# MATLAB Mini Course*

Kyle Handley

*University of Maryland - College Park*

September 3, 2009

**Place:** Experimental Economic Lab (EEL) Tydings 4104

**Time:** 2-5:30 pm, Sept 1, 3 and 8

**Office Hours:** None officially, but you can speak with me after classes or via email.

**Objectives:** This mini course will be a hands-on introduction to the MATLAB computing environment. We will cover the basics of MATLAB syntax and computation. We will quickly move on to more advanced topics of writing loops, optimization and basic dynamic programming. Students with some familiarity with MATLAB should still benefit from the course. New users should spend some extra time on self-study and experimentation. There will be no homework, no exams and I will not take attendance. Your only test is to learn something useful that will make life easier in Econ 701, 630, 741 or any research task that may require scientific computing.

**Structure:** There are 4 lectures–(1) Using MATLAB, (2) Loops, Efficiency and Monte Carlo, (3) Optimization and (4) Dynamic Programming to cover in 3 classes. See the table of contents for an outline. Lecture time will be roughly allocated as follows

Class 1: Lecture 1, start Lecture 2

Class 2: Finish Lecture 2, start Lecture 3

Class 3: Finish lectures 3 and 4

*Acknowledgements: Portions of these lecture notes have been adapted from material originally put together by Ethan Ilzetzki

# Contents

**I Using MATLAB** **4**

**1 Preliminaries** **4**

1.1 Accessing MATLAB in the Graduate Computer Lab . . . . . 4

1.2 Accessing MATLAB via Terpconnect . . . . . 4

1.2.1 Running interactively . . . . . 5

1.2.2 Running in the background . . . . . 5

1.3 HELP!!! . . . . . 6

1.4 Basic Windows . . . . . 6

1.5 Statements, expressions, and variables; saving a session . . . . . 7

1.6 Command line editing and recall . . . . . 8

**2 Matrices, operations and basic MATLAB functions** **8**

2.1 Entering matrices . . . . . 8

2.2 Generating a Matrix using Operations and Functions . . . . . 9

2.3 Loading a matrix . . . . . 9

2.4 Matrix operations, array operations . . . . . 9

2.5 Matrix building functions . . . . . 11

2.6 Referencing matrix elements, submatrices and colon notation . . . . . 11

2.7 Scalar functions . . . . . 13

2.8 Vector functions . . . . . 13

2.9 Matrix functions . . . . . 14

**3 Scripting and M-files** **14**

3.1 M-files . . . . . 14

3.1.1 Logical–relational operators and IF statements . . . . . 15

3.2 Functions . . . . . 17

**4 Other Basic Tools** **20**

4.1 Text strings, error messages, input . . . . . 20

4.2 Graphics . . . . . 21

**5 Appendix of useful Matlab Functions** **24**

**II Loops, Efficiency and Monte Carlo Simulation** **34**

**6 FOR and WHILE loops** **34**

6.1 FOR statements . . . . . 35

6.2 WHILE statements . . . . . 36

**7 Comparing efficiency of algorithms: clock, etime, tic and toc** **36**

**8 How MATLAB stores and retrieves data – writing efficient loops** **37**

**9 Basic Monte Carlo** **40**

9.1 Random Number Generator . . . . . 40

9.2 A simple Monte Carlo study . . . . . 41

**III Optimization** **42**

**10 Solving Nonlinear Equations** **43**

10.1 Bisection . . . . . 43

10.2 Newton's Method . . . . . 45

10.3 Function Iteration . . . . . 47

**11 The Optimization Toolbox** **48**

11.1 fzero . . . . . 48

11.2 fsolve . . . . . 48

11.3 Minimization and Maximization . . . . . 50

11.4 Unconstrained Example . . . . . 50

11.5 Example with Constraints . . . . . 51

11.6 Utility maximization example . . . . . 52

**12 Numerical Integration** **53**

**IV Dynamic Programming** **53**

**13 A Cake-Eating Example** **53**

**14 A Discrete, Stochastic, Cake Eating Problem** **59**

# Part I

# Using MATLAB

## 1 Preliminaries

MATLAB is an abbreviation for MATrix LABoratory. It is a matrix-based system for scientific calculations. You can solve numerical problems without necessarily having to write a long program. This course provides an introduction to MATLAB. It will provide the basics of MATLAB programming and applications (primarily) for macroeconomics and international finance.

### 1.1 Accessing MATLAB in the Graduate Computer Lab

In econ computer lab, you can simply:

1. Log On
2. Click on Start–Network Applications.
3. Click on Mathematics and then on MATLAB

### 1.2 Accessing MATLAB via Terpconnect

You need to install PuTTY (secure shell program), Kerberos and WINSCP (secure FTP software). You can find information on how to do this at http://www.helpdesk.umd.edu/documents/4/4372/

In short, putty acts like a linux window where you can type your commands, etc. Winscp looks like windows explorer and let you move files between your computer and the servers. Follow the instructions from OIT to install the programs. Open Winscp and connect to terpconnect.umd.edu See instructions at http://www.helpdesk.umd.edu/documents/4/4315/. You should be able to see your home directory. You can create your own directories and save up to 1GB of files

- Open the shell command window by clicking on the icon that looks like two computers connected by a yellow wire (you'll need to type your username and password again).
- Go to the directory where you have your matlab files by typing `cd directory_name`
- type `tap matlab`, this will load matlab into your session

### 1.2.1 Running interactively

If you want to run something while you are logged in, type `matlab` at the command prompt. You'll be in the main matlab command window. You can run an M-file just by typing the matlab file name. No figures will open so save them to a file within your M-file script (more on this later).

### 1.2.2 Running in the background

Eventually, you might be working with routines and algorithms that days rather than seconds to complete. First, you can speed the process up by running on the linux servers which are far more powerful and stable than your own PC or the grad lab computers. Second, you can run code in the background. This means you can log out, log back in to check on things, log back in later when the routine is finished and have everything saved in a nice little output file. The alternative is to tie up your home computer. Or, you could tie up a computer in the grad lab where (1) another student may shut it off if too many computers are locked out or (2) your program will crash because a PC can't perform complex calculations in a room that is often over $80^\circ$ $F$. In case of (1) or (2) you'll never know if a bug in your code, a malicious fellow student or a poorly ventilated lab caused your computer or program to die.

You can write up a script (see below) to run in the background using `nohup` which stands for "no hang up." The syntax is `nohup matlab -nodisplay <myfile.m> a.out &`

myfile.m is your main code (you must include inequality signs) and a.out is where you'll find whatever would appear in the main matlab windows (remember to save all your variables). `-nodisplay` is just a option that prevents an some extra writing. You can change the name of your output file to whatever you want. You can open the file by typing `emacs a.out` or by going back to WINSCP and opening it with any text editor.

## 1.3 HELP!!!

Beyond this basic primer, the best way to learn MATLAB is through hands-on experimentation. Many economists have MATLAB code they have used on their websites. Use Google and other web resources to find code snippets to get a sense of how experienced programmers have set up applications.

You should learn to use the `help` function. Typing `help` *command* will display detailed information about any MATLAB function. Try `help regress` to learn how you can run a basic regression in MATLAB.

## 1.4 Basic Windows

When starting MATLAB on the PC, several windows will appear. If you cannot see one of the windows, click on VIEW in the toolbar, and you will see a list of windows. The command window is the main control module. You can enter commands following the `>>` prompt. A command followed with a semicolon will be performed "ilently" – no output will be displayed. Output of commands not performed silently will be shown in the command window. The command history window shows you the past commands that you have entered in the command window, and makes it easy to repeat an earlier command (by double-clicking on that command). You can also see, repeat and modify past commands by pressing the up arrow key when in the command window. The current directory window shows you the directory from which you are currently working, and a list of recent files you have created or used in MATLAB. You can change the current directory using the controls at the top of this window. Saved files or loaded files will be to or from the current directory. However, a list of folders listed in file'set path will also be searched for .m files (more on this later). The workspace window displays all the variables that are currently stored in the current session. For example, entering `a=1` in the command window creates a (scalar) variable a, which is assigned the value 1. You will now see the variable a in the workspace. Double clicking on that variable will display it and even allow you to manually alter its value. There are other windows, but these will suffice for now.

## 1.5 Statements, expressions, and variables; saving a session

MATLAB is an expression language; the expressions you type are interpreted and evaluated. MATLAB statements are usually of the form `variable = expression` or *expression*.

Expressions are usually composed of operators, functions, and variable names. Evaluation of the expression produces a matrix, which is then displayed on the screen and assigned to the variable for future use. If the variable name and = sign are omitted, a temporary variable `ans` (for answer) is automatically created to which the result is assigned.

A statement is normally terminated with the carriage return. However, a statement can be continued to the next line with three or more periods followed by a carriage return. On the other hand, several statements can be placed on a single line if separated by commas or semicolons. If the last character of a statement is a semicolon, the printing is suppressed, but the assignment is carried out. This is essential in suppressing unwanted printing of intermediate results. MATLAB is case-sensitive in the names of commands, functions, and variables. For example, solveUT is not the same as solveut. The command `who` lists the variables currently in the workspace(useful if you are working on a server). A variable can be cleared from the workspace with the command `clear` *variablename* The command clear alone clears all nonpermanent variables. The permanent variable `eps` (epsilon) gives the machine precision-about 10-16 on most machines. It is useful in determining tolerences for convergence of iterative processes. You will learn more about this in Econ 630.

**Important.** We all write infinite loops and runaway output from time to time. A runaway display or computation can be stopped on most machines without leaving MATLAB with `CTRL-C` (or `CTRL-BREAK` ). On Unix/Linux machines, you have to use the `kill` "jobnumber" on jobs that you are running in the background. Consult a good linux manual or the web for more information about the unix command line.

Saving a session. When one logs out or exits MATLAB all variables are lost. However, invoking the command `save` before exiting causes all variables to be written to a file `matlab.mat`. When one later reenters MATLAB, the command `load` will restore the workspace to its former state. You can also save your session to a .mat file with the name of your choice by entering `save` *filename*.

### 1.6 Command line editing and recall

A convenient feature is use of the up/down arrows to scroll through the stack of previous commands. One can, therefore, recall a previous command line, edit it, and execute the revised command line. For small routines, this is much more convenient that using an M-file which requires moving between MATLAB and the editor. For example, If one wanted to compare plots of the functions $y = \sin mx$ and $y = \sin nx$ on the interval $[0, 2\pi]$ for various m and n, one might do the same for the command line:

```
m=2;
n=3;
x=0:.01:2*pi;
y=sin(m*x);
z=sin(n*x); now hit the ↑ key and edit replace y with z and m with n
plot(x,y,x,z)
```

## 2 Matrices, operations and basic MATLAB functions

### 2.1 Entering matrices

MATLAB works with essentially only one kind of object-a rectangular numerical matrix. All variables represent matrices. 1-by-1 matrices may be interpreted as scalars and matrices with only one row or one column may be interpreted as vectors. Matrices can be introduced into MATLAB in several different ways: " Entered manually " Generated by operations and functions " Created in M-files " Loaded from external data files.

Manual Entry of a Matrix

Either of the statements `A = [1 2 3; 4 5 6; 7 8 9]` or

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

creates a 3-by-3 matrix and assigns it to a variable A. The elements within a row of a matrix may be separated by commas or a blank space. A semi-column or a new line denotes the end of a row. You must surround the entire list of elements with square brackets, `[ ]`.

## 2.2 Generating a Matrix using Operations and Functions

A matrix can be generated from other matrices using simple operations. For example:
`B = A'` Creates a matrix B that is the transpose of the matrix A. Note that entering
`A'` Will display the outcome of that operation, but also create a matrix **`ans`** in the workspace with the result. A more complete overview of operations will be given in the following section. MATLAB also has a number of functions to easily create commonly used matrices (or to generate random ones). `B = eye(3)` will create a 3 by 3 identity matrix, for example. An overview of such functions is provided in section 4.

## 2.3 Loading a matrix

The load command reads binary files containing matrices generated by earlier MATLAB sessions, or reads text files containing numeric data. The text file should be organized as a rectangular table of numbers, separated by blanks, with one row per line, and an equal number of elements in each row. For example, outside of MATLAB, create a text file containing these four lines:

```
16.0 3.0 2.0 13.0
5.0 10.0 11.0 8.0
9.0 6.0 7.0 12.0
4.0 15.0 14.0 1.0
```

Store the file under the name magik.txt. Then the command

```
load magik.txt
```

reads the file and creates a variable, **`magik`**, containing our example matrix.

## 2.4 Matrix operations, array operations

The following matrix operations are available in MATLAB:

```
+   addition
```

- subtraction
* multiplication
^ power
' transpose
\ left division
/ right division

These matrix operations also apply to scalars (1-by-1 matrices). If the sizes of the matrices are incompatible with the matrix operation, an error message will result, except in the case of scalar-matrix operations (for addition, subtraction, and division as well as for multiplication) in which case the operation is performed between the scalar and each cell of the matrix separately. The "matrix division" operations deserve special comment. If A is an invertible square matrix and b is a compatible column, resp. row, vector, then
`x=A \ b` is the solution of $Ax = b$ and, resp.,
`x=b/A` is the solution of $xA = b$. In left division, if A is square, then it is factored using Gaussian elimination and these factors are used to solve $Ax = b$. If A is not square, it is factored using Householder orthogonalization with column pivoting and the factors are used to solve the under- or over- determined system in the least squares sense. Right division is defined in terms of left division by `b/A=(A' b')'`.

Array operations (dot operations). Matrix operations can also be performed on a element-by-element basis, by adding a period before the operator. These are also known as Hadamard operations. The following operators `.* , .^ , . , and ./,` can be used in such a way. For example, either

```
A=[1,2,3,4].*[1,2,3,4]
or
B=[1,2,3,4].^2
```

will yield `[1,4,9,16]`. This is particularly useful when using Matlab graphics. Note that there is no need for a ".+" or ".-" operator, since these operators work on a element by element basis by definition.

## 2.5 Matrix building functions

Convenient matrix building functions are

```
eye(n)      n by n identity matrix
zeros(m,n)  m by n  matrix of zeros
ones(m,n)   m by n  matrix of ones
diag(A)     returns diagonal elements of A as vector
triu(A)     upper triangular part of a matrix
tril(A)     lower triangular part of a matrix
rand(m,n)   m by n random matrix with uniformly distributed elements
```

If x is a vector, `diag(x)` is the diagonal matrix with x down the diagonal; if A is a square matrix, then `diag(A)` is a vector consisting of the diagonal of A . What is `diag(diag(A))`? Try it. Matrices can be built from blocks. For example, if A is a 3-by-3 matrix, then `B = [A, zeros(3,2); zeros(2,3), eye` will build a 5-by-5 matrix corresponding to that definition. The built-in functions `rand` and `magic`, provide an easy way to create random or specific matrices. The command rand(n) will create an n x n matrix with randomly generated entries distributed uniformly between 0 and 1, while rand(m,n) will create an m x n one. magic(n) will create an integral n x n matrix which is a magic square (rows and columns have common sum); hilb(n) will create the n x n Hilbert matrix, the king of ill-conditioned matrices ( m and n denote, of course, positive integers). Matrices can also be generated with a for-loop.

## 2.6 Referencing matrix elements, submatrices and colon notation

Individual matrix and vector entries can be referenced with indices inside parentheses. The element in row i and column j of A is denoted by A(i,j). Here are some examples

```
A = [1 2 3;4 5 6;7 8 9];
A(3,2); displays 8
A(2,1) + A(2,2) + A(3,2) the sum of row 2
A(:,2); read as all rows, column 2 and displays a column vector
```

We will later that there are better ways to sum rows and columns than the code in line 2.

The last line uses MATLAB's colon notation. This is a very useful way to access submatrices of a matrix. For example, `A(1:4,3)` is the column vector consisting of the first four entries of the third column of A. A colon by itself denotes an entire row or column: `A(:,3)` is the third column of A , and `A(1:2,:)` is the first two rows. Arbitrary integral vectors can be used as subscripts: `A(:,[1 3])` contains as columns, columns 1 and 3 of A . Such subscripting can be used on both sides of an assignment statement: `A(:,[1 3]) = B(:,1:2)` replaces columns 1 and 3 of A with the first two columns of B. Note that the entire altered matrix A is printed and assigned. Try it. Columns 1 and 3 of A can be multiplied on the right by the 2-by-2 matrix [1 2;3 4]: `A(:,[1 3]) = A(:,[1 3])*[1 2;3 4]` Once again, the entire altered matrix is printed and assigned. If x is an n-vector, what is the effect of the statement `x = x(n:-1:1)`? Try it. To appreciate the usefulness of these features, compare these MATLAB statements with a Pascal, FORTRAN, or C routine to effect the same.

Vectors and submatrices are often used in MATLAB to achieve fairly complex data manipulation effects. "Colon notation" efficiently generate vectors and matrices. Creative use of these features permits one to minimize the use of loops (which slows MATLAB) and to make code simple and readable. Special effort should be made to become familiar with them. The expression `1:5` is actually the row vector `[1 2 3 4 5]`. The numbers need not be integers nor the increment one. For example,

```
A=0.2:0.2:1.2
B=5:-1:1
```

generate these vectors

```
A =

    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000


B =

     5     4     3     2     1
```

The following statements will, for example, generate a table of sines.

```
x = [0.0:0.1:2.0]' ;
y = sin(x);
[x y]
```

Note that since sin operates entry-wise, it produces a vector y from the vector x.

## 2.7 Scalar functions

Certain MATLAB functions operate essentially on scalars, but operate element-wise when applied to a matrix. The most common such functions are

| | | | | |
|---|---|---|---|---|
| sin | asin | exp | abs | round |
| cos | acos | log (natural log) | sqrt | floor |
| tan | atan | rem (remainder) | sign | ceil |

## 2.8 Vector functions

Other MATLAB functions operate on a vector (row or column), but act on an m-by-n matrix (m¿=2) in a column-by-column fashion to produce a row vector containing the results of their application to each column. Row-by-row action can be obtained by using the transpose; for example, mean(A')'. A few of these functions are

| | | | |
|---|---|---|---|
| max | sum | median | any |
| min | prod | mean | all |
| sort | std | var | |

For example, the maximum entry in a matrix A is given by max(max(A)) rather than max(A).

### 2.9 Matrix functions

Much of MATLAB's power comes from its matrix functions. The most useful ones are

```
eig     eigenvalues and eigenvectors
chol    cholesky factorization
svd     singular value decomposition
inv     inverse
lu      LU factorization
qr      QR factorization
hess    hessenberg form
schur   schur decomposition
rref    reduced row echelon form
expm    matrix exponential
sqrtm   matrix square root
poly    characteristic polynomial
det     determinant
size    size
norm    1-norm, 2-norm, F-norm, infinity-norm
cond    condition number in the 2-norm
rank    rank
```

MATLAB functions may have single or multiple output arguments. For example, `y = eig(A)` or simply `eig(A)` produces a column vector containing the eigenvalues of A while `[U,D] = eig(A)` produces a matrix U whose columns are the eigenvectors of A and a diagonal matrix D with the eigenvalues of A on its diagonal. Try it. See the `help` file each command to understand its output options.

## 3 Scripting and M-files

### 3.1 M-files

MATLAB can execute a sequence of statements stored in files. Such files are called M-files because they must have suffix .m in their filename. Much of your work with MATLAB will be in creating

and refining M-files. M files can be created in MATLAB's m-file editor or any text editor. Using the MATLAB will automatically highlight code for you (as will other programs you may find you prefer). The m-file editor can be accessed by loading any .m file (even a not-yet-existent one) from the file'load menu. You can open a blank script editor as follows. Go to File – New – M-file to open the script editor. Do this now, because we'll use it below.

There are two types of M-files: script files and function files.

**Script files.**A script file consists of a sequence of normal MATLAB statements. If the file has the filename of rotate.m, for example, then the MATLAB command rotate will cause the statements in the file to be executed. Variables in a script file are global and will change the value of variables of the same name in the environment of the current MATLAB session.

**Function files.** Function files allow you to create new functions which will then have the same status as other MATLAB functions. Variables in a function file are by default local.

### 3.1.1 Logical–relational operators and IF statements

Relational operators check the relative conditions between two matlab objects. You will see these all the time with IF statments. The relational operators in MATLAB are

| | |
|---|---|
| `<` | `less than` |
| `>` | `greater tha` |
| `<=` | `less than or equal` |
| `>=` | `greater than or equal` |
| `==` | `equal` |
| `=` | `not equal.` |

**Note that "=" is used in an assignment statement while "==" is used in a relation.**
Relations may be connected or quantified by the logical operators

| | |
|---|---|
| `&` | and |
| `\|` | or |
| `~` | not. |

When applied to scalars, a relation is actually the scalar 1 or 0 depending on whether the relation is true or false. Try `3 < 5, 3 > 5, 3 == 5, and 3 == 3`. When applied to matrices of the same size, a relation is a matrix of 0's and 1's giving the value of the relation between corresponding entries. Try a = rand(5), b = triu(a), a == b. A relation between matrices is interpreted by while

and if to be true if each entry of the relation matrix is nonzero.

**texttttIF.** statements make heavy use of logical and relational operators. The general form of a simple if statement is

```
if relation
  statements
end
```

The statements will be executed only if the relation is true. Hence, if you wish to execute statement when matrices A and B are equal you could type

```
if  A == B
    statement
end
```

but if you wish to execute statement when A and B are not equal, you would type

```
if any(any(A ≠ B))
   statement
end
```

Or, more simply,

```
if A == B else
   statement
end
```

Note that the seemingly obvious `if  A ~= B, statement, end` will not give what is intended since statement would execute only if each of the corresponding entries of A and B differ. Type `help any` to see how the `any` function gets around this complication. Why is it applied twice in the code above? The functions any and all can be creatively used to reduce matrix relations to vectors or scalars.

Multiple branching is also possible, as is illustrated by

```
if n < 0
     parity = 0;
elseif  rem(n,2) == 0
     parity = 2;
else
     parity = 1;
end
```

In two-way branching the elseif portion would, of course, be omitted. The `for` statement permits any matrix to be used instead of 1:n. See the User's Guide for details of how this feature expands the power of the for statement.

### 3.2 Functions

We first illustrate with a simple example of a function file. Note that lines beginning with % are comments-MATLAB will ignore them. The comment lines between the definition of the function and the first command line function as the "help" for this function. They will be displayed when you enter help fucntionname.

```
function  a = randint(m,n)
%RANDINT  Randomly generated integral matrix.
%         randint(m,n) returns an m-by-n matrix with entries
%         between 0 and 9.
a = floor(10*rand(m,n));
```

A more general version of this function is the following:

```
function  a = randint(m,n,a,b)
%RANDINT  Randomly generated integral matrix.
%         randint(m,n) returns an m-by-n matrix with entries
%         between 0 and 9.
%         rand(m,n,a,b) return entries between integers  a  and  b .
if nargin < 3, a = 0; b = 9; end
a = floor((b-a+1)*rand(m,n)) + a;
```

This should be placed in a file with filename `randint.m` (corresponding to the function name). The first line declares the function name, input arguments, and output arguments; without this line the file would be a script file.

In the command window, or in any other m-file, the MATLAB statement z = randint(4,5) will cause the numbers 4 and 5 to be passed to the variables a and b in the function file with the output result being passed out to the variable z. Since variables in a function file are local, their names are independent of those in the current MATLAB environment. Note that use of nargin ("number of input arguments") permits one to set a default value of an omitted input variable-such as a and b in the example. A function may also have multiple output arguments (in fact, it is simply returning a vector or matrix instead of a scalar). For example:

```
function  [mean, stdev] = stat(x)
% STAT  Mean and standard deviation
%       For a vector x, stat(x) returns the
%       mean and standard deviation of x.
%       For a matrix x, stat(x) returns two row vectors containing,
%       respectively, the mean and standard deviation of each column.
[m  n] = size(x);
if m == 1
      m = n;      % handle case of a row vector
end
mean = sum(x)/m;
stdev = sqrt(sum(x.^ 2)/m - mean.^2);
```

Once this is placed in a file stat.m, a MATLAB command [xm, xd] = stat(x), for example, will assign the mean and standard deviation of the entries in the vector x to m and xd, respectively. Single assignments can also be made with a function having multiple output arguments. For example, xm = stat(x) (no brackets needed around xm) will assign the mean of x to xm. The following function, which gives the greatest common divisor of two integers using the Euclidean algorithm, illustrates the use of an error message.

```
function  a = gcd(a,b)
% GCD  Greatest common divisor
```

```
%       gcd(a,b) is the greatest common divisor of
%       the integers a and b, not both zero.
a = round(abs(a));  b = round(abs(b));
if  a == 0 & b == 0
    error('The gcd is not defined when both numbers are zero')
else
    while b  ≠ 0
        r = rem(a,b);
        a = b;  b = r;
    end
end
```

Some more advanced features are illustrated by the following function. As noted earlier, some of the input arguments of a function-such as tol in the example, may be made optional through use of nargin ("number of input arguments"). The variable nargout can be similarly used. Note that the fact that a relation is a number (1 when true; 0 when false) is used and that, when while or if evaluates a relation, "nonzero" means "true" and 0 means "false". Finally, the MATLAB function feval permits one to have as an input variable a string naming another function.

```
function  [b, steps] = bisect(fun, x, tol)
%BISECT Zero of a function of one variable via the bisection method.
%         bisect(fun,x) returns a zero of the function.  fun is a string
%         containing the name of a real-valued function of a single
%         real variable; ordinarily functions are defined in M-files.
%         x  is a starting guess.  The value returned is near a point
%         where  fun  changes sign.  For example,
%         bisect('sin',3) is pi.  Note the quotes around sin.
%
%         An optional third input argument sets a tolerence for the
%         relative accuracy of the result.  The default is eps.
%         An optional second output argument gives a matrix containing a
%         trace of the steps; the rows are of form  [c f(c)].

% Initialization
if nargin < 3, tol = eps; end
trace = (nargout == 2);
```

```
if x  ~= 0, dx = x/20; else, dx = 1/20; end
a = x - dx;  fa = feval(fun,a);
b = x + dx;  fb = feval(fun,b);

% Find change of sign.
while (fa > 0) == (fb > 0)
     dx = 2.0*dx;
     a = x - dx;  fa = feval(fun,a);
     if (fa > 0) ~= (fb > 0), break, end
     b = x + dx;  fb = feval(fun,b);
end
if trace, steps = [a fa; b fb]; end

% Main loop
while  abs(b - a) > 2.0*tol*max(abs(b),1.0)
     c = a + 0.5*(b - a);  fc = feval(fun,c);
     if trace, steps = [steps; [c fc]]; end
     if (fb > 0) == (fc > 0)
          b = c;  fb = fc;
       else
          a = c;  fa = fc;
     end
end
```

Some of MATLAB's functions are built-in while others are distributed as M-files. The actual listing of any M-file-MATLAB's or your own-can be viewed with the MATLAB command type functionname. Try entering `type eig, type vander, and type rank`.

# 4 Other Basic Tools

## 4.1 Text strings, error messages, input

Text strings are entered into MATLAB surrounded by single quotes. For example, `s = 'This is a test'` assigns the given text string to the variable s. Text strings can be displayed with the function disp. For example:

```
disp('this message is hereby displayed')
```

Error messages are best displayed with the function error

```
error('Sorry, the matrix must be symmetric')
```

since when placed in an M-File, it causes execution to exit the M-file. In an M-file the user can be prompted to interactively enter input data with the function input. When, for example, the statement `iter = input('Enter the number of iterations: ')` is encountered, the prompt message is displayed and execution pauses while the user keys in the input data. Upon pressing the return key, the data is assigned to the variable iter and execution resumes.

When in MATLAB, the command dir will list the contents of the current directory while the command what will list only the M-files in the directory. The MATLAB commands delete and type can be used to delete a diskfile and print a file to the screen, respectively, and chdir can be used to change the working directory. M-files must be accessible to MATLAB. On most mainframe or workstation network installations, personal M-files which are stored in a subdirectory of one's home directory named matlab will be accessible to MATLAB from any directory in which one is working. See the discussion of MATLABPATH in the User's Guide for further information.

## 4.2 Graphics

MATLAB can produce both planar plots and 3-D mesh surface plots. Planar plots. The plot command creates linear x-y plots; if x and y are vectors of the same length, the command plot(x,y) opens a graphics window and draws an x-y plot of the elements of x versus the elements of y. You can, for example, draw the graph of the sine function over the interval -4 to 4 with the following commands:

```
x = -4:.01:4;
y = sin(x);
plot(x,y)
```

Try it. The vector x is a partition of the domain with mesh size 0.01 while y is a vector giving the values of sine at the nodes of this partition (recall that sin operates entrywise). When in the graphics screen, pressing any key will return you to the command screen while the command shg (show graph) will then return you to the current graphics screen. If your machine supports multiple windows with a separate graphics window, you will want to keep the graphics window exposed-but moved to the side-and the command window active. As a second example, you can draw the graph of $y = \exp(-x^2)$ over the interval -1.5 to 1.5 as `x = -1.5:.01:1.5; y = exp(-x.^2); plot(x,y)`. Plots of parametrically defined curves can also be made. Try, for example, `t=0:.001:2*pi; x=cos(3*t); y=s`

The command grid will place grid lines on the current graph. The graphs can be given titles, axis labels, and text captions within the graph with the following commands which take a string as an argument.

| | |
|---|---|
| `title` | graph title |
| `xlabel` | x-axis label |
| `ylabel` | y-axis label |
| `gtext` | interactively-positioned text |
| `text` | position text at specified coordinates |

For example, the command `title('Best Least Squares Fit')` creates a title. The command gtext('The Spot') allows a mouse or the arrow keys to position a crosshair on the graph, at which the text will be placed when any key is pressed. By default, the axes are auto-scaled. This can be overridden by the command axis. If `c=[xmin,xmax,ymin,ymax]` is a 4-element vector, then `axis(c)` sets the axis scaling to the prescribed limits. Alone, axis freezes the current scaling for subsequent graphs; entering axis again returns to auto-scaling. The command `axis('square')` ensures that the same scale is used on both axes. See help axis for additional features. Two ways to make multiple plots on a single graph are illustrated by

```
x=0:.01:2*pi;
y1=sin(x);
y2=sin(2*x);
y3=sin(4*x);
plot(x,y1,x,y2,x,y3) passing 3 sets of y coordinates on the same x's
```

and by forming a matrix Y containing the functional values as columns `x=0:.01:2*pi; Y=[sin(x)', sin(2*`

Another way is with `hold on`. The command `hold on` on freezes the current graphics screen so that subsequent plots are superimposed on it. Entering `hold off` releases the hold. You can also toggle the hold on and off by simply entering hold. One can override the default line types and point types. For example,

```
x=0:.01:2*pi; y1=sin(x); y2=sin(2*x); y3=sin(4*x);
plot(x,y1,'--',x,y2,':',x,y3,'+')
 or you could have used the following 4 lines instead
hold on
plot(x,y1,'--')
plot(x,y2,':')
plot(x,y3,'+'), hold off
```

These commands render a dashed line and dotted line for the first two graphs while for the third the + symbol is placed at each node. When passing multiple formatting options at once, writing each `plot` command on a single line and using `hold on` can be easier to read and navigate later.

The line- and mark-types are

- Line types: solid (-), dashed (–). dotted (:), dashdot (-.)
- Mark types: point (.), plus (+), star (*), circle (o), x-mark (x)

See the help file for `plot` for more options.

The command subplot can be used to partition the screen so that multiple plots can be viewed simultaneously. This can be very useful for impulse response functions or when you want to show a lot of information on one page without cluttering a single plot with many lines.

`subplot(m,n,p)` divides the plot window into an m*n matrix of graphs and assigns the graph described in the following plot statement to the pth element of that matrix. See help subplot for more details. 3-D mesh plots. Three dimensional mesh surface plots are drawn with the function mesh. The command mesh(z) creates a three-dimensional perspective plot of the elements of the matrix z. The mesh surface is defined by the z-coordinates of points above a rectangular grid in the x-y plane. Try mesh(eye(10)). To draw the graph of a function z=f(x,y) over a rectangle, one first defines vectors xx and yy which give partitions of the sides of the rectangle. With the function `meshgrid`, one creates a matrix x, each row of which equals xx and whose column

length is the length of yy, and similarly a matrix y, each column of which equals yy, as follows: `[x,y] = meshgrid(xx,yy);`

One then computes a matrix z, obtained by evaluating the function f entry by entry over the matrices x and y, to which mesh can be applied. For example, draw the graph of $z = \exp(-x^2 - y^2)$ over the square $[-2, 2] \times [-2, 2]$ as follows (try it):

```
xx = -2:.1:2;
yy = xx;
[x,y] = meshgrid (xx,yy);
z = exp(-x.^2 - y.^2);
mesh(z)
```

Another example, To evaluate the two-dimensional *sinc* function, $\sin(r)/r$, between the x and y directions:

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(X,Y,Z)
```

# 5 Appendix of useful Matlab Functions

There are many other functions. There exist, in particular, several "oolboxes" of functions for specific fields, including signal processing, control systems, robust-control, system identification, optimization, splines, chemometrics, mu-analysis and synthesis, state-space identification, and neural networks. (The toolboxes, which are optional, may not be installed on your system.) These can be explored via the command help. Nonlinear Equations and Optimization

| | |
|---|---|
| `fminbnd` | minimum of a function of one |
| `fmins` | minimum of a multivariable fu |
| `fzero` | zero of a function of one varial |
| `fsolve` | solution of a system of nonline |

Note that functions are simply strings such as: `'x^2-x'`.

Column-wise Data Analysis

| | |
|---|---|
| max | maximum value |
| min | minimum value |
| mean | mean value |
| median | median value |
| std | standard deviation |
| sort | sorting |
| sum | sum of elements |
| prod | product of elements |
| cumsum | cumulative sum of elements |
| cumprod | cumulative product of elements |
| diff | approximate derivatives (differences) |
| hist | histograms |
| cplxpair | reorder into complex pairs |

Statistics

| | |
|---|---|
| abs | absolute value |
| corrcoef | correlation coefficients |
| cov | covariance |

Interpolation

| | |
|---|---|
| spline | cubic spline |

General

| | |
|---|---|
| help | help facility |
| demo | run demonstrations |
| who | list variables in memory |
| what | list M-files on disk |
| size | row and column dimensions |
| length | vector length |
| clear | clear workspace |
| computer | type of computer |

```
exit        exit MATLAB
quit        same as exit
```

Matrix/Array Operators

```
Matrix Operators              Array Operators
-------------------------------------------------
+    addition                 +    addition
-    subtraction              -    subtraction
*    multiplication           .*   multiplication
/    right division           ./   right division
\    left division            .\   left division
^    power                    .^   power
'    conjugate transpose      .'   transpose
```

Relational and Logical Operators

```
<           less than
<=          less than or equal
>           greater than
>=          greater than or equal
==          equal
~=          not equal
&           and
|           or
~           not
```

Special Characters

```
=           assignment statement
[           used to form vectors and matrices
]           see [
(           arithmetic expression precedence
)           see (
```

```
.           decimal point
...         continue statement to next line
,           separate subscripts and function arguments
;           end rows, suppress printing
%           comments
:           subscripting, vector generation
!           execute operating system command
```

Special Values

```
ans         answer when expression not assigned
eps         floating point precision
pi          pi
i, j        sqrt(-1)
inf         infinity
NaN         Not-a-Number
clock       wall clock
date        date
nargin      number of function input arguments
nargout     number of function output arguments
```

File Management

```
chdir       change current directory
delete      delete file
diary       diary of the session
dir         directory of files on disk
load        load variables from file
save        save variables to file
type        list function or file
what        show M-files on disk
fprintf      write to a file
pack        compact memory via save
```

Special Matrices

| | |
|---|---|
| compan | companion |
| diag | diagonal |
| eye | identity |
| gallery | different matrices |
| hadamard | Hadamard |
| hankel | Hankel |
| hilb | Hilbert |
| invhilb | inverse Hilbert |
| linspace | linearly spaced vectors |
| logspace | logarithmically spaced vectors |
| magic | magic square |
| meshdom | domain for mesh points |
| ones | constant |
| pascal | Pascal |
| rand | random elements |
| toeplitz | Toeplitz |
| vander | Vandermonde |
| zeros | zero |

Matrix Manipulation

| | |
|---|---|
| rot90 | rotation |
| fliplr | flip matrix left-to-right |
| flipud | flip matrix up-to-down |
| diag | diagonal matrices |
| tril | lower triangular part |
| triu | upper triangular part |
| reshape | reshape |
| .' | transpose |
| : | convert matrix to single column; A(:) |

Relational and Logical Functions

```
any         logical conditions
all         logical conditions
find        find array indices of non-zero values
isnan       detect NaNs
finite      detect infinities
isempty     detect empty matrices
isstr       detect string variables
strcmp      compare string variables
```

Control Flow

```
if          conditionally execute statements
elseif      used with if
else        used with if
end         terminate bif, for, while
for         repeat statements a number of times
while       do while
break       break out of for and while loops
return      return from functions
pause       pause until key pressed
```

Programming and M-files

```
input       get numbers from keyboard
keyboard    (temporarily) returns control from M-file to command prompt
error       display error message
function    define function
eval        perform operation in text
feval       evaluate function given by string
echo        enable command echoing
exist       check if variables exist
casesen     set case sensitivity
```

```
global      define global variables
startup     startup M-file
getenv      get environment string
menu        select item from menu
etime       elapsed time
lasterr       reports the last error that has occurred
try/catch     see help (useful for debugging)
```

Text and Strings

```
num2str     convert number to string
int2str     convert integer to string
setstr      set flag indicating matrix is a string
sprintf     convert number to string
isstr       detect string variables
strcomp     compare string variables
hex2num     convert hex string to number
```

Command Window

```
clc         clear command screen
home        move cursor home
format      set output display format
disp        display matrix or text
fprintf     print formatted number
echo        enable command echoing
```

Graph Paper

```
plot        linear X-Y plot
loglog      loglog X-Y plot
semilogx    semi-log X-Y plot
semilogy    semi-log X-Y plot
polar       polar plot
```

```
mesh        3-dimensional mesh surface
contour     contour plot
meshdom     domain for mesh plots
bar         bar charts
stairs      stairstep graph
errorbar    add error bars
```

Graph Annotation

```
title       plot title
xlabel      x-axis label
ylabel      y-axis label
grid        draw grid lines
text        arbitrarily position text
gtext       mouse-positioned text
ginput      graphics input
```

Graph Window Control

```
axis        manual axis scaling
hold        hold plot on screen
shg         show graph window
clg         clear graph window
subplot     split graph window
```

Graph Window Hardcopy

```
print       send graph to printer
prtsc       screen dump
meta        graphics metafile
```

Elementary Math Functions

```
abs         absolute value or complex magnitude
angle       phase angle
```

| | |
|---|---|
| sqrt | square root |
| real | real part |
| imag | imaginary part |
| conj | complex conjugate |
| round | round to nearest integer |
| fix | round toward zero |
| floor | round toward -infinity |
| ceil | round toward infinity |
| sign | signum function |
| rem | remainder |
| exp | exponential base e |
| log | natural logarithm |
| log10 | log base 10 |

Trigonometric Functions

| | |
|---|---|
| sin | sine |
| cos | cosine |
| tan | tangent |
| asin | arcsine |
| acos | arccosine |
| atan | arctangent |
| atan2 | four quadrant arctangent |
| sinh | hyperbolic sine |
| cosh | hyperbolic cosine |
| tanh | hyperbolic tangent |
| asinh | hyperbolic arcsine |
| acosh | hyperbolic arccosine |
| atanh | hyperbolic arctangent |

Special Functions

| | |
|---|---|
| bessel | bessel function |

| | |
|---|---|
| gamma | gamma function |
| rat | rational approximation |
| erf | error function |
| inverf | inverse error function |
| ellipk | complete elliptic integral of first kind |
| ellipj | Jacobian elliptic integral |

Decompositions and Factorizations

| | |
|---|---|
| balance | balanced form |
| backsub | backsubstitution |
| cdf2rdf | convert complex-diagonal to real-diagonal |
| chol | Cholesky factorization |
| eig | eigenvalues and eigenvectors |
| hess | Hessenberg form |
| inv | inverse |
| lu | factors from Gaussian elimination |
| nnls | nonnegative least squares |
| null | null space |
| orth | orthogonalization |
| pinv | pseudoinverse |
| qr | orthogonal-triangular decomposition |
| qz | QZ algorithm |
| rref | reduced row echelon form |
| schur | Schur decomposition |
| svd | singular value decomposition |

Matrix Conditioning

| | |
|---|---|
| cond | condition number in 2-norm |
| norm | 1-norm,2-norm,F-norm, infinity-norm |
| rank | rank |
| rcond | condition estimate (reciprocal) |

Elementary Matrix Functions

```
expm        matrix exponential
logm        matrix logarithm
sqrtm       matrix square root
funm        arbitrary matrix function
poly        characteristic polynomial
det         determinant
trace       trace
kron        Kronecker tensor product
```

Polynomials

```
poly        characteristic polynomial
roots       polynomial roots---companion matrix method
roots1      polynomial roots---Laguerre's method
polyval     polynomial evaluation
polyvalm    matrix polynomial evaluation
conv        multiplication
deconv      division
residue     partial-fraction expansion
polyfit     polynomial curve fitting
```

# Part II

# Loops, Efficiency and Monte Carlo Simulation

## 6 FOR and WHILE loops

In their basic forms, MATLAB flow control statements operate like those in most computer languages. Open the m-file editor and work through the following code fragments. You should **always write any kind of loops as a script**. Command line entry of `for, while` loops is messy and difficult to debug.

## 6.1 FOR statements

FORThe `for` statement performs a set of expression that depend on an index that is updated with each iteration. Type the following into your m-file script. Save the file as `myscript.m`. There are two way to run the script. Type `myscript.m` at the command line or from the m-file editor, hit the key `F5`

```
x = [];
for i = 1:n
x=[x,i^2 ]
end;
```

will produce an vector of length n and the statement

```
x = [];
for i = n:-1:1
x=[x,i^2 ]
end
```

will produce the same vector in reverse order. Try them. Note that a matrix may be empty (such as `x = []`). What is the reasoning for the syntax `x=[x,i^2]`?

Question: Do you know a better way to produce the output above? Try `x=1:n.^2`. This produces the same vector as the first block of code in only one line. How would you produce the second block in one line?

You can and frequently will used nested `for` statements too. The statements

```
for i = 1:m
    for j = 1:n
        H(i, j) = 1/(i+j-1);
    end
end
H
```

will produce and print to the screen the m-by-n Hilbert matrix. The semicolon on the inner statement suppresses printing of unwanted intermediate results while the last H displays the final result.

### 6.2 WHILE statements

The general form of a while loop is

```
while   relation
  statements
end
```

The statements will be repeatedly executed as long as the relation remains true. For example, for a given number a, the following will compute and display the smallest nonnegative integer n such that 2n ¿=a:

```
n = 0;
    while  2^n < a
         n = n + 1;
    end
    n
```

There are many applications where `for` and `while` loops can accomplish the same tasks, but we might prefer to use `while`. See section 10.2 for an example contrasting `for` and `while` loops.

## 7 Comparing efficiency of algorithms: clock, etime, tic and toc

One of the measures of the efficiency of an algorithm is the elapsed time. The MATLAB function clock gives the current time accurate to a hundreth of a second (see help clock). Given two such times t1 and t2, `etime(t2,t1)` gives the elapsed time from t1 to t2. One can, for example, measure the time required to solve a given linear system Ax=b using Gaussian elimination as follows:

```
t = clock;
```

```
x = A \ b;
time = etime(clock,t)
```

You may wish to compare this time-and flop count-with that for solving the system using `x = inv(A)*b`. The more convenient `tic` and `toc` will automatically count the time.

# 8 How MATLAB stores and retrieves data – writing efficient loops

MATLAB and FORTRAN use column-major order for storage as opposed to row-major order (used by C). In longer algorithms, you should be aware of how your code accesses elements from vectors and matrices. The following matrix

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

is stored in MATLAB as

```
1   4   2   5   3   6
```

If matlab has to retrieve the $A(2,1)$ and $A(1,2)$ elements this will take longer than retrieving $A(1,1)$ and $A(2,1)$. Even though 2 and 5 appear to be stored next to each other, they are in two different blocks of memory because MATLAB "thinks" in terms of columns. By contrast, C would store the matrix as

```
1   2   3   4   5   6
```

so that retrieving same-row data is faster.

This may seem like a trivial point, but the simple example below shows that we are talking about practical time savings by writing smarter code.

```
%%  EXAMPLE OF GOOD AND BAD LOOPS %%
clear

m=1000000;
n=2;
%A=randn(m,n);
%B=A';
```

```
%v=1:m*n;
v=ones(1,m*n);
A=reshape(v,m,n);
B=reshape(v,n,m);

%compute sums of the columns of A;
%then compute sums of the rows of B;

tic
sumBrows=0;
for i=1:m;
    for j=1:n;
        sumBrows=sumBrows+B(j,i);
    end;
end;
rowsumtime=toc

tic
sumAcols=0;
for i=1:n;
    for j=1:m;
        sumAcols=sumAcols+A(j,i);
    end;
end;
colsumtime=toc
```

This produces the following output on my machine

```
rowsumtime =

    3.5253


colsumtime =

    1.5118
```

First, summing over the rows in nested `for` loop takes over twice as long. This is because of how arrays are stored in memory. Second, are your times different than mine or the person sitting next to you? They most likely are because computing efficiency depends on what other jobs your CPU is processing. If you are checking your email, running iTunes and instant messaging while your code is running you might get faster results using row sums purely by coincidence.

Accessing matrices element by element like this is usually a bad idea even if you are being efficient. One way to speed up your algorithms is to use MATLAB's built in functions. Let's use the `sum` function to test this.

```
%% EXAMPLE: USING BUILT IN FUNCTIONS WILL USUALLY SAVE YOU TIME
%now let's try using matlab's sum function twice;

tic;
sumB=sum(sum(B));
sumfunBtime=toc

tic
sumA=sum(sum(A));
sumfunAtime=toc

%now let's use sum but tell it to sum across B's row's first

tic
sumB=sum(sum(B,2));
sumfunBtime2=toc
```

My computer gives the following output

```
sumfunBtime =


    0.0180


sumfunAtime =
```

```
    0.0069

sumfunBtime2 =

    0.0053
```

These are much much faster, but we're still getting slower times for row sums on the B matrix until we tell the `sum` function to sum the rows of B first. Why is this last option the fastest of all? [Hint: what is the dimensionality of the vector created by the inner `sum` operation?].

# 9 Basic Monte Carlo

We'll use a loop to run a Monte Carlo study of the basic linear regression model. At the same time, we'll get some practice using the random number generator.

## 9.1 Random Number Generator

The random number generator is a useful tool. The syntax is `rand(m,n)` to create a m by n matrix of uniformly distributed random numbers. **`randn`** produces standard normal random numbers. Try entering `A=rand(3,3), B=randn(3,3)` at the command line. My computer outputs

```
A =

    0.7640    0.6131    0.6479
    0.9230    0.4254    0.0345
    0.7807    0.9953    0.7850


B =

   -1.1080    0.2251   -1.6177
   -0.6113    1.2364   -0.3539
   -0.1456    1.9349   -1.4687
```

Do you get the same output? Most likely, you have something different. It is often necessary when debugging simulations or running another person's code on your machine to generate the same set of random numbers. We can do this by setting the RNG to the same state each time. Try the following code `rand('state',4104),A=rand(3,3),randn('state',4104),B=randn(3,3)`. Now you should get the same results. I'm using our room number as the seed. It's best to use something you can always remember later such as your birthday.

## 9.2 A simple Monte Carlo study

Now we'll combine random number generation with loops to do a monte carlo study. This will be useful practice for simulating RBC models later.

Save the following code as the m-file **mcs.m**

```
%%% A simple Monte Carlo Example %%

% run basic linear regression of y = b0 +b1*x + eps
% where x is normal rv and eps is normal (0,sigma) disturbance

% set params
b0=1;
b1=2;
sigma=2;

randn('state',4104);

% number of trials
m=1000;

% number of data points
n=100;

% generate arrays of data
% each set of trial data will be stored as a column
X=randn(n,m);
eps=sigma*randn(n,m);
```

```
y = b0+b1*X + eps;


% running with a loop
bhat=zeros(n,2);

for i=1:m;
    XpXinv=inv([ones(n,1),X(:,i)]'*[ones(n,1),X(:,i)]);
    XpY=[ones(n,1),X(:,i)]'*y(:,i);
    bhat(i,:) = XpXinv*XpY;
end;

meanvar=[mean(bhat);std(bhat)]

% running without using a loop
% sample covariance of x and y
mx=mean(X);
my=mean(y);
xy=(sum(X.*y-y*diag(mx)-X*diag(my))+n*mx.*my)/n;
% sample variance of x with df correction
xx=var(X)*(n-1)/n;

bhat2(:,2)=xy./xx;
bhat2(:,1)=my'-bhat2(:,2).*mx';


meanvar2=[mean(bhat2);std(bhat2)]
```

# Part III

# Optimization

## 10 Solving Nonlinear Equations

One of the most basic numerical problems encountered in computational economics is to find the solution of a system of nonlinear equations. Nonlinear equations generally arise in one of two forms. In the nonlinear root finding problem, a function f is given and one must compute an n-vector x, called a root of f, that satisfies

$$f(x) = 0.$$

For example, if you are looking for the steady state of a system defined by a set of differential equations, you would find its steady state by finding the root of the equations created by setting the first derivatives of certain objective functions to zero.

In the nonlinear fixed-point problem, a vector valued function g is given and one must compute an n-vector x called a fixed-point of g, that satisfies

$$x = g(x).$$

For example, finding the steady state of a difference equation can be formulated in this form. The vector $x$ might be the steady state capital stock, labor supply and consumption.

The two forms are equivalent. The root-finding problem may be recast as a fixed-point problem by letting $g(x) = x - f(x)$. Conversely, the fixed-point problem may be recast as a root finding problem by letting $f(x) = x - g(x)$.

### 10.1 Bisection

The bisection method is perhaps the simplest and most robust method for computing the root of a continuous real-valued function defined on a bounded interval of the real line. The bisection method is based on the Intermediate Value Theorem, which asserts that if a continuous real-valued function defined on an interval assumes two distinct values, then it must assume all values in between. In particular, if $f$ is continuous, and $f(a)$ and $f(b)$ have different signs, then $f$ must have at least one root $x \in [a, b]$.

The bisection method is an iterative procedure. Each iteration begins with an interval known to contain or to bracket a root of f, meaning the function has different signs at the interval endpoints. The interval is bisected into two subintervals of equal length. One of the two subintervals must have endpoints of different signs and thus must contain a root of f. This subinterval is taken as the new interval with which to begin the subsequent iteration. In this manner, a sequence of intervals is generated, each half the width of the preceding one, and each known to contain a root of f. The process continues until the width of the bracketing interval containing a root shrinks below an acceptable convergence tolerance.

For example, let's find root of function $y = f(x) = x^3 - 2$. Copy this code to a an m-file as save it as `bsect.m`.

```
%%%%%% Matlab Code to solve
%%%%%% y = f(x)  y = x^3-2

%% set parameters %%
a=1;
b=2;
tol=0.01;

%% initialize starting values %%
s = sign((a^3-2));
x = (a+b)/2;
d = (b-a)/2;

%% loop until convergence criterion met %%
while d>tol;
d = d/2;
if s == sign(x^3-2)
x = x+d;
else
x = x-d;
end
end
```

```
%% output the solution %%
disp('solution is') ; disp(x);
```

This method will find the root of the function, but is rather slow in doing so.

## 10.2 Newton's Method

Suppose an economist is presented with a demand function

$$q = 0.5p^{-0.2} + 0.5p^{-0.5},$$

representing the sum a domestic demand term and an export demand term at price $p$. Using standard calculus, the economist could easily verify that the demand function is continuous, differentiable, and strictly decreasing. Suppose that the economist is asked to find the price that clears the market of, say, a quantity of 2 units. Without using the Matlab implicit function `fsolve`, one can apply Newton's method. The algorithm for Newton's method is

1. Start from guess $x_k$
2. Take a linear approximation (first order Taylor approximation) of the function around $x_k$.
3. Find the zero of the linear approximation and use it as the next guess $x_{k+1}$ .
4. If the original function evaluated at $x_{k+1}$ is within our tolerance of zero, $x_{k+1}$ is the solution. Otherwise, return to step 2 with $x_{k+1}$ as the current guess.

Save the following example as an m-file called `newt.m` and run it.

```
p = 0.25; %initial guess

for i=1:100

Δp = (.5*p^-.2+.5*p^-.5-2)/(-.1*p^(-1.2) - .25*p^(-1.5));%how far are we from zero?
p = p - Δp; %ajust next guess by distance from zero

if abs(Δp) < 1.e-8 %is it within machine zero precision?
    break  %if yes, stop
```

```
end

end

display('the solution is'),disp(p);
```

That worked pretty well for this application, but what are some problems with the code above more generally? Newton's method will usually converge very quickly, but not always. What if we need more than 100 steps? How can we change this code to account for that (endogenously, rather than increasing the steps)? Let's write it up as a while loop. Paste the following into an m-file and call it `newt2.m`.

```
p = 0.25; %initial guess
count=0;
Δp=1;
noconv=0;

while abs(Δp)> 1.e-8;

    Δp = (.5*p^-.2+.5*p^-.5-2)/(-.1*p^(-1.2) - .25*p^(-1.5));%how far are we from zero?
    p=p-Δp;

    count=count+1; %increase counter
    if count>10000; %bail out of the loop after 10,000 steps
        noconv=1;
        break
    end

end;
if noconv==0;
    display('the solution is'),disp(p)
    display('steps to convergence'),disp(count)
else
    disp('no convergence, try new starting value')
end
```

In this specific example, the improved code doesn't make any difference to the solution but it is more robust to less well behaved functions. You may enter a function due to coding error or a badly specified model that is poorly behaved and has no zero. Newton's method to run out towards infinity, but the count breaker will stop this and tel you about it. You will encounter economic applications and assignments in Econ 630 where this is actually a problem. This function converge in 7 steps. Try setting the count max to 3 and see what happens.

### 10.3 Function Iteration

Function iteration is a relatively simple technique that may be used to compute a fixed-point $x = g(x)$. The technique is also applicable to a root finding problem $f(x) = 0$ by recasting it as the equivalent fixed-point problem $x = x - f(x)$. Function iteration begins with the analyst supplying a guess $x(0)$ for the fixed-point of g. Subsequent iterations are generated using the simple iteration rule

$$x(k+1) \leftarrow g(x(k)).$$

Since g is continuous, if the iterates converge, they converge to a fixed-point of g. In theory, function iteration is guaranteed to converge to a fixed-point of g if g is differentiable and if the initial value of x supplied by the analyst is "sufficiently" close to a fixed-point. Function iteration, however, often converges even when the sufficiency conditions are not met. Given that the method is relatively easy to implement, it is often worth trying. For example, suppose that one wished to compute a fixed-point of the function $g(x) = x^0.5$.

```
x=0.8;
maxit = 100;
tol   = 0.001;

for it=1:maxit
   gval =  x^.5;
   if norm(gval-x)<tol, break, end
   x = gval;
end

disp(x);
```

# 11 The Optimization Toolbox

MATLAB's Optimization Toolbox is an excellent application with a number of .m functions that help in solving these and other problems.

## 11.1 fzero

The function fzero finds the zero of a function of one variable. The command `x = fzero(fun,x0)` would assign to x the "zero" of the expression fun in the vicinity of `x0`.

The expression fun can be entered in one of two methods:

1. As a string (between single quotes:' ')
2. Defined in a function. For example

```
function y=example(x)
y = x^2-1;
return;
```

In the latter case, we'd enter the expression `@example` as the first argument of the fzero function.

Example: find the zero of the function $f(x) = x^3 - 2$.

## 11.2 fsolve

The function fsolve solves for the zero of a system of equations/functions. Multiple functions can be described as a single function with a vector as output.We'ell solve for the steady state of a simple RBC model. See this note for more details. Save the following m-file as `rbcss.m`.

```
function F=rbcss(x,alpha, beta,Δ,psi)
%steady state of simple RBC model
%x(1) is capital
%x(2) is labor
```

```
%x(3) is consumption

F(1) = 1/beta -1 - alpha*x(1)^(alpha-1)*x(2)^(1-alpha) + Δ
F(2) = psi*(x(3)/1-x(2)) - (1-alpha)*x(1)^alpha*x(2)^(-alpha)
F(3) = x(3)+Δ*x(1) - x(1)^alpha*x(2)^(1-alpha)
```

This function recasts the steady state equations so that a solution for capital, labor and consumption outputs the vector $F(x) = 0$. Note that I have separated the parameters from the endogenous variables. Now we can use `fsolve` to find the steady state solution with the following code

```
%%% find the solution to simple RBC model %%%

%%% set params %%%
alpha=.33;
beta=.99;
delta=.023;
psi=1.75;
x0=[20 1 2];
%%% set options on fsolve %%%
options = optimset('maxiter',100000','tolfun',10e-8,'tolx',10e-8);
[SS, fval, exitflag, output]=fsolve(@(x) rbcss(x,alpha, beta, delta, psi),x0,options);
SS
```

This will output the following

```
Optimization terminated: first-order optimality is less than options.TolFun.


SS =


   26.4133    0.8536    2.0419
```

The other output variables are

| | |
|---|---|
| fval | the value of the function at the solution, should be close to zero |
| exitflag | Flag for convergence, value of 1 means a solution was found otherwise check help file for error |
| output | contains information about the solution algorithm and iterations |

## 11.3 Minimization and Maximization

## 11.4 Unconstrained Example

Consider the problem of finding a set of values [x1, x2] that solves:

$$[x_1, x_2] = \arg\min \exp(x_1)[4x_1^2 + 2x_2^2 + 4x_1x_2 + 1]$$

To solve this two-dimensional problem, write an M-file that returns the function value. Then, invoke the unconstrained minimization routine fminunc. Just follow the steps below to minimize this function easily.

Step 1: Write an M-file `objfun.m`

```
function f = objfun(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
\end{lstlisgin}

Step 2: Invoke one of the unconstrained optimization routines
\begin{lstlisting}
x0 = [−1,1]; % Starting guess
options = optimset('LargeScale','off');
[x,fval,exitflag,output] = fminunc(@objfun,x0,options);
```

After 40 function evaluations, this produces the solution

```
x =
0.5000 -1.0000
```

The function evaluated at the solution x is returned in fval.

```
fval =
1.0983e-015
```

The initial guess for the vector $[x_1, x_2]$ affects the time to convergence, as well as the solution, if more than one local minimum exists. In the example above, `x0` is initialized to `[-1,1]`.

## 11.5 Example with Constraints

For routines that permit equality constraints, nonlinear equality constraints must be computed in the M-file with the nonlinear inequality constraints. For linear equalities, the coefficients of the equalities are passed in through the matrix Aeq and the right-hand-side vector beq. The next example demonstrates how this works with simple linear equality constraints.

The syntax is `[x,fval] = fmincon(@fun,x0,A,b,Aeq,beq)`. This minimizes **`fun`** subject to the linear equalities **`Aeq*x = beq`** as well as **`A*x <= b`**. Set **`A=[]`** and **`b=[]`** if no inequalities exist.

Suppose you would like to minimize the above function, but have the nonlinear equality constraint $x_1^2 + x_2 = 1$ and the nonlinear inequality constraint $x_1 x_2 \geq -10$.

Rewrite them as

$$x_1^2 + x_2 = 1$$

$$-x_1 x_2 - 10 \leq 0$$

and solve the problem using the following steps.

Step 1: Write an M-file **`objfun.m`**. We did this already in the last section.

Step 2: Write an M-file **`confuneq.m`** for the nonlinear constraints function

```
function [c, ceq] = confuneq(x)
c = -x(1)*x(2) - 10; % Nonlinear inequality constraints
ceq = x(1)^2 + x(2) - 1; % Nonlinear equality constraints
```

Note that the ordering of the non-linear constraints inside the contraint function matter. See the **`help`** file for **`fmincon`** for details.

Step 3: Invoke the constrained optimization routine

```
x0 = [-1,1]; % Initial guess
options = optimset('LargeScale','off');
[x,fval] = fmincon(@objfun,x0,[],[],[],[],[],[],@confuneq,options)
[c,ceq] = confuneq(x) % Check the constraint values at x
```

After 21 function evaluations, the solution produced is

```
x =
-0.7529 0.4332
fval =
1.5093
c =
-9.6739
ceq =
6.3038e-009
```

So far we have just performed minimization problems. To solve a maximization problem we proceed identically, after multiplying the objective function by -1.

## 11.6 Utility maximization example

Let us now solve a standard microeconomic maximization problem, using this procedure. Assume a consumer with the following separable CES utility over two goods $u(x1, x2) = x10.8 + x20.8$ The consumer has a wealth of $w = 10$ and faces prices $[p1, p2] = [1, 2]$.

Step 1: Write an M-file `utility.m`

```
function f = utility(x)
f = -((x(1)^.8)+(x(2)^.8))
```

Step 2: Invoke the optimization procedure

```
Aeq=[1 2];
beq=10;
x0=[8 1];
[x,fval] = fmincon(@utility,x0,[],[],Aeq,beq)
```

The output is:

```
Optimization terminated: Magnitude of directional derivative in search direction less than
2*options. TolFun and maximum constraint violation is less than options. TolCon.
x =
```

```
9.4118 0.2941
fval =
-6.3865
```

`fminsearch` is an additional minimization function that uses a different minimization method. Use MATLAB help for more information.

## 12 Numerical Integration

The quad function solves for the integral of a function. You might use this for econometrics applications or certain macro models where agents have to compute an expectation. Here's a quick example

Step 1: Write an M-file `functionname.m`

```
function f = funfunc(x)
f = 1./(x.^3-2*x-5);
```

Step 2: Integrate

```
Q = quad(@funfunc,0,2);
```

The above code integrates the function $f(x) = \frac{1}{x^3-2x-5}$ on the interval $[0, 2]$.

# Part IV

# Dynamic Programming

## 13 A Cake-Eating Example

Suppose that you are presented with a cake of size W. At each point of time, $t = 1, 2, \ldots$, you can eat some of the cake and may save the rest. Let $c_t$ be your consumption in period t, and let $u(c_t)$ represent the flow of utility from this consumption. We allow fore infinite horizon. One can consider solving the infinite horizon sequence problem given by

$$\max_{\{c_t\}_1^\infty, \{c_t\}_1^\infty} \sum_{t=1}^{\infty} \beta^t u(c_t) \tag{1}$$

Where

$$u(c) = \ln(c)$$

along with the transition equation

$$W_{t+1} = W_t - c_t \tag{2}$$

In specifying this as a dynamic programming problem, we write

$$V(W) = \max_{c \in W} u(c) + \beta V(W - c) \tag{3}$$

$V(W)$ is the value of the infinite horizon problem starting with a cake of size W. So in the given period, the agent chooses current consumption and thus reduces the size of the cake to $W' = W - c$, as in the transition equation. We use variables with primes to denote future values. The value of starting the next period with a cake of that size is then given by $V(W - c)$, which is discounted at rate $\beta = 0.8$.

For this problem, the state variable is the size of the cake (W) given at the beginning of each period. The state completely summarizes all past information that is needed for the forward-looking optimization problem. The control variable is the level of consumption in the current period, c. Note that c lies in a compact set. The transition equation can be rewritten as

$$W' = W - c \tag{4}$$

Alternatively, we can specify the problem as choosing tomorrow's state

$$V(W) = \max_{W' \in W} u(W - W') + \beta V(W') \tag{5}$$

Either specification yields the same result. But choosing tomorrow's state often makes the algebra a bit easier, so we will work with (5)

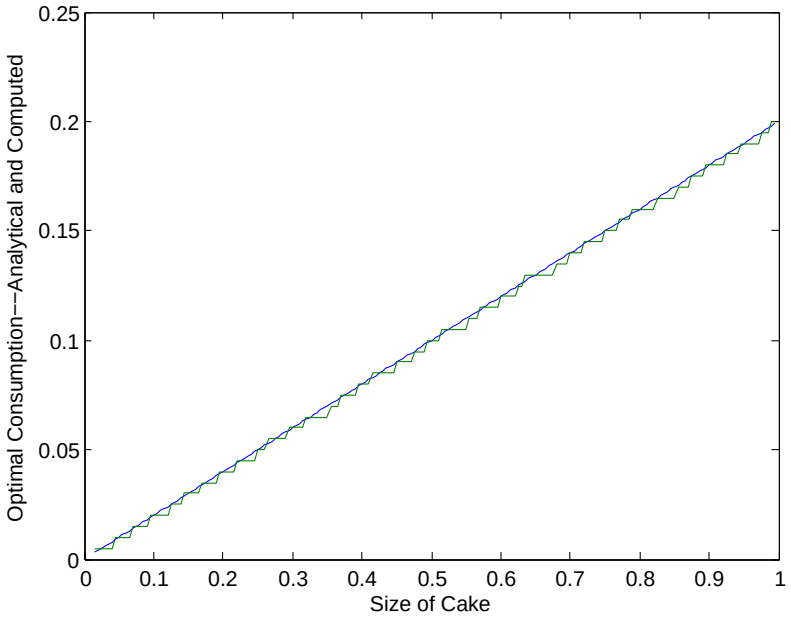
We are trying to find a function $V(W)$ that satisfies this condition for all W. In effect, we are looking for the fixed point of the contraction on $V(W)$ that is defined by equation (5). Fortunately the contraction mapping theorem tells us that not only does such a fixed point exist, but also we will converge to that fixed point through successive iterations on (5), regardless of our initial guess for $V(W)$. It is important to note that not all applications you will face will satisfy Blackwell's sufficient conditions for a contraction, so there is no guarantee that successive iterations will work, and if so, convergence may depend on your initial guess for $V(W)$. Even when Blackwell's sufficient

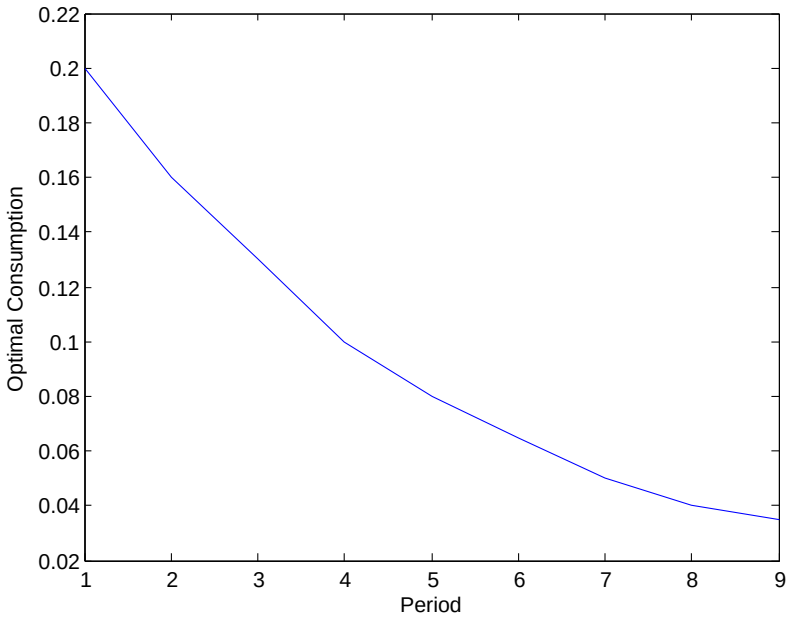
conditions do hold, the time required for your program to converge on the correct function $V(W)$ will depend on your initial guess. It is always worth spending some time to chose an intelligent guess for a starting point for your value function, rather than choosing an arbitrary function.

```
%% THIS FILE SOLVES THE DETERMINISTIC CAKE EATING MODEL
% based on example in Adda and Cooper, "Dynamic economics", MIT Press,
% pp. 16-20

clear;
maxIter = 3000;  % number of iterations
beta=0.8;    % discount factor
toler = 1e-10; % Tolerance

% state space
grid = 0.0001:0.005:1; % See comment 1
[y,dimW]=size(grid);

% Creating matrix that gives the consumption resulting from a choice % of W' (in columns)and given th
consumption = grid'*ones(1,dimW)-ones(dimW,1)*grid;
consumption(consumption<=0) = 1e-15; % Think why we do this.

% Initial guess for the value function
V = zeros(1,dimW);

gap=1;
round = 0;
converge=1;

% Value function iteration
while gap > toler;
    [newV,index] = max((log(consumption)+beta*ones(dimW,1)*V)');
    round = round+1;
    if round > maxIter;
        converge=0;
        break;
    end;
```

```
    gap = norm(newV-V);
    V = newV;
    end;

converge % Will tell us whether the value function coverged

% Plot analytical solution and approximated solution
analytic = (1-beta)*grid;
comput = grid-grid(index);

figure(1)
plot(grid(4:end),[analytic(4:end);comput(4:end)]);
xlabel('Size of Cake');
ylabel('Optimal Consumption--Analytical and Computed');

% Simulation
Windex = zeros(1,9);
Windex(1,1)=dimW;
for period = 1:10;
    Windex(1,period+1) = index(Windex(1,period));
end
Wsim = grid(Windex);
Csim = Wsim(1:9) - Wsim(2:10);

figure(2)
plot(1:9,Csim);
xlabel('Period');
ylabel('Optimal Consumption');
```

```
converge =

     1
```


Optimal Consumption--Analytical and Computed
Size of Cake
0.25
0.2
0.15
0.1
0.05
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1


Optimal Consumption
Period
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
1 2 3 4 5 6 7 8 9

**Comment 1:** In general it is up to you to choose how tight to create the grid for a given application. This poses a tradeoff. A high resolution grid will provide more precise solutions, but may take longer to converge (the curse of dimensionality). A lower resolution grid will converge faster, but will give less precise results. At the extreme, a sufficiently low resolution may lead you to absurd conclusions or odd results (what would have happened in the above example if you chose a grid of only 5 points, for example?).

A good strategy is to start from a low resolution grid, while you're still debugging your program and increase the grid's resolution once you know the program converges. Make sure, though, not to choose a grid with too small a resolution to obtain reasonable results.

**Comment 2:** Many matrix operations may seem complicated to set up. Most of them may be possible to replicate by using nested for loops to populate the matrix. This is a bad idea. MATLAB is a matrix-based application. It can perform matrix operations with (relative) ease. As model becomes more complex, and the curse of dimensionality starts to kick in, you will find computation time increasing exponentially when excessively using for loops. You should get used to working in matrix format from the outset.

**Food for thought:**

- What happens as you reduce the grid resolution? Why?
- What happens if you increase beta to 0.95? What's going on?!
- What happens if you try to simulate the model for 100 periods? Why do you obtain this result?
- Try to run the program with a resolution that is 1 or 2 orders of magnitude higher than in the current set up. Notice how long it takes the computer to perform the matrix operations, even for this relatively simple application.

All the above are examples of the great importance of choosing the proper resolution and of necessity of having a very good theoretical grasp of your model before solving it on the computer.

# 14 A Discrete, Stochastic, Cake Eating Problem

Let us now introduce uncertainty into the problem. We'll now assume that the agent is uncertain as to her future appetite. To allow for variations of appetite, suppose that utility over consumption is given by

$$\varepsilon u(c)$$

where $\varepsilon$ is a random variable representing the taste shock. For simplicity, assume that the taste shock takes on only two values: $\varepsilon_h > \varepsilon_l > 0$. Further we assume that the taste shock follows a first-order Markov process.

$$\pi_{lh} \equiv Prob(\varepsilon' = \varepsilon_h | \varepsilon = \varepsilon_l)$$

Suppose transition matrix is:

$$\begin{bmatrix} .6 & .4 \\ .2 & .8 \end{bmatrix}$$

Further suppose that the cake must be eaten in a single period. Perhaps we should think of this as the wine drinking problem, recognizing that once a good bottle of wine is opened, it must be consumed. Further we modify the transition equation to allow the cake to grow (depreciate) at rate $\rho = 0.95$, before being consumed. The cake consumption example is then a dynamic, stochastic discrete choice problem.

Let $V^E(W, \varepsilon)$ and $V^N(W, \varepsilon)$ be the values of eating a cake of size W now (E) and waiting (N), respectively, given the current taste shock Then

$$V^E(W, \varepsilon) = \varepsilon u(W) \tag{6}$$

$$V^N(W, \varepsilon) = \beta E_{\varepsilon'|\varepsilon} V(\rho W, \varepsilon') \tag{7}$$

where

$$V(W, \varepsilon) = \max\{V^E(W, \varepsilon), V^N(W, \varepsilon)\} \tag{8}$$

We solve the problem using the following code.

```
%%  THIS FILE SOLVES THE DISCRETE CAKE EATING MODEL —— Based on Cooper and
%  Adda
```

```
clear
clc
dimK=100;               % Size of cake space
itermax=60;             % maximum iteration
K0=10;
dimEps=2;               % size of taste shock space
ro=0.95;                % shrink factor for the cake
beta=0.95;              % discount factor

K=0:1:(dimK-1);         % grid for the cake
K=K0*ro.^K';   % Grid for cake size 1 ro ro^2
% K=K0*K';
eps=[.8,1.2];       % taste shocks
pi=[.6 .4;.2 .8];   % transition matrix

V=zeros(dimK,dimEps);   % Store the value function. Rows are cake size and columns are shocks
Vnext=zeros(dimK,dimEps);
dif=1;
Veat = kron(log(K),eps);
for iter=1:itermax;                   % loop for iterations
    iter;
    Vwait = (beta*pi*[V(1:dimK-1,:)' zeros(dimEps,1)])';
    for epscount=1:dimEps;
        Vnext(:,epscount)=(max([Veat(:,epscount) Vwait(:,epscount)]'))';
    end %epscount
    dif = norm(V-Vnext);
    V=Vnext;
    if dif<1e-6;
        break;
    end;
end  % end iteration loop

% plot the value function

figure(1)
plot(K,V)
title('Value Function')
```

```
xlabel('Size of Cake');
ylabel('Value Function');
legend('Low Taste','High Taste',0)
```