Handout: Good programming
607: Applied Macroeconometrics
Jon Faust

1 Good laboratory habits

Empirical work shares many (but unfortunately too few) attributes with laboratory experiments in physical sciences. Foremost, one hopes that it is replicable. Computer analysis of a fixed data set should, of course, be exactly replicable. Unfortunately, much work is very difficult to replicate. Good laboratory habits can guarantee replicability, however.

This document has a few tips that many of us have learned through painful experience are useful. You too will probably experience this at some point: you run a bunch of stuff on the computer and find a really exciting result. And you never again can figure out where it came from.

1.1 An ideal

Suppose you have a completed paper using some computer analysis. From the standpoint of replicable and auditable science, the ideal would be that you have a program or programs that if run with no other human intervention except starting the program will reproduce all your results.

Then you know you can replicate your results. If a question arises as to why you got some result, you will be able to trace it exactly (at least in principle).

Mark Watson regularly provides replication files on the web. I don't do that well, but I try.

1.2 Some basics

At the beginning of your program be sure that there is nothing in Matlab memory that (due to some programming error of yours) may interact with

your stuff.

The top of your program should contain:

```
clear;
clc;
close all;
```

These commands, particularly, the `clc` command clears most things in Matlab's memory. The other commands clear the screen and close any open files.

1.3 Random numbers

If you are using a random number generator, there is a certain *quasi-stochastic* element to your results. Unless you take steps, exact replication will be impossible. Of course, random number generators are in fact deterministic. If you want to exactly replicate a Monte Carlo, you simply need to know the state of the random number generator at the outset. Many random number generators have a single state variable that is called the seed. Thus, if you are using a Monte Carlo, you should seed the random number generator at the outset:

```
randn('seed',123);
rand('seed',123);
```

2 Programming Style: Modular programming

Good laboratory habits go beyond replicability. If your program has a bug and doesn't do what you think it does, having a replication file is useful in *ex post* diagnosis. Until the error is discoverd, however, it doesn't keep lyou from publishing or going to a policymaker with bogus stuff.

Through the years, a few good practices have proven useful in helping avoid problems.

- Use meaningful variable names. The natural language interpretation of your variable names should give a clue as to what the variable is.

- Put a comment in the code whenever you do something you think might puzzle you or some other reader at some point in the future.

  For example, suppose I want to know the probability that the sample mean of 200 iid standard normal random variables is greater than 2. The following will provide a good estimate:

  ```
  mean(mean(randn(200,10000))>2)
  ```

  It might take a reader a bit of time to confirm what this is doing.

- If the computational efficiency is not greatly different or the differences in efficiency don't matter, coding things in a readable way is better than coding things in the most computationally efficient way.

  For example, you all know how to run the Monte Carlo in the previous example using a loop to do the 10,000 Monte Carlo replications. This would be much clearer.

3 Modular programming

- The big picture: Think of the code like you think of standard written language. We break up writing into paragraphs where each paragraph, loosely speaking, develops a single idea. Computer code should mirror this with each paragraph being a separate routine (in Matlab, a separate `function`).

- A concrete advantage: Often in code, we re-use certain "paragraphs" again and again. By putting these in a separate routine, we can simply call the same code again and again, rather than repeating it.

  For example, if you need to generate, say, Gaussian AR(1) data in many different contexts, having a standard function to do this is very useful.

- A bigger advantage comes in modifying and debugging. Suppose you have written a function to generate a Gaussian AR(1) sample. You have used this for the last several problem sets and you discover a bug (or you decide you'd like to see what happens if the shocks are uniform instead of Gaussian).

You only need to modify the code in one place to be back in business. Then all the programs for problem sets that call this function can be re-run without modification.

- A decent rule of thumb, with many exceptions: If you find yourself repeating a block of code, you should have used a function. (One main exception is when the block of code is entirely trivial.)
- In old versions of Matlab, each function had to be in a separate M-file. More recently, functions that are only to be called by code within a single file can all be included in that file.
- Many of our assignments will be so simple that the advantages of modular programming will be minor, but please take a crack at using good modular programming in the future.