Getting Started with Matlab

Jerome Adda

February 4, 2003

Contents

1 Introduction

Matlab is a programming language which is used to solve numerical problems, including computation of integrations, maximizations, simulations. It is very similar to GAUSS, and closely related to more standard languages such as Fortran, Pascal or C++. Compared with the latter languages it is more user friendly and more matrix orientated. It is good at producing graphs.

2 Some Basic Features

2.1 Matlab Windows

When opening Matlab, you will see several windows, depending on the configuration. The Command window allows you to type in commands and display results. The Workspace window displays all the variables (matrices) in memory. The Command History window displays all the previous commands you have typed. Finally, the Directory window allows you to select a working directory.

2.2 Creating a Matrix

The basic object in Matlab is a matrix. To create a matrix, you need to give it a name (beware, Matlab is case sensitive):

```
A=[1 4 0; -1 2 1; 2 0 3]
```

This creates a 3 by 3 matrix and displays it on screen.

```
A=
     1 4 0
    -1 2 1
     2 0 3
```

There are other commands to create matrices:

```
B=ones(3,4)
```

creates a 3 by 4 matrix with ones. Similarly the command

```
C=zeros(2,3)
```

creates a 2 by 3 matrix with zeros. Finally,

```
D=rand(4,5)
E=randn(4,5)
```

creates two 4 by 5 matrices with random numbers drawn respectively from a uniform and a normal distribution.

Finally the command:

```
x=0:0.1:1;
```

creates a row vector with elements starting at 0 up to 1 with increments of 0.1:

```
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
```

This is very useful to create a grid.

2.3 Sum, Transpose and more

The usual matrix operations, sums and substractions, are defined in Matlab.

```
c=D+E; c=D-E;
```

Here the ; indicates that we do not want to see the result printed out on the screen. Note that the matrices should have the same dimension. To find the size of a matrix, you can use the command:

```
[rowD,colD]=size(D);
```

which returns the number of rows in a variable named rowD and the number of columns in colD. We could have used different names for these output variables. The multiplication is defined as:

```
c=D*E';
```

where ' denotes the transposition of a matrix, so that the sizes matches. There exists another operation which operates a multiplication cell by cell for two matrices of similar size. Suppose that:

```
A=[1 2;0 1];
B=[-1 0 ; 2 1];
c=A.*B
c=
  -1 0
   0 1
```

In this case, the first cell (row 1 column 1) of A is multiplied by the first cell of B and so forth. Similarly, we can define the division element by element:

```
c=A./B
warning: Divide by zero
-1 Inf
 0  1
```

The upper right cell now contains a value which is infinity, given that we have attempted a division by zero.

Finally, the inverse of a matrix is defined by the command inv:

```
inv(A)
 1 -2
 0  1
```

2.4 Min, Max, Mean...

Define a matrix

```
A=[1 2 3 4; 0 2 0 3; -1 3 1 2];
```

The function min returns the minimum values along columns:

```
 min(A)
 -1 2 0 2
```

To get the minimum along the rows, use the command:

```
 min(A,[],2)
 1
 0
-1
```

The 2 indicates that we are looking at the second dimension of the matrix. The min operator also works on higher dimensional matrices, by indicating along which dimension one want to find the minimum values. Similarly, the max operator returns the maximum along any dimension.

A number of Matlab operators are defined in a similar way:

```
 mean(A,2)
 2.50
 1.25
 1.25
```

returns the mean along the second dimension (rows). The operator median returns the median along a given dimension. The operator sum return the sum of all elements along a particular dimension:

```
sum(A,1)
 0 7 4 9
```

2.5 Concatenation of Matrices

Concatenation is the process of joining small matrices to make bigger ones. The pair of square brackets, [], is the concatenation operator. For example:

```
A=ones(2,2);
B=zeros(2,2)
c=[A B]
0 0 1 1
0 0 1 1
```

The two matrices A and B have been put side by side in c. We can also stack them:

```
c=[A;B]
 0 0
 0 0
 1 1
 1 1
```

2.6 Indexing a Matrix

To display or change only part of a matrix, we need to be able to refer to a particular cell or set of cells in this matrix.

```
A=[ 1 2; 0 3];
```

To change the value of the upper right cell, we would write

```
A(1,2)=3
1 3
0 3
```

where the first number indicates the row and the second the column. We can also change an entire column or row by using the command ":",

```
A(:,1)=zeros(2,1)
0 3
0 3
```

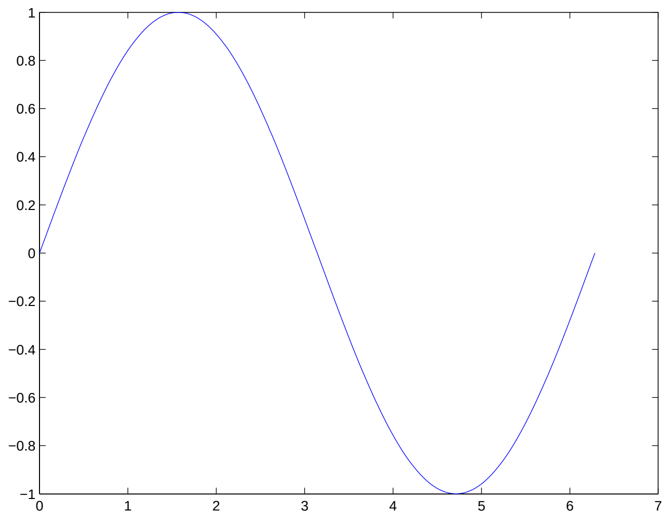
Note that we need to have conformable sizes. A(:,1) is a 2 x 1 vector and so is zeros(2,1).

3 Graphics

Matlab has extensive facilities for displaying vectors and matrices as graphs.

The basic command is `plot`. `plot(x,y)` creates a plot of y versus x:

```
x=0:pi/100:2*pi;
y=sin(x);
plot(x,y)
```



We can plot more than one line on this graph if `y` is a matrix. In this case each line has a separate color. You can embellish this graph by adding labels on the axes and a title:

```
xlabel('Year')
ylabel('Business Cycle')
title('Graph of Annual Business Cycle in Patagonia')
```

The command `axis([xmin xmax ymin ymax])` sets limits for the current x and y axis.

4 Creating a Batch File

For more complex calculations, involving a series of commands, working in the command window is rather difficult. We can write all the commands in a file and then run it. The matlab code files have an extension .m. To start one, click on File, New, M-file to open the Matlab editor. Then you can type in any command. To run the file, save it and go to the Matlab command window and type its name (without the extension .m). Or alternatively, there should be a run icon in the tool bar of the Matlab editor. For example, the following lines in a file named example1.m creates artificial data stored in `x` and builds a model:

$$y_i = x_i\beta + \varepsilon_i$$

and estimate $\hat{\beta}$ by OLS:

```
clear                              % clear the memory of all previous matrices
clc                                % clear the screen
sig=0.1;                           % defines the standard deviation of epsilon
x=rand(100,10);                    % creates a matrix of explanatory variables
b=rand(10,1);                      % creates a vector of coefficients
epsilon=randn(100,1)*sig;          % creates a vector of residuals
y=x*b+epsilon;                     % creates the dependent variable

bhat=inv(x'*x)*x'*y;               % OLS formula
[b bhat]                           % displays the true vector and the estimate
```

In this file the % is used to write comments next to the code. These comments are not taken into account by Matlab but are there to help the understanding of the code.

5 Flow Control

5.1 if Statement

The `if` statement evaluates a logical expression and executes a group of statement if the expression is true. The optional key words `else` and `elseif` provide for the execution of alternate groups of statements. An `end` keyword, which matches the `if`, terminates the last group of statements:

```
if a>b
  'greater'
 elseif a<b
  'less'
 else
  'equal'
end
```

5.2 Loops

Loops are used to repeat a group of statements a number of times. There are two commands used to execute loops, `for` and `while`. With the `for` command, Matlab repeats a group of statements a fixed number of times. A matching `end` concludes the statement:

```
m=zeros(22,1);
for i=1:22
 m(i,1)=sqrt(i);
end
```

This loop will fill in the vector `m` of size 22 by 1 with the square root of the index of the row. We could have written this more compactly as:

```
m=sqrt(1:22)';
```

Note that in many cases loops can be avoided by making use of matrix operations. This is useful as Matlab is not very quick in computing loops.

The `while` loop repeats a group of statements an indefinite number of times under the control of a logical condition. A matching `end` concludes the statement. For instance, if we want to approximate the value $\sqrt{2\pi}$, we can use the Stirling formula which states that:

$$\lim_n \frac{n!}{n^n e^{-n}\sqrt{(n)}} \simeq \sqrt{2\pi}$$

```
err=1;                          % initialize a variable with a big number
oldapprox=1;                    % stores the previous guess of (2 pi)^0.5
n=1;
while err>0.001                 % execute the loop until err is lower than .001
 oldapprox=approx;              % keep track of previous approximation
 approx= factorial(n)/(n^n)/exp(-n)/sqrt(n)
 err=abs(approx-oldapprox);     % calculate the error between two successive guesses
 n=n+1;                         % update the iteration number
end
[sqrt(2*pi) approx]             % displays the true value and the approximation.
```

6 Functions and Subroutines

When writing a big program, it is cumbersome to have all the codes in the same file. Sometimes, minor or technical computations are better handled separately as a subroutine or as called in Matlab a function. This is also useful when one has to call a same set of arguments repeatedly.

We can define functions in Matlab, by writing the lines of code into a separate file (with an extension .m). The function can accept input arguments and return output arguments. The name of the .m file and of the function should be the same. For instance, the approximation above can be written in a file `approximation.m`:

```
function [approx]=approximation(tol)
% Approximate the value sqrt(2*pi)
 err=1;                          % initialize a variable with a big number
 approx=1;                    % stores the previous guess of (2 pi)^0.5
 n=1;
 while err>tol                 % execute the loop until err is lower than .001
  oldapprox=approx;              % keep track of previous approximation
  approx= factorial(n)/(n^n)/exp(-n)/sqrt(n);
  err=abs(approx-oldapprox);  % calculate the error between two successive guesses
  n=n+1;                      % update the iteration number
 end
```

This function can then be called within another program or in the Command window, by typing

```
approximation(0.01)
```

This will produce an approximation, with a tolerance of 0.01.

Exercise 1: Construct a function called AR1 which simulates an AR(1) process over T periods:

$$y_t = (1-\rho)\mu + \rho y_{t-1} + \varepsilon_t \qquad \text{with } y_0 = \mu \quad \text{and } V(\varepsilon) = \sigma^2$$

The input parameters should be μ, ρ, σ and T. The output parameter should be a vector of size T x 1 containing the simulated values:

```
 [y]=AR1(mu,rho,sigma,100)
```

Exercise 2: Extend the previous function to simulate the AR(1) process simultaneously for N agents.

7 Help in Matlab

To search for help for a particular command, type `help` and the name of the command.

```
help factorial
```

This will display the syntax and some explanation for the particular command.

A more general command for help is `lookfor` which displays a list of commands which are related to the topic of interest:

```
 lookfor integral
```

will display all functions related to integration. A guide is also located at the address:

```
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml
```

8 Deterministic Cake Eating Problem

In this simple problem, the agent starts with a cake of size K. He has to decide how much to eat in all future periods. The program of the agent can be written using the Bellman equation:

$$V(K) = \max_c u(c) + \beta V(K - c)$$

where $V()$ is the value function, i.e. the total flow of utility over all future periods. β is the discount factor. This program can be rewritten as

$$V(K) = \max_{K'} u(K - K') + \beta V(K')$$

where K' is the size of the cake at the end of the period, using the fact that $K' = K - c$. To find the value function, we use the value function iteration methods. The iterations will be on

$$V_n(K) = \max_{K'} u(K - K') + \beta V_{n-1}(K')$$

So we start with a guess for $V_0(K)$ and iterate backwards. Suppose that $u(c) = \ln(c)$.

- we need a grid for the cake size. $\{K_0, \ldots, K_N\}$. This can be done in Matlab by typing:

  ```
  K=0:0.01:1;
  ```

  This defines an equally spaced grid between 0 and 1, incremented by steps of size 0.01.

- we need to store the successive values of V_n into a : N x $MaxIter$ matrix. We can initialize this matrix to contain zeros:

  ```
  V=zeros(N,MaxIter);
  ```

- compute the value for any given size of cake at the end of the period, i.e.:

  $$v_{ij} = u(K(i) - K(j)) + \beta V_{n-1}(K(j)) \qquad K(j) < K(i)$$

- search on the $\{K\}$ grid, the value K' which gives the highest utility flow. This amounts to finding the index j such that v_{ij} is maximal.

The program can be written as:

```
clear
MaxIter=30;
beta=0.9;
K=0:0.01:1;            % grid over cake size
[rk,dimK]=size(K);
V=zeros(dimK,MaxIter); % initialize the value function matrix

for iter=1:MaxIter    % iteration of the Bellman equation
   iter               % displays iteration number
   aux=zeros(dimK,dimK)+NaN; % matrix of all possible values of consumption
                             % initialize this to missing values.
                             % When taking the max, the missings are not
                             % taken into account.

   for ik=1:dimK           % loop over all size of cake at beginning of period
       for ik2=1:(ik-1)    % loop over all size of cake at end of period
           aux(ik,ik2)=log(K(ik)-K(ik2))+beta*V(ik2,iter);
       end
   end  % Note: these two loops do not fill in entirely the aux matrix
        % The size of cake next period has to be lower than the current one.

   V(:,iter+1)=max(aux,2);  % take the max value over vij ignoring
                             % the missing values
end

[Val,Ind]=max(aux,2);   % find the index of next period cake which
                        % maximizes the utility
optK=K(Ind);            % Tomorrow's optimal cake size
optK=optK+Val*0;
optC=K'-optK';          % optimal consumption

figure(1)
plot(K,V);
xlabel('Size of Cake');
ylabel('Value Function');

% plot optimal consumption rule as a function of cake size
figure(2)
plot(K,[optC K'],'LineWidth',2)        % plot graph
xlabel('Size of Cake');
ylabel('Optimal Consumption');
text(0.4,0.65,'45 deg. line','FontSize',18)   % add text to the graph
text(0.4,0.13,'Optimal Consumption','FontSize',18)
```

9 Discrete Cake Problem

In this problem, the consumer has to decide when to eat a cake. The agent can only take one of two actions, eat the entire cake now, or wait an additional period. If the agent decides to wait, the cake shrinks by a factor ρ each period. The agent wakes up each day with a taste shock ε drawn from a distribution. The program of the agent is:

$$V(K,\varepsilon) = \max[u(K,\varepsilon), \beta E_{\varepsilon'} V(\rho K, \varepsilon')] \qquad \rho \in [0,1]$$

where K is the size of the cake at the beginning of the period and β is the discount factor. The program of the agent is stochastic, so E'_ε is the expectation operator over future values of this taste shock, ε'.

As before, solving this program numerically involves several steps:

- defining a grid for the size of the cake. A smart way of defining the grid is $\{K_0, \rho K_0, \rho^2 K_0, \ldots\}$, where K_0 is the initial cake size. This way, the next cake size will be on the grid too.

  ```
  K0=1;  % max cake size
  K=K0*ro.^(0:dimK)';        % dimK is the size of the grid
  ```

- defining a grid for the taste shock:

  ```
  eps=(MinEps:0.05:MaxEps)';   % MinEps and MaxEps are the minimum
                              % and maximum value for eps
  dimEps=length(eps);         % size of the taste shock grid
  ```

- defining a transition matrix for ε, i.e.

  $$\pi_{ij} = Prob(\varepsilon' = j | \varepsilon = i)$$

  For simplicity, suppose that ε is i.i.d., so that $\pi_{ij} = 1/dimEps$

  ```
  pi=ones(dimEps,dimEps)/dimEps;
  ```

- initialize the matrix which contains the value function:

  ```
  V=zeros(dimK,dimEps);   % remember: rows are cake size
                          % and columns are taste shocks.
  ```

- Compute the utility of eating a cake of size $K(ik)$ now, with a taste shock $\varepsilon(ieps)$:

  ```
  Vnow=log(K(ik)*eps(ieps));
  ```

- compute the utility of waiting one more period if the cake is of size $K(ik)$ and the taste shock is $\varepsilon(ieps)$. The expectation is now a weighted sum given the discretization of the taste shock:

  ```
  Vwait=pi(ieps,:)*V(ik+1,:)';
  ```

- Construct a loop over
  - all possible taste shocks $\varepsilon(ieps)$.
  - all possible cake sizes $K(ik)$
  - all iteration of the Bellman equation.

Once the program has been solved, we can back out the threshold ε^* such that the agent is indifferent between eating the cake now and waiting:

$$u(K,\varepsilon^*) = \beta E_{\varepsilon'} V(\rho K, \varepsilon')$$

For any cake of size K, we can evaluate $E_{\varepsilon'} V(\rho K, \varepsilon')$ as the mean of the value function over all realization of ε, given the i.i.d. assumption.

```
EnextV=pi(ieps,:)*V(ik+1,:)';    % expected value next period
threshold(ik,ieps)=exp(EnextV-log(K(ik))); % matrix of thresholds
```

Next, we want to plot the threshold as a function of the cake size.

10 Simulation of a Model

We are now interested in the simulation of the stochastic discrete cake eating model. We would like to simulate the behavior of many agents over several periods to determine when the cakes will be eaten.

```
 clear
 dimK=100;
 beta=0.9;
 K0=1;  % max cake size
 K=K0*ro.^(0:dimK)';        % dimK is the size of the grid

 MinEps=0.95;
 MaxEps=1.05;
 pi=ones(dimEps,dimEps)/dimEps;   % transition matrix for i.i.d. process
 eps=(MinEps:0.05:MaxEps)';   % MinEps and MaxEps are the minimum
                             % and maximum value for eps
 dimEps=length(eps);         % size of the taste shock grid

 V=zeros(dimK,dimEps);   % Store the value function.
                         % Rows are cake size and columns
                         % are shocks
 auxV=zeros(dimK,dimEps);
 err=1;
for iter=1:itermax;       % iterating on the Bellman equation
    [iter]                % displays iteration number
    for ik=1:dimK-1;            % loop over size of cake
        for ieps=1:dimEps;      % loop over taste shocks
        Vnow=log(K(ik)*eps(ieps));  % utility of eating the cake now
        Vwait=pi(ieps,:)*V(ik+1,:)';
        auxV(ik,ieps)=max(Vnow,beta*Vwait);
    end  % end loop over taste shock
    end  % end loop over size of cake
    err=V-auxV;
    V=auxV;
end

threshold=zeros(dimK,dimEps); for ik=1:dimK-1;
    for ieps=1:dimEps;
        EnextV=pi(ieps,:)*V(ik+1,:)';
        threshold(ik,ieps)=exp(EnextV-log(K(ik)));
    end
end
```

Simulations

```
eps=rand(T,S);

eat=zeros(T,S);


for t=2:T    % loop over all periods
 for s=1:S   % loop over all agents
  eat(t,s)=eps>threshold(t,1);   % eating if shock is bigger than threshold
    if eat(t-1,s)==1   % check whether the cake has already been eaten
       eat(t,s)=1;
    end
end
```