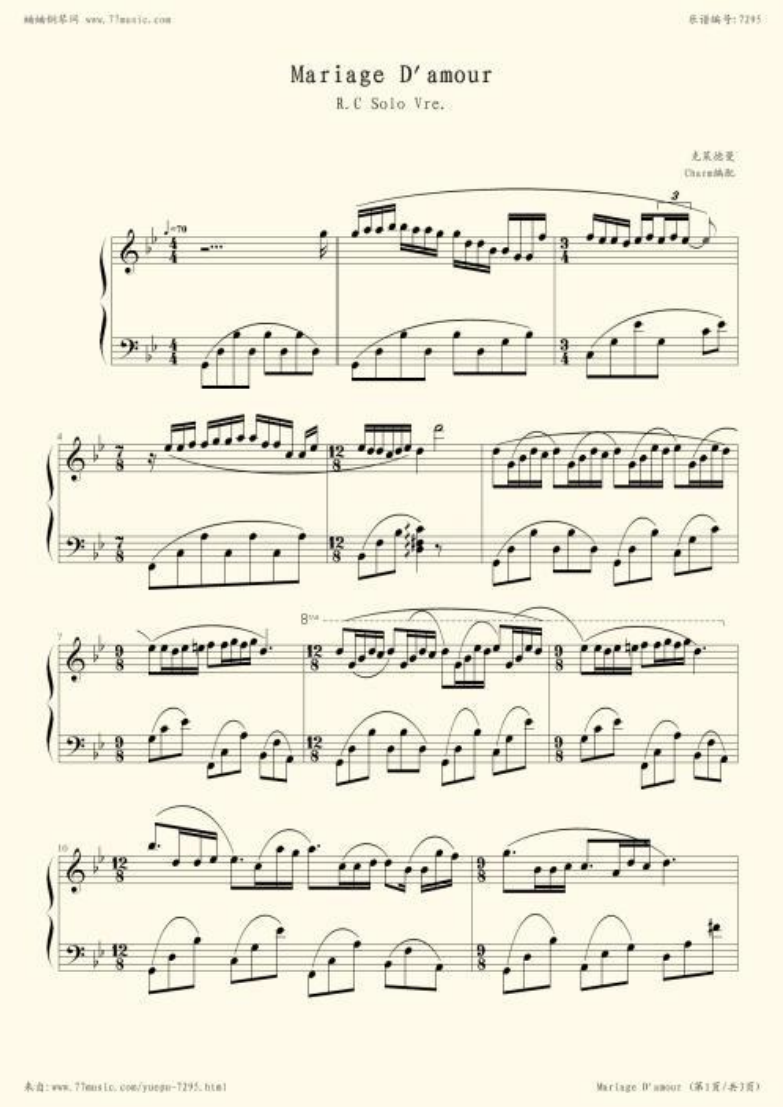
4.9 Object Oriented Programming

```
classdef mypoint2
    properties
            x;      %cordinate x
            y;      %cordinate y
        end
        methods
            function obj = mypoint2(a,b)
                if(nargin==0)
                    a = 1;
                    b = 1;
                end
                obj.x = a;
                obj.y = b;
            end
            function obj = normalize(obj)
                r = sqrt(obj.x.^2+obj.y.^2);
                obj.x = obj.x./r;
                obj.y = obj.y./r;
            end
            function plot(obj,plotstr)
                if(nargin==1)
                    plotstr = 'b*';
                end
                figure,plot(obj.x,obj.y,plotstr)
            end
        end
    end
```

4.10 Some interesting functions

- travel % Traveling salesman problem demonstration.
- vibes % Vibrating L-shaped membrane.
- teapotdemo % A demo that uses the famous Newell teapot to demonstrate MATLAB graphics features.
- life % MATLAB's version of Conway's Game of Life.
- makevase % Generate and plot a surface of revolution.
- truss % Animation of a bending bridge truss.
- fifteen % A sliding puzzle of fifteen squares and sixteen slots.
- xpquad % Superquadrics plotting demonstration.
- wrldtrv % Show great circle flight routes around the globe.
- spy penny why logo

4.10 Some interesting functions




```
fs = 8000; % sample rate
dt = 1/fs;
T16 = 0.20;
t16 = [0:dt:T16];
[temp k] = size(t16);
t4 = linspace(0,4*T16,4*k);
t8 = linspace(0,2*T16,2*k);
[temp i] = size(t4);
[temp j] = size(t8);% Modification functions
mod4 = sin(pi*t4/t4(end));
mod8 = sin(pi*t8/t8(end));
mod16 = sin(pi*t16/t16(end));

f0 = 2*220;
% reference frequency
ScaleTable = [2/3 3/4 5/6 15/16 ...
    1 9/8 5/4 4/3 3/2 5/3 9/5 15/8 ...
    2 9/4 5/2 8/3 3 10/3 15/4 4 ...
    1/2 9/16 5/8];
% 1/4 notes
do0f = mod4.*cos(2*pi*ScaleTable(21)*f0*t4);
re0f = mod4.*cos(2*pi*ScaleTable(22)*f0*t4);
mi0f = mod4.*cos(2*pi*ScaleTable(23)*f0*t4);
fa0f = mod4.*cos(2*pi*ScaleTable(1)*f0*t4);
so0f = mod4.*cos(2*pi*ScaleTable(2)*f0*t4);
la0f = mod4.*cos(2*pi*ScaleTable(3)*f0*t4);
ti0f = mod4.*cos(2*pi*ScaleTable(4)*f0*t4);
do1f = mod4.*cos(2*pi*ScaleTable(5)*f0*t4);
re1f = mod4.*cos(2*pi*ScaleTable(6)*f0*t4);
mi1f = mod4.*cos(2*pi*ScaleTable(7)*f0*t4);
fa1f = mod4.*cos(2*pi*ScaleTable(8)*f0*t4);
so1f = mod4.*cos(2*pi*ScaleTable(9)*f0*t4);
……
```

Homework 3

HW3-1. The probability that exactly k successes will occur out of a series of n independent yes/no trials, each of which has a probability of success p, is given by Equation

$$P(k;n,p)=\binom{n}{k}p^k(1-p)^{n-k}$$

where the expression $\binom{n}{k}$ is called Binomial Coefficient. The value of Binomial Coefficient is given by

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

1. Define an anonymous function to calculate $P(6;10,0.5)$
2. Write a function *Binomial_Coefficient* that will determine the probability that exactly k successes will occur out of n trials for a specified probability of success p. The function should include *two nested functions*, one to calculate the values of the binomial coefficient and one to calculate the value of $p^k(1-p)^{n-k}$. *[* Do not use built-in function of factorial]*
3. Write a script which calls above function for the value $P(6;10,0.5)$ and displays the output.
4. Call the above function again to find the Binomial Probability Sums $\left(\sum_{x=0}^{5}P(k;n,p)\right)$ and display the outputs *[* Shown Below]* where k=0-5,n=5 and p=0.6

```
Ans =

    0.0102    0.0870    0.3174    0.6630    0.9222    1.0000
```

Submit homework online before Oct 22,2019

Homework 3

HW3-2. (a) Write a function titled leap_year that takes a range from the user and returns the total number and the names of all leap years

1. If the year is evenly divisible by 4, go to step 2. Otherwise, go to step 5.
2. If the year is evenly divisible by 100, go to step 3. Otherwise, go to step 4.
3. If the year is evenly divisible by 400, go to step 4. Otherwise, go to step 5.
4. Output: the year is a leap year (it has 366 days).
5. Output: the year is not a leap year (it has 365 days).

(b)Write a script which calls your function with a range from [1952 to 2019] and displays the output .

HW3-3. Take a random row matrix with 20 elements. Use *Quick Sort* & Sort A in increasing order using *Recursion function*. (write a function that call itself inside for sorting purpose)

A=randi([-500,500],1,20)

QuickSort(https://en.wikipedia.org/wiki/Quicksort)

Submit homework online before Oct 22,2019

Thanks