THE AUSTRALIAN ECONOMIC REVIEW
THE UNIVERSITY OF MELBOURNE
MELBOURNE INSTITUTE
Check for updates
The Australian Economic Review, vol. 00, no. 00, pp. 1–12 DOI: 10.1111/1467-8462.12567

Perspective

Programming Style: Suggested Guidelines for Writing Code

Joe Hirschberg and Jenny Lye

Abstract

Since computer code has become elemental in current research it has become imperative that code should be written clearly and in good style. A program written in a clear, well-written style is easier to debug and is more useful to those who may want to replicate your work, extend it, speed it up or borrow from it. Clearly, written code is also a useful tool in the training of future researchers. In this paper, we suggest a set of style guidelines and we provide examples of how particular elements of code can be done using the statistical packages Stata, SAS and R. We also present recommendations for how to write code.

*Joe Hirschberg, Department of Economics, The University of Melbourne, Victoria 3010, Australia. Corresponding author: Clarke, email <j.hirschberg@unimelb.edu.au>.

1. Introduction

It is now common procedure for academic journals to require the submission of all the code and data used in the generation of the results published in the journal to ensure the replicability of the conclusions presented. This has become especially important with the proliferation of highly complex estimation procedures used in most current journal articles. Also, it is usually a requirement that data and computer programs be submitted as part of a submission of a research essay or dissertation. Although the policy may be quite specific as to the nature of the data, there are few details or guidelines for the code. For example, Vilhuber (2021) uses five pages to define the policy for *American Economic Association* publications but scant attention is given to the condition of the code submitted. There is more information at <https://www.aeaweb.org/journals/data/faq#package>. However, aside from assuming the program can be run to produce the results in the publication, there is no condition as to the 'readability' of the code to ensure that it does what it purports to do. Frequently, the code is written in a manner that is difficult to follow thus, although the code may generate the reported results it may not be performing the task described in the paper it supports. One way to avoid these pitfalls is to write code that is clearly written and uses good style.

Few programs work or perform the task perfectly the first time. This implies that they need to be debugged. A well-written program is easier to debug and maintain and is more useful to others who may want to replicate

your work, extend it, speed it up or borrow from it.[1] Good style helps to concentrate on parts of code elements that can be checked separately. Additionally, good code is a useful tool in the training of future researchers.

For most applications, the construction of code involves assembling data into a form that can be used by computation programs (for example, regressions) that have been written by professional programmers. Programs have been written to invert matrices, multiply matrices and find the eigenvalues of matrices. The algorithms for these computations have been the subject of intense investigation by many authors over the past 70+ years. Many statistical packages have optional elements that allow you to design your own special purpose estimation procedure, however, the focus in this paper is the construction of code to manipulate data transparently so that an analysis can be replicated, and assumptions used made clear. We will draw on statistical packages such as Stata, SAS and R as examples.

2. Style Guidelines

The history of computer coding is not long, given the widespread need for code grew up in the 1960s. The early forms of code still influence coding elements and conventions currently in use. In the 1960s code was primarily stored on 80-column punch cards and the primary language for writing computer code used in scientific applications, such as mathematics, engineering, economics and statistics, was FORTRAN (FoRmula TRANslation).[2] This language still exists and the conventions for the code have carried through to current computer languages of today many of which started as FORTRAN programs. Basic rules for the code included: that there could be leading blanks and that the variable mnemonics must start with a letter, be in capital letters and be no longer than eight characters.

Due to the need to conserve storage in early computers (they may have had as little as 124 k of memory) it was considered important to conserve space in the computer. This space was required for the operating system, the program and the data. Thus, early FORTRAN code was exemplified by very tight code with few extra spaces that had little 'wasted space' for comments and indications of the coding process. The basic form of storage for code and data was on cards where the standard 2-foot card box held 2,000 cards. This would have included the programs and the data, which could easily be dropped and put out of order. This meant that programs were often kept as short as possible.

Unfortunately, this brevity has continued to this day where many current programs have been written with almost no extra spaces and with very few and often short comments since they follow the program styles of these early codes. This has been the case even though a standard PC now has more than 10,000 times the space for code and data can be spooled in and out of memory from disks that hold terabytes of data. In part, this may be due to current programs that employ routines with code that may have been written over 50 years ago. In addition, some modern programmers seem to feel compelled to write in the same style.[3]

In the rest of this section, we outline a set of recommended style guidelines that are designed to aid in the understanding of the code, its maintenance and assist in reducing the likelihood of introducing errors.

2.1 Documentation Comments

Always include a preliminary comment line or lines containing the name of the program file (so you can find it later), the version number of your program, your name or initials, and the date the program was last modified. The preliminary comments should also include a description of what the program does and how it may differ from any related programs (for example, Cox 2005; Nagler 1995). For example, you may have written this program to perform an analysis done by another program but only on a subset of the data or using a modified specification or method of estimation (GMM vs 2SLS). In addition, if the code follows a specific article make sure to include complete references to the literature and the source information for any data that you are reading.

Programs can be of any length. The days of limited space when programmers prided themselves on their brevity and every line in a computer was an additional card in the deck they needed to carry around are over. It is better to be verbose than terse. This will be especially useful when you are trying to understand at a glance what you did or may even help spot mistakes. Comments should be used to explain what a piece of code does or what it is supposed to do. If you copy one program to make another many of these comments will stay the same from program to program. While it may be cumbersome to keep lots of comments it will pay off in the long run.

In both Stata and SAS, comments can start with a/* and end with a*/on as many lines as you want. There are alternatives. In both SAS and Stata, you can start a comment line with a *. However, in Stata, this will only comment for the line the * is on while in SAS it will only end when a semicolon is found. In Stata, you can also use the three slashes///after which all else on the line will be ignored. In R, any statement beginning with a # is a comment, however, R does not support multi-lined comments so if a comment runs over several lines there must be a # on every line of the comment. The colour for comments is different from the colour of commands in each of the programs to make them even more clear to the user. Examples of writing a comment for each of these programs are given in Figure 1.

2.2 Names/Mnemonics

Choose meaningful descriptive names for programs, sub-routines, variables and macros. Names chosen for your program should not conflict with anything that already exists, although the use of sequential names may help you find those files later and understand what they contain, (see also Cox 2005; Gentzkow and Shapiro 2014; Nagler 1995; Wassberg 2020). Names of variables are often referred to as mnemonics because they are intended to remind one of the underlying concept or variables.

A. Use standard abbreviations when possible. For example, use *avg* rather than *av* or *aver*, use *sd* instead of *stndv* or *standev*, use *max* instead of *mx* or *maxim*. Use trailing numbers when they are related or for different things. For example, *avg_male*, *avg_female*, or *max_1* and *max_2*.

B. Add long descriptive variable labels when possible. And the format of the responses in the case of a survey (that is, 1 = yes, 2 = no, 3 = don't know, etc.).

C. The case of the letters used may or may not be important to the program. Two variables can be called the same name but with different capitalization in both Stata and R as they are case-sensitive. However, SAS does not differentiate between upper- and lower-case characters in the code, thus you could alternate the case and it will not differentiate between them.

D. In Stata and SAS the names of variables cannot start with numbers or special characters (~!@#$%^&*()[]{}|\/?><,.) but they can start with an underscore (_). In R they can begin with an underscore (_) or a dot (.). However, if the variable name begins with a dot symbol it should not be a followed by a numeric digit.

E. Some words are reserved for names of special variables by many programs. In SAS the names: _n_, _name_, _type_. In Stata, some of the special names are _n, _N, _coef, and _cons. Use similar names and abbreviations for command options that are in common use for other cases. For example, use ii and jj for a subscript or loop counter instead of i and j because they are similar but will not be confused with an automatic function in SAS for the identity matrix and the matrix of all the same values. Note also that in all of the programs the names of functions such as log(), sin(), exp() are reserved.

Figure 1 An Example of Comment Writing.

SAS

```
/*      program parallelassay_pokies.sas

    SAS 9.4   Hirschberg & Lye  22/12/2020

    Program to perform Assay analysis to estimate the equivalent value for a
    dummy variable. Use the pokies data for smoking bans near the border to
    determine what change in the number of EGMs is equivalent to being on the
    border based on the data from Hirschberg J. & J. Lye (2010), 'The indirect
    impacts of smoking bans in gaming venues', Chapter 11, in Current Issues in
    Health Economics, Contributions to Economic Analysis, eds D. Slottje and
    R. Tchernis, Emerald Press, Sydney.
*/
```

Stata

```
/*
   program parallelassay_pokies.do

   version 16.1  Hirschberg & Lye  22/12/2020

   Program to perform Assay analysis to estimate the equivalent value for a
   dummy variable. Use the pokies data for smoking bans near the border to
   determine what change in the number of EGMs is equivalent to being on the
   border based on the data from Hirschberg J. & J. Lye (2010), 'The indirect
   impacts of smoking bans in gaming venues', Chapter 11, in Current Issues in
   Health  Economics,  Contributions to Economic Analysis, eds D. Slottje and
   R. Tchernis, Emerald Press, Sydney.

*/
```

R

```
#       program parallelassay_pokies.R
#
#       R version 3.6.3  Hirschberg & Lye  22/12/2020
#
#       Program to perform Assay analysis to estimate the equivalent value for a
#       dummy variable. Use the pokies data for smoking bans near the border to
#       determine what change in the number of EGMs is equivalent to being on the
#       border based on the data from Hirschberg J. & J. Lye (2010), 'The indirect
#       impacts of smoking bans in gaming venues', Chapter 11, in Current Issues in
#       Health Economics, Contributions to Economic Analysis, eds D. Slottje and
#       R. Tchernis, Emerald Press, Sydney.
#
```

2.3 Missing Values

Data on an observation(s) can sometimes be missing and recorded by using a missing value code such as −1 or 99. These codes will cause problems if you are trying to generate statistics as these values will be interpreted as numbers. It is important to check that they have been interpreted as missing values or how they need to be recoded for this to happen (see also Nagler 1995). Both SAS and Stata use a dot (.) to depict missing values and in R they are represented by the symbol 'NA'.

2.4 Defining Equation Expressions

The results of the equation of expressions in most computer languages are not the same as an algebraic expression as we would use in mathematics. It is important to realize that the computer evaluates the right side of the expression and then defines the left side as

the result. Thus, if $y = 10$ and we have an expression such as: $y = y + 1$, the result would be $y = 11$. Some programs (such as R) use instead of (or as well as) = to emphasize that a computer evaluates equal signs in code in a manner that is not equivalent to the use of equal signs in mathematics.

When typing expressions spread them out so they are readable (see also Cox 2005). Some rules to follow are as follows.

A. When in doubt as to the order of the computation use parentheses. The expression $y=x+z/q$ is more obvious as $y = x + (z/q)$. In most computer programs the division and multiplication operators are evaluated before addition and subtraction. But to be sure use parentheses.

B. There should be spaces around each binary operator except for power expressions such as ** and ^. For example, while $z = x + y$ is clear, x**2 is preferable to x **2.

C. Overall readability is paramount; so in the following identical equations the second expression is preferable to the first:time = hours + minutes/60+seconds/360, time = hours + minutes/60 + seconds/360.

D. Put a space after each comma in a function, for example, c = sum(x, y, z).

E. Use blank lines to separate code so that blocks of code are used for related processes. This approach makes it easier to see what parts of the code belong to each other. Further, it also makes it easier for a block of code to possibly be used in other applications.

2.1 Code Width

Code width should be limited to make it easier to read (see also Wassberg 2020). One way to do this is to limit code to the first 80 columns (72 are even better). This conforms to the width of a punch card image from the days of the punch cards and makes code more readable; although modern code editors can display wider lines of more than 200 on a screen, shorter lines are generally easier to read than longer lines. This may involve spreading commands into multiple lines. However, the awkwardness of viewing (and understanding) long lines outweighs the awkwardness of splitting commands into two or more physical lines.

In some code, such as in SAS, long lines of code can be written on multiple lines since the program will assume each line is part of the same command until a delimiter (a semicolon in SAS) is found. In Stata, there is no line delimiter (although you can specify one—it is not commonly used in most Stata codes). The end of a command only is determined by the end of the line, which is a code that is normally not printed by the do-file editor; however, you can request as an option that they be printed on the screen. In some cases, this can lead to very long lines that are very hard to read and debug. Alternatively, commands can be extended over multiple lines with a blank and three diagonal bars ('///'). In *R*, code can be spread over multiple lines by ending the incomplete line with an operator such as +,/, < or by leaving a bracket open. An example of rewriting a regression with a long list of regressors over multiple lines in all programs is given in Figure 2.

Figure 2 Writing a Command Over Multiple Lines.

SAS	*Stata*	*R*
1⊟Proc reg ; 2 model y_new d_cac d_rac d_rcac d_peh d_bieh 3 d_cd d_dw d_cw n_br d_acevap 4 d_ed1 d_ed2 hh_inc zz4 ;	reg y_new d_cac d_rac d_rcac d_peh d_bieh /// d_cd d_dw d_cw n_br d_acevap /// d_ed1 d_ed2 hh_inc zz4	reg = lm(y_new ~ d_cac + d_rac + d_rcac + d_peh + d_bieh + d_cd + d_dw + d_cw + n_br + d_acevap + d_ed1 + d_ed2 + hh_inc zz4, data = df)

2.2 Program Clarity

Write programs for clarity and not necessarily for efficiency. Remember that a program with fewer lines does not necessarily make it easier to read or understand (see for example, Wassberg 2020). Unless the routine is part of a very large-scale simulation it is usually better to allow the routine to take a little longer and write the code in a plain style, than attempt to save a few seconds by using a piece of clever code that may induce errors. It is also better to rely on routines that are already written rather than defining your own.

3. Program Elements

In this section, we detail, via examples, how particular elements of code can be done in the three programming languages we consider. In each case, we provide examples of the code for each coding situation using the elements of the code that would be most useful. In most cases, the example code provided could be modified for particular cases and they are presented for comparison purposes and not to provide the most efficient or 'cleanest' approach.

3.1 Logical Operators

Logical operators are used to carry out Boolean operations that are widely used in constructing variables that are conditional on specific conditions. An overview of these operators for each of the computer programs is given in Table 1. One of the most used expressions is for the comparison of values and while the symbols for these comparisons are similar across R, Stata and SAS there is one major exception. The Boolean algebra operator for equal is a double equal sign (==) in both Stata and R while it is a single equal sign in SAS.

Table 1 Summary of Logical Operators.

Operator	*SAS*	*Stata*	*R*
Exactly equal to	=	==	==
Not equal	~=	!= (or ~ =)	!=
Greater than	>	>	>
Less than	<	<	<
Greater than or equal	>=	>=	>=
Less than or equal	<=	<=	<=
Or	|	|	|
And	&	&	&

The result of the Boolean expression can be used to construct a numeric value of either a 0 or 1. Figure 3 provides an example where the result of any of the statements will be a new variable *vic* that equals 1 when the state is equal to 'Victoria' and 0 when the state is not equal to 'Victoria'. In the R command '*as.numeric*' is used to convert true and false values to 1 and 0. Note that the SAS expression could be confused since it appears to have two equal signs in the expression while both the Stata and R expressions use the logical equals (==) for the Boolean algebra comparison.

In addition, we can also use a mix of Boolean algebra with definitional equations. An example is given in Figure 4 where we define a variable z that is equal to x only when the state is Queensland and otherwise is equal to 0.

Figure 3 Construction of Binary Variables.

SAS	***Stata***	***R***
vic = (state = "Victoria")	gen vic = (state =="Victoria")	vic <- as.numeric(state =="Victoria")

Figure 4 Combination of Boolean Algebra with Definitional Equations.

SAS	***Stata***	***R***
z = x * (state = "Queensland")	gen z = x * (state == "Queensland")	gen z <- x * as.numeric(state =="Queensland")

Figure 5 Summary Statistics by Group.

SAS	*Stata*	*R*
`Proc means data=new; by firm; var y x1-x3 ;`	`by firm, sort: sum y x1 x2`	`by(data = df, df$firm, summary)`

3.2 Subsets of Observations

Often when working with datasets it is necessary to only use subsets of observations or variables in that file. For example, in Figure 5 we illustrate how to obtain the summary statistics for variables y, x_1 and x_2 for different groups of observations, in this case, different firms. In Stata, we use a *by* command in combination with a *sort* command as the observations need to be ordered by type of firm. In R we also use a *by* command where df is the dataset or data frame and df$x is the syntax for referring to the variable x in the data frame df.

Figure 6 illustrates restricting the sample to run a regression where the observations only correspond to the firm *Chrysler*. In Stata, we use the *if* qualifier and in R we use the *subset* command where the logical equals (==) is used in both cases to restrict the firm to only those observations corresponding to the *Chrysler* firm. In SAS code one specifies the data set to be used with a (where =) clause that specifies conditions for which observations are to be used.

In Figure 7 we restrict the sample to run a regression only for certain observations in the sample, say observations 61–80. The *in* qualifier is used in Stata whereas in R we restrict the rows of the data frame. In SAS, code one specifies the dataset option for the first observation to be used as *firstobs*= and the last observation to be used as *obs* = .

3.3 Conditional Evaluation

Using a conditional *if-else* statement allows control over the flow of the program. For example, in Figure 8 we define a value for a scalar y that depends on the value of another scalar x. In all computer programs rules for the writing of conditional statements must be followed.

In Stata, the open brace must appear on the same line as the *if* or *else* and nothing may follow the open brace except comments. The first command to be executed must appear on a new line. The close brace must appear on a line by itself. However, in R the *else* must be on the same line as its preceding closing curly bracket. Alternatively, for simple cases like this, an *if-else* statement can be used where the first argument is the condition to be tested and the second corresponds to the result if the test is true and the third is the

Figure 6 Subsets of Sample by Variable.

SAS	*Stata*	*R*
`proc reg data =test(where=(firm ="Chrysler")); model y = x1 x2 x3 ; run;`	`Regress y x1 x2 x3 if firm == "Chrysler"`	`summary (lm(y~x1+x2, data = subset(df, df$firm == "chrysler")))`

Figure 7 Subsets of Sample by Observation.

SAS	*Stata*	*R*
`proc reg data=test(firstobs=61 obs=80); model y = x1 x2 x3 ; run;`	`regress y x1 x2 x3 in 61/80`	`summary (lm(y1 ~ x1 + x2 + x3, data = df[61:80,] ))`

Figure 8 Using Conditional Statements.

SAS	*Stata*	*R*

`if x = 1 then y = 2 ;`
`else y = 3 ;`
	`if x == 1 { scalar y = 2 }`
	`else { scalar y = 3 }`
		`if (x==1) { y<-2 } else { y<-3 }`
		or,
		`y <- ifelse(x==1,2,3)`

corresponding result if the test is false. Since SAS uses delimiters, the statement can be on one line or multiple lines.

3.4 Loops

Loops allow the same command to be run for several variables at one time rather than writing separate code for each variable. Stata has commands '*foreach*' and '*forvalues*' where the *foreach* command loops through a list while the *forvalues* command loops through numbers. A similar command to *foreach* in R is '*lapply*', which applies a function to each element of a list that can be constructed using the '*cbind*' command. In SAS, one can write a macro routine where the procedures can be cycled through with replacement of the elements of a list.

In Figure 9 we illustrate how to repeat a regression with different dependent variables labelled *y*1 to *y*5 using the same set of explanatory variables. In Stata, we use the *foreach* command where *varlist* is used to loop over the different dependent variables. In the *regress* statement, single quotes are used around the dependent variable where the left quote ‘ differs from the right quote’ (see Cox 2020). In R *cbind* is used to construct a vector y consisting of *y*1 to *y*5. The command '*lapply*' is used to loop over the different dependent variables using a *function* statement to repeat the regression using the *lm* command where the *summary* command is used to show the results of each of the separate regressions. Because many of the SAS regression procedures allow for separate dependent variables regressed on the same regressors this becomes fairly straightforward.

In Figure 10 we show how to construct five new variables of random uniformly distributed numbers. In Stata we use the *forvalues*

Figure 9 Repeating a Regression with Different Dependent Variables. That the rules regarding the layout are the same as described under Conditional Evaluation.

SAS	*Stata*[1]	*R*
`proc reg data=test;` `model y1-y5 = x1-x3; run;`	`foreach y of varlist y1 y2 y3 y4 y5 { regress `y' x1 x2 x3 }`	`y <- cbind(y1, y2, y3, y4, y5)` `lapply(1:5, function(x) summary(lm(y[,x] ~ x1+x2+x3)))`

Figure 10 Constructing Variables of Random Numbers.

SAS	*Stata*[1]	*R*

`data new ; array xx x1-x5 ;`
`do ii = 1 to 100 ;`
`do over xx ; xx = uniform(0) ; end;`
`output ;`
`end;`
`run;`
	`set obs 100`
	`forvalues i=1(1) 5 { generate x`i'=runiform() }`
		`for (i in 1:5){`
		`assign(paste("x",i,sep=''), runif(100) )`
		`}`

command to generate $x1$ to $x5$ with each variable consisting of 100 observations of random uniformly distributed numbers generated using the command *runiform*(). In R we use a *for* loop to repeat the commands five times and *assign* and *paste* commands to construct the five different variables $x1$ to $x5$. The *sep* in the *paste* command defines what separates the variables. The observations for each variable are generated using the command '*runif*()'. In SAS we use a data statement to create a new data set called *new* by cycling over each observation and writing out the values in a loop. We use an *array* for five x's (this could be any size), called by the temporary name *xx*. Next, we cycle the creation of the five uniform RVs 100 times and write them out to the data set.

3.5 Sub-programs

User-defined programs can be written to reduce duplication in your code. In Figure 11 we write a program to convert a temperature recorded in Fahrenheit to Celsius degrees. In Stata, we write the program in a *do-file* called *convert*. The program begins with '*program*' followed by its name *convert*. The sequence of commands of the program follows and it is completed with an *end* statement. In this program '1' is used to represent the number that will be the input into the program that is, the number to be converted. In R we write a function called *convert*. The body of the function appears between curly brackets and consists of a sequence of R commands. The last command is the object that is returned. The SAS '*proc fcmp*' program will create a function routine that uses the standard code as used in a data statement to provide a result. In the example below, the function '*f_to_c*()'can be used in a data step and some procedures. It would also be possible to define a look-up table in SAS using *proc format* so that when the temperature is to be used it can be converted to either.

3.6 Simulations

Simulations can be used to understand the properties of econometric estimators and statistics computed from sample data. A pseudo-random number generator is used to generate sets of artificial data assuming a data generation process. Using the artificial data, estimators and test statistics are constructed and their properties examined.

In Figure 12 we present programs that use simulation methods to examine the properties of OLS estimators from a simple regression. Values for the population parameters are set and the dependent variable is constructed by generating pseudo-random numbers for the random error term assuming it has a standard normal distribution.

In Stata, we write a program called *reg_ols* that generates the dependent variable y and then estimates the simple regression model given data for the explanatory variable x and saves the OLS estimates. The program is called using the *simulate* command where we generate 5,000 replications using *reps*() and we save the results to a Stata data file called *results*. The computer-generated sequences of random numbers can be exactly replicated by

Figure 11 Program to Convert a Temperature Recorded in Fahrenheit to Celsius Degrees.

SAS	*Stata*	*R*

/*
Use the program fcmp to create a function called f_to_c( )
*/
options cmplib=sasuser.funcs;
proc fcmp outlib=sasuser.funcs.example;
function F_to_c(fd) ;
cd = ((fd - 32) /9) * 5 ;
return(cd) ;
endfunc;
run;
/*
Use as cd = f_to_c(fd)
*/
	/* call the program convert */
	program convert
	/* the formula with '1' used to represent the number to convert */
	scalar cel = ((`1'-32)/9)*5
	/* display the result */
	display cel
	/* close the program */
	end
	/* command to run the program */
	convert 85
		# function convert
		# return the value xc
		# convert from the input xf
		convert = function(xf) {
		xc = (xf-32)*5/9
		return(xc)
		}
		# command to call function convert
		convert(85)

Figure 12 Simulation for Regression.

SAS

```
*    Repeat the data 5000 times ;

     data temp ; set test ;
          do ii = 1 to 5000 ;
          y = 2 + 5*x1 + rannor(45321) ;
          output ;
          end;
     run;
*    Sort the data by the value of ii ;

     proc sort data = temp ; by ii ; run;
*    Run the regression on each case with a by variable
     saving the results in dataset called est ;

     proc reg noprint data=temp outest=est ; by ii ;
     model y = x1 ; run;
*    Compute the statistics for the means of the coefficients
;

     proc means data=est ; var intercept x1 ; run;
*    Alternative approach using IML  ;

     proc iml ;
     use test ; read all into x ;
     n = nrow(x) ; k = ncol(x) + 1 ; x = j(n,1,1) || x ;
     ixpxx = sweep((x`*x),(1:k)) * x` ;
          do ii = 1 to 5000 ;
          y = 2 + 5 #x[,2] + rannor(J(n,1,(45321))) ;
          b = b // (ixpxx * y)` ;
          end;
     print "means of coefficients",
          (b[:,]) [c={"intercept" "x1"}];

          quit; run;
```

Stata

```
/* Program to run a simple regression
  and save OLS estimates */
   capture program drop reg_ols
   program reg_ols, rclass
       gen y = 2 + 5*x + rnormal(0,1)
       regress y x
       return scalar b2=_b[x]
       return scalar b1=_b[_cons]
       drop  y
   end


/* read data in for the explanatory variable x */
   use xdata , clear

/* use simulate command to generate 5000 replications
   and save the OLS estimates in the file results */
   simulate  r(b1)  r(b2), seed(45321) reps(5000) nodots ///
   saving(results, replace): reg_ols

/* use summarize command to look at mean of
   the simulated 5,000 OLS estimates  */
   summarize _sim_1 _sim_2
```

R

```
# To replicate the order of
# the number generation process
set.seed(45321)

# set sample size and
# number of simulations
n <- 500; mc <- 5000

# initialize b1 and b2
b1 <- numeric(mc);  b2 <- numeric(mc)

# load the data for the explanatory variable
xdata <- read.csv(file='xdata.csv', header=TRUE)

# Loop to run simple regression
# and save OLS estimates:
for(j in 1:mc) {
  y <- 2 + 5*xdata$x + rnorm(n,0, 1)
  b <- coefficients( lm(y~xdata$x) )
  b1[j] <- b["(Intercept)"]
  b2[j] <- b["xdata$x"]
}

# Mean of the 5,000 simulated estimates
mean(b1);   mean(b2)
```

setting the *seed*() option. The data for *x* is the same for each replication and is read in from a Stata data file before the *simulate* command. Summary statistics are then used to examine the properties of the OLS estimates.

To perform the simulation in R we first set the seed for the computer-generated sequence of random numbers using the command '*set.seed*()'. The sample size and number of replications are set and the vectors b1 and b2, where the results of the OLS estimates will be stored from each of the replications, are initialized. The data for *x* is the same for each replication and is read from an external file. Within a *for* loop, *y* is generated and the OLS estimates are stored in the vectors b1 and b2.

In SAS we can generate a new data set with replications of the original dataset. First, we generate new data with a simulated value for the dependent variable. This is done by creating a dataset with replications of each observation (here this is set to 1000). These new values have a simulated value of the dependent variable (*y*) based on the formula and the random error term. In the second step this data set is sorted by the iteration number of the simulation (*ii*). The third step estimates the regression with a *by* option so that the regression is run 1000 times. By using the 'noprint' option and the outset= options we can save the results in a new dataset called *est*. The last step uses the data set called *est* to compute the means and other statistics for each of the coefficient estimates. Alternatively, a far more efficient method (using 10 times less CPU time) would be to use a *Proc IML* (Interactive Matrix Language) code to perform the simulation as shown in the second line in Figure 12. In this program, the vector defined as: is computed once (referred to as *ixpxx*) and the only change for each simulation is the generation of a different vector of the dependent variable. Thus, the *i*th coefficient set would be defined by: for each simulated vector $\mathbf{Y}_i$.

4. Final Recommendations

A. Only write a program when it is necessary to. Always first check before writing a program to see if one already exists or at least could be modified. Remember, it is often useful to 'steal' code from other programs that perform a similar function. Try searching for other programs by googling for them first. However, when using code from other programs make sure you work through the code and understand every line. Start with a first draft, which is then maybe revised several times.

B. Begin by getting a version of your program working first and then concern yourself about making it your ideal program (see for example, Wassberg 2020). Writing a program is not dissimilar to writing a paper and

you may find that your first attempt needs to be revised multiple times. Your final program may not even retain much from your first version. Similarly, as with writing a paper, set aside more time than you think you will need to write a program.

C. If possible, always test a smaller version of your program first. For example, when writing a program to perform a simulation use a reduced number of iterations first to check the program is working.

D. Write a program that only does what it needs to do. Do not make it more complicated than is required because this often results in errors being introduced into the code. In many cases, researchers use a series of programs to perform such processes as cleaning data, estimating a similar model, and presenting results in tabular and graphical form. These programs are then collected and called by a master program for the entire analysis or for a subpart of the analysis.

E. Programs not only need to work correctly but also be easy to read, understand and maintain. Adopt a consistent style for your program that will make your code more readable, and easier to find errors and maintain. For example, in a 'for loop' assign the closing brace '}' with the beginning of the first command in the loop, as shown in Figure 12 with the R code. Try breaking large amounts of code into logically arranged sections by grouping related lines of code (see for example, Boswell and Foucher 2011, Wassberg 2020). Keep some lines between the different sections to separate them. Further, delete any unnecessary code.

F. Further, separate tasks that can be performed sequentially in different programs. For example, data cleaning and data analysis should be considered as two separate tasks. This approach helps to identify errors. The sequence the programs need to be run in can be identified by placing a number indicating the order of execution before the name of the program (see for example, Nagler 1995, Orozco et al. 2020).

G. Do not sacrifice clarity in your program over optimizing for speed or memory use. It is more important that it is clearly, cleanly written and easily understood.

H. Adopt the '*Don't Repeat Yourself*' or DRY principle where you avoid having the same code repeated in two or more places (Hunt and Thomas 1999). For example, suppose you want to run numerous regressions using several of the same regressors in each regression. Instead of repeating the regressors for each regression, they could be set up as a list or group before running the regressions and then this list or group used in each specification. This approach also has the advantage that if you want to include an additional regressor or delete a regressor then you only need to make a change to the list or group.

I. Keep different versions of your program. Make sure that you date them and use a naming convention to identify the stage of the version. This is useful so that you can always return to earlier versions of the program to make comparisons and may help identify mistakes. Alternatively, use version control software to track successive versions of a piece of code (see for example, Gentzkow and Shapiro 2014, Orozco et al. 2020).

J. Do not feel that all the analysis needs to use the same software. As we discuss here different programs have strengths in different areas of research. SAS is a legacy routine that includes many operations for data organisation that allow for the construction and storage of complex data sets. However, the estimation procedures are not updated at the same frequency as either Stata or R. In many ways, R is the replacement for SAS, however the syntax and debugging process suffers from its strength—that it is free-ware and it has

many different contributors. Many of the most recent procedures described in the statistics literature employ R as the preferred coding language. Stata also suffers from this deficiency; however, to a lesser extent, the base procedures have a more structured syntax than the contributed ones. StataCorp sells Stata and, also publishes a scholarly journal that provides insights into the use of new estimation programs. This makes it possible to go to one place to find the latest procedures. Unlike either SAS or R, Stata explicitly has procedures designed for applied econometrics and programs written in it are one of the most widely used software found on the *American Economic Review* resources website (Fiva, Værøy and Herrera 2021). For this reason, it is not unusual to find the code used for applied economic research to use SAS or R for data manipulation and Stata for the estimation and presentation of results.

DATA AVAILABILITY STATEMENT

Not applicable.

Endnotes

1. Many of these details reflect longstanding and general advice (for example, Kernighan and Plauger 1978).

2. As referred to in the recent film 'Significant Figures'.

3. Programs to invert matrices and sort data are based on algorithms that have been available in FORTRAN since the 1960s.

ACKNOWLEDGMENTS

Open access publishing facilitated by The University of Melbourne, as part of the Wiley - The University of Melbourne agreement via the Council of Australian University Librarians.

ORCID

Joe Hirschberg http://orcid.org/0000-0001-8354-5433

References

Boswell, D. and Foucher, T. 2011, *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. O'Reilly Media Inc, Sebastopol, CA, USA.

Cox, N. 2005, 'Suggestions on *Stata* programming style', *The Stata Journal*, vol. 5, pp. 560–566.

Cox, N. 2020, 'Speaking Stata: Loops, again and again', *The Stata Journal*, vol. 20, pp. 999–1015.

Fiva, J., Værøy, T. and Herrera, F. 2021, August 9, 'For applied economics @*Stata* is still the only game in town', viewed 3 December 2021, <https://twitter.com/JFiva/status/1424679980538241025?s=20>.

Gentzkow, M. and Shapiro, J., 2014, 'Code and Data for the Social Sciences: A Practitioner's Guide', viewed 11 October 2021, <https://web.stanford.edu/~gentzkow/research/CodeAndData.pdf>.

Hunt, A. and Thomas, D. 1999, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, Boston, MA, USA.

Kernighan, B. and Plauger, P. 1978, *The Elements of Programming Style*. McGraw–Hill, New York.

Nagler, J. 1995, 'Coding style and good computing practices', *Political Science and Politics*, vol. 28, pp. 488–492.

Orozco, V., Bontemps, C., Maigné, E., Piguet, V., Hofstetter, A., Lacroix, A., Levert, F. and Rousselle, J. 2020, 'How to make a Pie: Reproducible Research for Empirical Economics and Econometrics', *Journal of Economic Surveys*, vol. 34, pp. 1134–1169.

Vilhuber, L. 2021, 'AEA data and code availability policy', *AEA Papers and Proceedings*, vol. 111, pp. 818–823.

Wassberg, J. 2020, *Computer Programming for Absolute Beginners: Learn Essential Computer Science Concepts and Coding Techniques to Kick-Start Your Programming Career*. Packt Publishing.