Matlab Introduction Documentation

Release 1.0

Lutz Hendricks

Aug 11, 2019

Contents

Contents:

CHAPTER 1

The Basics

1.1 Installation and Configuration

Instructions for installation from ITS

1.1.1 User Interface

The Matlab window has a bunch of sub-windows.

Of these I only keep 3:

1. file browser: shows files in current directory
2. command window: here you can type commands for interactive calculations
3. editor: here you edit program files

   Tip: undock the editor, so it sits in its own window.

1.1.2 Configuration

Options can be set in `preferences` as usual (keyboard shortcuts, colors, etc).

Options can also be defined in `startup.m` in the home directory. This is where you want to put any code that should run every time Matlab starts.

You also need a startup file for each project, so that Matlab can find your code.

1.2 Key Features of Matlab

Matlab is a mathematical programming language.

Matlab is *interpreted* (or just-in-time compiled, as opposed to compiled).

This means that it can be used interactively by typing commands at the prompt >> in the command window.

But most of your code will be written to program files (matlab `functions`).

Functions can be run from the command prompt.

Commands are either built-in or written by the user. Every command is a stored in an M-file.

If you type `sum([1,2,3])` at the command prompt, Matlab will look for `sum.m` and run it. How does Matlab know where to look? It uses the Matlab Path

1.2.1 Matlab Pros

It is easy to learn and use.

- The main reason is that Matlab is *dynamically typed*. That is, a variable can change type as it is modified.
- Arguments are passed to functions without declaring their types.
- The documentation is outstanding.

Matlab has a large standard library.

- But there is little (high quality) user contributed code.
- Most of it lives on the Matlab file exchange. But this is not curated in any way. Use at your own risk.

1.2.2 Matlab Cons

Relative to compiled languages (such as `C`) and relative to typed languages (such as Julia), Matlab is slow.

Because the types of function arguments are not checked (unless the user does so), “interesting” errors arise.

1.3 Interactive Matlab

In the command window, type your commands at the >> prompt.

Matlab returns the answers on the screen.

Example

```
>> date
ans =
26-Jul-2015
```

Typing date at the command prompt executes the built-in `date` command.

> Tip: When you write your own programs, make sure all names are *unique*. If you write a function called `date`, there will be trouble.

`date` returns a string containing the date.

Its return value is shown on the screen after `ans =`.

To get rid of the `ans =` part, we could use `disp(date)`.

Example

```
>> dateString = date;
>> disp(dateString);
26-Jul-2015
```

> Note: Matlab displays the result of any command, unless the command is ended with `;`.
>
> If you ever see a lot of numbers appearing on the screen "out of nowhere," you probably forgot the `;` at the end of a line.

##Doing Math

Matlab has built-in operators for common math operations:

```
a = 2; b = 7;
disp(b*a);
14

y = b ^ a;
disp(y);
49
```

Built-in functions:

```
>> sin([1,2])
ans =
    0.8415    0.9093
```

A neat feature: most functions work on *arrays* of inputs. One command returns the `sin` values for many numbers in one fell swoop.

1.4 Getting Help

To find out more about a particular command, type

```
help [command name]
```

To find a command for a given task, use the Matlab help system.

1.4.1 Matlab tutorials

EconPhd

Mathworks

List of free Matlab books

MIT course

CHAPTER 2

Writing Solid Code

2.1 General Concepts

2.1.1 Structured Programming

Break down a process into chunks.

Package each chunk into a separate function.

A key idea: aside from an explicitly specified set of inputs and outputs, the function is independent of the rest of the world (*encapsulation*).

Write your code *top down by stepwise refinement*

- Start with an outline of the steps.
- For each step:
  - if it's trivial: write it out in pseudo code.
  - it it's not: write another outline for that step
- Finally, translate the pseudo code into code.

Each function should be *short* and perform exactly *one task*.

2.1.2 Write reusable code

- Write each function to be sufficiently general, so it can be reused in other projects.
- Over time, you will accumulate a library of code that can be used over and over.
- Example: Write code that produces marginal products, average costs, etc for a CES production function.

2.1.3 Notes on Writing Code

Read a good book on best practices in programming.

1. I see a lot of very poorly written code that is impossible to understand and not robust. Do yourself a favor and save a lot of time down the road by learning to how write quality code.
2. A book I like is "Writing Solid Code."

2.2 Some rules

1. No literals as in `x=zeros([5,3])` or for `i1 = 1 : 57`. It's not robust and hard to read.
2. No global variables.
3. Don't worry about speed. Worry about robustness and transparency.
4. Unique names: I suffix all functions I write with a project code (e.g. `var_load_sc.m`, `var_save_sc.m`, etc). It avoids naming conflicts with other projects.
5. Your code should contain lots of self-testing code. Most code is so fast that the loss of speed is irrelevant. If it is relevant, have a switch that globally switches test code on and off.
6. Avoid using reserved words, in particular `i` as an index.

2.2.1 Style matters

This point is hard to overstate. It is extremely important to write code that is easy to understand and easy to maintain.

In practice, you often revisit programs months or years after they were written. They need to be well documented and well structured.

The programs needed to solve a stochastic OLG model have thousands of lines of code. The only way to understand something this complex is to break it into logical, self-contained pieces (a function that solves the household problem, another that solves the firm problem, etc.).

One example of how important this is:

Air traffic control centers still operate with hardware from the 1970s. The reason is that nobody understands the software well enough to port it to new hardware.

The FAA has already spent billions of dollars on unsuccessful attempts to rewrite this mess.

Another example is the Space Shuttle, which runs (now "ran") on hardware from the 1960s. The reason is again that the software engineers can no longer understand the existing code.

There are many books on good programming style. One that I like is *Writing Solid Code* by Steve Maguire. Read it!

2.2.2 Avoid literals

Your code should rarely use specific values for any object. When you refer to an object, do so by its name.

For example, create variables to hold directory names and constants. The reason is that code is otherwise hard to change and maintain.

Imagine you set some parameter `sigma=2`, but refer to it as `2` in your code instead of `sigma`. If you decide to try `sigma = 3`, you need to locate and change every occurrence of `sigma` in your code. It's a mess.

The Golden Rule is: Every literal must have a name. Its value is defined in one place only.

Related to this: do not hard-code *functional forms*.

If you want to compute the marginal product of capital, write a function for it. Otherwise, if you want to switch from Cobb-Douglas to CES, you have to rewrite all your programs.

- Object oriented programming makes it easy to swap out entire parts of a model. We will talk about this later.

2.2.3 Self-Test Code

Your code should test itself automatically and periodically.

Embed error catching code everywhere (use valideattributes).

Catching bugs early makes them easier to find.

A trick to prevent your code from getting slowed down by self-testing:

- add a debugging switch as an input argument to each function (I call it `dbg`).
- if `dbg` is 0: go for speed and turn off self-testing
- if `dbg > 10`, run all self-test code

The process is then:

1. Write code. Make sure it runs (correct syntax).
2. Make sure it is correct (run all self-test code – slow)
3. When you are confident that your code is good, set `dbg = 0` and go for speed
4. But every now and then, randomly switch `dbg` on so that self tests are run (little cost in terms of run time; a lot of gain in terms of confidence in your code).

2.2.4 Automated Unit Testing

The golden rule:

> When you write a function, write a test function to go with it.

It is hard to overstate the importance of automated testing. It gives you peace of mind. When you change some code, you can simply rerun your test suite and ensure that nothing has been broken.

The key is to fully automate the testing. Your project should have a single function that runs all tests in order.

All programming languages have unit testing frameworks that make it easy to automate this process. Matlab's framework is described here.

2.2.5 Optimization

Optimization refers to program modifications that speed up execution.

> Think before you optimize!

Most code runs so fast that optimization is simply a waste of time.

Also: Beware of your intuition about where the program spends most of its time.

Here is an example: Consider the function that solves a stochastic OLG model.

It turns out that it spends 80% of its time running the Matlab interpolation function `interp1`!

There is little point optimizing the rest of the code.

To find out what makes your program slow, run the Matlab profiler.

Some of Matlab's built-in functions are extremely slow.

- Two examples are `interp1` and `sub2ind`.
- It is easy to write replacements that run ten times faster.
- The Lightspeed library contains faster versions of built-in functions.

2.3 Common mistakes

2.3.1 Passing arguments in the wrong order.

Matlab does not check the types of arguments.

Often functions have lots of input arguments.

It is easy to confuse the order and write `myfun(b,a)` instead of `myfun(a,b)`.

To avoid this: check that inputs have admissible values.

2.3.2 Passing too few arguments.

Matlab permits to omit input or output arguments when calling a function.

It is useful to check that the number of input arguments is as expected using nargin.

2.3.3 Reusing variable names.

Matlab does not permit explicit declaration of variables. It is therefore easy to use a variable name twice without noticing.

2.3.4 Indexing problems.

It is easy to make mistakes when extracting elements from matrices. This is especially true for code that wraps a loop into a single line of code.

For example, this is easy to read:

```
for ix = 1 : nx
  zV(ix) = xV(ix+2) + yV(nx + 2 - ix);
end
```

This is the same thing, more compact but harder to read:

```
zV = xV(3 : nx+2) + yV(nx+1 : -1 : 2);
```

Tip: Write out code explicitly. Once it works, one can still make it faster (if that is even worthwhile).

Another common indexing mistake is to use too few arguments. For example:

```
x = rand([3,4]); y = x(3);
```

This should produce a syntax error, but it does not. Instead, it flattens x into a vector and then takes the 3rd element.

2.4 Material for Economists

Quantitative Economics by Sargent and Stachursky

- a really nice collection of lectures and exercises that covers both programming and the economics of the material (in Julia and Python)

Tony Smith: Tips for quantitative work in economics

2.5 Matlab Material

1. Mathworks style guidelines
2. Datatool style guidelines
3. Johnson: Elements of Matlab Style (Book)
4. Good Matlab Practices
5. Best practices for scientific computing

2.6 Material Not for Economists

1. Lifehacker: teach yourself how to code

CHAPTER 3

Data Types

Matlab knows the following data types:

1. **Matrices** of floating point numbers. Vectors and scalars are special cases.
2. Text **strings**. These are really vectors of characters to Matlab.
3. **Structures**.
4. **Cell** arrays.

Then there are more specialized datatypes such as `tables`.

You can also define your own data types. We will talk about that more later (see object oriented programming).

The type of a variable is **not fixed**.

The following is perfectly fine code (and a popular source of errors).

```
a = 1;
a = 'a string';
```

3.1 Vectors

To create a vector, simply fill it with values:

```
a = [1, 2, 3];
disp(a);
1 2 3
```

Matlab knows row and column vectors:

```
b = [1; 2; 3];
disp(b);
1
```

(continues on next page)

(continued from previous page)

```
        2
        3
        disp(a*b);
        14
```

3.1.1 The colon operator

The colon `:` makes a vector of sequential numbers:

```
        disp(1 : 4);
        1 2 3 4
        % Now with step size 3
        disp(1 : 3 : 10);
        1 4 7 10
```

3.1.2 Indexing

To extract elements from a vector, hand it a list (vector) of indices.

```
>> a = [2, 3, 4, 5];
>> disp(a[1, 3]);
2 4
```

The vector of indices can be constructed with the colon operator:

```
        a = 11 : 20;  disp(a(2 : 2 : 6));
        12 14 16
```

Any vector can be used to index elements:

```
        idxV = [2 5 8];  disp(a(idxV));
        12 15 18
```

Then there is logical indexing.

```
>> a = 11 : 20; idxV = (a > 15);
>> disp(idxV)
     0     0     0     0     0     1     1     1     1     1
>> disp(a(idxV))
    16    17    18    19    20
```

See also the find function for a similar idea.

Exercises

Start with `x = 1 : 3 : 30`.

1. Find all even elements.
2. Find all elements between 5 and 20.
3. Set all even elements to their negative values.

3.2 Matrices

A matrix is an n-dimensional array of numbers. One can also have arrays of other data types (see *cell arrays*).

To create a matrix, simply fill it with values.

```
        a = [1 2 3; 4 5 6];  disp(a);
        1 2 3
        4 5 6
```

Many commands work directly on matrices.

```
        a = [1 2 3]; b = [2; 1; 1]; disp(a * b);
        7
        disp(b * a);
        2 4 6
        1 2 3
        1 2 3
```

To extract elements:

```
        c = b*a;  disp(c(1:2, 2:3));
        4 6
        2 3
```

To extract all elements:

```
        disp(c(1,:));
        2 4 6
```

But note: c(:) yields all elements flattened into a vector!

```
        disp(c(:)');
        2 1 1 4 2 2 6 3 3
```

To extract a list of elements, use `sub2ind`.

```
c = [1 2 3; 4 5 6];
idxV = sub2ind(size(c), [1,2], [2,3]);
>> c(idxV)

ans =

     2     6

>> disp([c(1,2), c(2,3)])
     2     6
```

3.2.1 Matrix Pitfalls

Incorrect Indexing

```
        c = [1 2 3; 4 5 6]; disp(c(5))
        3
```

`c(5)` should be a syntax error, but instead yields the 5th element of the flattened `c(:)` matrix!

Matrix dimensions change when you add elements.

```
>> x=1;
>> x(2,2) = 2

x =

     1     0
     0     2
```

This is a common source of bugs and one of the most counterproductive features of Matlab.

3.3 Multi-dimensional matrices

Matlab matrices can have more than 2 dimensions.

```
        a = rand([3,2,5]);
        size(a)
        ans =
        3     2     5

        a(:,:,3)
        ans =
        0.9218    0.4057
        0.7382    0.9355
        0.1763    0.9169
```

Sub-matrices work just like ordinary 2-dimensional matrices.

But: `a(:,1,:)` is not a 2D matrix. It's a 3D matrix with a singleton 2nd dimension.

3.3.1 Matrix Exercises

1. Construct a matrix `A` with elements `[2,4,...,20]` in row 1 and `[1,4,7,...,28]` in row 2.
2. Replace row 1 with its square.
3. Find all columns where row 1 > row 2.
4. Let `x=ones(10,1)`. Compute `Ax`.

3.4 Structures

Structures are containers for variables of different types.

They are defined by simply adding element to a blank structure.

3.4.1 Example:

Store the contact information for a person.

```
        contactS.Name = 'John Doe';
        contactS.Age = 37;
```

The elements are accessed by name:

```
        disp(contactS.Age);
        37
```

One can have structure arrays, but they are tricky because each element must be a structure with the same fields.

Often, *cell arrays* are more useful.

3.4.2 Where structures are useful

Use a structure to pass a large number of arguments to a function.

- Example: Set of parameters and prices for solving a household problem.
- Example: Our models store all fixed parameters in a structure that is passed to essentially all functions.

Structures can make code robust against changes.

- Example: Add a preference parameter to the model. Only the household code needs to be changed. Other programs use the same structure to look up model parameters.

3.5 Text Strings

To Matlab, a text string is a vector of characters. And a character is represented by a number.

Therefore: most vector operations work on text strings.

Example:

```
myName = 'Hendricks'; disp(myName(5:8));
rick
disp(myName(5) == 'r');
1
```

3.5.1 sprintf

sprintf produces formatted text from (usually) numerical inputs.

The syntax is almost the same as in `C`. Read the manual for details.

Example:

```
        sprintf('Integer: %i. Float: %5.2f. String: %s', 5, 3.71, 'test')
        ans =
        Integer: 5. Float: 3.71. String: test
```

3.6 Cell Arrays

A cell array is an n-dimensional array of mixed data.

Each cell can hold a different data type.

```
>> x = {'abc', 17; [3,4,5], {1, 2}}
x =
    'abc'            [       17]
    [1x3 double]     {1x2 cell}
>> disp(x{1,1})
abc
>> disp(x{2,1})
     3     4     5
```

Uses:

- most common: replacement for structure array when one is not sure that all elements have the same fields.

3.7 Numeric Precision

By default, numeric variables are stored as `double` (double precision float, 64 bit).

Even if you define `x=1`, it is a `double`.

```
>> x=1;
>> class(x)

ans =

double
```

If you work with large (simulated) datasets, you may want to store matrices in formats that take less storage.

```
>> x = ones([1,3], 'uint8')

x =

    1    1    1
```

This leads to some nice opportunities for errors about which Matlab helpfully does not complain (another Matlab bug).

```
>> x(2)=1e5

x =

    1   255    1
```

The number assigned to `x` was too large for an 8 bit integer. It gets truncated without warning.

Worse, applying common numeric operations to integers returns integers:

```
>> y = x/2

y =

    1   128    1

>> class(y)

ans =

uint8
```

Rule of thumb: Store large datsets in low precision. Make everything double the moment you load it.

CHAPTER 4

Editing and Debugging

4.1 The Editor

Matlab has a built-in editor.

Benefit: Integration with the command processor. When a crash occurs, you can inspect variables in the editor.

4.1.1 Editor Tips

Cell mode

- Start a line with `%%` and it becomes a collapsible `cell`. That makes moving around the editor easier.
- One can do other things with cells (e.g., run a portion of code; collapse it; etc.)

Comments

- Use **lots** of comments to make your code readable.
- Block comments are enclosed with `%{...%}`

Mlint

- highlights questionable syntax or errors

4.2 Debugging

Debugging means locating program errors.

4.2.1 `keyboard` command

A useful command for debugging is keyboard. It halts program execution and returns the user to the command line. Any commands can then be executed as if the program itself contained those commands. In particular, one can inspect the values of all local variables.

Setting break points has a similar effect.

4.2.2 Debugging mode

Matlab can switch to a debugging mode that allows the user to inspect the state of a program when it crashes.

To switch on debugging mode, type `dbstop error`

To end debugging mode type `dbclear all`

When a program crashes, it is halted and the command prompt is activated. The effect is exactly the same as placing a `keyboard` statement at the point where the program crashed.

To end debugging, type `dbquit` This stops the program.

4.2.3 Tips

Write **test functions** for every sub-routine you write. Never assume that a new program will run correctly.

Write a `test_all` functions that runs all the tests in one go.

Embed **self-test code** in your programs.

Check everything you can.

- Is the number of input arguments correct?
- Are the input arguments of the right type and dimension?
- Are they in admissible ranges?
- Are intermediate results admissible?
- Are the output arguments acceptable?

By checking all the time, errors are spotted as early as possible which makes them much easier to fix.

See validateattributes for an easy way of implementing all of this.

CHAPTER 5

Functions and Scripts

5.1 Encapsulation

A key idea of structured programming:

- package code into self-contained functions
- avoid side effects

A function should only change the rest of the world through the outputs it explicitly returns. This is called encapsulation.

For this to work, all variables inside a function must be invisible to other code – they are **local**.

5.1.1 Example:

```
function y = f(x)
        a = 5;
        y = x + a;
end

% Command line:
>> a = 3; y = f(0); disp(y)
5
>> disp(a)
3
% Now the converse
function y = g(x)
        z = x + a;
end
>> g(0)
% Error: `a is not defined`
```

5.2 Namespaces

A namespace is a set of functions or scripts that can "see" the same objects.

Example:

- In the function `f(x)` above, `a` was in `f(x)`'s namespace, but not in `g(x)`'s.
- Conversely, the `a` inside `f(x)` is local and not in the command line's namespace.

In Matlab, there are only 2 namespaces:

1. Global: this is what you can access from the command line.
2. Local: inside a function.

Note: This is not exactly true because there are nested functions.

This means in particular that all global variables and all functions are visible everywhere. Their names must be unique.

Other languages have more control over namespaces (using "modules").

5.3 Functions versus Scripts

Scripts are simply collections of commands that are run as if they were typed at the command prompt.

Scripts run in the global workspace. This is, generally speaking, not good. They create side effects.

Rule of thumb:

> Always use functions, never scripts!

Functions are similar to scripts with one crucial difference:

- All variables inside a function are **private**.
- Other functions cannot see the variables inside a function (*encapsulation*).
- Any variable a function should know must be passed to it as an input argument.
- Any variable to be retained after the function finishes must be passed out of it as a return argument.

An important principle of structured programming:

> **Package a well-defined task into a function with a simple interface.**

A function looks like this

```
function [y1, y2] = fname(x1, x2)
        % Do stuff with x1 and x2 to generate y1 and y2
end
```

This would be stored in a text file called `fname.m`.

A side note: Even built-in Matlab commands are often written in Matlab.

Try `open sub2ind`. You get to see (and you can modify) the source code for the built-in `sub2ind` command (`sub2ind.m`).

Now try `open zeros`. This opens `zeros.m`.

- you get an m-file that is blank except for comments
- this means: `zeros` is not written in Matlab (it's truly built-in)

5.4 The Matlab Path

The Matlab path determines which m-files are visible in the global namespace (and inside all functions).

In contrast to other languages, this is an all or nothing affair: If a function is visible somewhere, it is visible everywhere.

When the user issues a command, such as `m = zeros([3,4]))`, Matlab searches for a matching m-file, in this case `zeros.m` and runs it.

It does not matter whether `zeros` is a built-in command or an m-file written by a user.

Where Matlab looks is defined by the `matlab path`.

This is a list of directories that Matlab goes through when it looks for functions. In case you wonder, this is done as a `cell array` of strings.

You can see what's on the path by typing `path`.

You add directories to the path with addpath.

In addition, Matlab searches the current directory (the one shown in the file browser).

Therefore, every time you write new code, you need to put the directories on the `path` before the code can be called.

There is no way to allow one function to access some code without changing the `path` globally. This differs from other languages. It makes it hard to organize code.

There is one exception: local functions are only visible to functions within the same m-file.

5.5 Organizing Code

When you write new commands, you need to make sure Matlab can find them. This is best accomplished in two ways:

1. For files that belong to the one particular project, place them in a project directory. Then switch the current directory to that directory using `cd`.

   Or put the directory on the `path` using a startup routine.

2. For files that are **shared** between several projects, place them in a special directory (`blah/shared`). Place this directory on the path: `addpath('blah/shared')`.

Over time you will write many general purpose routines that should be stored in this `shared` directory. Reusable code!

5.5.1 Name Conflicts

Now you run into a problem: **name conflicts**.

Each time you write a new function, you need to make sure that there is no other function with the same name.

There are 2 ways of ensuring this:

1. Suffixes: For each project, invent a suffix and append it to every function name.

   Such as `plot_821.m`

2. Packages: If you place an m-file into a directory that starts with `+`, let's say `+econ821`, you can access it like this: `econ821.plot(x)`.

   For this to work, the parent dir of `+econ821` must be on the `path`.

   This is also useful to organize your code within a project (which may contain hundreds of functions).

5.6 Example of a Function

Imagine we want to compute the sum of 3 variables.

Type the following code in a text editor:

```
function xSum = sum3(x1, x2, x3);
% This function sums 3 variables
        sum1 = x1 + x2;
        xSum = sum1 + x3;
end
```

Save it as `sum3.m` in a directory Matlab knows to find.

Row 1 defines the name of the function, its input arguments and its output arguments.

Row 2 is a comment starting with %. Matlab ignores it.

Rows 3 and 4 contain the commands to be executed.

To run `sum3.m`: At the command prompt type

```
        sum3(1, 2, 3)
        ans =
        6
```

To illustrate that the variables inside `sum3.m` are private, try typing `sum1` at the command prompt. The result:

```
??? Undefined function or variable 'sum1'.
```

Matlab complains that variable `sum1` does not exist.

`sum1` was visible inside of sum3.m, but not outside of it.

5.7 More on Private Variables

Variables inside of `sum3.m` are also not visible in functions called from within `sum3.m`.

Example

Modify `sum3.m` to read:

```
        function xSum = sum3(x1, x2, x3);
                sum1 = x1 + x2;
                sum3sub;
                xSum = sum1 + x3;
        end
```

Then create the function:

```
        function sum3sub
                disp(sum1)
        end
```

Running `sum3.m` results in an error: sum1 is not found.

5.8 Global Variables

To make a variable visible from anywhere, define it as a `global` (in `sum3.m` and `sum3sub.m`):

```
global sum1;
```

Now running `sum3.m` does not yield an error message.

> Remark: Avoid globals where possible.
>
> A function should be a self-contained unit with a clear interface to the outside world (via its input and output arguments).
>
> Globals create confusion.

5.9 Nested Functions

If you need a function to see the local variables in another function, nest it inside the other function.

Example:

```
function f
        a = 5;
        disp(g(17));
        % After calling g(), a == 52

        % Nested function can see `a`
        function z = g(x)
                z = x + a;
                a = 52;
        end
end
```

The main use of this: optimization. See below.

5.10 Passing by Value

In Matlab, all function arguments are passed **by value**.

That means, Matlab creates a copy of the variable and passes it to the function.

However, Matlab is smart enough not to create a copy when the function does not change the variable ("copy on change").

Still, this can be very inefficient. If you pass a gigantic array to a function and change one element inside the function, the entire array needs to be copied.

There is no way of passing by reference.

5.11 Examples

5.11.1 Example: Cobb-Douglas production function

This is a general purpose Cobb-Douglas function

Notable features:

- block comment describing inputs and outputs

  this is all the rest of the world needs to know about the function
- a unique name
- self-test code governed by debugging switch `dbg`

5.11.2 Example: Finding the Zero of a Function

Task: Find the solution to the equation \(f(x)=\ln(x)-5=0 \).

Set up a function that returns the deviation f(x)

```
function dev = dev1_821(x);
dev = log(x) - 5;
end
```

Use the built-in function `fzero` to search for the solution:

```
fzero('dev1_821', [0.1 100])
ans =
7.3891
```

What if I want to pass additional parameters to the objective function? Make it a nested function.

Example here.

5.12 Example: Two period household

Household solves \(\max u\left(c,g\right)\)

subject to \(y=c+s \) and \(g=z+sR\)

A solution: \(c,g,s\)

that solve 2 budget constraints and Euler equation

\(u_{c}=u_{g} R \)

Assume \(u\left(c,g\right)=\frac{c^{1-\sigma}}{1-\sigma}+\beta\frac{g^{1-\sigma}}{1-\sigma} \)

5.12.1 Pseudo code

This is not a trivial program to write. So we break it down into trivial steps.

See Tips on programming

We design top-down.

Level 1:

Task: Find optimal \(c\).

1. Set parameters. Set up a grid of values for c

2. For each c: Calculate deviation from Euler equation.
3. Find the c with the smallest deviation.

   Note: Usually one would not restrict \(c\) to lie on a grid.

Level 2:

Task: Calculate deviation from Euler equation.

Given: guess for \(c\), parameter values

1. Use budget constraints to calculate \(s,g\)
2. Return deviation: \(dev=u_{c}-u_{g}R \)

Level 3:

Utility function.

- Return \(u_{c} \) and \(u_{g} \)
- for given \(c,g\) and parameters.

5.12.2 Code

We write the code bottom up.

Utility function:

- Allow matrix inputs (cM, gM).
- Parameters as arguments.
- This should really be a general purpose function (my library contains an OOP version).

Sample call:

```
>> hh_example_821(2, 0.5, 1.04, 0.9, 1.5)
c = 1.224490   Dev = 0.034947
```

5.13 Exercises

1. Write a CES utility function that computes \(u'(c)\) and \(u(c)\).
2. Write a function that computes the inverse of \(u'(c)\).
3. Write a test function that checks properties of the utility function: 4. The inverse of the inverse equals \(u'(c)\). 5. Marginal utility is decreasing.

Extra credit:

1. Package all of that into an object (a user defined data type).
2. Now write all of this for \(u(c)=e^{-\phi c}\).
3. In your test function, set things up so that you only need to change a single line of code to test both utility functions (the benefit of OOP in action).

CHAPTER 6

Matlab Oddities and Pitfalls

6.1 Matrix Indexing

Matlab is perfectly fine with invalid matrix indices.

The following code should produce a syntax error:

```
x =
     1     2     3
     4     5     6

>> x(5)
ans =
     3
```

This is a very common source of bugs.

6.2 Load and Save

The `load` command is insane. It writes variables to the workspace that cannot be seen in the code. Always use load as a function as in `m = load(filename)`.

Even then `load` is insane. To retrieve the contents of `m` you need to know its name. Example:

```
x = 10;
save('test.mat', 'x');
m = load('test.mat');
disp(m.x);
10
```

It's best to write a wrapper for `load` and `save` so that `load2('test.mat')` returns `10` instead of `m`.

6.3 Numeric Precision

Matlab “downgrades” numeric precision to the lowest common denominator in calculations.

```
x=ones([1,1], 'uint8'); y = 1.2345;
>> z=x/y;
>> disp(z)
    1
>> class(z)
ans =
uint8
```

One solution: convert everything to `double` as soon as you load it.

Always check that intermediate results are still `double` (using validateattributes).

CHAPTER 7

Graphics

Matlab contains a large number of functions that plot data in 2D and 3D.

The most important is `plot`.

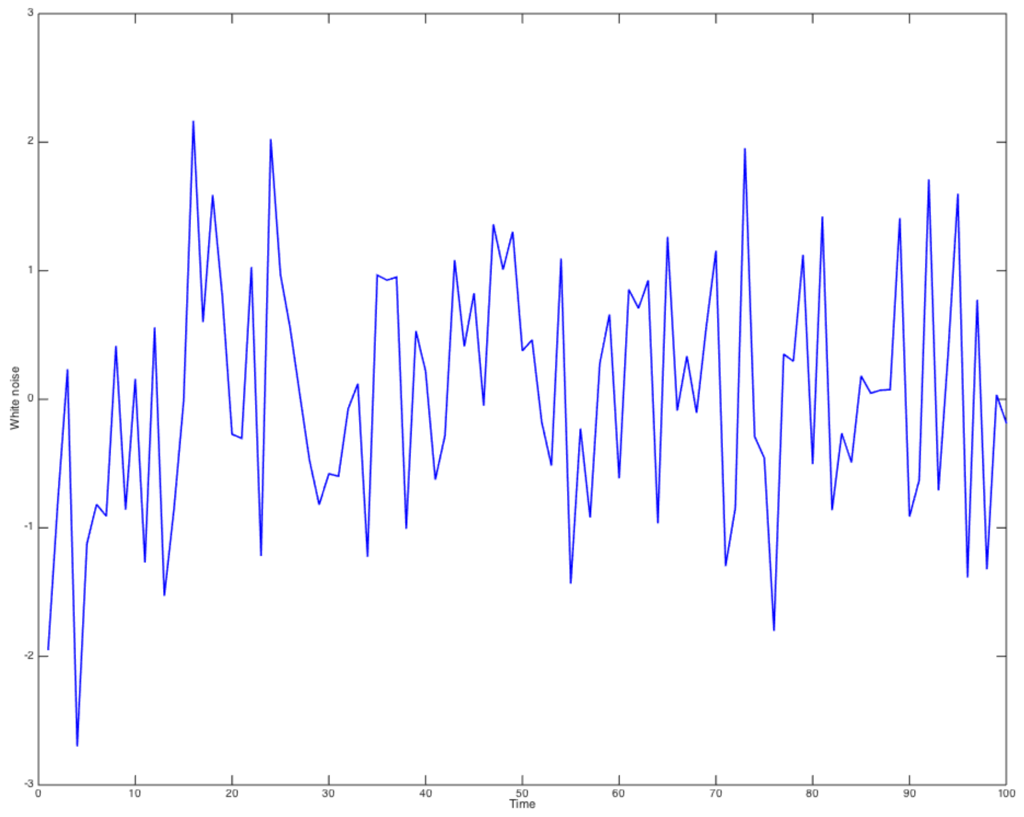
7.1 Example: Plotting a white noise process

```
% Seed the random number generator. For reproducability
rng(21);
% Draw iid normal random variables
xV = randn([100,1]);

% Line plot
%         'b-' specifies a blue solid line
%         fh is a file handle. It can be used to set plot options
fh = plot(xV, 'b-');

% Formatting
xlabel('Time')
ylabel('White noise')
```

The result: a poorly formatted figure (saved here by hand as `png`):



figWhiteNoise

7.2 Saving Plots

Now for the bad news: Generating reasonably formatted plots in Matlab is surprisingly hard.

The only reasonable way of doing so uses Export_fig.

Even then it is complicated. . .

CHAPTER 8

Numerical Optimization

The goal: minimize \(f(x)\) subject to \(l \le x \le u\) where

- \(x\) is a vector
- \(l, u\) are lower and upper bounds
- \(f(x)\) returns a scalar

Two basic approaches: use gradients or gradient-free.

Trade-offs:

- when gradient based algorithms work, they are much faster than gradient-free ones
- gradient-free algorithms tend to be more robust / more likely to find the minimum

8.1 Gradient-Free Algorithms

The leading candidate is Nelder-Mead.

The idea can be visualized for the 2D case (\(x\) is length 2). Start from a triangle and try to stretch it or flip it over until you find a point that is better than any one previously known. Repeat until you can't make any more progress.

This simple algorithm is surprisingly robust. If speed is not a major issue, this is a good starting point (see guvGuide).

In matlab, the algorithm is fminsearch.

8.2 Gradient-Based Algorithms

The idea: evaluate enough points to build an approximation of the function to be optimized around the current point (typically quadratic). Then try to find a better point around the minimum implied by the approximate function.

In Matlab, one algorithm is fmincon.

Caveats:

- The vast majority of gradient based algorithms assume that the objective function can be solved to very high precision. This is rarely the case in economic problems.
- If the objective function is not continuous in \(x\), gradient based algorithms tend to have problems. They try to construct function approximations using very small step sizes. This is an issue when models are solved using simulated individual histories, especially with discrete choice.
- imfil is one of the few (Matlab) solvers that can handle noisy problems.

8.3 Problems

There is no way to guarantee that a solver finds a **global** minimum. Especially gradient based algorithms like to get stuck at local minima. The only way of mitigating the problem: run the solver from many starting points (expensive).

8.4 Solver Collections

Matlab's own solvers (in the Optimization toolbox) are not always state of the art. It is worth experimenting with alternatives.

For Matlab, a good one is NLOpt.

A list of solvers is here and here.

8.5 Tips and Tricks

Before passing your arguments into a solver, run them through a transformation that makes them lie in a fixed interval (e.g. \([1,2]\)). Solvers like well-scaled guesses. The transformation makes it easier for them.

CHAPTER 9

Matlab Exercises

Try the exercises at Quantitative Economics.

Some of the code for these exercises is in the `examples` folder in my github repo.

Some of my general purpose reusable code is posted in the `shared` folder in my github repo.

9.1 Read the Matlab documentation on the following subjects:

- Data types: matrices, strings, structures.
- Indexing to access matrix elements.
- Operators: `*, .*, /, ./`
- String handling: sprintf
- Loops: while, for.
- Functions: input/output arguments, global/local variables.
- Optimization: fzero, fsolve, optimset.
- Graphics: plot.

9.2 Getting Started

1. Compute the mean and standard deviation for weighted data

   Input: data and weights (matrices)

   Output: mean and standard deviation for each column

   My solution

2. Compute the cdf for weighted data.

   Input: data and weights

   Output: cumulative percentiles and their values

   My solution

3. Compute points at the percentile distribution for weighted data (e.g. median)

   My solution

9.3 Root finding

Write a Matlab program that numerically finds the solution of

\(f\left(x,a\right) =x^{2}-a=0 \) for \(x\geq 0 \)

- Write the deviation function \(f(x) \)
- Plot \(f(x) \) for \(x \in [0,4] \) .
- Find the solution for \(a = 4 \) using `fzero`.
- Find the solution for \(a = 4 \) using `fsolve`.

Try other algorithms, such as `fminsearch` for fun.

9.4 Growth Model

Consider a growth model given by the Bellman equation

\(V(k) = \max u(c) + \beta V(f(k) - c) \)

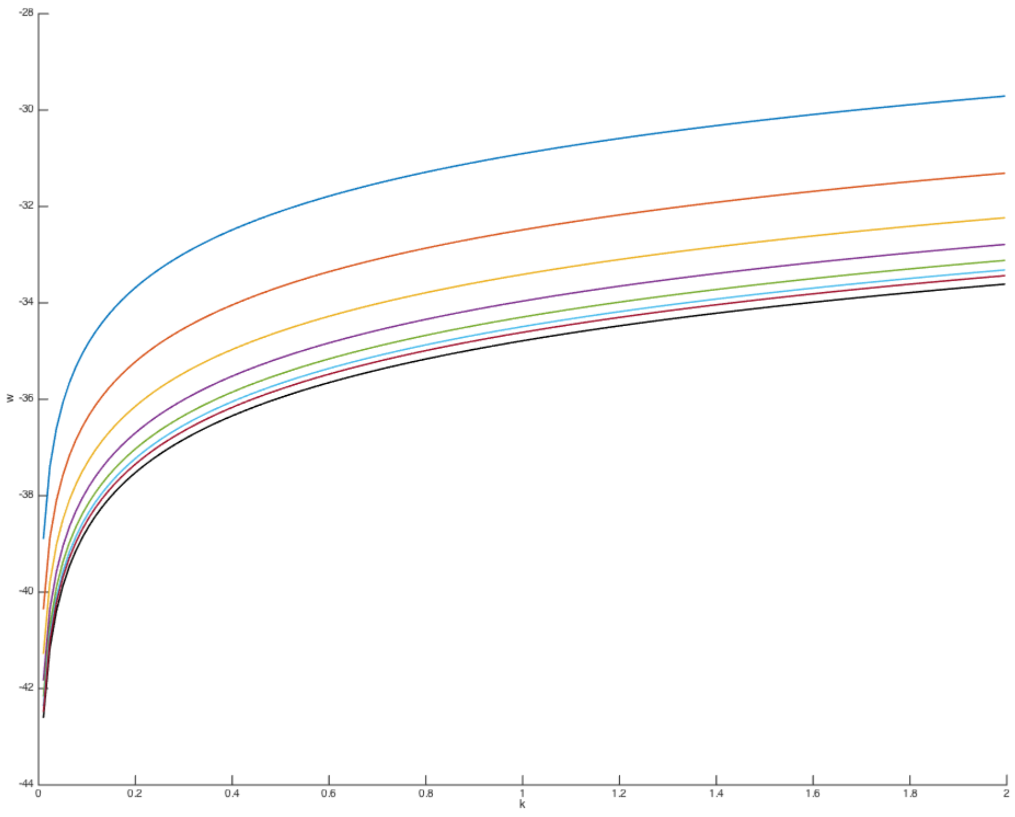
with \(f(k) = k^\alpha\).

Solve the growth model by value function iteration.

Steps:

1. Set up a grid for \(k \).
2. Start from an arbitrary guess \(V_{0}(k) \).
3. Iterate over the Bellman operator until \(V \) converges.

Along the way, plot the value functions to show how they converge to the true one. Like this:



GrowthValueIter

See QuantEcon and my Matlab solution.

9.5 Useful Exercises From QuantEcon

You should also write test routines for each function.

1. Draw discrete random numbers

   Given: probability of each state

   My solution

2. Simulate a Markov Chain

   See QuantEcon for the setup and Julia code.

   Also compute the stationary distribution

   My solution

   Also try their Exercise 1 (illustrate the central limit theorem).

3. Approximate an AR1 with a Markov chain from QuantEcon.

   Do this at home by simply "translating" the Julia code into Matlab.

   You will see that the syntax is very similar.

My solution

4. Simulate random variables to illustrate the Law of Large Numbers and the Central Limit Theorem.

   See QuantEcon for the setup and Julia code.

   My solution

CHAPTER 10

Object Oriented Programming (OOP)

10.1 The Idea

OOP takes structured programming to the next level. Structured programming encapsulates local data in a function. The user does not need to know anything about the function other than the interface (inputs and outputs).

OOP recognizes that some groups of functions "hang together" because they operate on the same object. One idea is to group these functions together.

The second idea is that certain persistent data "belong to" an object. They should only be manipulated by functions that also "belong to" the object.

OOP therefore bundles data (called `properties`) and functions (called `methods`) together.

10.1.1 Example: Utility function

\(u(c,l) = c ^ (1-\sigma) / (1-\sigma) + \phi \log(l)\)

Persistent data include: parameters (\(\sigma, \phi\)).

Methods include: compute \(u_{c}, u(c,l)\), inverse marginal utility, indifference curves.

10.2 Benefits

There is nothing that OOP can do that could not be done without OOP. The benefits lie in code organization.

The programmer sees all methods that operate on the object in one place. That makes it easier to

- test the code
- modify the code
- ensure consistency

Since all code is in one place, it is easy to swap out.

Imagine you want to compute a model with different utility functions. With OOP, all you need to do is swap out the utility function object. Ideally, the other code remains unchanged.

10.3 Examples From My Library

CRRA utility

CES production function

Enum data type.

10.4 References

Matlab documentation on object oriented programming.

CHAPTER 11

Organizing Code for a Complex Model

Note (2019-July): An updated, cleaner version of this in `Julia` can be found in my `Julia github` repo (module `modelLH`).

11.1 Goals

1. Be able to solve many different versions of the model. Changes to the model, such as replacing a utility function, should have no effects on most of the code.

11.2 Setting Parameters

The problem: one typically solves several versions of a model. How to ensure that all the right fixed / calibrated parameters are set for each version?

Example: A model is solved with different production functions, say Cobb-Douglas or CES.

Here is a possible workflow:

1. Define a `ModelObject` class. It has no properties. It has methods related to setting parameters that are common to all model objects.
    1. Methods:
        1. `set_calibrated_params`: copy calibrated parameters from a structure into the object.
        2. `set_default_params`: set all potentially calibrated parameters to their default values (for testing)
2. Define a class for each **abstract model object** (naturally a sub-class of `ModelObject`). Example: `UtilityFunction`.
3. Define a class for each **specific** model object. Example: `UtilityLog` (naturally a sub-class for the abstract `UtilityFunction`).
    1. Properties:

        1. all fixed and calibrated parameters
        2. switches that govern its behavior, such as: which parameters are calibrated?
    2. Methods:
        1. `calibrated_params`: returns default values and bounds for potentially calibrated parameters.
        2. `set_switches`: sets switches that are not known when the object is constructed.
        3. Methods that are specific to the object (e.g. compute marginal utility)
4. Write a **test** function for each abstract object and run it on each specific object.
    1. All sub-classes of `UtilityFunction` must have the same methods, so they can be tested using the same code.
5. Define a class that defines the model.
    1. It instantiates one object for each model object.
    2. For each model version, some defaults are overridden. That either means: a different object is constructed (e.g., a different utility function); or switches in the object are set.
    3. Run `set_switches` on all objects (now that we know all model settings). At this point, the definition of the entire model is neatly contained in this object.
6. At the start of the calibration loop: 4. Make a list of all calibrated parameters. `pvectorLH` does that.
    1. Make a vector of guesses for the calibrated parameters (`pvectorLH.guess_make`).
    2. Feed it to the optimizer.
7. In the calibration loop:
    1. Recover the parameters from the guesses (`pvectorLH.guess_extract`).
    2. Copy the calibrated parameters into the model objects (`set_calibrated_params`).
    3. Solve the model… 4. Note that nothing in the mode code depends on which objects make up the model (e.g. log or CES utility).

It is now trivial to replace a model object, such as a utility function. Simply replace the line in the model definition that calls the constructor for `UtilityLog` with, say, `UtilityCES`. The code automatically updates which parameters must be calibrated and which methods are called when utility needs to be computed.

11.2.1 Alternative Approach

1. Assign each model version a number.
2. Write a file that sets "named" choices for each model version.
    1. E.g.: `cS.prodFct = 'CES'`
3. For each named choice, define an `object` with the properties / methods:
    1. provide a list of all potentially calibrated parameters (in my code, these would be `pstructLH` objects)
    2. provide methods that are needed to solve the model (e.g.: return marginal products etc)
4. Collect all potentially calibrated parameters into a `pvectorLH` object
    1. This can make a vector of guesses that can be passed into optimization functions (such as `fmincon`)
    2. It can take the `guess` vector and transform it into the parameters to be calibrated

    3. For model objects that can be swapped out, simply question the object to get a list of all parameters it needs.

5. In the model solution code, you don’t need to know what kind of production function is used. Key is that all of them have the same methods.

Now, to solve the model with a different production function, simply call the constructor for the appropriate object. The rest is automatic.

CHAPTER 12

Indices and tables

- genindex
- search