# RA Manual: Notes on Writing Code

Matthew Gentzkow*
Jesse M. Shapiro
*Chicago Booth*

June 25, 2012

## 1 Introduction

Every step of every research project we do is written in code, from raw data to final paper. Doing research is therefore writing software.

Over time, people who write software for a living have learned a lot about how to write it well. We follow their lead. We aim to write code that would pass muster if we worked at Google or Microsoft.

Economists sometimes write code that is like stream-of-consciousness: a more or less random series of steps that happen to produce the right result. A good way to generate this kind of code is to use Stata interactively for an hour and then copy and paste the list of commands into a text editor. This code will do what it is supposed to do. But it will be very difficult for someone other than the person who produced it–or even for that same person after a day or two–to read and understand it. It will be virtually impossible to modify or extend it. And if anything about the environment changes–if you try to run it on a different computer, say, or change the name of an input file–it will break.

It is obvious that Google Maps or Microsoft Word could not be written this way. The code for these programs must be written so that many people over many years can read and understand it. It must have a logical structure that makes it easy to fix, modify, and extend, and allows people to take pieces developed for one problem and apply them to another. It must be robust enough to remain viable in a constantly changing environment. And it must be efficient. Although our projects are orders of magnitude smaller, our code needs to fulfill these same requirements.

This document lays out some broad principles we should all follow. Both this document and our understanding of what makes good code are constantly evolving. Few of us have formal training and we're learning as we go. If you look at code we've written in the past, you'll see that most of it fails some of the criteria below and much of it fails most of the criteria. We encourage you to invest in reading more broadly about software craftsmanship, looking critically at your own code and that of your colleagues, and suggesting improvements or additions to the principles below.

Apply these principles to every piece of code you check in without exception. It is tempting to check in bad code first and plan to clean it up later. But we are convinced that this way of working is inefficient. How you write your "first drafts" is up to you, but you should invest the time to make your code clean sooner rather than later.

You should also take the time to improve code you are modifying or extending even if you did not write the code yourself. A core of good code plus a long series of edits and accretions equals bad code. The problem is that the logical structure that made sense for the program when it was small no longer makes sense as it grows. It is critical that we regularly look at the program as a whole and improve the logical structure through reorganization and abstraction. Programmers call this "refactoring." Even if your immediate task only requires modifying a small part of a program, we encourage you to take the time to improve the program more broadly. At a minimum, you should guarantee that the code quality of the program overall is at least as good as it was when you started.

*These notes are prepared for internal use by our research assistants. Please do not excerpt or borrow without attribution.

As with other writing there is a certain amount of subjectivity in judging what is good code. Choices involve tradeoffs. The right choice in one language may be the wrong choice in another. Use your judgment, ask lots of questions, and be on the lookout for issues that you face which would be productive for us to discuss as a group.

# 2 Code should be logical

### Despise redundancy

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
> *The Pragmatic Programmer*

If you remember nothing else from this code manual, remember this: *redundancy is evil*.

Amateur code is plagued by redundancy. Blocks of code are copied and pasted. Constant values are hard coded and repeated multiple times. A new script is created by starting with an old one, modifying a few lines, and then hitting "save as." As a result, the same piece of knowledge (an instruction to the computer, a constant value, an algorithm, an algebraic formula) is represented multiple times.

Why is this evil? Most obviously, it means that someone who wants to change some piece of knowledge must change it in multiple places. If it can be represented in multiple places, the coder is unlikely to know where to find it so they must go searching throughout the program. Often, a piece of knowledge is changed in one place but not another leading to nasty bugs. The logical structure of the code is obscured because it's not obvious to the reader whether the knowledge represented in one place is meant to be the same as that in another place, or if not how they are meant to be different. (There is little worse than having to put two large blocks of code side by side and try to diff them by hand.) In a moderately complex program, one reaches a point where understanding or modifying it is virtually impossible and it's better to throw it out and rewrite it from scratch.

*Never copy and paste code.* Or, if never is a bit too strong, do this rarely, reluctantly, when you are sure there is no alternative, only for small snippets of code, and only when you will make significant changes after pasting.

When you are tempted to copy and paste, step back and ask yourself: What is the piece of knowledge I'm duplicating? How can I restructure the logic of the code so it has a single authoritative representation. Maybe you should use a loop. Maybe you should abstract some set of steps as a function. Maybe you should store some options, input parameters, or metadata separately.

### Separate functional code and metadata

This is a corollary to the previous point. Not only should every piece of knowledge have a single, unambiguous, authoritative representation, but that representation should be where the reader would expect to find it.

Code should be used to represent algorithms. Metadata (either set off separately within a script or stored in a separate file) should be used to represent the quantities that define specific instances of those algorithms–parameters, options, physical quantities, and so on.

Never hard code real numbers. E.g.,

```
effect_size = coefficient * population / 3
height = measurement / 7.12
```

What is 3? What is 7.12? The reader of the code has to guess. If someone realizes later that 3 should be 4, they have to go hunting through the code to find it. As already noted, hard coded values often are referred to multiple times, introducing redundancy. It is much better to write

```
num_cities = 3
height_conversion_factor = 7.12
effect_size = coefficient * population / num_cities
height = measurement / height_conversion_factor
```

The key point is, 3 is not a good name for the number of cities. It's ambiguous—it could refer to 3 cities or 3 apples or 3 kilograms. And it will end up pointing to the wrong piece of information if the number of cities changes. The name `num_cities` is better because it is both descriptive and appropriately abstract.

Similarly, never hard code references to arrays. It is common to see code like this:

```
parameters[27] = alphahat
parameters[28] = betahat
```

If the parameter vector changes, someone will have to go through the code and change all the 28's to 29's. Someone who wants to refer to the value of alpha later has to keep track of where in the parameter vector it is located. This is a recipe for buggy code and frustration. It is better to write

```
index_alpha = 27
index_beta = 28
parameters[index_alpha] = alphahat
parameters[index_beta] = betahat
```

or to make parameters a struct object so you can write

```
parameters.alpha = alphahat
parameters.beta = betahat
```

If the same constants are used in multiple scripts, they should be stored in a separate text file.

**Abstract if and only if it improves the code you are writing right now**

Abstraction means changing the boundary between algorithm and inputs/metadata. The code

```
egen total_income = total(income), by(state)
egen total_obs = total(1), by(state)
gen y = (total_income - income) / (total_obs - 1)
```

is an algorithm that computes leave-out mean income by state. One could instead abstract this algorithm so it could work for counties, cities, etc. by defining a function that allows the user to say

```
leave_out_mean_income, gen(y) byvar(state)
```

or abstract it further so it could also work for variables other than income

```
leave_out_mean, gen(y) byvar(state) meanvar(income)
```

If you have redundant code, you should abstract. If your program uses a leave-out-mean by state and also a leave-out mean by county, it's time to write the leave_out_mean function.

But what if your code currently only computes the leave-out mean once, for income by state? Abstraction may still make the logical structure of the code clearer. In the first version of the leave-out mean code above, the reader has to spend a few seconds staring at the code to realize what it does. Many readers don't care about the details of how the leave-out mean is computed. The generalized code is clearer, and writing the generalized code doesn't take much more time than writing the specific code.

Abstraction is not always a net gain, however. One could in principle generalize the function above to compute many different kinds of means so the user could write

```
compute_mean, gen(y) byvar(state) meanvar(income) ///
  meantype(leave_out)
```

Chances are, this will take a lot of time and will not do all that much to improve the readability of the code. Pushing the abstraction too far can actually make code less clear:

```
compute_something, gen(y) byvar(state) inputvar(income) ///
  comptype(leave_out_mean)
```

The other advantage of abstraction is that if someone needs to compute another leave-out mean later, they have a tool at the ready. However, as a general rule it's best not to invest time in generalizing code if there is no immediate gain. It's tempting to add all kinds of functionality to programs that might be useful someday. But if it's not useful right now, there's a good chance it won't ever be used in which case the generalization is a waste of time.

### Write functions and files with a clear purpose

A reader should be able to look at the name of a function or file and guess what it does. This is good:

```
set memory 2g
set_path
define_rhs_variables
define_lhs_variables
```

This is bad:

```
do_preliminaries_part_1
do_preliminaries_part_2
```

Files called "figures.do" and "tables.do" are better than files called "analysis.do" and "more_analysis.do.

This requires that objects not only have descriptive names, as discussed above, but that they be defined in such a way that each object has a clear, intuitive purpose.

This is one of the main reasons why you should keep functions and files short. It's unlikely that the contents of a 500-line script will be intuitive.

Often, it's not obvious how to take a long and unwieldy script and make it short and purposeful. When this is difficult, it's usually a sign that you need to step back and think about the logical structure of the program as a whole.

### Use the right data structures

One of the main features of modern programming languages (object-oriented or not) is that they allow users to define a rich array of data structures.

Data structures affect efficiency. Choosing between arrays, stacks, hash tables, binary trees, and so forth is a key part of algorithm design. These considerations are second order in much of what we do, however, because most of the statistical analysis we do naturally operates on arrays.

Data structures also affect clarity. This

```
param = estimate_model(y1, z, x1, x2, x3, clist, plist, alg, ///
  x4, x5, verbosity)
```

is much harder to make sense of than this

```
param = estimate_model(lhs_var, rhs_vars, options)
```

Likewise, this

```
[div_est,div_all_est,multi_est] = compute_diversity_statistics...
    (Xsim,Ysim_est,input.maxnum,eq_est,simind);
[div_rand,div_all_rand,multi_rand] = compute_diversity_statistics...
    (Xsim,Ysim_rand,input.maxnum,eq_rand,simind);
[div_repshare50,div_all_repshare50,multi_repshare50] =...
     commpute_diversity_statistics...
    (Xsim,Ysim_repshare50,input.maxnum,eq_repshare50,simind);
```

is harder to read than this

```
dstat.est = compute_diversity_statistics...
    (Xsim,Ysim_est,input.maxnum,eq_est,simind);
dstat.rand = compute_diversity_statistics...
    (Xsim,Ysim_rand,input.maxnum,eq_rand,simind);
dstat.repshare50 = compute_diversity_statistics...
    (Xsim,Ysim_repshare50,input.maxnum,eq_repshare50,simind);
```

especially as the number of parallel calls gets large.

Another example of how using the right data structure greatly improves clarity is when creating tables in Matlab. This

```
tabletext = ...
[div_est, div_all_ex_est, div_all_est;...
 div_coll_p, div_all_ex_coll_p, div_all_coll_p;...
 div_rand, div_all_ex_rand, div_all_rand;...
 div_coll_ad, div_all_ex_coll_ad, div_all_coll_ad;...
 div_joint_op, div_all_ex_joint_op, div_all_joint_op;...
 div_repshare50, div_all_ex_repshare50, div_all_repshare50;...
 div_coll_rd, div_all_ex_coll_rd, div_all_coll_rd;...
 ];
```

is not as clear as, and much harder to maintain than, this

```
rows = {'est','coll_p','rand','coll_ad','joint_op','repshare50','coll_rd'};
cols = {'div','div_all_ex','div_all'};
tabletext = zeros(length(rows),length(cols));
for i = 1:length(rows)
    for j = 1:length(cols)
        tabletext(i,j) = colstat{j}.(rows{i}).(cols{j});
    end
end
```

### Make your functions shy

A reader should know exactly which variables a function uses as inputs and which variables it can potentially change.

Most functions should explicitly declare their inputs and outputs and should only operate on local variables. Make the set of inputs and outputs as small as possible; the functions should be reluctant to touch any more data than they need to. For example, if a function only depends on the parameter beta, pass it only beta and not the entire parameter vector.

Use global variables rarely if ever.

Stata poses a special problem because the data in memory is by definition a global variable. From the following code

```
use x y z using autodata.xls, clear
prepare_data
update_variables
merge_new_data
regress productivity y_average z_average
```

there is no way to tell what are the inputs and outputs to the `prepare_data`, `update_variables`, and `merge_new_data` functions and no way to tell where the `productivity`, `y_average`, and `z_average` variables came from. When possible, write functions that only operate on variables that are explicitly passed to the function and do not otherwise touch the data in memory; if a function creates new variable(s), the names

of these variables should typically be specified as part of the function call. This should always be true of ado files; for programs only defined and used within a given .do file, it's a matter of judgment. When the number of variables needed from a dataset is not too large, list the variables explicitly in the `use` command. Always list variables explicitly with the `keep()` option when using the `mmerge` command.

**Use overriding methods instead of switches where appropriate**

For some projects we use object-oriented programming (OOP). In this paradigm we define classes of objects which have associated properties and methods. Instead of, say, creating a several seperate variables and defining a function to operate on them, we can define an object class which has the variables as properties and the function as a method. We can also define subclasses, which inherit all the properties and methods of the superclass except those explicity changed in the subclass file.

There are several reasons why we might want to use OOP. One reason is that creating classes and subclasses is a much clearer way of coding a model with several different variants than writing a program with a lot of if/else switches.

For example, suppose we have a class Model with a method DrawRandVar which draws a random variable from a certain distribution. Suppose that usually we want the variable to be drawn from the standard normal distribution, but we also want to look at results which require it to be drawn from a uniform distribution. We could code this as an option switch:

```
function draw = DrawRandVar(obj, option)
    if option == 'uniform'
        draw = rand;
    else
        draw = randn;
    end
end
```

However, we could also create a subclass UniformModel of the class Model which contains the overriding method:

```
function draw = DrawRandVar(obj)
    draw = rand;
end
```

If you are working with a model coded in OOP, you should think carefully about whether to include switches or write subclasses with overriding methods. Code with lots of switches can become harder to follow and parse, and creates a false sense of generality – if we add an option switch to this method we may need to add it to others to make the model consistent, this would be easier to acheive using a subclass. Furthermore, switches are checked every time a method is called, unnecessarily increasing the computational cost of running the method.

Note that when writing subclasses, you should be careful to keep superclass methods intuitive and 'shy'. For example, if a superclass method takes an input which is redundant in the superclass and all but one of several subclasses, it may be better to refactor to avoid this redundancy. Having arguments which are reduntant in the superclass makes it more difficult for another user to understand what the superclass method does. On the other hand, if subclass methods only modify one or two lines of a long superclass method, you should think about abstracting out these lines as a separate method in the superclass. This new method can then be modified in the subclass, avoiding the redundancy introduced by copying a long method. In general, you should think about structuring superclass methods and their corresponding subclass methods in the most intuitive way possible.

# 3 Code should be readable

### Keep it short

No line of code should be more than 100 characters long. All languages we work in allow you to break a logical line across multiple lines on the page (e.g, using "///" in Stata or "..." in Matlab). You may want to set your editor to show a "margin" at 100 characters.

Long scripts should be factored into smaller functions. Individual functions should not normally be more than 40 lines long. (In the past, we did not regularly apply this rule to Stata code. You will therefore see legacy .do files that are long series of commands broken up with comments rather than separated into functions. For new code, we expect you to use the functional style.)

### Order your functions for linear reading

A reader should be able to read your code from top to bottom without skipping around. Subfunctions should therefore appear immediately after the higher level functions that call them. For example, suppose you have a top-level function called main which calls func_I and func_II, which in turn call sub-functions func_1 and func_2 respectively. The script should look like

```
function main
    [code that calls func_I and func_II]
function func_I
    [code that calls func_1]
function func_1
    [...]
function func_II
    [code that calls func_2]
function func_2
    [...]
```

The alternative ordering would be by function hierarchy: main, func_I, func_II, func_1, func_2. This arrangement makes that levels at which the different functions live clearer, but is harder to read linearly. It is possible that a script is so long, and has so many levels of functional hierarchy, that nobody would ever read it linearly. In this case, one could argue that arranging by level makes more sense. But the correct response in this case is usually to break the script into multiple files and preserver the property that each individual file can be read beginning-to-end.

### Choose descriptive names

Good names replace comments and make code self documenting. It's better to write

```
gen income_percap = income / population
```

than

```
* Define income per capita variable
gen ipc = income / population
```

It's better to write

```
function [name] = lookup_name(id_number)
```

than

```
%
% lname: This function takes an id_number as input and
%     returns the name of the corresponding person.
%
function [output] = lname(input)
```

It's better to have a file called "appendix_tables.do" with no header comment than a file called "atab.do" with a header comment explaining that the file produces tables for the appendix.

By default, names for variables, functions, files, etc. should consist of complete words. Only use abbreviations where you are confident that a reader not familiar with your code would understand them and that there is no ambiguity. Most economists would understand that "`income_percap`" means income per capita, so there is no need to write out `income_percapita`. But `income_pc` could mean a lot of different things depending on the context. Abbreviations like `st`, `cnty`, and `hhld` are fine because we use them throughout our code. But using `blk_income` to represent the income in a census block would be confusing.

Avoid having multiple objects whose names do not make clear how they are different: e.g., scripts called "state_level_analysis.do" and "state_level_analysisb.do" or variables called `x` and `xx`.

Names can be shorter or more abbreviated when the objects they represent are used frequently and/or very close to where they are defined. E.g., it is sometimes useful to define short names to use in algebraic calculations. This is hard to read:

```
log_coefficient = log((income_percap' * income_percap)^(-1) *///
  income_percap' * log_wage)
```

This is better:

```
X = income_percap
Y = log_wage
log_coefficient = log((X'*X)^(-1)*X'*Y)
```

**Use white space and indents to make code scannable**

Carriage returns should separate blocks of related commands. Major sections of code can be separated with two carriage returns.

Use spaces rather than tab characters to indent. Many editors automatically insert spaces when you press tab. Others allow this as an option.

Commands inside functions or loops should be indented four spaces.

Logical lines broken across multiple rows (e.g, using "///" in Stata or "..." in Matlab) should be indented two spaces and aligned vertically for clarity. E.g.,

```
var inset = {
  top: 10,
  right: 20,
  bottom: 15,
  left: 12
};
```

or

```
function [data, options] = format_raw_data(
  str dataset_name,
  int population,
  double density,
  struct options,
)
```

Equals signs, arithmetic operators, etc. should be padded with spaces. Function arguments should be separated with a space after the comma:

```
my_function(x, y, z)
```

rather than

```
my_function(x,y,z).
```

By default, use this form for braces defining blocks of code

```
for i = 1:5 {
    ...
}
```

rather than this form

```
for i = 1:5
{
    ...
}
```

or this form

```
for i = 1:5
{
    ...
    }
```

A large number of related commands that suggests a tabular format should be aligned vertically for easy scanning. E.g.,

```
cats                 = 10;
dogs                 = 4;
tyrannosaurus_rexes  = 1;
pigeons              = 2;
pigs                 = 8
dragons              = 25;
other_animals        = 5;
```

**Pay special attention to coding algebra**

Make sure that key calculations are clearly set off from the rest of the code. If you have a function called `demand()` with 15 lines of setup code, 1 line that actually computes the demand function, and 5 more lines of other code, that 1 line should be set off from the rest of the code so it is obvious to a reader scanning the document.

Often, it is a good idea to isolate algebraic calculations inside sub-functions. E.g.,

```
function [quantity] = demand(X, beta) {
    ...
    a bunch of commands to compute utilities and pop
    ...
    shares = logit_demand_function(utilities)
    quantity = shares * pop
}
function [shares] = logit_demand_function(u) {
    shares = exp(u) ./ repmat(sum(exp(u)), 3, 1)
}
```

This both makes the main function easier to read for someone who is not interested in the details of the computation and makes the code easy to modify if the user decides they want to replace logit demand with something else.

Break complicated algebraic calculations into pieces. Programming languages have no objection to definitions like

```
gen percap_gdp_real = ///
  (consumption + govt_expenditures + exports - imports - taxes) * ///
  10^6 / (price_index * pop_thousands * 1000)
```

or far longer ones. But a human may find it easier to parse the following:

```
gen gdp_millions_nominal = ///
  (consumption + govt_expenditures + exports - imports - taxes)
gen gdp_total_real = gdp_millions * 10^6 / price_index
gen pop_total = pop_thousands * 10^3
gen gdp_percap_real = gdp_total_real / pop_total
```

Complex calculations are better represented in mathematical notation than in code. For such calculations, write out the algebra in Lyx and save the document to the /docs/ folder adjacent to the script in question. Then add a comment in the code point the reader to the file name and revision number they should refer to.

**Make logical switches intuitive**

When coding switches, make sure that the conditions are intuitive. Often there is more than one way to express a logical condition. Choosing the most intuitive expression makes the logical meaning of the switch clear, and helps users parse the code quickly. In the following example using Matlab code, suppose x is a vector of 0s and 1s:

```
if max(x) == 0
    y = 0
end
```

This block of code is logically equivalent to:

```
if all(x == 0)
    y = 0
end
```

Both switches check whether the vector x contains only 0s. However, the first condition parses as "if the maximum entry of x if equal to 0", while the second parses as "if all entries of x are zero". The second test is better because it is logically identical to what we want to check, whereas the first test relies on the fact that all entries of x are greater than or equal to 0.

**Use enough comments and no more**

Eliminate redundant comments. There is no need for a comment that says "performs analysis" at the top of a file called analysis.do. There is no need for a comment that says "define log population" before the line

```
lpop = log(pop)
```

The purpose of each script should be clear. If the file name by itself does not make the purpose clear, there should be a comment at the top of the file to clarify.

The required inputs for every function should be clear. For example, a function may require that a certain input be a positive integer. If the name of the function and the name of the inputs does not make this clear, there should be a comment at the top of the file to clarify.

Comments should be used to explain code that might otherwise seem unintuitive to the user. For example, the function below translates the mean and standard deviation of a variable into parameters of an underlying lognormal distribution. A comment is in order so that the user knows what is going on:

```
program define simulate_spending
    syntax, gen(name) mean_spending(real) sd_spending(real)
    * using known results to translate moments of spending into
    * parameters of lognormal distribution
    * see http://en.wikipedia.org/wiki/Log-normal_distribution
    local sigma = sqrt(ln(1 + (‘sd_spending’/‘mean_spending’)^2))
    local mu = ln(‘mean_spending’) - 0.5 * (‘sigma’^2)
    display ‘sigma’
    display ‘mu’
    gen ‘gen’ = exp(rnormal(‘mu’, ‘sigma’))
end
```

In long scripts, comments can be used to make clear how the code is organized. In many languages, the modular structure of the code makes clear how it is organized. This is natural in Matlab, Java, etc. In Stata, we might write

```
* PROGRAMS
program main
    prepare_data
    run_regressions
    output_tables
end
program prepare_data
    X = format_x_vars
    Y = format_y_vars
    ...
end
program format_x_vars
    ...
end
program format_y_vars
    ...
end
program run_regressions
    ...
end
program output_tables
    ...
end
* EXECUTE
main
```

The reason for the separate “EXECUTE” portion at the bottom of the script (rather than simply executing “main” at the top of the script) is that Stata reads in programs in order of appearance. Writing the script in this way allows main to call functions that are written below main.

Note that much of our legacy Stata code does not use functions to separate blocks of code, and instead uses comments to give the code an outline form and make it easy to scan. Instead of the above, we would write

```
*****************************
* Prepare data
*****************************
* Format X variables
...
* Format Y variables
```

```
...
*****************************
* Run regressions
*****************************
 ...
*****************************
* Output tables
*****************************
 ...
```

If you include a comment as a header like this for one major block of code, you should include a similar header for every block of code at the same logical place in the hierarchy. This is a case where redundant comments are allowed. The comments are not there to provide information, but to make the code easy to scan. These guidelines only apply to modifying legacy code; new Stata code should use functions to separate blocks of code.

### Be consistent

There are many points of coding style that are mostly a matter of taste. E.g., sometimes people write variable names like `hhld_annual_income` and other times like `hhldAnnualIncome`. Although some people have strong feelings about which is better, we don't particularly care. What is important is that we all use consistent conventions. This is especially important within scripts: if you are editing a program in which all scripts use two-space indent you should use two-space indent too, even though that breaks our normal rule (or use grep to update the script to four-space indent).

On many of these points, we have not established clear conventions. When in doubt, use the convention you have seen most often in our code.

### Avoid commands that make code hard to read

Certain commands inherently harm readability, such as the "eval" command in Matlab. We definitely want to avoid code like the following:

```
for i = 1:length(cflist)
    eval(strcat('dstat.',cflist{i},'=...
        compute_diversity_statistics...
        (Xsim,Ysim_',cflist{i},',input.maxnum,eq_',cflist{i},',simind);'));
end
```

In cases that we have to use these commands, we should be especially careful and use them wisely. In the case of the "eval" command, readability drops significantly as the complexity of the statement being evaluated increases. Therefore, we should keep these statement short and try to only use "eval" on the right hand side of evaluation. We can refactor the code above as

```
for cf = cflist
    cf = cf{:};
    thisy = eval(['Ysim_',cf]);
    thiseq = eval(['eq_',cf]);
    dstat.cf = compute_diversity_statistics...
        (Xsim,thisy,input.maxnum,thiseq,simind);
end
```

Note that aside from restricting the use of "eval" command, we also used another trick on the fact that Matlab loops operate on lists and arrays. Therefore, we can write

```
for cf = cflist
    ...
end
```

Rather than

```
for i = 1:length(cflist)
    cf = cflist{i}
    ...
end
```

# 4 Code should be robust

**Check for errors**

Programming languages typically come with debugging tools and informative error handling. These are our first line of defense in error-handling. Often they are enough. For example if in Stata I write

```
gen str x = "hello"
gen y = x^2
```

then Stata will return:

```
type mismatch
r(109);
```

Typically, this will be enough information to alert the user to the fact that the code failed because the user attempted to square a string. Adding additional error-checking to check that x is not a string will add one or more lines of code with little gain in functionality.

However, there are some circumstances in which error-checking should be added to code.

Error-checking should be added for robustness. For example, if the wrong argument will cause your script to become stuck in an infinite loop, or call so much memory that it crashes your computer, or erase your hard drive, you should include code to ensure that the arguments satisfy sufficient conditions so that those outcomes will not occur. If those outcomes will occur given the arguments specified, the code should "throw an error" (exit and tell the user why).

Error-checking should be added to avoid unintentional behavior. For example, suppose function `multiplybytwo()` multiplies a number by 2 but is only written to handle positive reals. For negative reals it produces an incoherent value. Because a user might expect the function to work on any real, it would be a good idea to throw an error if the user supplies a negative real argument. (Of course, it would be even better not to write a function with such confusing behavior.)

Error-checking should be added to improve debugging efficiency. For example, suppose that function `norm()` requires a vector input. A user unintentionally passes the function a matrix and it takes 30 minutes to figure out what the problem is because the language's error message is not sufficiently informative. In this case, it might be worth a few minutes of coding time to add code to `norm()` that checks whether the input is a vector and returns an explicit error otherwise. Chances are that this will save a few more 30-minute debugging sessions in the future and be worth the time.

Note that there is an intrinsic tradeoff between time spent coding error-handling and time spent debugging. It is not efficient to code explicit handling of all conceivable errors. For example, it is probably not worth adding a special warning in the case where the user passes a string to `norm()`, because the user is unlikely to make that mistake in the first place.

Error checking code should be written so it is easy to read. It should be clearly separated from other code, either in a block at the top of a script or in a separate function. It should be automated whenever possible. If you find yourself writing a comment of the form

```
% Note that x must be a vector
```

ask yourself whether you can replace this with code that throws an error when `isvector(x)` is false. Code is more precise than comments, and it lets the language do the work.

Note that our usual warning against redundancy applies to error-checking. If you have a large program many of whose functions operate on a data matrix X, and there are various conditions that the data matrix must satisfy, write a function called `is_valid_data_matrix()` rather than repeating all the validation checks at the top of each function.

**Write tests**

Real programmers write "unit tests" for every piece of code they write. These scripts check that the piece of code does everything it is expected to do. For a demand function that returns quantity given price, for example, the unit test might confirm that several specific prices return the expected values, that the demand curve slopes down, and that the function properly handles zero, negative, or very large prices. For a program to compile, it must pass all the unit tests. Many bugs are thus caught automatically. A large program will often have as much testing code as program code.

Many people advocate writing unit tests *before* writing the associated program code.

Economists typically do not write unit tests, but they test their code anyway. They just do it manually. An economist who wrote a demand function would give it several trial values interactively to make sure it performed as expected. This is inefficient because writing the test would take no more time than testing manually, and it would eliminate the need to repeat the manual tests every time the code is updated. This is just a special case of the more general principle that any manual step that can be turned into code should be.

All of our derived directories include a test script (checkdta.do) that runs at the end of the make process and confirms that all output files have the expected properties, including unique and non-missing keys.

Every code library should have test scripts for all its functions. Every derived or analysis directory that includes functions that are generalized to be used outside of a single script (e.g., an .ado file called by multiple .do files) should include a test script for these functions. Including test scripts for functions used only within a given script is optional.

The key lines of tests scripts should be written in the form of assert commands. To make the code as reasonable as possible, and to allow the code to handle failed tests without breaking, we use testing libraries. In Stata, we use the commands testgood.ado and testbad.ado in /lib/stata/gslab_misc/. In Matlab, we use the xunit testing library stored at /lib/third_party/matlab_xunit. This is a Matlab version of the standard testing library used in Java, C++, etc.

Test scripts should be named simply "test" (i.e., test.do, test.m, etc.) if there is only one of them in a directory. If there are multiple test scripts, they should be placed in a sub-directory of the /code/ or library directory called /test/, and should have names that being with "test_" and indicate what objects they are testing (e.g., test_file1.do, test_file2.do, test_function1.m, test_function2.m).

Test scripts should be written up to the same standards as any other piece of code. The test script for a data file, for example, should replace comments or documentation explaining what properties the variables in the data file are expected to have.

Test scripts should not be contextual. That is, they should test the abstract properties of the function, rather than the current uses of the function. So, if your function's job is to compute the ratio of two scalars and you are currently using it to compute per capita GDP, the associated test script should pass abstract scalars to the function, not GDP and population. It might, for example, check that the function works even if the scalars are negative (which GDP and population never are). The goal of testing in this way is to ensure that the function can be trusted to perform as advertised even in contexts that have not yet arisen. And doing this allows the test script to encode the programmer's understanding of the properties of the function. In that way it serves as a form of documentation, with the added advantage that the documentation is sure to be correct if the test runs successfully.

Test scripts should be exhaustive of categories of cases, not of cases. It is not possible to test the "ratio" function on all possible scalars. A better bet is to test it on representative cases (positive reals, negative reals, complex numbers, etc.).

When you find a bug in a code library you have also found a bug in its unit tests, because they did not catch the bug. Hence, when practical, you should fix both the bug *and* the tests. That way, if the bug recurs, the tests will catch it next time.

# 5 Code should be efficient

### Profile slow code relentlessly

Languages like Matlab and R provide sophisticated profiling tools. For any script for which computation time is an issue (typically, anything that takes more than 10 seconds or so to run), you should profile frequently. The profiler often reveals simple changes that can dramatically increase the speed of the code.

Profiling Stata code is more difficult. Often, the sequential nature of the code means it is easy to see where it is spending time. When this is not the case, insert `timer` functions into the code to clarify which steps are slow.

Speed is the only justification we allow for violating the other coding principles we articulate above. Sometimes we are calling a function so many times that the tiny overhead cost of good, readable code structure imposes a big burden in terms of run-time. But these exceptions are rare and occur only in cases where computational costs are significant.

### Store "too much" output from slow code

There is an intrinsic tradeoff between storage and CPU time. We could version no intermediate data or results, and rerun the entire code pipeline for a project back to the /raw stage each time we change one of the tables. This would save space but would require a tremendous amount of computation time. Or, we could break up a project's code into hundreds of directories, each of which does one small thing, and store all the intermediate output along the way. This would let us make changes with little computation time but would use a lot of storage (and would likely make the pipeline harder to follow). Usually, we compromise, aggregating code into directories for conceptual reasons and to efficiently manage storage.

It is important to keep this tradeoff in mind when writing slow code. Within reason, you should err on the side of storing too much output when code takes a long time to run. For example, suppose you write a directory to estimate several specifications of a model. Estimation takes one hour. At the time you write the directory, you expect to need only one parameter from the model. Outputting only that parameter is a mistake. It will (likely) be trivial to store estimates of all the model parameters, and the benefits will be large in compute time if, later, you decide it would be better to report results on two or three of the model's parameters.

If estimation is instantaneous, re-estimating the model later to change output format will not be costly in terms of compute time. In such cases, concerns about clarity and conceptual boundaries of directories should take priority over concerns about CPU time.

### Separate slow code from fast code

Slow code that we plan to change rarely, such as code that estimates models, runs simulations, etc., should ideally be separated from fast code that we expected to change often, such as code that computes summary statistics, outputs tables, etc. That way, we can repeatedly modify the presentation of the output without having to rerun the same estimation step over and over.

Consider again a directory that estimates several specifications that take, together, an hour to run. The code that produces tables from those specifications will likely run in seconds. Therefore, it should be stored in a separate directory. We are likely to want to make many small changes to how we format the output, none of which will affect which specifications we want to run. It will not be efficient to have to repeatedly re-estimate the same model in order to change, say, the order of presentation of the parameters in the table.

Again, if instead estimation were very fast, it might be reasonable to include the script that produces tables inside the same directory as the script that estimates models. In such a case, the decision should be based on clarity, robustness, and the other principles articulated above, rather than on economizing CPU time.

# 6 Recommended Reading

Brooks, Frederick P. 1995. *The Mythical Man-Month.* Addison-Wesley: New York.

Hunt, Andrew and David Thomas. 2000. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley: New York.
Martin, Robert C. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall: New York.