

MATLAB Tutorial

(Version: MATLAB Student R2013a)

- Getting Started with MATLAB (P.2-4)
- Functions of each window (P.5-8)
- MATLAB Basics (P.9-17)
- Data Import / Export (P.18-21)
- Graphics (P.22-24)
- Loop Structures : *for* and *while* loop (P.25-26)
- Conditional Statements : *if* statements (P.27-28)
- User Written Functions : Function m-files and anonymous function (P.29-32)
- Optimization Solvers : *fminsearch*, *fminunc* and *fmincon* (P.33-42)
- Exercises (P.43-51)

MATLAB Desktop Windows

- When you start MATLAB, the desktop appears in its default layout (Figure. 1)
The desktop includes the following windows:

- ***The Command Window***

- This is where you can enter and execute MATLAB commands and display any output

- ***The Command History Window***

- This contains a list of commands issued from the command window

- ***The Workspace or Variables Window***

- The objects created during the current session are listed

- ***The Current or Working Folder Window***

- This is your project directory

MATLAB Desktop Windows (Cont'd)

■ There are three tabs across the top of the MATLAB desktop (Figure 1)

- ***HOME, PLOTS, APPS***

- *You mainly use HOME tab*

■ Clicking each tab displays the ribbon beneath the tabs (Figure. 2)

- The contents of the ribbon depend on which tab is active
- The contents are divided into groups

Ex. *HOME* tab is divided into six groups: *FILE, VARIABLE, CODE, SIMULINK, ENVIRONMENT, RESOURCES*

- You may often use *FILE* and *VARIABLE* group in the *HOME* tab
 - *FILE* group : You will find the resources necessary to manage your files
 - *VARIABLE* group : *Contains the facilities to import / save / edit data*

■ To the right of the tabs, there is a set of icons giving quick access to some functions, *HELP*, and a search field for the help documentation

Default layout of MATLAB Desktop Window

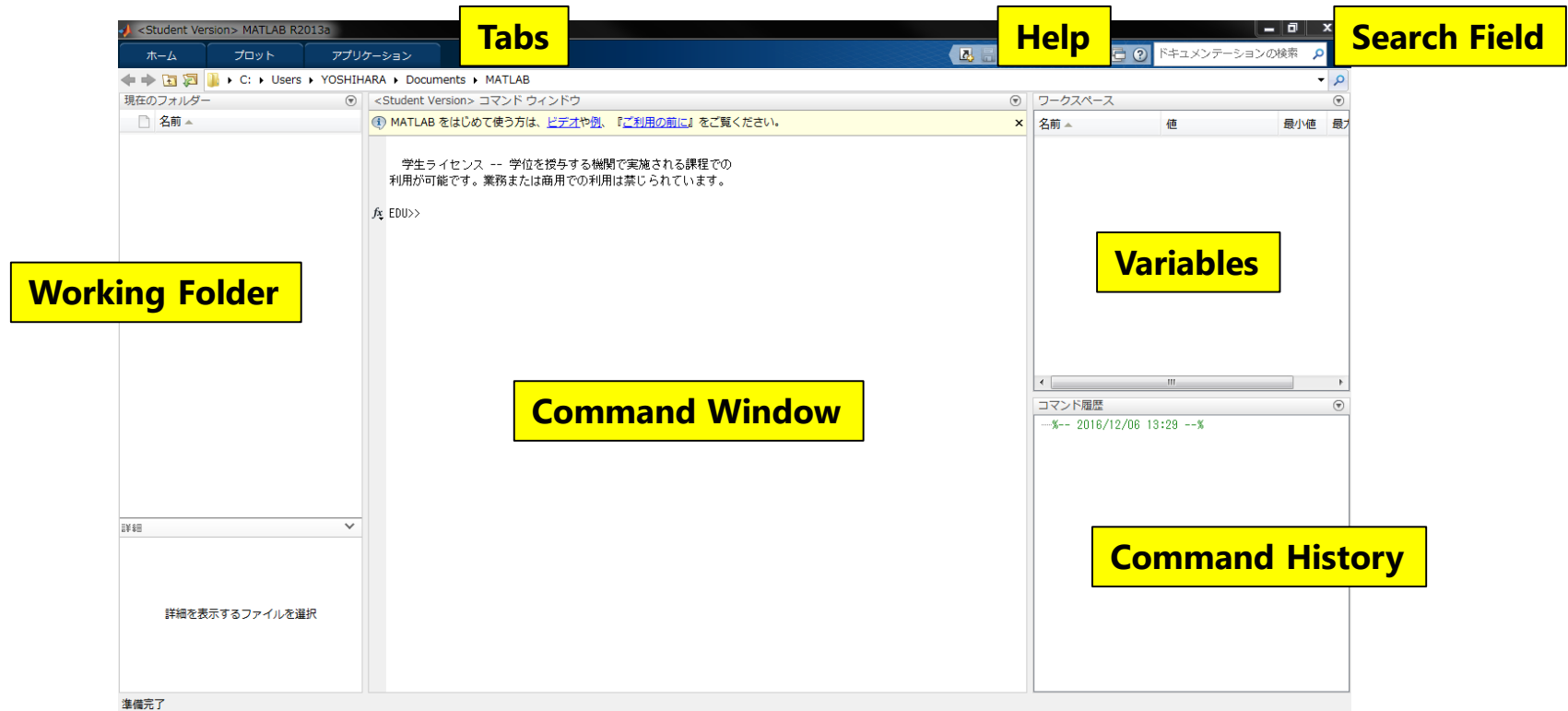


Figure. 1 Default layout

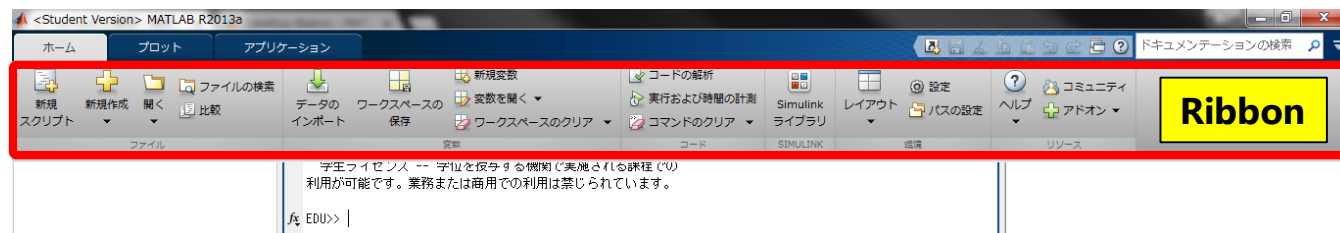


Figure. 2 Ribbon

Command Window

- It is the interface which we communicate with MATLAB
- The simplest use of the command window is as a calculator.
Most calculations are entered almost exactly as one would write them.
 - Type the following commands next to "*EDU>>*" in the command window, and press enter and check the answer

```
EDU>> 1+1  
ans = 2  
EDU>> 5*2  
ans = 10
```

```
EDU>> a = 10/2  
a = 5  
EDU>> a  
a = 5
```

```
EDU>> b = 10^2;  
EDU>> b  
b = 100  
EDU>> 1+1 ...  
+1+1  
ans = 4
```

【NOTE】

- The object "*ans*" contains the result of the last calculation
- For example, you can also create an object "*a*" that hold the result
- The arithmetic symbols $+$, $-$, $*$, $/$ and $^$ have their usual meanings
- " $=$ " is the assignment operator
- "*EDU>>*" is the MATLAB command prompt
- " $;$ " at the end of a command suppresses the output
- If a statement is not fit on one line and you want to continue it to a second line, type an ellipsis (...) at the end of the first line
- You can use the up down arrow keys to recall previous commands you typed
- You can clean up the command window by typing "*clc*"

Command History, Working Folder and Workspace

■ The Command History Window

- This contains all commands previously issued unless they are deleted
- You can execute any command in this window by double clicking it
- You can also access this window using the up down arrow keys at the MATLAB command prompt in the command window

■ The Working Folder Window

- This window displays the contents of the working directory
 - All contents for the project must be in the same directory
- The name of working directory is given in the row below tabs
- You can open any file by double clicking on the file name in this window

■ The Workspace Window

- This window contains a list of the variables already defined
- You can view and edit any variable by double clicking on it in this window
- You can erase all variables in this window by typing "*clear*" in the *command window*

Editor Window

- You can save your commands in an m-file and run the entire set or a selection of the commands in the m-file
- The MATLAB Editor has facilities editing and saving your m-file, for deleting commands or adding new commands to the file before re-running it
- You can set up a new m-file by clicking “new” in the HOME tab. Just write the following commands in the m-file, and save the file as “*vol_sphere.m*” in the current working directory.

vol_sphere.m

```
1  % This m-file calculates the volume of a sphere
2  r = 2;
3  volume = (4/3) * pi * r^3;
4  string = ['The volume of a sphere of radius' num2str(r) 'is' num2str(volume)];
5  disp(string);
```


Editor Window (Cont'd)

- Return to the command window and just type `"vol_sphere"`
- If you write down the above command correctly, MATLAB will process this as if it were a MATLAB instruction, and you can find the following statement in the command window

The volume of a sphere of radius 2 is 33.5103

- You can change the value of the radius of the sphere in the editor window and try to re-run the file
- The Editor Window is a programming text editor with various features color coded. This color coding helps you to identify errors in a program.
 - Comments are in green
 - variables and numbers in black
 - incomplete character strings in red
 - language key-words in blue

Creating Vectors and Matrices

■ The basic variable in MATLAB is an *Array*.

- The numbers can be regarded as (1×1) arrays, the column vectors as $(n \times 1)$ arrays and matrices as $(n \times m)$ arrays
- MATLAB can also work with multidimensional arrays

■ Creating vectors and matrices

```
1  x = [1, 2, 3, 4]      % 1 × 4 row vector
2  y = [1; 2; 3; 4]      % 4 × 1 column vector
3  z = [1, 2, 3; 4, 5, 6] % 2 × 3 matrix
```

```
x =
    1    2    3    4
y =
    1
    2
    3
    4
z =
    1    2    3
    4    5    6
```

- The matrix is entered row by row. ";" in the bracket means a line feed
- The elements in a row are separated by spaces or commas
- Anything after "%" is regarded as a comment which is not executed by MATLAB

Basic Matrix Operations

1 $x = [1, 2; 3, 4]$

2 $y = [5, 6; 7, 8]$

$x =$

1 2
3 4

$y =$

5 6
7 8

3 $x + y$

4 $x - y$

5 $x * y$

$ans =$

6 8
10 12

$ans =$

-4 -4
-4 -4

$ans =$

19 22
43 50

6 $x .* y$

7 $x ./ y$

$ans =$

5 12
21 32

$ans =$

0.2000 0.3333
0.4286 0.5000

- The +, -, * and / operators do standard matrix addition, subtraction, multiplication and division respectively.
- The dot operators .* and ./ provide element by element operations
- Be careful with the dimensions of matrices when you do matrix operations

Matrix Operations with Scalar and Exponents

1	<code>a = 2;</code>						
2	<code>x = [1, 2; 3, 4];</code>						
3	<code>z = [1, 2; 3, 4];</code>						
4							
5	<i>% Matrix Operations with Scalar</i>						
6	<code>x + a</code>	<code>ans =</code>		<code>ans =</code>		<code>ans =</code>	
7	<code>x - a</code>						
8	<code>x * a</code>	3 4	-1 0	2 4	0.5000	1.0000	
9	<code>x / a</code>	5 6	1 2	6 8	1.5000	2.0000	
10							
11	<i>% Exponents</i>						
12	<code>x ^ a</code>	<code>ans =</code>		<code>ans =</code>		<code>ans =</code>	
13	<code>x.^ a</code>	7 10	1 4	1 4			
14	<code>x.^ z</code>	15 22	27 256	9 16			

- Adding a scalar to or multiplying a scalar by a matrix does the operation on each element of the matrix
- It is possible to
 - raise a matrix to some power ("`x^a`" above)
 - raise each element of a matrix to a power ("`x.^a`" above)
 - raise the elements to specified powers ("`x.^z`" above)

Handling Parts of a Matrix

1	<code>x = [1, 2, 3];</code>	
2	<code>y = [1, 2, 3; 4, 5, 6];</code>	
3		
4	<code>x(2)</code>	<code>ans = 2</code>
5	<code>x(1 : 2)</code>	<code>ans = 1 2</code>
6		
7	<code>y(2, 3)</code>	<code>ans = 6</code>
8	<code>y(1, 2 : 3)</code>	<code>ans = 2 3</code>
9	<code>y(2, :)</code>	<code>ans = 4 5 6</code>

- You can extract any elements of a vector or a matrix as above
 - For example, "`y(2, 3)`" extracts (2, 3)-element of a matrix `y`
- ":" operator generates a sequence of coordinates to be extracted from a matrix
- Using ":" on its own extracts the entire column or row
- You can also assign any values to certain elements of a matrix
 - For example, you can assign 10 to (2, 3)-element by the command "`y(2, 3) = 10`"

Stacking Matrices

```

1  x = [1, 2; 3, 4]
2  y = [5, 6, 7; 8, 9, 10]
3  z = [0, 1; 1, 0; 0, 1]
4  -----
5  % Stacking Matrices
6  a = [x, y]
7  b = [x; z]

```

$x =$		$y =$			$z =$	
1	2	5	6	7	0	1
3	4	8	9	10	1	0
					0	1

$a =$					$b =$	
1	2	5	6	7	1	2
3	4	8	9	10	3	4
					0	1
					1	0
					0	1

- You can stack some matrices and create a new matrix
 - Be careful with the dimensions of matrices to be stacked
- You can also use the commands "*cat*", "*horzcat*" and "*vertcat*" to stack matrices. For more details, see MathWorks support page.

Special Matrices

```
1  x = eye(2)
2  y = ones(3)
3  z = zeros(1, 4)
```

```
x =
    1    0
    0    1
```

```
y =
    1    1    1
    1    1    1
    1    1    1
```

```
z =
    0    0    0    0
```

- `"eye(n)"` returns an $n \times n$ identity matrix
- `"ones(n, m)"` returns an $n \times m$ matrix with all elements equal to 1.
Just type `"ones(n)"` returns an $n \times n$ matrix with all elements equal to 1.
- `"zeros(n, m)"` returns an $n \times m$ matrix with all elements equal to 0.

Mathematical Functions

■ Mathematical functions are applied on an element by element basis

- The following functions are some of commonly used mathematical functions
- For more details of the mathematical functions, see MathWorks support page
 - `"pi"` returns the value of $\pi=3.1415\dots$
 - `"exp(A)"` returns the exponential e^A for each element of a matrix A
 - `"exp(1)"` returns the value of $e=2.7183\dots$
 - `"log(A)"` returns the natural logarithm $\ln(A)$ of each element of a matrix A
 - `"sqrt(A)"` returns the square root of each element of a matrix A
 - `"inv(A)"` returns the inverse matrix of A
 - `"sum(A)"` returns a row vector containing the sum of each column when A is a matrix. It returns the sum of the elements when A is a row or column vector.
 - `"sum(A, N)"` returns the sum along dimension N . For example, if A is a matrix, `"sum(A, 2)"` returns a column vector containing the sum of each row.
 - `"diag(A)"` returns the diagonal elements when A is a square matrix. It returns a matrix with diagonal elements A and zeros elsewhere when A is a column vector

Sequences

```

1  [0 : 2 : 10]
2  [3 : 2 : 8]
3  [1 : 5]
4  [1 : 3]'
```

```
ans =
    0    2    4    6    8   10
```

```
ans =
    3    5    7
```

```
ans =
    1    2    3    4    5
```

```
ans =
    1
    2
    3
```

- "*[first : increment : last]*" returns a row vector whose elements are a sequence with first element "*first*", second element "*first+increment*" and continues while the new entry is less than "*last*"
- If only two numbers are specified, the increment is assumed to be 1
- The transpose of a matrix is denoted by a quote mark (*A'* is the transpose of *A*)

Random Number Generators

```
1  rng(1)
2  x = rand(1, 4)
3  y = randn(3, 3)
```

$x =$

0.4170 0.7203 0.0001 0.3023

$y =$

-0.7585 -0.5727 -0.1969
-1.1096 -0.5587 0.5864
-0.8456 0.1784 -0.8519

- The basic random number generator is "*rand()*" which generates pseudo uniform random numbers in the range $[0, 1]$
- "*randn()*" generates random numbers which follow a standard normal distribution $N(0, 1)$
- By "*rng(seed)*" command, it is possible to "seed" the random number generator where "seed" is an integer between 0 and 2^{32}
 - This enables you to produce replicable simulations using random numbers

Import from Excel format files

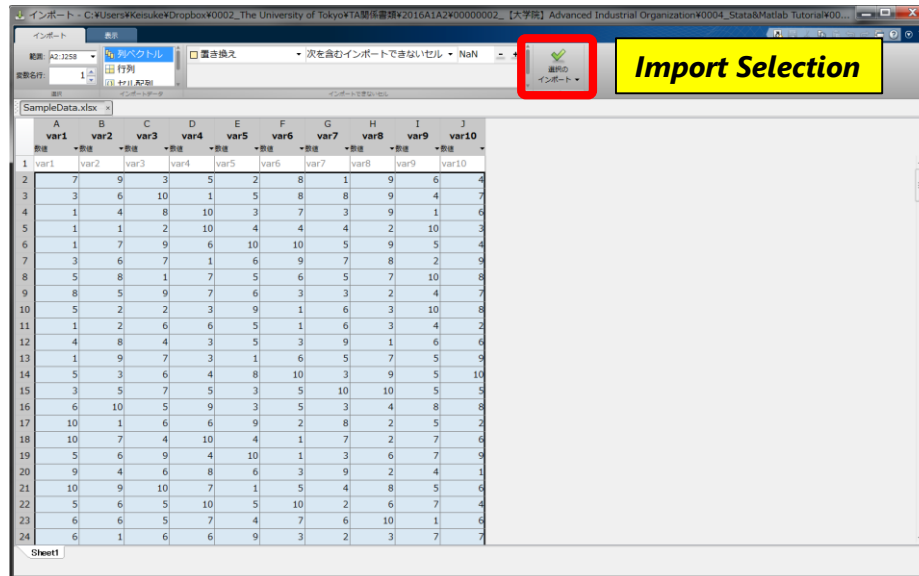
- The simplest way to import the data is just clicking "*Import Data*" in the *HOME* tab and navigate to the data file and open it

HOME tab



Figure. 3 Import Data

- You will be presented with a *Data Import Window* like Figure. 3 below



- In this window, you can
 - specify the range of data to be imported
 - specify which row contains the variable names
 - import the data as column vectors or a single matrix
 - specify how to deal with unimportable cells
- After specifying these items, *click the "Import Selection"* and the data are imported

Figure. 4 Data Import Window

Import from CSV format files

■ Importing from Excel format files (Cont'd)

- You can also import the data by using "*xlsread*" command in the command window

```
Data = xlsread('SampleData.xlsx')
```

■ Importing from text files

- If your data are in comma separated value (CSV) format, the same procedure allows you to import the data
- There are additional options available that allow you to specify the delimiter, combine delimiters and specify decimal point
- You can also use "*csvread*" command in the command window to import the data (The data file must contain only numerical values)

```
Data = csvread('SampleData.csv')
```

Export to Excel and CSV files

■ Export to Excel format files

- You can export the data to Excel format files by using "*xlswrite*" command
- The following command writes the matrix "*Data*" to the Excel format file "*SampleDataExport.xls*" in the current working directory

```
xlswrite('SampleDataExport.xls', Data);
```

- If A is $n \times m$ matrix, the numeric values in the matrix are written to the first n rows and m columns in the first sheet in the spreadsheet

■ Export to CSV format files

- You can export the data to CSV format files by using "*csvwrite*" command
- The following command writes the matrix "*Data*" to the CSV format file "*SampleDataExport.csv*" in the current working directory

```
csvwrite('SampleDataExport.csv', Data)
```

MATLAB specific data format “.mat”

■ You can also use “.mat” format which is MATLAB specific binary format

- It is not easily readable in most other softwares
- This file can not be examined in a text editor since it is a binary format

■ You can save the contents (variables) of the workspace by using “save” command

- The following command saves all contents of the workspace in the file “*filename.mat*” in the current working directory

```
save('filename')
```

- You can select the variables to be saved.
For example, the following command saves *var1* and *var2* in the file “*filename.mat*”

```
save('filename', 'var1', 'var2')
```

■ Similarly, the commands “load('filename')” and “load('filename', 'var1', 'var2')” load the contents or the specified variables of “*filename.mat*” into the workspace

Basic Plotting Function: plot

■ The basic plotting command in MATLAB is “*plot*” command

- “*plot(x)*” plots the data in *x* against versus its index
- “*plot([x, y])*” plots the data in *x* and *y* against versus their index in the same graph
 - If the input is a matrix, “*plot*” plots the data of each column against versus their index

Example of “*plot*” command

```
1  x = randn(50, 1);  
2  y = randn(50, 1);  
3  plot(x) ----- Figure. 5  
4  plot([x, y]) ----- Figure. 6
```

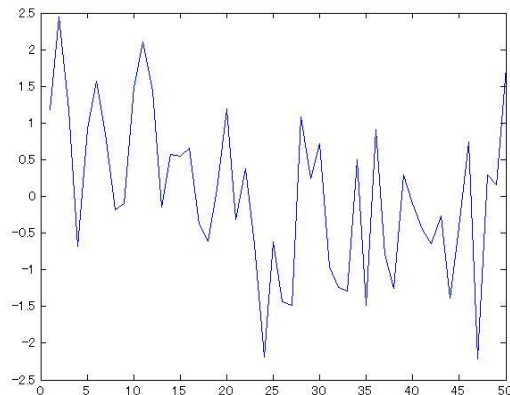


Figure. 5 Plot 1

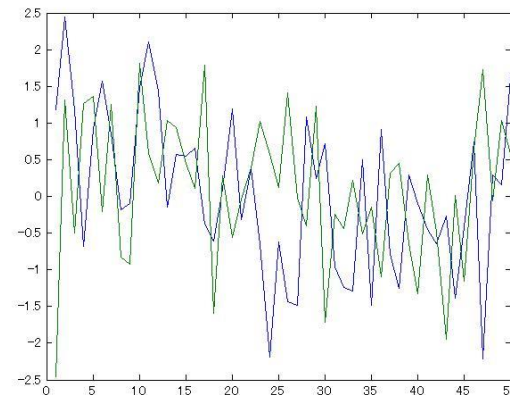


Figure. 6 Plot 2

Histogram

■ To draw a histogram, you can use *"hist"* command

- *"hist(x)"* draws a 10-bin histogram of the vector *x*
- *"hist(x, m)"* draws a *m*-bin histogram of the vector *x* (*m* is a scalar)
- *"hist(x, c)"* draws a histogram of the vector *x* using the bins specifies in *c* (*c* is a row vector)

Example of drawing a histogram

1	<code>x = randn(5000, 1);</code>		
2	<code>hist(x)</code>	-----	Figure. 7
3	<code>hist(x, 100)</code>	-----	Figure. 8
4	<code>hist(x, -4.0:0.2:4.0)</code>	-----	Figure. 9

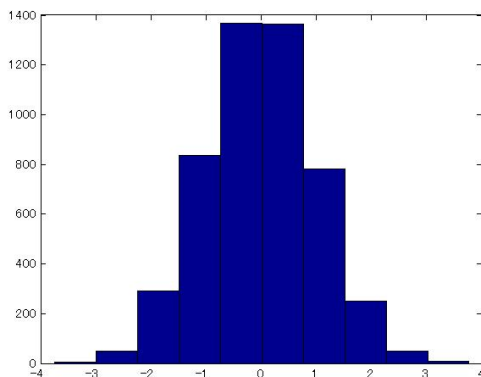


Figure. 7 Histogram 1

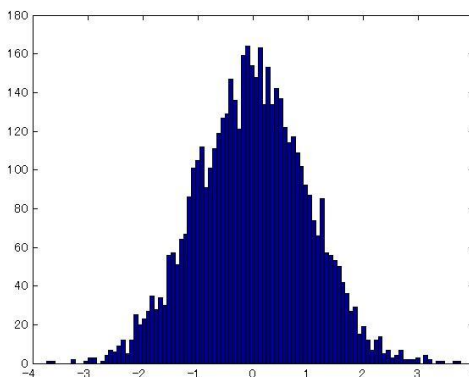


Figure. 8 Histogram 2

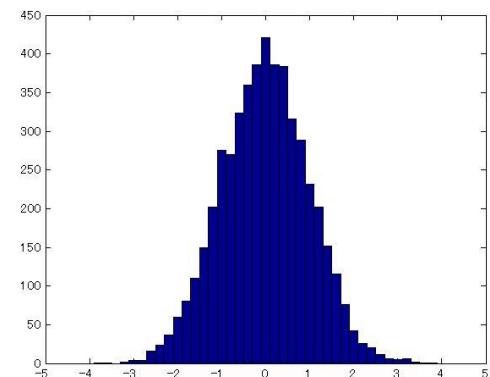


Figure. 9 Histogram 3

Scatter Plot

■ To draw a scatter plot, you can use "*scatter*" command

- "*scatter(x, y)*" plots the data in x as horizontal axis and in y as vertical axis

Example of scatter plot

```
1  x = randn(50, 1);  
2  y = randn(50, 1);  
3  scatter(x, y, 'rx')  
4  xlim([-5, 5])  
5  ylim([-5, 5])  
6  title('Scatter Plot')  
7  xlabel('x')  
8  ylabel('y')
```

Range of each axis

Figure title, axis label

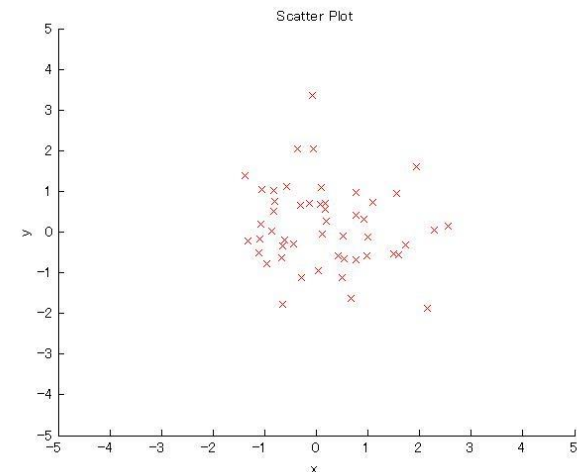


Figure. 10 Scatter Plot

【NOTE】

- You can specify various options like color, marker symbol, line style and so on
 - In the above example, 'rx' at the third input argument in "*scatter*" means red and x-mark
- You can also specify the range of each axis, figure title, axis label as in the above example
- For more details of plotting functions, see MathWorks support page

for loop

■ You often need to run the same commands a number of times

- You can use "*for*" and "*while*" loop to achieve such a goal

for loop

The basic form of for loop is

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

- The *expression* is usually a row vector
- The statements are processed for each of the values in expression
- Used when one knows when and how many times an operation will be repeated (cf. *while* loop)

Example of for loop

```
1  for i = 1:5
2      x(i) = i;
3  End
4  x
```

x = 1 2 3 4 5

Example of nested for loop

```
1  for i = 1:3
2      for j = 1:3
3          sum = i + j;
4          fprintf('%d + %d = %d %n', i, j, sum);
5      end
6  end
```

```
1 + 1 = 2
1 + 2 = 3
1 + 3 = 4
2 + 1 = 3
2 + 2 = 4
2 + 3 = 5
3 + 1 = 4
3 + 2 = 5
3 + 3 = 6
```

※ By "*fprintf*" command, MATLAB displays multiple variable values on the same line in the command window. "*disp*" command has the same function. "*%n*" indicates that the cursor will move to the next line.

while loop

while loop

The basic form of *while* loop is

```
while conditions  
    statements  
end
```

- Same basic functionality as the *for* loop
- The statements are repeated so long as the conditions are true (cf. *for* loop)

Example of *while* loop

```
1  a = 0;  
2  while a < 10  
3      a = a + 1;  
4      fprintf('a = %d \n', a);  
5  end
```

```
a = 1  
a = 2  
a = 3  
a = 4  
a = 5  
a = 6  
a = 7  
a = 8  
a = 9  
a = 10
```

【NOTE】

- You can mandatorily abort the iteration by typing "*Ctrl+c*" in the command window
- You should use matrix statements in preference to the loop because the former is more efficient and easier to use.

if statements

- You often need to run commands if certain conditions hold
 - You can use "if" command to achieve such a goal

if statements : The basic form of *if* statement is

```
if conditions
  statements
end
```

- The *statements* are only processed if the *conditions* are true

```
if conditions
  statements1
else
  statements2
end
```

- The *statements1* are processed if *conditions* are true, and the *statements2* if false

```
if conditions1
  statements1
elseif conditions2
  statements2
else
  statements3
end
```

- The *statements1* are processed if *conditions1* are true, the *statements2* are processed if *conditions2* are true, and otherwise, the *statements3* are processed

if statements (Cont'd)

■ The *conditions* above can include the following operators

`==` : equal

`&&` : logical and

`~=` : not equal

`||` : logical or

`<` : less than

`xor` : logical exclusive or

`>` : greater than

`all` : true if all elements of vector are nonzero

`<=` : less than or equal to

`any` : true if any element of vector is nonzero

`>=` : greater than or equal to

Example of *if* statement with *for* loop

```
1  x = (-1) + 2 * rand(5, 1);
2  for i = 1 : 5
3      if x(i) > 0
4          d(i) = 1;
5      else
6          d(i) = 0;
7      end
8      fprintf('d(%d) = %d \n', i, d(i));
9  end
```

```
d(1) = 1
d(2) = 0
d(3) = 1
d(4) = 0
d(5) = 1
```

Function m-files

- One of the most useful facilities in MATLAB is the facility to write one's own functions and use them in the same way as official functions in MATLAB
 - The following m-file "*normaldensity.m*" is an example of user written function which estimates the density function of a normal distribution

$$\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$


normaldensity.m

```
1  function f = normaldensity(z, mu, sigma)
2
3  % This function calculates the density function of the normal distribution
4  % with mean "mu" and standard deviation "sigma" at a point "z"
5
6  % "sigma" must be a positive non-zero real number
7  if sigma <= 0
8      fprintf('Invalid input\n');
9      f = NaN;
10 else
11     f = (1 / (sqrt(2 * pi) * sigma)) * exp(-(z - mu)^2 / (2 * sigma^2));
12 end
```

Function m-files: Note

- The function m-file starts with keyword "*function*" which indicates that this m-file is a function m-file
- "*f*" in the first line is the value that will be returned when the file has been run
- It is possible to have more than one function in a function file.
Only the first function (main function) in the file is visible to the calling program.
Second and subsequent function can be seen by the main function.
- The name of the function m-file must be the same as the name in the first line

normaldensity.m *function f = normaldensity(z, mu, sigma)*



Must be the same name

- The function m-file must be saved in the current working directory

How to use function m-files

■ How to use the function "*normaldensity*"

- In the command window, type "*normaldensity(z, mu, sigma)*" with given values for *z*, *mu* and *sigma*
- For example, if you want to evaluate the standard normal density function at zero, just type "*normaldensity(0, 0, 1)*" and you can obtain the following result

```
EDU>> normaldensity(0, 0, 1)  
ans = 0.3989
```

- In addition, if you want to plot the standard normal density function, just type the following command and you can obtain a figure

```
fplot('normaldensity(x, 0, 1)', [-5, 5])
```

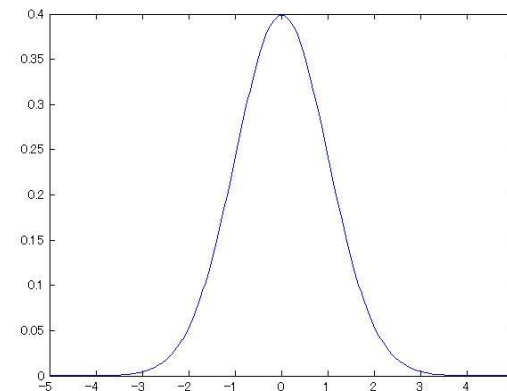


Figure. 11 Standard Normal Density Function

Anonymous functions

■ You can define your own function without maintaining a separate m-file

- The first line in the box below shows how to define the normal distribution function in the previous example as an anonymous function

```
1  f = @(z, mu, sigma) (1 / (sqrt(2 * pi) * sigma)) * exp(-(z - mu)^2 / (2 * sigma^2));  
2  f(0, 0, 1)
```

```
ans = 0.3989
```

【NOTE】

- "*f*" in the first line is a function handle for the anonymous function.
This can be passed as an input argument to another function
 - For example, in the maximum likelihood estimation, you can pass the likelihood function defined as the anonymous function to a general optimization routine
- "@" constructs the function. The parentheses "()" immediately after "@" contain the input arguments of the following function

Unconstrained Optimization Solver: `fminsearch`

- "`fminsearch`" finds the minimum of unconstrained multivariable function using derivative-free method

- "`fminsearch`" uses the simplex search method which is a direct search method that does not use numerical or analytic gradients

- Nonlinear programming solver that searches for the minimum of a problem specified by

$$\min_x f(x) \quad \text{where } f(x) \text{ is a function that returns a scalar and } x \text{ is a vector or a matrix}$$

- Syntax

$[x, fval, exitflag, output] = fminsearch(fun, x0, options)$

【Input Arguments】

- `fun` : Function to minimize
- `x0` : Initial point
- `Options` : Optimization options

【Output Arguments】

- `x` : Solution
- `fval` : Objective function value at solution
- `exitflag` : Reason "`fminsearch`" stopped
- `output` : Information about the optimization process

※ For more details of each argument, see MathWorks support page

Unconstrained Optimization Solver: `fminsearch` (Cont'd)

■ Examples : Using an anonymous function

- Consider minimizing the following function which is minimized at the point $x = [1, 1]$ with minimum value 0

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

- Set the initial point to $x_0 = [0, 0]$ and minimize using "`fminsearch`"

```
1 fun = @(x) 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;  
2 x0 = [0, 0];  
3 [x, fval] = fminsearch(fun, x0);  
4 x  
5 fval
```

```
x = 1.0000    1.0000  
fval = 3.6862e-10
```

Unconstrained Optimization Solver: `fminsearch` (Cont'd)

■ Examples : Using a function m-file

- Consider the same problem above, but here, the value of an objective function is given by executing the following m-file

Obj_fminsearch.m

```
1  function f = Obj_fminsearch(x)
2  f = 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;
```

```
1  x0 = [0, 0];
2  [x, fval] = fminsearch(@Obj_fminsearch, x0);
3  x
4  fval
```

```
x = 1.0000    1.0000
fval = 3.6862e-10
```

Unconstrained Optimization Solver: `fminunc`

- `"fminunc"` also finds the minimum of unconstrained multivariable function, but you can use the information of derivatives
 - You can choose trust-region (default) or quasi-newton algorithm in `"fminunc"`
- Nonlinear programming solver that solves the same problem as `"fminsearch"`
- Syntax

`[x, fval, exitflag, output, grad, hessian] = fminunc(fun, x0, options)`

 - Input and output arguments except for `"grad"` and `"hessian"` are the same as in `"fminsearch"`
 - `grad` : Gradient at the solution
 - `hessian` : Approximate Hessian matrix
- Examples : Supplying the gradient
 - Consider the same problem as in `"fminsearch"`, but here, try to minimize the objective function using the information of derivatives
 - `"fminunc"` can be faster and more reliable when you provide derivatives

Unconstrained Optimization Solver: fminunc (Cont'd)

■ Examples : Supplying the gradient (Cont'd)

- The objective function $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ has the following gradient

$$\nabla f(x) = \begin{pmatrix} -400(x_2 - x_1^2)x_1 - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{pmatrix}$$

- You can specify the gradient in addition to the objective function in the function m-file as follows

Obj_fminunc.m

```

1  function [f, g] = Obj_fminunc(x)
2
3  % Objective function
4  f = 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;
5
6  % Gradient
7  if nargin > 1
8      g = [-400 * (x(2) - x(1)^2) * x(1) - 2 * (1 - x(1));
9           200 * (x(2) - x(1)^2)];
10 end

```

"nargout" is the number of function output arguments. If the gradient is required by "optimoptions" below, the commands in line 8 and 9 is executed

Unconstrained Optimization Solver: fminunc (Cont'd)

■ Examples : Supplying the gradient (Cont'd)

```

1  x0 = [0, 0];
2  options = optimoptions('fminunc', 'Algorithm', 'trust-region', 'GradObj', 'on', 'Display', 'iter');
3  [x, fval] = fminunc(@Obj_fminunc, x0, options);
4  x
5  fval

```

You can specify algorithm here

You can specify whether you use gradient or not

Iteration	$f(x)$	Norm of step	First-order optimality	CG-iterations
0	1		2	
1	1	1	2	1
2	0.953125	0.25	12.5	0
	⋮			
14	1.04121e-05	0.0322022	0.0759	1
15	1.2959e-08	0.00556537	0.00213	1
16	2.21873e-14	0.000223722	3.59e-06	1

You can view iterations by 'Display', 'iter' option above

Local minimum possible.

fminunc stopped because the final change in function value relative to its initial value is less than the default value of the function tolerance.

<stopping criteria details>

$x = 1.0000 \quad 1.0000$
 $fval = 2.2187e-14$

Constrained Optimization Solver: `fmincon`

- “`fmincon`” finds the minimum of constrained nonlinear multivariable function
 - You can choose several algorithms like interior-point (default), `sqp`, and so on
- Nonlinear programming solver that searches for the minimum of a problem specified by

$$\begin{array}{lll}
 \min_x f(x) & \text{s.t.} & c(x) \leq 0 \quad \text{Nonlinear Inequality Constraints} \\
 & & ceq(x) = 0 \quad \text{Nonlinear Equality Constraints} \\
 & & A \cdot x \leq b \quad \text{Linear Inequality Constraint} \\
 & & Aeq \cdot x = beq \quad \text{Linear equality Constraint} \\
 & & lb \leq x \leq ub \quad \text{Bound Constraint}
 \end{array}$$

■ Syntax

```
[x, fval, exitflag, output, lambda, grad, hessian]
= fmincon(fun, x0, A, b, Aeq, beq, lb, ub, nonlcon, options)
```

- You have to specify some arguments in terms of constraints as input arguments
- Output arguments except for “`lambda`” are the same as in “`fminunc`”
 - `lambda` : Lagrange multipliers at the solution

Constrained Optimization Solver: fmincon (Cont'd)

■ Examples : Linear Inequality and Equality Constraints

- Consider the following constrained minimization problem

$$\min_x f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

$$\text{s.t.} \quad x_1 + 2x_2 \leq 1$$

$$2x_1 + x_2 = 1$$

$$\left\{ \begin{array}{l} \frac{(1, 2)}{A} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \frac{1}{b} \\ \frac{(2, 1)}{Aeq} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \frac{1}{beq} \end{array} \right.$$

```

1 fun = @(x) 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;
2 x0 = [0.5, 0];
3 A = [1, 2];
4 b = 1;
5 Aeq = [2, 1];
6 beq = 1;
7 [x, fval] = fmincon(fun, x0, A, b, Aeq, beq);
8 x
9 fval

```

```

x = 0.4149  0.1701
fval = 0.3427

```

Constrained Optimization Solver: fmincon (Cont'd)

■ Examples : Bound and Nonlinear Inequality Constraints

- Consider minimizing the same objective function above subject to bound constraints, and within a circle centered at $[1/3, 1/3]$ with radius $1/3$

$$\min_x f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

$$\text{s.t.} \quad 0 \leq x_1 \leq 0.5, \quad 0.2 \leq x_2 \leq 0.8$$

$$\left(x_1 - \frac{1}{3}\right)^2 + \left(x_2 - \frac{1}{3}\right)^2 \leq \frac{1}{3}^2$$

- You have to specify the nonlinear constraints in the function m-file as follows.

CircleConst.m

```
1 function [c, ceq] = CircleConst(x)
2 c = (x(1) - 1/3)^2 + (x(2) - 1/3)^2 - (1/3)^2;
3 ceq = [];
```

No nonlinear equality constraints

Constrained Optimization Solver: `fmincon` (Cont'd)

■ Examples : Bound and Nonlinear Inequality Constraints (Cont'd)

```
1  fun = @(x) 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;
2  x0 = [1/4, 1/4];
3  A = [];
4  b = [];
5  Aeq = [];
6  beq = [];
7  lb = [0, 0.2];
8  ub = [0.5, 0.8];
9  nonlcon = @CircleConst;
10 [x, fval] = fmincon(fun, x0, A, b, Aeq, beq, lb, ub, nonlcon);
11 x
12 fval
```

No linear inequality and equality constraints

Bound Constraints

Nonlinear constraints specified in "*CircleConst.m*"

$x = 0.5000 \quad 0.2500$
 $fval = 0.2500$

【NOTE】

- You can specify various options, choose a specific algorithm, and also provide gradient for faster and more reliable computations as we did in "*fminunc*" example

Computing OLS Estimator: What to do

- In exercise 1, we try to compute the OLS estimator by the following three different methods, and check whether you can obtain the same estimates for β_0 , β_1 and β_2

(I) Using analytical formula

$$\beta_0 = (X'X)^{-1}X'y$$

(II) Using MATLAB optimization solver "*fminsearch*"

(II)-1 Using matrix operation to compute the objective function (sum of squared residuals)

$$\beta_1 = \operatorname{argmin}_{\beta} (y - X\beta)'(y - X\beta)$$

(II)-2 Using "*for*" loop to compute the objective function

$$\beta_2 = \operatorname{argmin}_{\beta} \sum_{i=1}^n (y_i - X_i\beta)^2$$

Computing OLS Estimator: How to do

■ How to implement

Step.1: Create "*main.m*" file and write the codes from page 45 to 47

Step.2: After that, create "*func1.m*" and "*func2.m*" files,
and write the codes in page 48, respectively

Step.3: Then, execute the program by typing "*main*" in the command window

- You can also execute the program by copying and pasting the program in "*main.m*" in the command window
- If you correctly write the programs, you can obtain the same values for variables "*beta0*", "*beta1*" and "*beta2*".
Check whether you obtain the same results.

Comments

- You MUST put three ".m" files in the same directory
- You need NOT to execute the programs in "*func1.m*" and "*func2.m*" because the codes (line 48 and 57) in "*main.m*" automatically call these files, respectively. You can execute whole programs just by executing the program in "*main.m*" file.

MATLAB Codes

main.m

```

1  %=====
2  % Exercise: Compute the OLS Estimator
3  %=====
4
5  % Variable declaration
6  global n y X K
7
8  % Load dataset
9  DATA = load('DatasetForExercise.csv');
10
11 % Substitute the first column of "DATA" into "y": Dependent variable
12 y = DATA(:,1);
13
14 % Substitute the second to fifth column of "DATA" into "x1"
15 x1 = DATA(:,2:5);
16
17 % Substitute the number of rows of "DATA" into "n"
18 n = size(DATA,1);
19

```

※ "size(A)" returns the number of elements in each dimension of a matrix.

"size(A, N)" returns the number of rows of a matrix A if $N = 1$ and columns if $N = 2$.

MATLAB Codes (Cont'd)

main.m (Cont'd)

```
20 % Create a explanatory variable matrix "X"
21 % with "X1" and the vector that all elements are 1(constant term)
22 X = [x1, ones(n,1)];
23
24 % Divide the second column of "X" by 10
25 X(:,2) = X(:,2)/10;
26
27 % Divide the third column of "X" by 1000
28 X(:,3) = X(:,3)/1000;
29
30 % Substitute the number of columns of "X" into "K"
31 K = size(X,2);
32
33 %=====
34 % ① OLS Estimates by analytical formula: Regress "y" on "X"
35 %=====
36 beta0 = inv(X'*X)*X'*y;
37
```

MATLAB Codes (Cont'd)

main.m (Cont'd)

```
38 %=====
39 % ② OLS Estimates by matlab optimization solver "fminsearch"
40 %    using matrix operation to compute the objective function in "func1.m" file
41 %=====
42
43 % Initial values for optimization
44 initial = ones(5, 1);
45
46 % Minimize the objective function using "fminsearch"
47 options = optimset('Display','iter');
48 [beta1, f1] = fminsearch(@func1, initial, options);
49
50 %=====
51 % ③ OLS Estimates by matlab optimization solver "fminsearch"
52 %    using "for" loop to compute the objective function in "func2.m" file
53 %=====
54
55 % Minimize the objective function using "fminsearch"
56 options = optimset('Display','iter');
57 [beta2, f2] = fminsearch(@func2, initial, options);
```


MATLAB Codes (Cont'd)

func1.m

```
1  function f = func1(beta)
2
3  % Variable declaration
4  global y X
5
6  % Compute the value of objective function (sum of squared residuals)
7  residual = y - X * beta;
8  f = residual' * residual;
```

func2.m

```
1  function f = func2(beta)
2
3  % Variable declaration
4  global n y X
5
6  % Compute the value of objective function (sum of squared residuals)
7  f = 0;
8  for i = 1:n
9      sq_resid = (y(i)-X(i,:) * beta)*(y(i)-X(i,:) * beta);
10     f = f + sq_resid;
11 end
```

Estimating Discrete Choice Model

■ In exercise 2, we try to estimate a discrete choice model by maximum likelihood

- Assume that the utility function for consumer i purchasing product j

$$u_{ij} = \mathbf{X}_{ij}\boldsymbol{\beta} + \varepsilon_{ij} = \beta_0 + \beta_1 d_i^1 x_j^1 + \beta_2 d_i^2 x_j^2 + \varepsilon_{ij}$$

where

- $\mathbf{d}_i = (d_i^1, d_i^2)$: Consumer characteristics ($i=1, \dots, 10,000$)
- $\mathbf{x}_j = (x_j^1, x_j^2)$: Product characteristics ($j=0, 1, \dots, 3$) ※ $j=0$: Outside goods
- ε_{ij} : i.i.d. utility shock which follows Type I extreme value distribution
- $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2)$: Parameters to be estimated

- Estimate the model by maximum likelihood

HINT

The likelihood function is written by

$$\prod_i \prod_j \left[\frac{\exp(\mathbf{X}_{ij}\boldsymbol{\beta})}{1 + \sum_k \exp(\mathbf{X}_{ik}\boldsymbol{\beta})} \right]^{y_{ij}}$$

where $y_{ij} = 1$ if a consumer i purchases a product j , and $y_{ij} = 0$ otherwise.

You can maximize the likelihood function by "*fminsearch*" or "*fminunc*".

MATLAB Codes

main.m

```

1  %=====
2  % Discrete Choice Model
3  %=====
4
5  % Variable declaration
6  global N d1 d2 x1 x2 choicedum
7
8  % Load Dataset
9  load ChoiceData.mat           % Choice data, Consumer characteristics
10 load ProductCharacteristics.mat % Product characteristics
11
12 N = size(ConsumerID, 1);
13 Choice = Choice + 1;
14 choicedum = dummyvar(Choice);
15
16 % Initial Value of parameters (Estimates may depend on initial values...)
17 init = [0; 0; 0];
18
19 % Maximize the likelihood function
20 options = optimoptions(@fminunc, 'Algorithm', 'Quasi-Newton', 'Display', 'iter', ...
21     'MaxFunEvals', 10000, 'TolFun', 1e-6, 'TolX', 1e-6);
22 [beta, Obj, exitflag, output, grad, hessian] = fminunc(@obj, init, options);
23
24 % Standard Error of MLE (Approximation) when using fminunc
25 se = sqrt( diag( inv( hessian ) ) );

```

MATLAB Codes

obj.m

```
1  function f = obj(theta)
2
3  % Variable declaration
4  global N d1 d2 x1 x2 choicedum
5
6  X1 = [zeros(N, 1), d1 * x1(1), d2 * x2(1)]; % product0 (outside goods)
7  X2 = [ones(N, 1), d1 * x1(2), d2 * x2(2)]; % product1
8  X3 = [ones(N, 1), d1 * x1(3), d2 * x2(3)]; % product2
9  X4 = [ones(N, 1), d1 * x1(4), d2 * x2(4)]; % product3
10
11 % Exponential mean utility for each consumer, for each product
12 expdelta = [exp(X1 * theta), exp(X2 * theta), exp(X3 * theta), exp(X4 * theta)];
13
14 % Objective function (log-likelihood function)
15 f = - sum( sum( log( (expdelta ./ repmat(sum(expdelta, 2), 1, 4)) .^ choicedum ), 2 ) );
```