Prof. Dr. Johannes Pfeifer
University of Mannheim

Fall Term 2015
M.Sc. Program

ADVANCED MACROECONOMICS

-Matlab Handout-

This is a short, self-contained introduction into Matlab that touches upon most issues you need to know in order to get started and be able to solve a fair amount of economic problems. Advanced topics are indicated with an asterisk in the heading.

1 What is Matlab

Matlab is a commercial software package, developed by Mathworks. More specifically, it is a **programming language** with a graphical user interface and extra tools. It includes many technical, mathematical, and statistical commands, such that it is mainly used in engineering, economics, and similar sciences.

It is a **function-oriented** language, i.e. you can write your own commands, in this case called functions. You can write such a function, which can accept input arguments and deliver output, save it as a file, and then use it as a command in your main program file. In recent years, Mathworks has started to incorporate elements of **object-oriented** programming. You will encounter primitive examples of object-oriented programming in the form of structures, but mostly you will deal with the **function-oriented** part of the language.

This property allows any programmer to write own functions and distribute them. Examples are `hpfilter.m` by Lawrence Christiano and `solab.m` by Paul Klein. Matlab programs have the extension `.m`. Also Mathworks writes large sets of functions, so called "toolboxes", for certain areas (such as finance and aerospace dynamics) and sells them. For many programs in economics, you will only need the Optimization and Statistics Toolboxes. There are also some free toolboxes on the internet. The most important ones for economics are the CompEcon-Toolbox (`http://www4.ncsu.edu/~pfackler/compecon/`) and LeSage's Econometrics Toolbox (`http://www.spatial-econometrics.com/`), which is not to be confused with Matlab's more recent proprietary Econometrics Toolbox.

Most important is the `help` function. Set the cursor to a written command and press F1 or type `help command` to know how a certain command is implemented. Start with `help help`. If you look for something, but don't know the command name, use `search keyword`, or better the search option in the help window (F1).

2 How to Start

When opening Matlab in the default specification (everything can be changed according to your taste), you have the command window, the workspace, the current directory, and the command history in one big window. Figure 1 shows the default setup.

- **Command Window**: Here you can type commands 'on the spot', i.e. they get executed, but not saved in a file. Furthermore, possible output from programs gets presented here.
- **Workspace**: An overview over all variables in the memory. Double clicking on a variable inside the workspace opens the variable editor (which looks similar to Excel) where you can see (and sometimes change) the variable's value.
- **Command History**: A list of recently used commands in the command window. You can double-click on them to execute them again.
- **Current Directory**: The directory where you currently are. Note that you need to be in the directory of a certain file in order to execute it (if it is not in a directory listed under the 'Set Path' option under 'Home').

Furthermore, the editor opens upon startup. This is a text editor to write your own Matlab programs. It is designed for programming Matlab, certain commands get different colors, and loops get indented automatically. Each command goes into a single line. If you want to spread command over several lines, you have to tell Matlab by putting a `'...'` at the end of the respective line. You can also add comments to your code. Comments are set by putting a percent sign `'%'` in front. They are not executed when your code is run and appear in green font.

There are two ways to run your code. You can run your whole program in the editor by pressing F5 or by going to the Tab "Editor" and clicking on the green arrow button that says "Run". This requires that the program is saved to the harddisk. All unsaved changes will automatically be saved.

If you do not want to execute your whole code, but only parts, you can select the code you want to run and press F9.

Sometimes, you want to terminate the execution of code, e.g. when you did a programming error and Matlab entered an infinite loop. In this case, click into the command window and press *Ctrl+C* to stop the execution of the current listing.

If you do not assign your computations to a pre-specified variable, Matlab automatically stores the *last* computation result in the variable `ans` (short for answer) in the workspace.

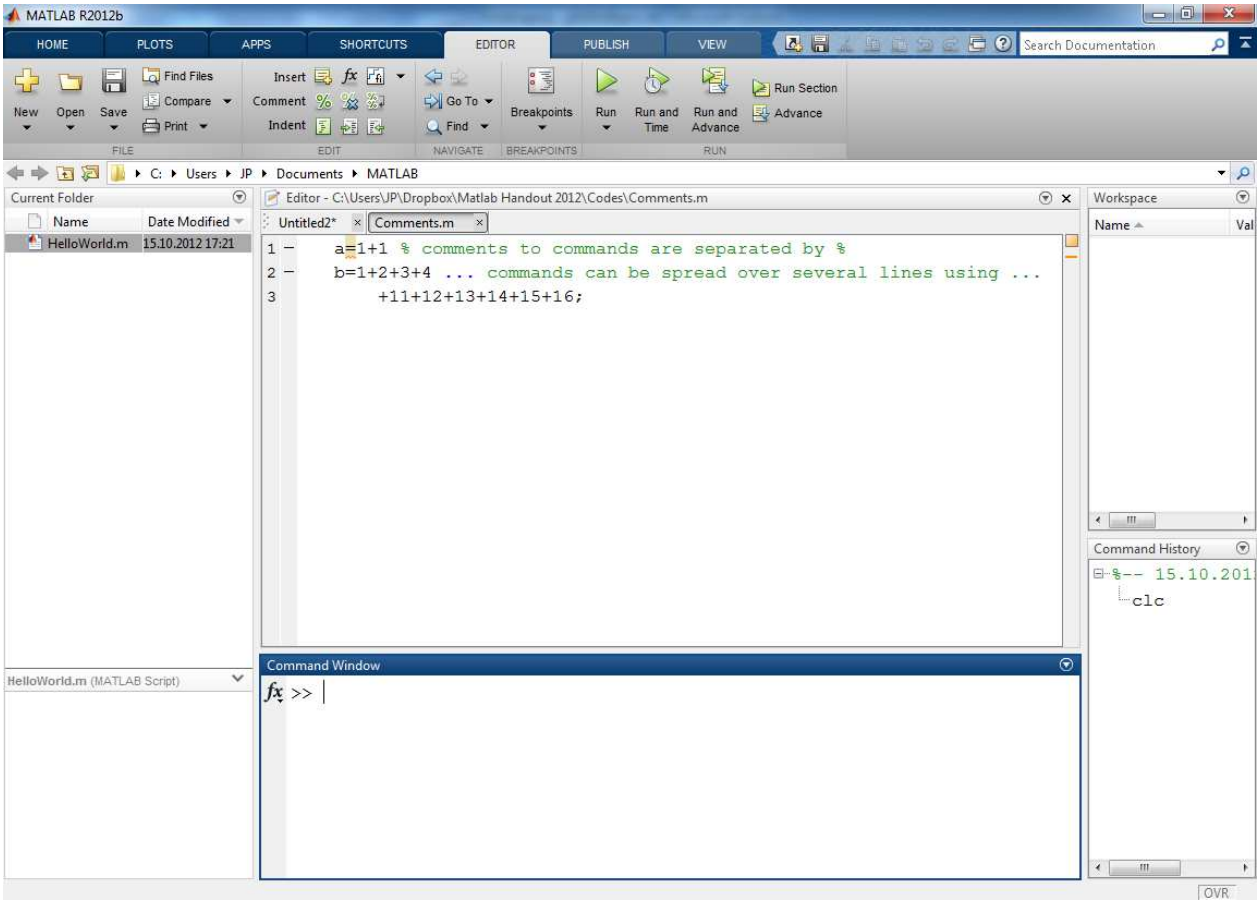

Figure 1: Matlab's user interface

3 Setting Paths

Generally, if you want to execute a function or access a file in Matlab, you must make it known to Matlab that this command or file exists. There are two ways to do this. First, the file or function is located in the Current Directory (shown immediately below the menu bar). In Figure 2, you can see that the Current Path is set to a different folder (`C:\Users\JP\Documents\MATLAB`) than the file `example1.mod` (`C:\dynare\4.4.3\examples`). Thus, Matlab will not know this file. You can easily change the path to the directory of this file by right-clicking on the file-name and selecting the "Change current folder to..." as shown in the figure. The second possibility is to add the folder with the desired functions/files to the Matlab Search Path. This can either be done by right-clicking on the file and selecting the "Add to Search Path" option or by going to 'Home' → 'Set Path' and selecting the folder(s) you want to add. Note that adding to the Search Path via the right-click option only adds to the Path for your current Matlab-session. In contrast, the 'Set Path' menu option is more flexible and allows for storing the path for future Matlab-sessions as well as automatically adding all subfolders of a directory.

In general, you should reserve permanently adding folders to your Search Path for tool-

boxes like the Econometrics toolbox or packages like `Dynare`.[1]

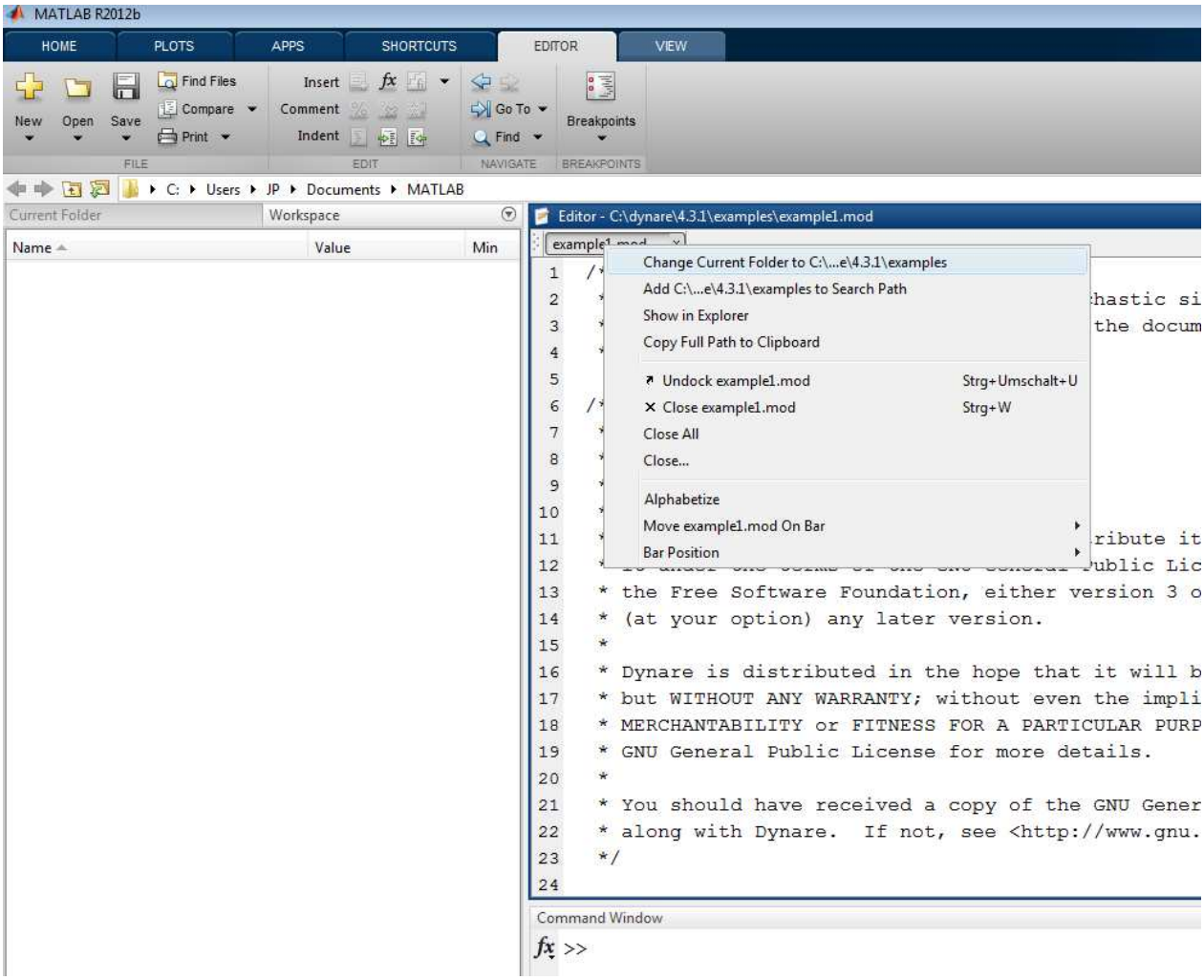

Figure 2: Matlab's user interface

4 Variable Names

Variable names in Matlab must start with a letter and can be followed by letters, digits, and underscores. Special character or spaces are not allowed, as is the case with Matlab keywords like `end` or `if`. Matlab is case sensitive, so `A` and `a` are not the same variable.
When naming variables, try to avoid naming variables like existing functions as this might lead to conflicts. In particular, avoid Greek letters as almost every single one is also the name of a built-in Matlab function. Instead of `alpha` use `alppha`. Use `betta` instead of `beta`. Moreover, do not use `i` as an index variable as it indicates an imaginary number.

[1]Dynare is a package for solving and simulating deterministic and stochastic macroeconomic models (`www.dynare.org`). After installing the program, you have to add its \matlab-subfolder to your path for the program to run in Matlab.

When in doubt, write down a variable name, place the cursor in the name, and press F1. If there is no corresponding help entry, it is safe to use the variable name.
If you get strange error messages when accessing a variable or calling a function `some_name`, try using `which some_name` to find out which function or variable Matlab has been calling. This also helps to detect problems with conflicting paths.

5 Decimal Separator, Scientific Notation, Inf, and NaN

Matlab uses the point as its decimal separator, while the comma is used to separate columns within a matrix or different function arguments. It also supports scientific notation. `1e5` is equal to 1×10^5.
There are two special values you will sometimes encounter in Matlab. `NaN`, short for 'Not a Number', represents undefined or unrepresentable computation results like 0/0. Usually, if it pops in your computation, there is something wrong. `Inf` and `-Inf` represent plus and minus Infinity. `Inf` results e.g. from division by 0 like 1/0.

6 Matrices and Operators

Definition of matrices
Matlab's internal organization is based on matrices. This makes working with matrices especially easy. For example, you define a matrix `A` as `A=[1 2; 3 4]`, where a blank or a comma separates columns and the semicolon rows. Similarly define `B=[2 3; 4 5];` (the semicolon at the end of the line suppresses the output in the command window).

Matrix operations
It is easy to (matrix-) multiply the matrix `A` with `B` by typing `A*B`. If you want to multiply each element in `A` with the corresponding element in `B`, use the dot operator: `A.*B`.

Matrix left and right division
The same is valid for divisions. Matrix right divide `A/B` is roughly the same as `A*inv(B)`. It is only roughly the same, because the answer is computed with a different algorithm than when computing the inverse explicitly. The corresponding element-wise operation `A./B` divides every element of `A` by the corresponding element of `B`.

More important than matrix right division is matrix left division. Left divide `A\B` is roughly the same as `inv(A)*B`). If `X` is an m by n matrix and `y` is a column vector with m components, then `b = X\y` is the least square solution to the system of equations `y=X*b`. As the solution is computed using the QR-decomposition, `b=X\y` is a numerically more efficient way to **perform a regression** than using the formula `b=(X'*X)^(-1)*X'*y`.
In contrast, the element-wise operation `b=A.\B` that multiplies every element of `B` with the inverse of the corresponding element of `A` is rarely used.

Single out elements of matrices

Single elements can be taken from a matrix according to `Matrix(row,column)`, e.g., `A(1,2)`. Hence, try to work with matrices instead of scalars, it is much faster. Also higher-dimensional arrays can be used: `C(:,:,1)=A; C(:,:,2)=B;`, where the colon stands for 'all rows' or 'all columns', respectively.

The Colon operator

To vectorize a matrix use the colon operator `Matrix(:)`. For example, the colon operator applied to the matrix `A=[1 2; 3 4]`, `D=A(:)`, returns: `D=` $\begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \end{bmatrix}$.

Note that Matlab follows column-major order. Elements of an array are counted along the respective column: the first overall element of `A` is `A(1,1)`, the second overall element is `A(2,1)`, the third overall element is `A(1,2)` and so on. Vectorization thus simply arranges all array elements consecutively in one column vector.

The keyword `end`

Matlab uses the keyword `end` for two purposes. The first use is to address the last element of a matrix or vector. This very useful as you do not need to explicitly keep track of the size of the array. For example, `A(1,end)` would deliver the matrix element in the last column of the first row. You can also perform integer arithmetic with `end` when addressing matrix elements. Think of `end` as a placeholder for the number elements along this matrix dimension. For example, `A(end,end-1)` delivers the element in the last row and the second-to-last column.

The second use it to signal the end of a control statement. It will be covered in Section 9.

Transpose a matrix or vector

Use `'` to transpose matrix. Given `D`, the command `D'` returns: $\begin{bmatrix} 1 & 3 & 2 & 4 \end{bmatrix}$.

Sum the elements of a matrix

Using the command `sum` you can sum the elements of a vector or the elements of matrix, either along the columns or the rows. Examples: `sum(D)` returns 10. The command `sum(A,1)` computes the sum of each column, i.e., returns $\begin{bmatrix} 4 & 6 \end{bmatrix}$. The command `sum(A,2)` yields $\begin{bmatrix} 3 \\ 7 \end{bmatrix}$.

Determine the size of a matrix

To determine the size of matrix or a vector use `size()`. The command `size(A)` returns [2 2], indicating that the matrix A has the size of 2 columns and 2 rows. Similar to the command `sum`, the command `size` can be used along a specific dimension. For example, `size(A,1)` returns the numbers of rows, `size(A,2)` returns the number of columns. If you are working with one-dimensional arrays you can also use `length(A)`. But be careful. If your array has more than one dimension, `length()` will automatically return the size along the largest dimension, which sometimes is not the dimension you thought it was.

Define special matrices

The commands `zeros`, `eye`, `ones` define special matrices. `G=zeros(4,2)` defines the matrix `G` that has zeros as entries and is of size [4, 2]. A similar command is `ones`. The command `H=eye(4)` defines the matrix `H` as the 4 by 4 identity matrix.

Single out the diagonal of a matrix

The command `diag` returns the diagonal of a matrix. For example, `diag(A)` returns [1 4].

Concatenation

You can easily combine matrices to a new matrix: `K=[ A B]` defines the new matrix `K=` $\begin{bmatrix} 1 & 2 & 2 & 3 \\ 3 & 4 & 4 & 5 \end{bmatrix}$

Modulo

The modulo operator `mod` gives the remainder of a division. `mod(7,4)` returns 3 as 7/4=1 with remainder 3.

Absolute value

The absolute value of a number or array can be taken using `abs(x)`.

7 Cleaning Up

The command `clear all` clears all variables from the memory (including globals, see Section 12), while `clear` only deletes variables within the current workspace/function. `close all` closes all open figures. `clc` is used to clear the command window.

Often programs you will write should start with a `clear all`, because you want to avoid having holdovers from previous programs floating around in the memory. However, avoid using `clear all` commands within functions as they will also delete breakpoints (see the section on debugging).

Note that this behavior has changed since Matlab 2015b. Here, `clear all` will not delete breakpoints, but it will delete all variables and precompiled function from the workspace. This means that using `clear all` will interfere with the newly introduced *Just In Time (JIT) Compiler* and will increase execution time. Thus, you should avoid putting `clear all` within your scripts in larger projects. If you do not use global variables, a simple `clear` should suffice.

8 Variable Types

Matlab distinguishes several different data types that are created by using different variable assignments. In contrast to compiled languages the variable type is not explicitly specified with a separate token. Different variable types have different symbols in the workspace as shown in Figure 3.

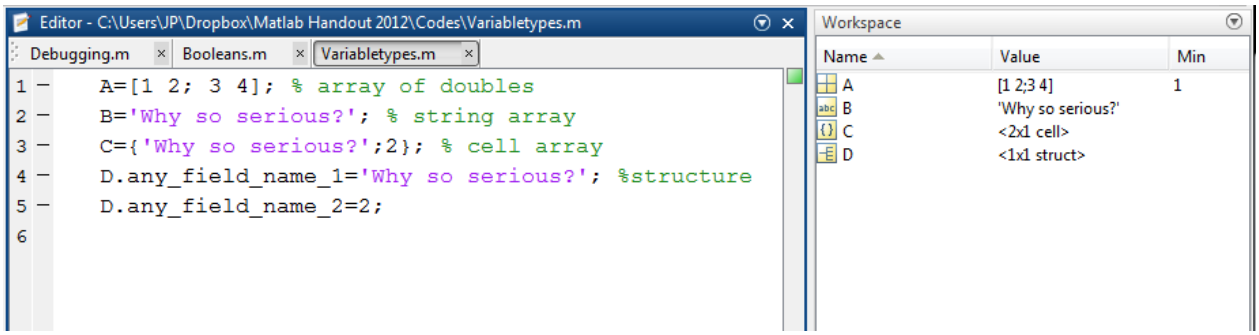

Figure 3: Variable Types

Doubles

Numbers are usually stored in so called `doubles`. An assignment of the type `A=[1 2; 3 4]` creates a new array `A` of the type `double` in the workspace that stores the numbers. `Double` is short for Double-Precision Floating Point and is the numeric data type typically used in Matlab.

String Arrays

The second important variable type are `string arrays`, which are used to store characters instead of numbers. They are created by using quotes in the assignment:
`B='Why so serious?'`.

Apart from the basic variable types `double` and `string arrays`, there are two additional types called `cell arrays` and `structures`. Both `cell arrays` and `structures` function like a cabinet with different drawers. Each drawer can hold a different data type, i.e. you can use them to organize your workspace.

Cell Arrays

`Cell arrays` are formed by using curly brackets in the assignments. For example,
`C={'Why so serious?'; 2}`
creates a 2 by 1 `cell array` where the first element, `C{1,1}`, is a string and the second element `C{2,1}` is a double.
Note that there are two ways of addressing contents of a `cell array`. Using curly brackets, as mentioned before, returns the content of the respective cell in the form of its native data type. In contrast, using round brackets returns a subset of the cell array: `C(2,1)` results in a 1 by 1 `cell array` containing a `double`.

Structures

`Structures` are multidimensional Matlab arrays with elements that are accessed by textual field designators instead of array indices. Continuing the example from before, you can create a `structure` D containing the `string array` 'Why so serious?'as the first field and the number 2 as the second field by typing
`D.any_field_name_1='Why so serious?'` ;
`D.any_field_name_2=2`

The dot is used to separate the respective field names from the name of the underlying structure that is accessed. You are most likely to encounter structures when using one of the Econometrics Toolboxes.

Boolean

`Booleans` are logical data types that can store only two values: `true` (1) or `false` (0). They appear naturally in testing for logical conditions. For example, after initializing the variables `a` and `b` in the first lines of Figure 4, the second line tests if `a` is equal to `b`. Note the difference between assignments, using the `'='`, and tests for equality with `'=='`. The answer to this logical test is obviously `false`. Hence, the answer is stored in the workspace in `ans` as of type `Boolean`, indicated by the check mark. For the purpose of logical statements, numbers other than 0 evaluate to `true`. Logical statements can be combined using the logical operators *and* (&) and *or* (|). In scalar statements, typically the short-circuit operators && and || are used. Negation is implemented using the tilde '~'.

Commonly used logical operators are
`any(A,2)` and `all(A,2)`,
where the second input argument gives the dimension along which to check the condition (here: along the columns, i.e. 2). `any` checks whether the array `A` contains at least one element that is non-zero along the specified dimension, while `all` checks whether all elements are different from 0. By default, both `any` and `all` move along the first dimension, i.e. the rows. `any(A)` results in a row vector indicating whether the any row of the respective column contained a non-zero element. Thus, for checking whole matrices, you can stack the calls, e.g. `any(any(A))`.

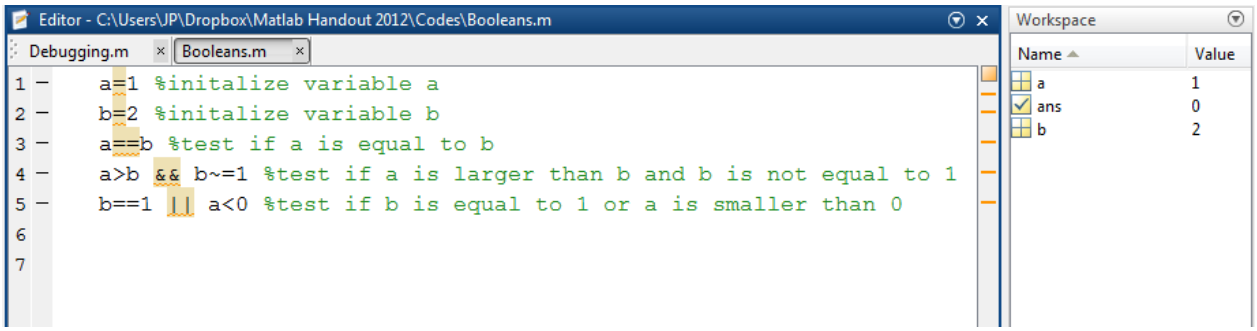

Figure 4: Booleans

9 Control Statements

As most programming languages, Matlab features two general types of control statements that allow you to determine which pieces of your code should be executed: branching statements and loops. All of these statements are closed by using the keyword `end`.

Branching statements

There are two branching statements that are very similar to regular English. The most important one is the `if`-statement, which allows to make code execution dependant on whether certain logical conditions are satisfied. For the logical conditions, you can use every statement that evaluates to a Boolean. Consider for example the following code:

```
mynumber = input('Enter an integer:');

if mod(mynumber,2)==0
   disp('Even');
elseif mod(mynumber,2)==1
   disp('Odd');
else
   error('Number is no integer');
end
```

After entering a number via the keyboard and saving it into `mynumber`, the expression `mod(mynumber,2)==0` tests whether the number is divisible by 2 without remainder. If this is the case and the expression evaluates to `true` (1), the program displays that the number is even. If `mod(mynumber,2)==0` evaluates to `false` (0), the `elseif` part checks whether `mod(mynumber,2)==1`, i.e whether the entered number is an odd number. Depending on the result, either `Odd` is displayed or the `else`-part is executed and the program displays an error-message.
The `if-end`-statement can be used as a standalone statement or can be expanded by adding additional `elseif` and `else`-clauses as in the example above. The `else`-part is a catch-all for all cases not considered explicitly in the `if` and `elseif`-conditions. It is considered bad programming practice to overly rely on simple `else`-clauses, because they liberate the programmer from explicitly specifying in which cases the statements will reached. This often results in bugs, because the code in the `else`-block is executed in cases the programmer simply forgot to think about.

Less frequently used is the `switch-case-end` statement, an example of which is given here for completeness:

```
switch mynumber
    case -1
       disp('negative one');
    case 0
       disp('zero');
    case 1
       disp('positive one');
    otherwise
       disp('other value');
end
```

Control loops

Control loops allow repeatedly executing the same lines of code without explicitly repeating them every time. Thereby, they allow efficiently making use of recursive program statements. `for-end`-loops execute for a predefined number of times:

```
total_sum=0;

for ii=1:1000
   total_sum=total_sum+ii;
end
```

At the beginning of the statement, `ii` is set to 1 and incremented by 1 in every iteration until in the last iteration `ii` is equal to 1000. The part `1:1000` is short for `1:1:1000`, i.e. the set that contains the numbers from 1 to 1000 in steps of 1. After the `ii=`-part you can specify any set you want. To only add the odd numbers from 1 to 1000, you could have used `for ii=1:2:1000`. You can also count down: `for ii=1:-1:-1000`.

In contrast to `for`-loops where the number of times a given code is repeated is pre-specified, `while`-loops repeat code until a termination criterion is reached. Consider the computation of a factorial:

```
n = 1;
nFactorial = 1;

while nFactorial < 1e10
   n = n + 1;
   nFactorial = nFactorial * n;
end
```

The inner part of the loop is repeated as long as the condition `nFactorial < 1e10` evaluates to `true` (1). As soon as it evaluates to `false` (0), execution is stopped. It is important to initialize the logical condition correctly before the loop as otherwise the loop may not be executed even once.

10 Logical Indexing

Logical indexing is a very powerful but initially often confusing technique that works on arrays. Its power mostly derives from the fact that it allows programmers to avoid using loops, which are usually the slowest part of Matlab codes.
Consider a case where you have a vector `a=randn(1000,1)` with 1000 random numbers and you want to delete all elements that are smaller than 0. One way to do this involves writing a loop that iterates over all 1000 elements, checks whether the respective element is smaller than 0, and deletes it. The faster way of doing so is logical indexing. The statement `a<0` would deliver a 1000 by 1 vector of Boolean into `ans`, with a 1 indicating all elements that

are smaller than 0 and which should be deleted. This Boolean vector can be used to address the rows of the vector `a` that should be deleted in the following way:
`a(a<0,:)=[];`
It means that all rows where `a<0` evaluates to `true` are set to an empty matrix, i.e. are deleted.

11 Functions

Calling Matlab functions
Functions are called by providing them with input arguments. They then return output arguments. Virtually all Matlab functions have a minimum number of function arguments, but may be called with optional arguments. If the optional arguments are omitted, the default is used. Consider for example an m by n array of doubles `X`. If you use `d = max(X)`, the function `max` returns a 1 by n vector `d` with the single largest element of each column. By default the maximum is taken along the rows of `X`.
If the maximum should be taken along the `k`-th dimension, it may be provided as an additional input argument:
`d = max(X,[],k)`
The reason for including the empty matrix [] as the second input argument is that `max(X,Y)` delivers the largest elements contained in X or Y along the default dimension. For example, `max([2;4],[1;5])` yields `[2;5]`.

Similarly, most functions provide at least one output argument. But there may be additional optional output arguments that can be obtained by calling the function with additional output arguments. For example,
`[d indices] = max(X,[],k)`
provides both the largest elements along dimension `k` in `d` and in `indices` the corresponding index number of the largest elements.

When calling a function with more than one output argument, the output always takes the form of an array, i.e. they are enclosed in square brackets. In recent versions of Matlab, output of not required output arguments may be suppressed with a '~' and separating the arguments with a comma:
`[~, indices] = max(X,[],k)`

Writing your own functions

An **example**: you write a file, called `compute_variance.m`, taking as an input a vector of numbers `vector`. It calculates the variance of these number and delivers it as an output.

The first line of the function file has to have the following structure:
`function [output] = name_of_the_function(input)`.

Continuing our example, the function for computing the variance would be

```
 function [outvalue]=compute_variance(vector);
var_mean=mean(vector);
outvalue=1/(length(vector)-1)*sum((vector-var_mean).^2);
```

After saving, you can use
`Variance=compute_variance(vector)`
in your main file, which will assign the value of the variance of `vector` to the variable `Variance`.

It is also important to know that the name of the function should correspond to the name of the m-file. If there is a difference, Matlab uses the file name, not the name declared in the function to call it. Moreover, never use spaces in function names!

12 Local vs. Global Variables

Matlab functions by default use local variables. That means, variables defined inside a function are only known to this function and cannot be accessed from other functions, unless you explicitly hand them over as a function input. In our `compute_variance`-function example from the last section, the variable `var_mean` was defined inside of the function. Even after running `compute_variance`, you cannot access this variable from the main script. It will not be in the workspace as it was local to the function `compute_variance`.
The same applies to `outvalue`, which is also local to the function. However, `outvalue` was the output argument of the function, meaning that after the function ran, its value will have been assigned to `Variance` in the main script.

There is also a concept called `global`-variables. If you put
`global var_name`
at beginning of your main script, where `var_name` is the name of the variable you want to make known to other functions, you can access the same variable in subfunctions by also putting
`global var_name`
in those functions. All changes done to this variable will be visible to all other functions/scripts where the `global`-command has also been set.

WARNING: While this may seem like an easy and simple way to get around the specification of input- and output arguments of a function, we strongly advise you to *never* use global variables, unless you really know what you are doing. It is extremely bad programming style. By not clearly defining the scope of a variable, you will quickly lose track of when which function changed the respective variable. Moreover, you might run into conflicts between local and global variables. You have been warned!

13 Statistical Functions

Matlab incorporates many statistical functions. Some of them require the presence of the Statistics Toolbox.

Descriptive Statistics
Important functions are `mean(A,dim)`, `median(A,dim)`, and `std(A,flag,dim)`, where `A` is an array of doubles, dim denotes the dimension along which the statistic is computed (e.g. 1 for along the rows), and `flag` is a Boolean being 0 if N-1 should be used in the divisor to generate an unbiased estimate and being 1 if division by N should be used to compute the standard deviation.

The minimum and maximum and the indices of the respective elements along dimension `k` can be computed using `[d, indices]=min(X,[],k)` and `[d, indices]=max(X,[],k)`, while the `p`-th quantile of `X` is obtained using `quantile(X,p)`.

Statistical distributions
To generate an m by n array of pseudo-random numbers from the uniform and the normal distribution use the `rand(m,n)` and `randn(m,n)` commands.

Matlab also allows easily evaluating density and cumulative density functions. For the normal distribution with mean `mu` and standard deviation `sigma`, the respective commands are `normpdf(x,mu,sigma)` and `normcdf(x,mu,sigma)`. Beta and Gamma distributions can be evaluated using `betapdf/betacdf` and `gampdf/gamcdf`.

14 Rounding

Matlab supports self-explanatory functions for rounding:
`round(x)` for the integer closest to `x`
`floor(x)` for the closest integer smaller than `x`
`ceil(x)` for the closest integer larger than `x`.

15 Plots

Often the easiest way to plot a single data series is to open the series in the variable editor by double clicking on it in the workspace, then select the data you want to plot, and click on the plot button.

More generally, to plot data, use the `plot(x-data,y-data,options)` command, where `options` can be for example `'b:'` implying a blue, dotted line. To label the x-axis and the y-axis use `xlabel('name the x axis ')` and `ylabel('name the y axis')`, respectively. Add a title using the command `title('this is my title')` and a legend using `legend('name the first series','name the second series')`.

Usually, plots are saved as .fig-files and actually contain the respective data points, i.e. can be changed using the plot tools. Every plot can be saved and converted into a pdf by the command:
`print('-dpdf','name the pdf file')`.
However, printing into pdf-files often leaves a large amount of whitespace around the figures. It is preferable to print into eps-files:
`print('-depsc2','name the eps file')`
and then use the program eps2pdf.

16 Load / save data

16.1 The quick and dirty way: command form

To save the complete workplace, simply use the command: `save anyname`. To save only the variables `A` and `B` under the filename `anyname` use
`save anyname A B`
The corresponding file will be named `anyname.mat`. Remember to never use spaces or exotic characters in filenames!

To load the saved data use the command `load anyname` or `load anyname A` to only load variable `A`.

16.2 The recommended way: function form

The above way of loading and saving via the command form is quick to write for the programmer as you do not need to type any special characters. But it is quite slow to execute for Matlab, because the Just-In-Time compiler has a hard time parsing what the elements of the command mean. Faster code results from using the functional form of the commands like:
`save('anyname','A ','B')`

17 Data

There exists an import wizard to import data into Matlab. Another convenient way to import data is to copy the dataset into a Matlab file. This has the advantage that you can write comments directly to it (name, range, frequency, source).

To compute the first differences of a time series `data` use the command `diff(data)`. The command `corr([data1,data2])` computes the correlation between the `data1` and `data2`.

When plotting time series data, it is useful to have a timeline on the x-axis. To define `time`, use the following command: `time=start_date:frequency:end_date`, where frequency is equal to 0.25 for quarterly data and 1 for annual data. For instance `time=1947:0.25:2010.75` yields a time line from the first quarter 1947 to the fourth quarter in 2010.

Often data needs to be imported from Excel using `xlsread`:

```
[numerical_data text_data]=xlsread('C:\Some Folder\...
Datafile.xlsx','Name of the Excel Sheet','A1:X10');
```

18 *Dynamic Field Referencing

A very useful feature when using structures is dynamic field referencing.[2] It allows easily accessing and creating subfields from arbitrary strings with the field name given by the string. For example, the code

```
for ii=1:2
   my_structure.(['field_ ',num2str(ii)])=ii;
end
```

will create a structure named `my_structure` with two subfields that have the names `field_1` and `field_2` and store the value of the respective index variable `ii`.

19 *The `eval` command

The `eval` command in Matlab allows executing a given string as Matlab code. As such, it can be very powerful, but it also makes for very unreadable and often very slow code. For this reason, you should try to avoid it where possible – which is often the case. For example, in Section 18 we learned how to dynamically create field names. The same can be done using `eval`:

```
for ii=1:2
   eval(['structure.field_',num2str(ii),'=ii;']);
end
```

But this way of doing it is orders of magnitude slower than dynamic referencing.

20 *Debugging

Matlab has several ways of helping you to find errors in your code. The most basic tool is Matlab's syntax help. Figure 5 shows what happens if you accidentally use an invalid character in a variable name. In this case the usually green square at the top right corner

[2] See http://blogs.mathworks.com/loren/2005/12/13/use-dynamic-field-references/ for details

turns red to indicate a syntax error has been found. Moreover, a red horizontal bar appears next to the respective line. On mouse-over an explanation is displayed.

Often the square at the top right corner is light orange. This indicates that Matlab has found warnings. Warnings are no syntax error. Your code will usually run, but there are ways to improve it. You should follow the suggestions to warnings in order to improve your programming style.

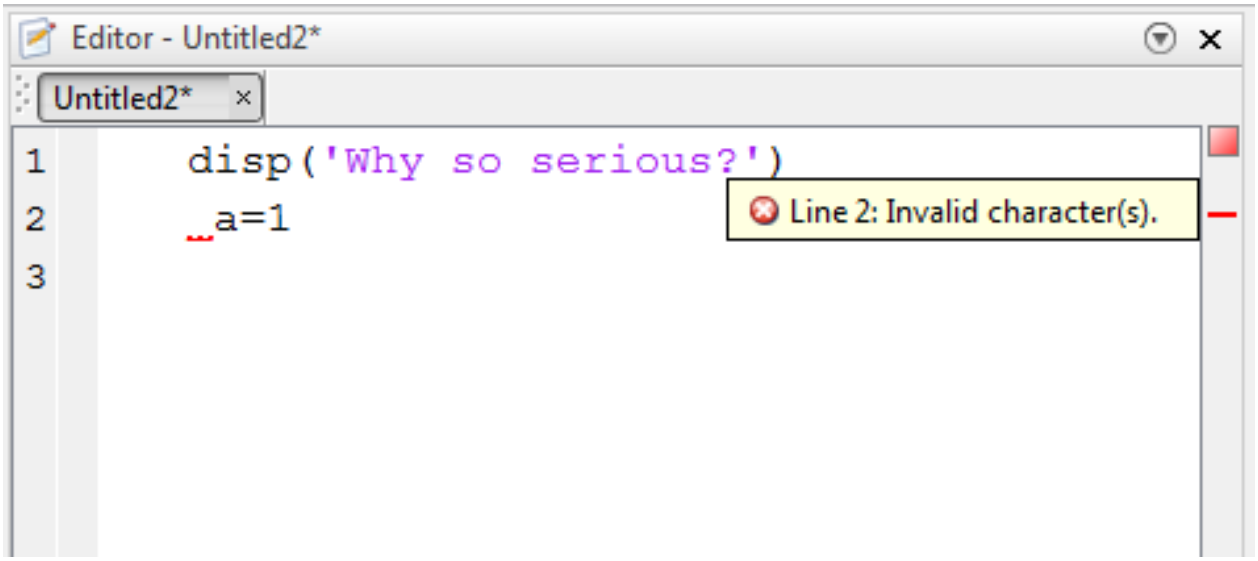

Figure 5: Matlab's syntax help

Often your code will run, but will not return the desired results for unknown reasons. In this case, the Matlab debugger can be used to trace out the problems. When you click on the black bar to the right of the line number, Matlab will set a `breakpoint` at this line.[3] A breakpoint appears as a red dot to the right of the line number as shown in Figure 6. If you run your code, Matlab will stop before executing this line. A green arrow at the beginning of the line shows where the execution of the program currently is. You can now use selective code evaluation with F9 to break down the computations into smaller bits. With F10 or the button with the sheet of paper and the blue arrow on the right you can execute the code line by line. Look at the variable values in the workspace to see if the computations behave as you expected.

If the problem occurs within a function, set a breakpoint at the beginning of the line where the function is called. When Matlab reaches this breakpoint and stops, use the button with the two sheets of paper and the blue arrow on the left or F11 to step into this function. Matlab will automatically open the source code of the function and allow you to continue debugging.

[3]Be aware that in Matlab versions before 2015b a `clear all` statement will also clear any breakpoints you have set.

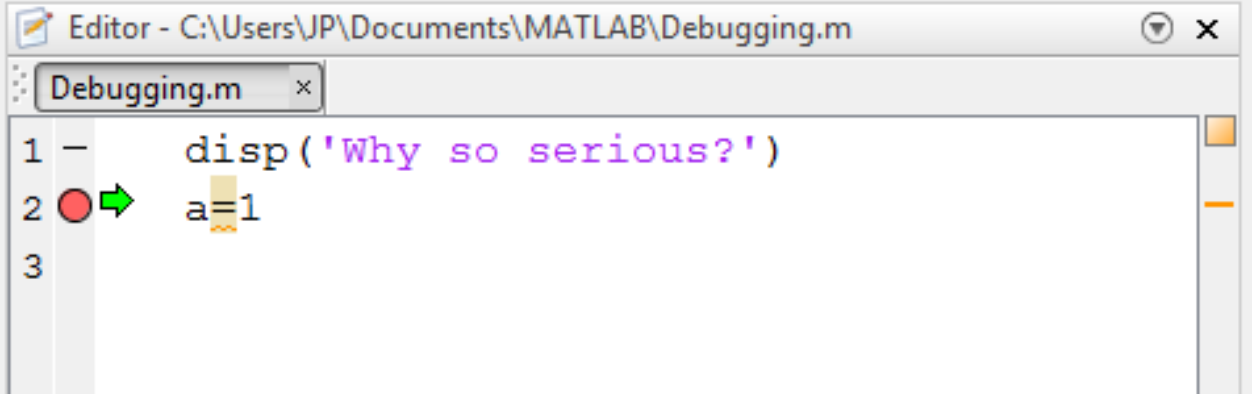

Figure 6: Matlab's debugger

21 *Numerical optimization

Consider you want to numerically find the root of the function $y = x^2 + 3x - 10$, i.e. you want to solve the equation $0 = x^2 + 3x - 10$. To do so, you can use the Matlab function `fsolve`, which is for solving non-linear systems of equations. The command is
`[x_opt, fval, exitflag]=fsolve(@(x) x^2+3*x-10,0)`
`x_opt` is the solution returned for `x`, `fval` the function value `y` at `x_opt`, and `exitflag` a code for the reason the algorithm terminated. If solving the equation was successful, `fval` should be 0 and `exitflag` be equal to 0. Looking at the input arguments, the `@(x)` is called a `function handle`. It indicates for which variable in the following equation it should be solved. The second argument, the 0 is the starting value for the algorithm. As the parabola in the example has a unique maximum, the starting value does not matter. The algorithm will always converge to the same value. However, for more complicated function, `fsolve` is only able to find local solutions and the starting value matters.

You can also provide your own functions to Matlab's optimization routines. Say you have written a function computing `y` for a given `x` in above equation:

```
function outvalue =simple_parabola(x)
outvalue=x^2+3*x-10;
end
```

In this case, you would provide a function handle to the function `simple_parabola` to `fsolve`:

```
[x_opt, fval, exitflag]=fsolve(@simple_parabola,0)
```

We can only touch upon the issues of numerical optimization and there are many intricacies involved. A good starting point is often the Matlab help, which provides good guidance on which algorithm/solver to use for the problem at hand. In general, functions differ whether your problem i) is nonlinear, ii) differentiable, iii) is univariate or multivariate, iv) involves solving or minimizing a function, and/or iv) features constraints.
Important solvers for economic applications are i) `fminbnd`, a fast Matlab routine that finds the minimum of single-variable continuous function on a fixed interval and is thus often used

to solve for the optimal labor supply, and ii) Chris Sims `csminwel.m`, a popular solver that minimizes a scalar-valued continuous function over a vector of arguments.

22 *Precedence Rules

Matlab mostly uses precedence rules from standard algebra. Expressions enclosed in brackets get the highest precedence followed by exponentials, multiplication/division, and addition/subtraction. That is `(1+2)*3` evaluates to 3 while `1+2*3` evaluates to 7. Moreover, precedence naturally evolves from left to right.
Note that things may become dangerous when mixing `true` matrix algebra like matrix multiplication with element-wise operations. For example `[1 0;0 1].*[1 1;0 1]*[1 2;3 4]` is not the same as `[1 0;0 1].*([1 1;0 1]*[1 2;3 4])`. Hence, when using element-wise operations it is advisable to voluntarily use rather more then fewer brackets.

23 *Object-Oriented Programming

Object-oriented programming is increasingly becoming an important part of the Matlab design, most importantly in the proprietary Econometrics Toolbox. The important difference to function-oriented programming is that `objects` are used, which are data structures consisting of data fields as well as methods/functions to work on the data or to interact with other objects. For example, if you want to initialize a random number generator using the Mersenne Twister-algorithm and with seed 1, you can create a `RandStream` object `s` by typing:

```
s = RandStream('mt19937ar','Seed',1);
```

Now you can use the method `get` on the object `s` to read out its properties:

```
s.get;
```

The difference to the `structures` encountered before is that we are calling a function after the dot, not a data field. To see the current state, type `s.State`. To generate a random number from this object, type `rand(s)`. If you now look at the current state, you will see that it has changed because you have just drawn a random number from this stream. To reset the random number generator to its initial state, you can use
`s.reset;`.

24 *Profiling Your Code

If you want to know how fast your code runs and where the bottlenecks are, you can use Matlab's builtin profiler. Just go to 'Editor' and choose 'Run and Time'. Matlab will then show you the execution time and the time spent in certain functions/lines of your code. A quick and easy way to compare code alternative is the `tic toc`-construct. Consider the following listing that compares using a loop to delete negative values of a vector with the logical indexing shown in section 10:

```
a=randn(10000,1);
b=a;
tic
iter=1;
while iter<=length(a)
     if a(iter,1)<0
          a(iter,:)=[];
     else
          iter=iter+1;
     end
end
toc

tic
b(b<0,:)=[];
toc
```

After initializing the random vectors, the two different codes are inserted between a `tic toc`-construct. After running the code, Matlab displays the runtime in the command window:

```
Elapsed time is 0.149053 seconds.
Elapsed time is 0.001669 seconds.
```

indicating that the logical indexing is faster by several orders of magnitude.

25 *Advanced Plotting

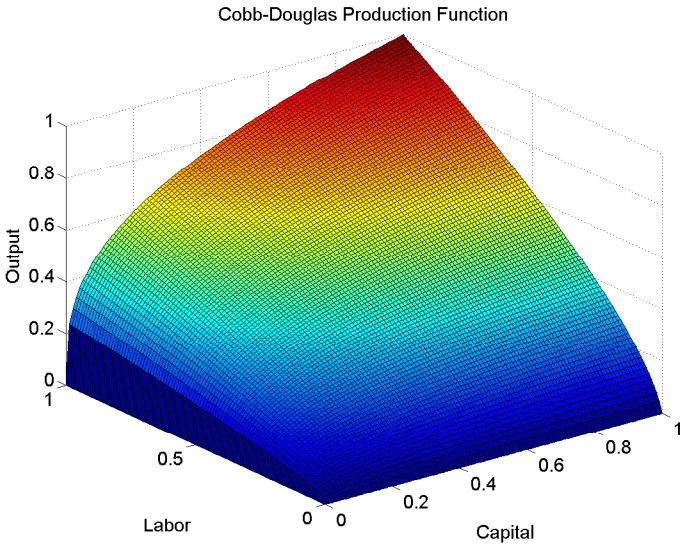

Figure 7: Cobb-Douglas production function

Matlab is also capable of creating 3-D-Graphs like the one shown in Figure 7.[4] It shows

[4] A good tutorial can be found at http://www.bu.edu/tech/research/training/tutorials/

a Cobb-Douglas production function along the capital and labor dimensions
```
k=[0:0.01:1]';
l=[0:0.01:1]';
```
with a capital share of
```
alppha=0.3175;
```

Before being able to plot the function, we need to define a two-dimensional grid of combinations of capital and labor for which we want to plot the function. This is done be the `meshgrid`-function:
```
[K,L]=meshgrid(k,l);
```

Given those combinations in matrix form, we can use element-wise operations to compute the production for the grid combinations of labor and capital:
```
Y=K.^alppha.*L.^(1-alppha);
```

Finally, we create figure with figure handle `h` (a figure handle is like a name which we can later use to address the figure):
```
h=figure
```
and use a `surf`-plot (short for surface) with y,x, and z given by capital, labor, and output:
```
surf(K,L,Y)
```

Lastly, label the axes in font size 14:
```
xlabel('Capital','FontSize',14)
ylabel('Labor','FontSize',14)
zlabel('Output','FontSize',14)
```
title the graph:
```
h=title('Cobb-Douglas Production Function')
```
increase the font size of the title
```
set(h,'FontSize',14);
```
and increase the font size of the axes
```
set(gca,'FontSize',14);
```
where `gca` stand for "get current axes".

Note that Matlab internally uses two different types of renderers to create figures: `painters`, which is fast and typically used for easy and small objects, and `zbuffer` that works better for complex scenes. Usually, you don't need to worry about this as Matlab chooses the one it needs. However, when printing graphics that use shading like the 3-D plot of the isoquants of a Cobb-Douglas production function, Matlab uses the `zbuffer`-option to create the figures. But if you export this figure into a pdf- or eps-file, Matlab suddenly switches to `painters`. This leads to a loss in the color gradient observed in bottom of the second picture. You can circumvent this issue by manually specifying the renderer in the `print`-command, i.e. for example
```
print -depsc2 -zbuffer CobbDouglasGraphzbuffer
```

---

visualization-with-matlab/.

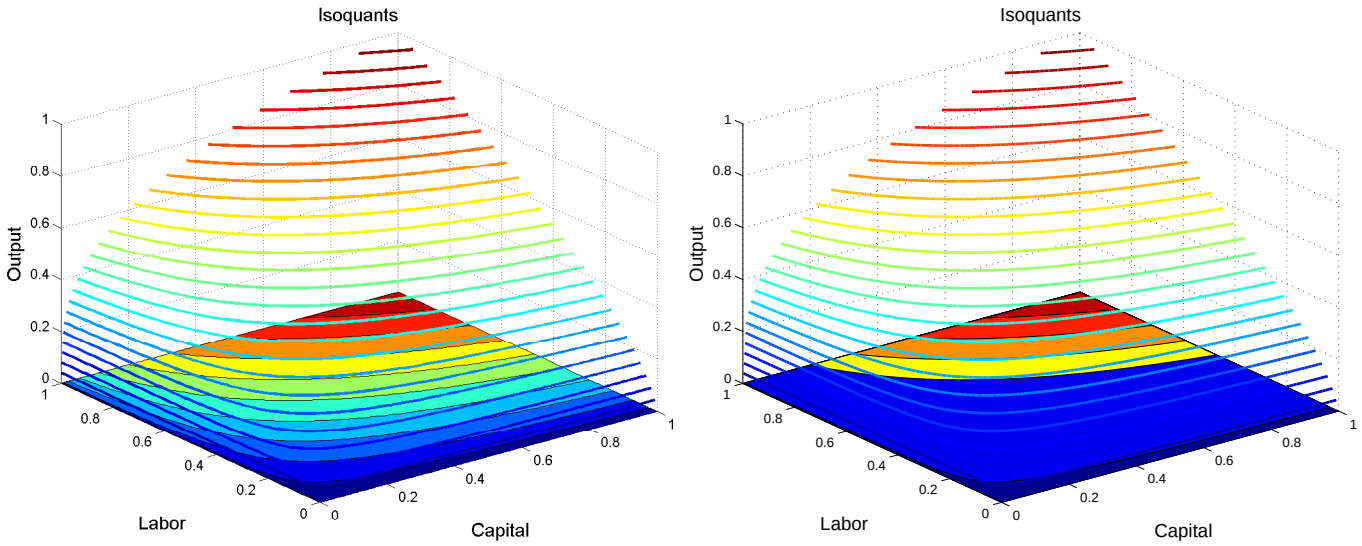

Figure 8: Cobb-Douglas isoquants. Left: printed with the `zbuffer`-option; right: printed with the default `painter`-option