Microdata and Matlab: An Introduction

Abi Adams Damian Clarke Simon Quinn

January 23, 2014

Contents

Introduction

> Brian: Look, you've got it all wrong! You don't need to follow me. You don't need to follow anybody! You've got to think for yourselves! You're all individuals!
>
> Crowd: *Yes! We're all individuals!*
>
> Brian: You're all different!
>
> Crowd: *Yes, we are all different!*
>
> *Monty Python, "The Life of Brian"*

Two things are true about the seven billion people on this planet. First, no two of us are the same. Second, we all respond — in more or less sophisticated ways — to the incentives that we face. These two characteristics — *heterogeneity* and *agency* — are part of what it means to be human. They also present a fundamental challenge for us as economists: ideally, we should build and estimate models that treat people as responding to incentives, and that allow different people to face different circumstances with different preferences. This book is designed as a practical guide for theory-based empirical analysis in economics. In short, it is about fitting models to data while respecting the heterogeneity and the agency of the people whose behaviour we study.

At its core, this is what economics is about: fitting models to data. Sometimes,

the data might simply be casual observations of the world: as economists, we might develop 'stylised facts' about the way that people behave, and try to build new models to explain what we see. Other times, we use aggregate figures, on concepts like GDP and inflation. And sometimes, we use data that has been collected from individuals: individual households, individual firms, individual workers, and so on. This is what we term 'microdata' — and that's what we will focus on in this book. In sum, this is a book about fitting microeconomic models to microdata, to improve our understanding of human behaviour.

MATLAB is *not* the central plot of this story — though it is certainly a lead character. Our goal in this book is not, in any general sense, to teach MATLAB. There are plenty of good books available to do that.[1] Rather, our goal is to discuss a series of standard problems in applied microeconometrics, and show how MATLAB can be used to tackle each one. Of course, it is quite unlikely that *any* of the specific models we study will fit perfectly any particular empirical problem that you face — but that, in a sense, is exactly the point. There are many excellent textbooks that cover standard microeconometric methods, and several excellent software packages for implementing those methods — often requiring just a single line of code for any given estimator.[2] Of course, all of these methods can be implemented in MATLAB, but this is not where MATLAB's comparative advantage lies.

Rather, the beauty of MATLAB is its extraordinary flexibility. MATLAB opens entire classes of new models — and, therefore, new ideas — that standard econometrics packages do not allow. Of course, when it comes to econometric algorithms, there will always be an important role for pre-bottled varieties off the shelf. But in this book, we will brew our own...

[1] For example, you could see Hahn and Valentine's Essential Matlab for Engineers and Scientists'.
[2] Without loss of generality, think of (i) Cameron and Trivedi (2005) and (ii) Stata...

Chapter 1

Entering the 'Matrix Laboratory'

1.1 Introduction

> Let's start at the very beginning,
> A very good place to start.
>
> *Hammerstein, "The Sound of Music"*

MATLAB is a computer language for doing maths. Its name is short for '**mat**rix **lab**oratory', and its purpose is simple: to provide a very powerful and very flexible way of solving mathematical problems. In this chapter, we will run a series of exercises to illustrate the simplicity with which MATLAB handles matrices. This will provide a foundation for more complicated concepts and structures that we will cover later.

The simplest way to interact with MATLAB is through the 'command line', and this is where we will begin. The command line can operate like a calculator.

We can see this by a none-too-complicated calculation:

```
>> 1 + 1
ans =
     2
```

We can use the command line to create matrix variables to store our results. Let's start with a simple variable, `y`:

```
>> y = 1 + 1
y =
     2
```

As its name suggests, MATLAB is designed to deal with matrices very simply and effectively; in MATLAB, we can enter any variable as a matrix, simply by using commas to separate columns and semi-colons to separate rows. For example, let's create a simple 3×2 matrix (which we will call '`x`'), and then multiply that matrix by two:

```
>> x = [1, 2; 3, 4; 5, 6]
x =
    1   2
    3   4
    5   6

>> 2*x
ans =
     2   4
     6   8
    10  12
```

We can check which matrices are stored in memory by using the commands `who` (for a short summary) and `whos` (for a longer summary):

```
>> who
Your variables are:
ans  x    y


>> whos
  Name      Size            Bytes  Class     Attributes
  ans       3x2                48  double
  x         3x2                48  double
  y         1x1                 8  double
```

MATLAB reports that we have three matrices in memory: `ans`, `x` and `y`. We should not be surprised to see `x` and `y` in memory; we just created these matrices, and we can check their contents simply by entering the matrix names at the command line:

```
>> x
x =
     1     2
     3     4
     5     6

>> y
y =
     2
```

The matrix `ans` may be more confusing. This matrix stores MATLAB's most recent answer that has not been stored in any other matrix. If we enter `ans` at the command line, we will ask MATLAB to recall its response to our earlier expression '`2*x`':

```
>> ans
ans =
     2     4
     6     8
    10    12
```

Notice that if we enter another expression that is not assigned to any other matrix, MATAB will use `ans` to store this new expression:

```
>> 5 * 5
ans =
    25

>> ans
ans =
    25
```

MATLAB has a very large range of mathematical operators. But our goal here is *not* to provide any comprehensive discussion of these. MATLAB provides excellent help files (to say nothing of a large range of online resources), and we don't want to use this book to describe in detail what is available elsewhere. For example, to learn about MATLAB's arithmetic operators, a researcher could simply search online to find the relevant help page. To learn the syntax of a particular command, we could use MATLAB's extensive help documentation from the command line:

```
>> help ones
```

Instead of discussing an ungainly list of commands and operations at this point, we will explore different techniques as they become relevant for our analysis of various microeconometric models. And so we begin — with an illustration of the most popular microeconometric technique of them all...

1.2 Running an OLS regression

A simple way to become familiar with the basic workings of an econometric program is to run an Ordinary Least Squares regression. In some ways, this is the "Hello World!" of the applied researcher. "Hello World!" is the test program which many computer programmers run when they first learn a language —

to discover its basic syntax, and to ensure that it is running correctly. Such programs simply print the words "Hello World!" and then terminate.[1] While the OLS regression requires a few more lines than printing a simple statement, it will allow us to work with the basic building blocks that we have already introduced.

Almost every applied research is familiar with Stata, and almost everyone who is familiar with Stata has, at some point or another, come across the `auto.dta` dataset. This is a dataset included by default when Stata is installed, and contains data on a series of models of cars in 1978. We will briefly[2] ask that you open Stata and, using the auto dataset, run a regression of mileage per gallon upon the car's weight and price. We will denote mileage per gallon by the $N \times 1$ vector $\boldsymbol{y}$, and will use the $N \times 3$ vector $\boldsymbol{X}$ to stack values of (i) price, (ii) weight, and (iii) the number 1. Our OLS model is, of course:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \tag{1.1}$$

where $\boldsymbol{\beta}$ is a 3×1 vector of parameters. We denote the OLS estimate of $\boldsymbol{\beta}$ as $\hat{\boldsymbol{\beta}}$; we can find $\hat{\boldsymbol{\beta}}$ straightforwardly in Stata...

```
. sysuse auto
(1978 Automobile Data)

. reg mpg price weight, noheader
```

```
------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       price |  -.0000935   .0001627    -0.57   0.567     -.000418    .0002309
      weight |  -.0058175   .0006175    -9.42   0.000    -.0070489   -.0045862
       _cons |   39.43966   1.621563    24.32   0.000     36.20635    42.67296
------------------------------------------------------------------------------
```

Let's now write these three variables to the file `auto.csv`:

[1] In MATLAB, such a program would be quite simple, containing just `disp('Hello World!')`.
[2] And apologetically, for any readers expecting that this book would be based entirely on MATLAB...

```
. outsheet mpg price weight using auto.csv, nonames comma
```

To run the same regression in MATLAB, we first need to import the data in `auto.csv`. Before being able to import this data, we must ensure that our current working directory contains the auto file. In order to move to this file, we use the commands `pwd` (print working directory), `cd` (change directory) and `ls` (list the contents of the current directory). After choosing the correct working directory, we can import using `dlmread`:[3]

```
>> DataIn   = dlmread('auto.csv');
>> X        = DataIn(:, 2:3);
>> size(X)

ans =

    74     2

>> X        = [X, ones(74,1)];
>> y        = DataIn(:,1);
```

The above block of code involves various new commands, so ensure that you are able to run each command without problems in your MATLAB window. Try running each command without the semi-colon at the end of the line; this will allow you to see the full output each time. The most important command is `dlmread`, which allows us to read-in the data from our file `auto.csv`. Here we store the entire dataset as a matrix named `DataIn`. Remember that if we are interested in ensuring that auto.dta has imported correctly, we could use the command `whos('DataIn')` to see the details.

Here we start to see how MATLAB is structured around matrices. Rather than storing data as a series of scalar variables that can be viewed in a browser, we

[3] We encourage you to type all code displayed above in to your MATLAB window manually. If you copy and paste the above code directly, you may find that you have trouble with the single quote operators around 'auto.csv'. In reality, what MATLAB requires is a straight apostrophe (which looks like this: ') around the file name. If you enter the typical single quote operators (which look like this: '') you may find that MATLAB will not allow you to move on to the next line at the command prompt. If this happens, you can exit a given line by pressing the control and c keys in unison.

have stored our data as 74×3 matrix, which then must be manipulated if we are interested in working with a matrix of independent variables and a vector for the dependent variable `mpg`. This is what we do in the remaining lines of the above block of code: first we extract our 74×2 matrix `X` and add a vector of ones, and then we create the vector `y`. It is worth noting here that the notation `DataIn(:, 1)` implies that we take data from every single row (':') of column 1 in matrix `DataIn`. This code is also useful in illustrating precisely *how* as microeconomists we are likely to deal with matrices in MATLAB. Whilst the previous section of this chapter has suggested that we can enter matrices by hand at the command line (parsing with commas and semi-colons), it is unlikely that this will be a frequent exercise. Generally we will either read in data directly as a matrix (as we have done here), or will use MATLAB's matrix-based operations to simulate data from economic models.

Now that we have two matrices (`X` and `y`) that contain the relevant data from Stata's auto dataset, we can run our regression. This requires little more than introductory econometrics, namely the formula:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}(\boldsymbol{X}'\boldsymbol{y}). \tag{1.2}$$

MATLAB's syntax follows equation 1.2 quite closely. The only specialised function that we require is `inv`, which allows us to invert our $\boldsymbol{X}'\boldsymbol{X}$ matrix:

```
>> XX=X'*X;
>> Xy=X'*y;
>> beta=inv(XX)*Xy

beta =

   -0.0001
   -0.0058
   39.4397
```

Here we have calculated our coefficient matrix `beta`. You will notice that our

result is equal to those coefficients which we calculated earlier in Stata.[4] We may also be interested in ensuring that beta is correct to more than four decimal places. We can do this by changing MATLAB's output format to the *long* format, (`>> format long`) which displays up to 15 digits for 'double' variables.[5] Try changing the format and then printing out beta. How does this compare to what we did earlier in Stata? (To change back to the traditional output format, just enter `format` once again.)

For those of you interested in jumping ahead at this point, we refer you to exercise (a) at the end of this chapter. This exercise provides a chance for you to 'get your hands dirty' with some coding of your own...

1.3 The beauty of functions

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> ... Readability counts...
>
> *Tim Peters, The Zen of Python*

So far, our analysis has simply involved typing instructions into MATLAB's command line. This is effective, but not very efficient. What we need is a method of saving commands so that we can run them later — and, if necessary, run them many times, with different data, different parameters, and different options. In MATLAB, we can do this with an 'M-file'. We enter M-files through MATLAB's Editor window, which we can access by typing `edit` at the command line. M-files are to MATLAB what do-files are to Stata.

[4] Note that there is nothing (except perhaps a desire for clear exposition) that stops us from calculating β in a single step. This would look like: `beta=inv(X'*X)*X'*y`, or alternatively using the functionality of MATLAB's backslash (`mldivide`): `beta = (X'*X)\(X'*y)`.

[5] We resist the temptation to dive into a tangential discussion about the different precision with which MATLAB can store numbers. You can look this up by searching for concepts like 'double' and 'single'.

The most useful application of an M-file is to define a *function*.[6] In MATLAB, a function is a special type of program. Functions are special for three reasons:

(i) Functions can accept *inputs*.

(ii) Functions can return *outputs*.

(iii) Each function is *self-contained*; this means that each function can access *only* those variables that are passed to it as an input, and can store variables *only* through returning them as outputs.[7]

1.4 A function for OLS regressions

The easiest way to understand functions in MATLAB is through an example. We will return to the OLS regression that we ran earlier.[8] A regression is a perfect candidate for a function; each OLS regression is computationally equivalent, however the inputs and outputs for each regression will vary depending upon the name of the $\boldsymbol{X}$ and $\boldsymbol{y}$ variables that we are interested in analysing. In this way, we may be interested in permanently having a function available that we can call to calculate regression results. This is useful to save time when typing in commands at the command line, and to limit careless mistakes from typing in the calculation of $\boldsymbol{\beta}$ many times.

Here is an example of a function that we have written, called `regress`. You should be able to open this file in the MATLAB editor by opening the file `regress.m`.

[6] We can also use an M-file to define a *script*, but we will not spend much time discussing scripts.

[7] Strictly speaking, a function could save a variable to a file on the disk — but this is an unusual exception to the rule, and not one that we will often want to use.

[8] OLS is a useful illustration of MATLAB's basic concepts and basic functionality. But we will leave these sorts of standard econometric applications after this chapter. In many respects, it is much easier to implement standard estimators in Stata — and, if you *would* prefer to use MATLAB, there is an extensive set of functions already available through the Econometrics Toolbox.

```
function [beta, se] = regress(y,X)
% regress(y, X) runs an OLS regression of the n*k matrix
% of k independent variables, on the dependent variable y
% (an n*1 vector).  regress returns a parameter vector of
% coefficients called beta and se.

%*** (1) Calculate the coefficients ***

beta    =   (X'*X)\(X'*y);

%*** (2) Calculate the standard errors ***
yhat    =   X*beta;
u       =   yhat - y;
N       =   length(y);
K       =   size(X, 2);
sigma   =   sum(u.*u)/(N-K);
v_mat   =   sigma * inv(X'*X);      % Covariance matrix
se      =   diag(sqrt(v_mat));      % Standard errors

return
```

There are a number of things worth highlighting here, either because they are required for the code to run, or because they are good practice when writing functions.

(i) The first line of the function tells MATLAB (a) the *name* of the function (`regress`), the *inputs* to the function (`y` and `X`), and (b) the *outputs* from the function (`beta` and `se`). The first line shows the correct syntax for this; we always start a function by some version of:

```
function [output] = name(inputs)
```

Critically, this does *not* mean that when we call the function `regress` we must use variables named `y` and `X`. Instead, it just means that, *within the*

program, the variables we have introduced will be *locally* referred to as `y` and `X`. This will become apparent when we run the function shortly.

(ii) There are various lines of text which immediately follow the first line, each of which is prefaced by the % symbol. MATLAB reads the % symbol as saying 'skip this line'. In this way, the lines of comments can then be thought of as an explanation (either for other users, or for ourselves in the future) to help understanding of our code. As an added benefit, those lines of comments which directly follow the function are included as the help file to the function. When we type `>> help regress` at the command line, the output will remind us what we need to input, and what we should expect as output.

(iii) The function assigns values to the matrices `beta` and `se`. These are the names of the outputs in the first line. This means that when the function finishes running, it will return as outputs these assigned values.

(iv) Note that the function generally 'looks nice'.[9] In particular, note that there are subheadings to show the main parts of the calculation, comments (after the '%' symbol) to explain the operation of several of the lines of code, and the '=' signs are tabbed to the same alignment.

Let's use our function to repeat the regression from section 1.2. Assuming that the 'auto' data from section 1.2 is still in memory, we need simply pass this data to the function using the syntax of `regress` that we have defined. We can do this from the command line[10]:

```
>> regress(y, X)

ans =
```

[9] Even if we say so ourselves...

[10] If you don't save the function in MATLAB's current working directory (which we can see using the `cd` command), you will need to tell MATLAB where the M-file can be found. This can be done by using the `addpath` command. For example, if you've saved the function in a folder called 'C:/MATLABcourse/', you should enter `addpath C:/MATLABcourse`. Doing this, you'll come across a nice time-saving feature of MATLAB: tab completion. If you enter part of the path and press the tab key, MATLAB will complete the path address if only one unique ending exists, or list all available ways the path could end if multiple endings are possible.

```
   -0.0001
   -0.0058
   39.4397
```

As explained earlier, we do *not* need to refer to our variable names as `y` and `X`; these are the names that MATLAB will use *within* the function `regress`, but this does not constrain the way that we *use* that function. For example, let's create two new variables, taking the values of `y` and `X`, and run the regression again:

```
>> barack = y;
>> hilary = X;
>> regress(barack, hilary)

ans =

   -0.0001
   -0.0058
   39.4397
```

This is fine if we just want to display our regression results on the screen — but what if we want to store the results in a variable (say, `OLS_beta`)? This is straightforward: we simply assign the variable as in section 1.2, but have the variable refer to a calculation using our function:

```
>> OLS_beta = regress(barack, hilary)

OLS_beta =

   -0.0001
   -0.0058
   39.4397
```

As expected, the function `regress` returns the same regression results as in section 1.2. However, if you compare the output to the definition of the function

`regress`, you might ask a curious question: *whatever happened to the variable* `se`? When we programmed the file `regress.m`, we specified the output as '`[beta, se]`' — but, so far, `regress` has reported only `beta`. The reason is that, when we ran `regress`, we have only *asked* for `beta`: if we call a function from the command line, or assign the result of a function to a single variable, MATLAB will only return the *first* output variable. We can recover `beta` *and* `se` by assigning *both* of these variables jointly:

```
>> [OLS_beta, OLS_se] = regress(barack, hilary)

OLS_beta =

   -0.0001
   -0.0058
   39.4397


OLS_se =

    0.0002
    0.0006
    1.6216
```

We might even want to ask MATLAB to report a horizontal concatenation of `OLS_beta` and `OLS_se`, just to make things look nice:

```
>> [OLS_beta, OLS_se]

ans =

   -0.0001    0.0002
   -0.0058    0.0006
   39.4397    1.6216
```

Hopefully, we can now start to appreciate the beauty and simplicity of functions. *Yes*, it is true that functions can save us from repeating a lot of unnecessary

typing — and, *yes*, it is true that functions can help to avoid careless mistakes. But the true beauty of functional programming is that we can replace a *number* with the *solution to a mathematical expression* — and do so with a syntax that is both simple and intuitive. For this reason, functions will be fundamental to everything we do in the rest of this book.

1.5 A simple utility function

We are, of course, familiar with many types of functions from our microeconomic theory. One particularly important building block is the utility function, which we use to map the quantity of goods an individual consumes to his or her payoff. Needless to say, this has all the ingredients to be used in a MATLAB function: it accepts inputs (goods consumed), it returns outputs (utility), and it is self-contained, depending entirely upon the inputs and a number of technology parameters.

Let's consider the Cobb-Douglas utility function. In this case our output will be utility, u, and our inputs good 1, x_1, and good, x_2. For now, we will take a very simple form of the Cobb-Douglas function:

$$u(x_1, x_2) = x_1^{1/2} \cdot x_2^{1/2}.$$

Let's have a look at how this would be set up in MATLAB.

```
function u = utility(x1, x2)

% Function to calculate utility given a two-good
% Cobb-Douglas specification.

u   =   (x1^0.5) * (x2^0.5);

return
```

Hopefully, this works well; we can confirm, for example, that the function returns correct answers for a few different bundles...

```
>> utility(1, 4)

ans =

     2

>> utility(3, 3)

ans =

    3.0000
```

Of course, we rarely want to use MATLAB merely to calculate a *single* number. We need an elegant way of dealing with multiple possible combinations of x_1 and x_2. Suppose that, for some reason, we want to find utility for $x_1 = 5$ and $x_2 \in \{1, \ldots, 10\}$. Let's create a vector `x1` and a scalar `x2` to represent this:

```
x1  = [1:10]';
x2  = 5;
```

We now have ten combinations of (x_1, x_2) for which we need to find $u(x_1, x_2)$. We *could* have `utility` operate ten *separate* times — for example, using a loop.[11] But this is very inefficient. Instead, we should have `utility` run *once*, and operate on the entire matrix `x1`. This is known as *vectorising*. Having defined `x1` and `x2`, we should be able simply to enter:

```
utility(x1, x2)
```

But we have a problem! Look again at the function `utility`. As written, the function works perfectly well for *scalars* `x1` and `x2`, but doesn't work for vectors (or, more generally, for matrices). This is because the operators used there

[11] We will introduce loops in Chapter 3.

— the power operator and the multiplication operator — are understood by MATLAB to refer to matrices. We managed to get the correct answers when entering two scalars (for example, when we calculated `utility(1, 4)`), because the scalar/matrix distinction did not matter in this simple case. But our function doesn't work for the more general case.

Fortunately, MATLAB has an elegant solution: we can modify both the power operator and the multiplication operator so they work 'element-by-element'. For both the power operator and the multiplication operator, we can do this by introducing a leading '.'. Let's go back and fix our function to allow for this...

```
function u = utility(x1, x2)

% Function to calculate utility given a two-good
% Cobb-Douglas specification.

u   =   (x1.^0.5) .* (x2.^0.5);

return
```

We now have a utility function that is correctly defined for matrices. This is very powerful — among other advantages, we can now visualise our function very efficiently. Let's suppose that we want to see how our function behaves for $(x_1, x_2) \in [0, 3] \times [0, 3]$. We can create a *meshgrid* to cover this two dimensional space (discretised on unit intervals):

```
>> [x1, x2] = meshgrid([0:3], [0:3])

x1 =

     0     1     2     3
     0     1     2     3
     0     1     2     3
     0     1     2     3
```

```
x2 =

     0     0     0     0
     1     1     1     1
     2     2     2     2
     3     3     3     3
```

Hopefully, it is clear what is going on here: we have defined a matrix `x1` and a matrix `x2` such that `x1` and `x2` cover the grid $\{0, 1, 2, 3\} \times \{0, 1, 2, 3\}$. With a single operation, we can now calculate utility for this entire grid:

```
>> u       = utility(x1, x2)

u =

         0         0         0         0
         0    1.0000    1.4142    1.7321
         0    1.4142    2.0000    2.4495
         0    1.7321    2.4495    3.0000
```

We can then visualise this with the '`surfc`' command:

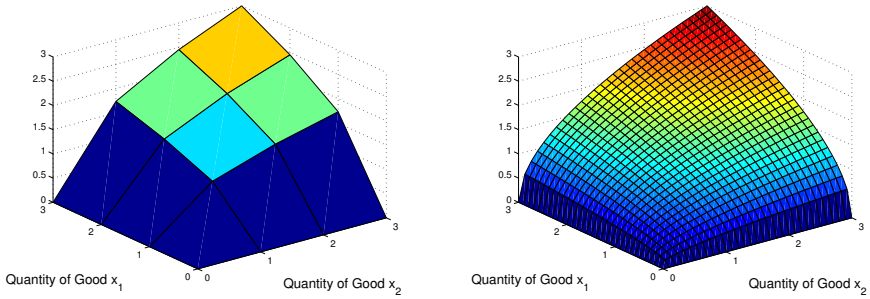
```
>> surfc(x1, x2, u)
```

Of course, we would really like to visualise this over a finer grid. This is easy using `meshgrid`:

```
[x1, x2]    = meshgrid([0:.1:3], [0:.1:3]);
u           = utility(x1, x2);
surfc(x1, x2, u)
```

As you will see in figures 1.1(a) and 1.1(b), the difference is quite stark, although

the complexity in coding each example is virtually identical once we have set up our function. We hope that this is something which holds throughout much of this book: while some things may seem initially quite simple, the basic methods presented here can be generalised to solve and visualise functions of arbitrary complexity. We will consider a more complicated function in the exercises that follow.



(a) Rough Mesh Grid

(b) Fine Mesh Grid

Figure 1.1: Cobb-Douglas Utility

1.6 Review and exercises

COMMAND	BRIEF DESCRIPTION
`whos`	Describes variables currently in memory
`help`	Describes a function along with its syntax
`lookfor`	Searches all M-files (including help files) for a keyword
`ones`	Creates an array of ones
`cd`	changes current working directory
`ls`	Lists content of current working directory
`pwd`	Prints the location of the current working directory
`dlmread`	reads in a comma-separated values file
`disp`	Prints text to the output window
`inv`	Inverts a matrix
`size`	Displays the size of an array
`format`	Sets the format of the output
`mldivide`	An efficient way to solve matrix division
`function`	Define a function which can be called from the command line
`addpath`	Adds a directory to the places MATLAB searches when a command is called
`meshgrid`	Replicates vectors to form a rectangular matrix
`surfc`	Draws a three-dimensional surface plot

Table 1.1: Chapter 1 commands

Exercises

(a). We have used MATLAB to recreate Stata's point estimates in a regression function. Can you now generate the same standard errors? We suggest you try this exercise *before* looking at the regress function in section 1.4, or skip to exercise (b) if you've already worked through this section carefully.

(b). Suppose that we want to generalise our utility function slightly, so that we have:

$$u(x_1, x_2, \alpha) = x_1^{\alpha} \cdot x_2^{(1-\alpha)}.$$

Create a new function, `utility2.m`, to accommodate this. Check that `utility2.m` matches the behaviour of `utility.m` for the special case $\alpha = 0.5$. Repeat the visualisation exercise. How does variation in α change the shape of u?

(c). Suppose that a consumer has a utility function of the form:

$$u(x) = -\exp(-r \cdot x).$$

Suppose that x is drawn from a Normal distribution with mean μ and variance σ^2 (that is, $\mu \sim \mathcal{N}\left(\mu, \sigma^2\right)$). An insurance company offers the consumer a product with a guaranteed lower limit, g. In effect, the insurance company says, "*If you buy our product and $x < g$, we will pay the difference, so you will get g. If you buy our product and $x > g$, we will do nothing, so you will just keep x.*".

(i) Interpret the paramter r.

(ii) What is the certainty equivalent if there is no insurance product?

To answer this question, you may rely on the following result (which applies for this special 'exponential-normal' case):

$$\mathbb{E}\left(u\right) = -\exp\left[\frac{1}{2} \cdot \sigma^2 \cdot r^2 - \mu \cdot r\right]. \tag{1.3}$$

(iii) What is the expected utilty with the insurance product? (Assume that the insurance company provides the product for free.)

To answer this question, you may rely on the following result (which, again, applies just for this special 'exponential-normal' case):

$$\mathbb{E}\left(u \,|\, x > g\right) = -\exp\left[\frac{1}{2}\cdot\sigma^2\cdot r^2 - \mu\cdot r\right]\cdot\left[\frac{1-\Phi(a+\sigma\cdot r)}{1-\Phi(a)}\right], \qquad (1.4)$$

where $a = \dfrac{g-\mu}{\sigma}$, and $\Phi(\cdot)$ is the *cdf* of the Normal.

(iv) (For MATLAB...) Assume now that $\mu = 0$ and $\sigma^2 = 1$. Define $s(g,r)$ as the consumer's surplus — in utility terms — from having the insurance product. Show the function $s(g,r)$ for $(g,r) \in (-3,3)\times(0,1.5)$.

Chapter 2

Optimisation

> Mason: Are you sure you're ready for this?
> Goodspeed: I'll do my best.
> Mason: Your *best*?!
>
> *The Rock*

Now the fun begins. Constrained optimisation is a fundamental tool of economists. As Lazear (2000) put it:

> Even when evidence suggests that the theories are wrong, we do not drop the assumption of maximisation. Instead, our approach is to think more carefully about the nature of the model set up, but not about the rationality of the individuals making the choices. Economists are rarely willing to assume that individuals simply do not know what they are doing.

Essentially, there are two ways that optimisation matters for economists:

(i) We need to use optimisation *ourselves*, in order to find the best possible fit for our model (where 'best' is defined by some particular objective function).

(ii) We treat *agents* as optimising — for example, we model consumers as maximising utility, firms as maximising profits, and so on.

In many respects, these two concepts are fundamentally different. On the one hand, we treat agents *as if* they optimise, as a way of *specifying* a model. On the other hand, we *actually* optimise, as a way of *estimating* our model. We should *always* keep these two concepts distinct in our minds. Yet each concept involves an optimisation problem — and the principles and methods that we use for the two problems are remarkably similar. In this chapter, we will consider each in turn.[1]

2.1 Estimating models with optimisation

Before we go too far down the path of examining how *other* microeconomic agents optimise, we'll consider how we as researchers use optimisation. As we suggested in point (i) above, this process starts with some objective function that we seek to resolve. Then, by using MATLAB's optimisation commands (which we'll discuss over the course of this chapter), we ask MATLAB to find the highest (or more precisely, the lowest), value for this function given the entire domain of inputs.

As we already have a useful benchmark from our regression estimates in section 1.2, this seems like a reasonable place to start. In chapter 1, we discussed how OLS regression estimates could be determined by inverting matrices. Of course every time we run an OLS regression, we are also solving for some well defined objective function. There are a number of ways that we could think about this objective function. The typical way is as the sum of the squared error terms (or sum of squares), however, you will perhaps remember that OLS estimates can also be calculated by maximum likelihood.

[1] And soon, we will consider both togther...

In the case of maximum likelihood, we are able to form an objective function by relying on the assumption of normality of the stochastic error term. As is likely laid out in your favourite econometrics text, the log-likelihood looks like this:

$$\begin{aligned} & L(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y} \mid \boldsymbol{x}) \\ & \quad = -\left(\frac{N}{2}\right) \ln 2\pi - \left(\frac{N}{2}\right) \ln \sigma^2 - \left(\frac{1}{2\sigma^2}\right) (\boldsymbol{y} - \boldsymbol{\beta}\boldsymbol{x})'(\boldsymbol{y} - \boldsymbol{\beta}\boldsymbol{x}) \end{aligned} \tag{2.1}$$

This is all well and good, but now, how do we actually maximise something like this in MATLAB? Fortunately given our work in Chapter 1, MATLAB's optimisation routines can be largely based around functions. Essentially, we tell MATLAB to find the highest value of the function we define by varying the values of the input parameters. In the above case this implies finding the highest value for L, by searching over combinations of the parameters $\boldsymbol{\beta}$ and σ^2.

The maximisation process thus begins by converting our objective function into a MATLAB function. It is convenient to optimise over a *single* vector, rather than over two separate objects — so we define $\boldsymbol{\theta} = \left(\boldsymbol{\beta}', \sigma^2\right)'$, and optimise in terms of $\boldsymbol{\theta}$:

```
function[ML] = normalML(theta,y,x)
% NormalML(theta,y,x) calculates the likelihood function given
% a matrix of covariates x and a dependent variable y with an
% unobserved stochastic error term which is distributed accor-
% ding to a normal distribution.
%
% The likelihood function is evaluated at the coefficients
% theta, which must be specified by the user.  These should
% include as many coefficients as are to be estimated in the
% model, and finally an estimate of sigma.
%
% See also mle

%*************************************************************
%*** (1) Form stats
%*************************************************************
```

```
N    =   length(y);
K    =   size(x,2);
sig  =   theta(K+1);
beta =   theta(1:K);
u    =   y - x*beta';

%*************************************************************
%*** (2) Likelihood function
%*************************************************************
ML   =  -(N/2)*log(2*pi)-(N/2)*log(sig^2)-(1/(2*sig^2))*(u'*u);
ML   =  - ML

return
```

The penultimate line of this code looks remarkably similar to (2.1). The remainder of the code just consists of determining a number of other parameters (such as N) based on the data in question. Perhaps the last line is the most confusing. The reason why we reverse the sign of our maximum likelihood value is that all of MATLAB's optimisations are based on *min*imisation. As a result, in cases such as this and many of the others which we'll go through in this book where we are actually interested in maximisation, we will just undertake a simple transformation such as this when calculating our objective function.

Once we've defined our objective function in this way, we can use MATLAB's clever range of solvers to determine the optimal value. For our ML example, we'll focus on the `fmincon` function. This function finds the minimum value, while allowing for both equality and inequality constraints, as well as the option of defining upper and lower bounds for parameters. If we were to set this out on paper it would look something like the following:

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{such that} = \begin{cases} c(\mathbf{x}) \leq b \\ ceq(\mathbf{x}) = beq \\ lb \leq \mathbf{x} \leq ub, \end{cases} \tag{2.2}$$

where further details regarding each constraint can be found in the MATLAB help file for `fmincon`. However, this may all be rather abstract, so let's have

a look in practice. To do this, we'll return to our parameter estimates from section 1.2. There we had defined a matrix $\boldsymbol{X}$ and a vector y which had come from the trusty auto dataset. We'll ask you to re-enter or recall these matrices (perhaps they are still in MATLAB's working memory, in which case you need do nothing), and we'll use this data to see whether our likelihood function allows us to recover the regression estimates we calculated in the previous chapter.

In the case of this problem, there is no clear upper and lower bound that we necessarily want to impose on our parameters, but we'll define some values to limit the domain over which MATLAB searches.[2] Here we are going to be estimating four parameters which correspond to the three $\hat{\beta}$'s, along with $\hat{\sigma}^2$. We'll set up the following lower and upper bounds for each of these parameters:

```
>> lb  = [-1000, -1000, -1000, 0];
>> ub  = [1000, 1000, 1000, 100];
```

We'll also define an initial range of values from which MATLAB should begin its search. Although we have quite a good idea of what these parameters should look like from our regression in chapter 1, we'll pretend this isn't the case, and be reasonably agnostic with our initial choice:

```
>> theta0  = [0, 0, 0, 1];
```

Finally, we want to define a number of optimisation options. While MATLAB makes a range of assumptions when optimising (such as the optimisation method to use, the number of iterations, and so forth), we are able to control all these ourselves. We define a few of these below, although the full list of options can be seen by typing `optimset` in your MATLAB window.

```
>> opt = optimset(`TolFun',1E-20,`TolX',1E-20,`MaxFunEvals',1000);
```

[2] MATLAB also has a function for unconstrained optimisation: `fminunc`. But our problem *is* constrained, because we have $\sigma^2 > 0$. Further, even if our problem were *not* constrained by economic or econometric theory, it may still be useful to use `fmincon`, and to impose some bound constraints on 'reasonable' values of the parameter space.

So, finally, we have all our ingredients; we've defined an objective function, the constraints, the starting point, and the optimisation options. With all of this, it's now just a matter of letting MATLAB get to work! We issue the `fmincon` command below:

```
>> fmincon(@(theta)normalML(theta,y,X), theta0, [], ...
                            [], [], [], lb, ub, [], opt);
```

We agree that this looks a little bit complicated. It is, however, about as involved as a typical MATLAB command is ever going to get. You'll note that at its heart it is just a call to `fmincon`, and we've included all the parts we've just defined above (the starting point `theta0`, the upper and lower bounds, and the optimisation options `opt`). The tricky part is in deciphering the first argument which starts with the @ symbol. In MATLAB parlance this is known as a "function handle", however this is essentially little more than the name of our function, and an indicator of which parts of the function we are maximising over. The `normalML(theta,y,X)` part of this argument just says that we use the function `normalML` and the data y and X which we defined above,[3] while the `@(theta)` part says that we are maximising over `theta`. The remainder of the arguments are simply made up of empty braces. While we won't concentrate too much on this for now, this should offer us some hint about how to set up problems where explicit equality and inequality constraints are required.

We encourage you to play around with this code until you feel comfortable with the various moving parts. For example, try varying the optimisation settings, the lower and upper bound, and the starting point. Depending upon the settings you use, the output will look something like the following:

```
Local minimum possible. Constraints satisfied.

fmincon stopped because the predicted change in the objective function
is less than the selected value of the function tolerance and constraints
are satisfied to within the default value of the constraint tolerance.
```

[3] If you've skipped over chapter 1 before reading this, you'll need to refer to section 1.2 in order to generate X and y.

```
No active inequalities.

ans =

   -0.0001   -0.0058   39.4396    3.3842
```

Importantly, we see that with these settings our ML estimator does a good job in recovering the correct estimate for $\boldsymbol{\beta}$ that we estimated earlier.

The `optimset` options provide a large range of options which are worth being familiar with when solving for minimums in this way. If, for example, we are interested in producing a graph of convergence of the `'PlotFcns','optimplotfval'`, while if we are interested in seeing a larger range of output including the procedure and the value of the objective function, we could specify `'Display','iter'` as part of our `options` command. In figure 2.1 we see the output from the `PlotFcns` option. As we discussed earlier, we have taken the absolute value of the likelihood function so it appears to be converging on a positive value from above, although in reality it will be converging on a negative value from below. Of course, as we see in our estimates for $\boldsymbol{\beta}$, this has no effect on the values produced.

2.2 Solving models with optimisation

Maximum likelihood, along with many of the other estimation techniques based around optimisation that we'll see in this book, are perfect candidates for MATLAB's `fmincon` routine. However, as we discussed in the opening of this chapter, there is an entirely different reason why we as microeconomists might be interested in maximisation. Constrained maximisation is at the heart of many, or even most, of our typical microeconomic models.

Let's start by considering a standard optimisation problem in consumer theory: that of a consumer determining his or her optimal consumption of two goods to maximise a Cobb-Douglas utility function. We are by now quite familiar with this utility function in MATLAB. We return to this example — admittedly simple

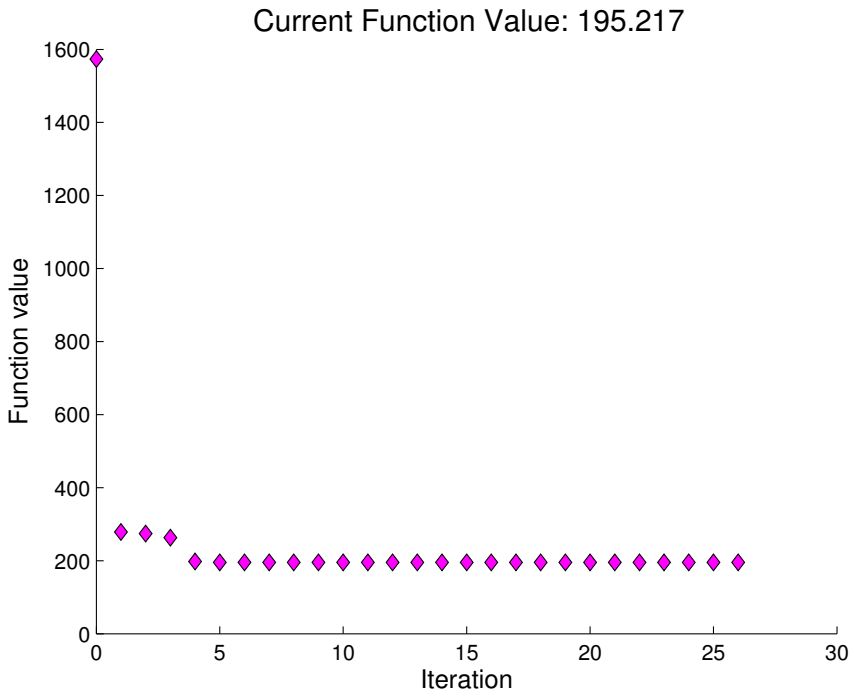


Figure 2.1: Convergence of (the absolute value of) the likelihood function

— to further demonstrate MATLAB's optimisation routines, and incorporate some features and checks we haven't yet seen in the previous section.

As in the previous chapter, we'll asume here that the consumer's utility function is $U = x_1^{1/2} x_2^{1/2}$, so that her maximisation problem is:

$$\max_{x_1, x_2} x_1^{1/2} x_2^{1/2} \qquad \text{subject to} \qquad I = p_1 x_1 + p_2 x_2. \tag{2.3}$$

Fortunately, we've already defined this function in our `utility.m` code. Graphically, we are interested in finding how the consumer can optimally maximise the utility presented in figure 1.1(b). This utility comes entirely from the quantities of each good consumed (displayed on the x- and y-axes), and is calculated on the z-axis. Of course, were the consumer not subjected to some binding budget constraint she would consume the maximum amount of both goods and arrive

at the maximum utility of 3 units (which corresponds to the red portion of the figure).

Below we set up a very slight variation of `utility.m` function from chapter 1. This brief function will, like its analogue from the previous chapter, return the value `u` for utility, and takes as arguments the quantity of the two goods consumed. Note that here we define our input argument `X` as a single vector rather than the slightly more verbose option:

`function u = utility(x1,x2).`

There is a functional reason for defining `X` in this manner. When our function accepts only one argument, we can avoid using the quite ungainly function handle that we used in our call to `fmincon` in section 2.1. Specifying the arguments this way rather than in the more extensive `x1,x2` form is, in reality, only a matter of style, as later in the function we can simply 'unpack' our vector into individual elements, as we see in the first two command lines. Also, as we discussed in our ML example above, in the final line we return the negative of our typical notion of utility, given that MATLAB will be minimising this function.

```
function u = cobbdouglas(X)
% Function to calculate utility given a Cobb-Douglas specif-
% ication and varying the quantites of goods consumed (vector
% X requires two goods).

x1  =  X(1);
x2  =  X(2);
u   =  -(x1^0.5)*(x2^0.5);

return
```

With the function that we will ask MATLAB to optimise now in hand, we can jump right into the business of actually optimising. Once again we will rely on the `fmincon` command given that in this case we are dealing with explicit constraints.

As before, We can create a series of matrices (or vectors) that indicate to `fmincon` what our initial guess and constraints are. Let's imagine for example that our consumer has a total income of $100, faces goods prices of $4 and $7, and our initial guess is that she would consume 15 and 5 units respectively.[4]

```
>> I  = 100;
>> P  = [4,7];
>> G  = [15,5];
>> lb = [0,0];
>> ub = [25,100/7];
```

Here we have introduced the right side of our budget constraint (the prices) as `P`, the left side as `I`, our initial guess as `G`, and an upper and lower bound for the consumption of each good (`ub` and `lb` respectively). Now, let's see how this all comes together with the `fmincon` command:

```
>> [consumption, u, exitflag] = fmincon(@cobbdouglas,G,P,I,[],...
                                         [],lb,ub);

consumption =

   12.5024   7.1415


u =

  -9.4491


 exitflag =

    5
```

This call to `fmincon` looks pretty similar to the version in section 2.1. However,

[4] It is not particularly important that the starting point respects the income constraint, as MATLAB will deal with that once optimisation begins.

you will notice a few differences. Perhaps most obviously, the first argument is now signficantly simpler. We were able to avoid the entire business of the function handle given that the objective function `cobbdouglas` only depends upon one argument (`X`), so MATLAB knows that it must optimise with respect to `X`. Another difference is that here we ask for various objects to be returned: `consumption`, `u` and `exitflag`. MATLAB understands this to mean that it should return: (i) the optimal input values for `X`, (ii) the value of the objective function at this optimum, and (iii) a code explaining how our optimisation method arrived at the final answer. When we estimated regression coefficients using ML we did not explicitly request the second and third option — so, as is always the case with MATLAB functions, we were only returned the optimal input values.

The value for the objective function (`u`) needs relatively little explanation, however we will want to consider the exit flag quite carefully (both in this explanation, and more generally whenever we optimise in MATLAB). There are a wide range of reasons why MATLAB can decide that it has 'found' the minimum value, and all of these are controlled using the `optimset` options that we've discussed previously. As is often the case, MATLAB's help file offers a good starting point. The file for `fmincon` lists all possible exitflags, and the reason why the optimisation routine terminates in each case. We will see for example that an exitflag of 5 is returned when the derivative at the final point was arbitrarily small. Of course, we can never be entirely sure that this corresponds to a local minimum. Perhaps if we were to set finer search criteria (using `optimset`), MATLAB would actually be able to find a smaller value for the objective function. Similarly, it may help MATLAB if we were to provide an analytical expression for the Jacobian.[5] In general, it is also a good idea to restart your optimisation from the final point, and see whether any improvements can be made. Let's try that here by plugging our `consumption` result back into `fmincon`:

```
>> [consumption,u,exitflag] = fmincon(@cobbdouglas,...
                                      consumption,P,I,[],[],lb,ub)

 consumption =

    12.5000    7.1429
```

[5] This often requires great cost — both in terms of time and emotion — to the programmer.

```
u =

  -9.4491


exitflag =

     1
```

We see a small change in our consumption values, and that now we are returned an exitflag of 1. Again referring to the help file, this implies that the constraint violation and tolerance of the objective function are less than the (very small) values which we have specified in the optimisation options. This is a reasonably satisfying result, and subsequent tests from this starting point suggest that no further improvements are found. Usually, '`exitflag = 2`' is a reasonable sign of success — though, as the previous discussion indicates, deciding on convergence criteria is often more a matter of art than science.

The rest of the command is issued exactly as we saw in the previous section. This involves passing `fmincon` the constraints and inital guess as parameters: first the command name, second our initial guess, then the left-hand side of the inequality constraint[6], followed by the right-hand side of the inequality constraint, and finally the upper and lower bounds we have defined for consumption of each good. Once again, we have included empty braces `[]` in our code. This tells MATLAB that there is no strict equality constraint in this case. As always, we refer you to the help files (or the command line!) to clear up any lingering doubts.

This code likely seems quite simple, and perhaps you are left wondering why we would even bother to do something like this in MATLAB. This is admittedly a very simple function — but simple functions are useful building blocks for more complicated analysis. Let's imagine for example that we wanted to nest

[6] Note that here we treat the income constraint as non-binding, although generally it always will. An individual *could* choose to spend less than all their income, although the form of the Cobb-Douglas utility function implies that maximisation of utility is achieved by spending all income on consumption.

utility maximisation inside a more extensive model. This would then involve defining a (potentially much) more complex function, which calls our simple Cobb-Douglas maximisation technique. Below we provide an example of such a function which is built around our previously defined function:

```
function [u,c] = budget(P,I,X)
%Budget presents the budget constraint faced by an individual,
%and calculates their optimal consumption based upon a Cobb-
%Douglas utility function defined in the function utility.m
%
%The syntax of the function is budget(P,I,X) and the function
%accepts as arguments a 2*N vector of prices, P, the total inc-
%ome, I, and the initial guess used in the fmincon command, X.
%
%see also fmincon

%************************************************************
%*** (1) Unpack relevant parameters from function call
%************************************************************
p1  =   P(1);
p2  =   P(2);
lb  =   [0,0];
ub  =   [I/p1,I/p2];

%************************************************************
%*** (2) Determine utility maximising consumption
%************************************************************
[c, u] =   fmincon(@cobbdouglas,X,P,I,[],[],lb,ub);

%************************************************************
%*** (3) Create graphical output
%************************************************************
x2  =   0:I/p2;                     % all possible values of x2
x1  =   (I-p2.*x2)./p1;       % values of x1 corresponding to x2
fig =   plot(x1,x2,'LineWidth',2);
hold on
```

```
x1  =   u.^2./x2;                  %calculate utility everywhere
axis([0,I/p1,0,I/p2]);                                  %set axes
plot(x1,x2, 'color','r','LineWidth',2)

xlabel('quantity of good x_1', 'FontSize', 14);
ylabel('quantity of good x_2', 'FontSize', 14);
title('Cobb-Douglas Utility Maximisation', 'FontSize',16);

return
```

Here you will see that we are dealing with a function "`budget`" which lets us calculate all possible points on a budget constraint for any set of incomes and prices $\{I, x_1, x_2\}$, and also plots the utility function of the consumer which corresponds to utility maximisation. Whilst we haven't introduced a great many new commands in this example, there are a number of complex elements which are worth examining (and changing when you run this in your MATLAB session).

Finally, let's try out this command and see what we get.

```
>> P=[100,200];
>> I=10000;
>> X=[40,20];
>> budget(P,I,X);
```

We see that as well as our output from the utility optimisation we get a plot of the consumer's budget constraint and the appropriate utility function (figure 2.2).

2.3 Simulating model solutions

At its core, microeconometrics is about heterogeneity. If everyone faced the same set of choices, with the same constraints, and had the same preferences,

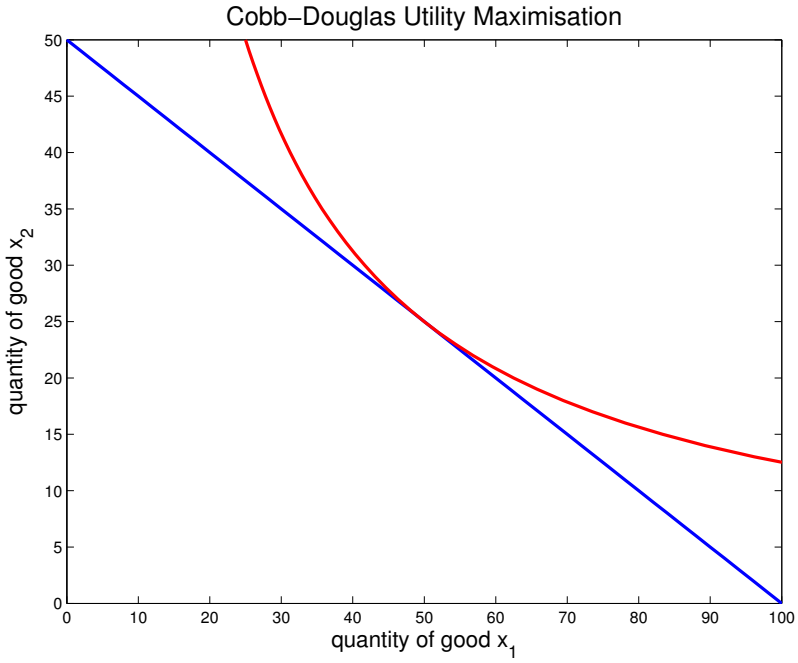


Figure 2.2: Single utility curve and budget constraint

then we would all behave in the same way — and there would be little advantage to collecting any dataset greater than $N = 1$. In many standard econometric models, we allow heterogeneity to enter through an additively separable 'error' term. Of course, there is nothing inherently wrong with this approach — but we may want to explore the consequences of introducing random variation in other parts of a model. We end this chapter with a simple illustration of how we might do this in MATLAB.

MATLAB offers us a wide range of tools to consider drawing random numbers from the specific distributions which are likely to underlie our Monte Carlo Simulations. Table 2.1 lists a number of these distributions, and their associated MATLAB commands:

It is sometimes argued — particularly in development economics — that different economic actors face different prices, based on their individual characteristics. (For example, information asymmetries may lead different firms to face

Table 2.1: Random Number Generators

Distribution	Command
uniform	`rand()`
normal	`randn()`
lognormal	`lognrnd()`
multivariate normal distribution	`mvnrnd()`
many others...	`random(distbn,)`

different factor costs.) Suppose, then, that we maintain our earlier model of a consumer with Cobb-Douglas utility. However, suppose that we now assume that, across the population, consumers face uniform variation in the price of good 1:

$$p_1 \sim U(100, 150). \tag{2.4}$$

Suppose that we are interested in characterising the resulting distribution of consumption bundles. Given the tools that we have discussed, this is easy. The following script shows a simple illustration of how this can be done. We leave it to you to step through the script to understand how it works; in the exercises that follow, we suggest two extensions.

```
%************************************************************
%*** (1) Setup, simulation of random variation
%************************************************************
clear
rng(1)
reps        = 100;

pshock      = [rand(reps, 1) * 50, zeros(reps, 1)];

I           = 10000;
P           = [50, 200];
x0          = [1, 1];
lb          = [0, 0];

c           = NaN(reps, 2);
```

```
opts        = optimset('algorithm', 'sqp', 'display', 'off');

%*************************************************************
%*** (2) Determine optimal consumption in each case
%*************************************************************
for count = 1:reps

    TempP       = P + pshock(count, :);

    ub          = I./TempP;
    c(count, :) =   fmincon(@cobbdouglas,[1, 1], TempP, ...
                        I,[],[],lb,ub, [], opts);

end

%*************************************************************
%*** (3) Visualise results
%*************************************************************
subplot(1,2,1)
scatter(c(:, 1), c(:, 2))
axis([min(c(:,1))-5, max(c(:,1))+5, ...
      min(c(:,2))-5, max(c(:,2))+5]);
xlabel('Good 1 Consumption')
ylabel('Good 2 Consumption')

subplot(1,2,1)
cdfplot(c(:, 1))
xlabel('Good 1 Consumption')
ylabel('F(p_1)')

return
```

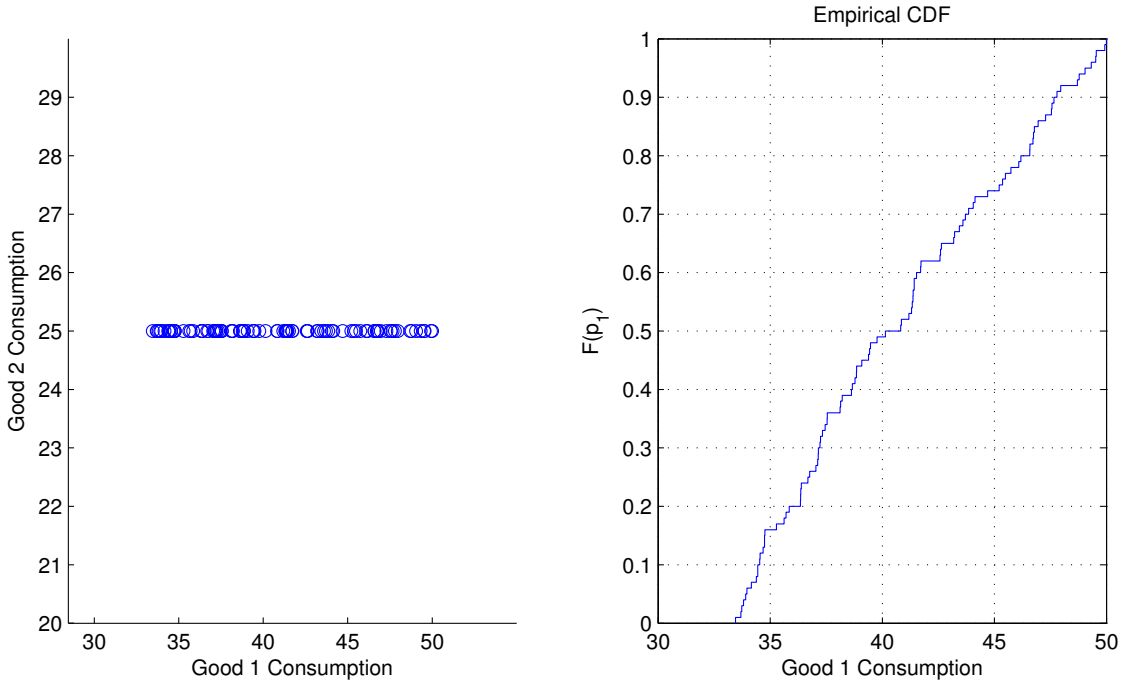


Figure 2.3: Incorporating heterogeneity in Cobb-Douglas utility maximisation

2.4 Review and Exercises

COMMAND	BRIEF DESCRIPTION
`length`	Determines the number of rows in a matrix
`pi`	The mathematical constant π
`fminunc`	Routine to minimise an objective function with no constraints
`fmincon`	Routine to minimise an objective function subject to linear or nonlinear constraints
`optimset`	Provides control over the optimisation process in MATLAB
`plot`	Draw a two-dimensional graph
`hold on`	Keep current plot in graph window, and add another plot to the output
`axis`	Set minimum and maximum for graph axes
`xlabel`	Label x-axis in the plot window (allows for LaTeX style parsing)
`title`	Add a title to the plot window
`rand`	Allows for a (pseudo-)random draw from a uniform(0,1) distribution
`randn`	Allows for a (pseudo-)random draw from a normal distribution
`rng`	Sets MATLAB's psuedo random number generator at a replicable point
`NaN`	Pre-fills a matrix with 'empty' values
`for`	Repeats a command or series of commands a specified number of times
`subplot`	Allows for multiple graphs on the same plot window
`scatter`	Bivariate scatter plot
`cdfplot`	Draws the empirical CDF of a vector

Table 2.2: Chapter 2 commands

Exercises

(i) Write an alternative maximum likelihood estimator. Rather than an OLS estimator, try generating MATLAB code to estimate a probit model. Remember, that in this case the log likelihood function looks like: $L(\boldsymbol{\beta}; \mathbf{y}|\mathbf{x}) = \sum_{i=1}^{N} \{y_i \cdot \ln \Phi(\boldsymbol{\beta x_i}) + (1 - y_i) \cdot \ln[1 - \Phi(\boldsymbol{\beta x_i})]\}$. Benchmark this code using the auto data set as before. Estimate `probit foreign length weight` in Stata. Ensure that your code replicates these results in MATLAB.

(ii) Using our Cobb-Douglas utility function, simulate variation in income. Use simulated results to plot Engel curves for x_1 and x_2.

(iii) Using our Cobb-Douglas example, suppose that consumers with higher income *also* tend to face lower costs for good x_1. Show how a simulation method could be used to think about consumption bundles in this case. (*Hint*: `mvnrnd` may be useful...)