Coding Style and Practice: Are all programs equal?

Joe Hirschberg and Jenny Lye[1]

Department of Economics
University of Melbourne
Victoria 3010
Australia.

Abstract

Research in applied economics requires the use of specialised software for data analysis. The scripts or code for these software routines can be written in many ways however it has recently become standard procedure for these scripts to be submitted along with the research document to allow for the replication of results. Often scripts written for one project serve as the basis for future projects that employ the same data and/or techniques of analysis. In addition, it has become increasingly common for economic research to be performed by teams that need to share scripts. For these reasons it is important to follow style guidelines so that the details of the analysis can be followed and verified.

This paper presents a style guide for coding in three commonly used software packages used for data analysis in applied economic research and we also perform an analysis of code used for papers published in the 2020 volume of the *American Economic Review* and the *Papers and Proceedings* of the 2020 meeting of the American Economic Association.

Key words: *Stata*, *R*, *SAS*, Replication, Cluster analysis of code

JEL: C8, Y1, A23, C55, L86

[1] Corresponding author: Hirschberg, email j.hirschberg@unimelb.edu.au. This paper is based in part on lectures the authors have given in lectures given to graduate students on research methods and applied econometrics at The University of Melbourne. We wish to thank Lars Vilhuber, Jon Fiva and Dan Millimet, for their comments on earlier drafts. We especially wish to thank Jon Fiva for allowing us to have access to the data for the survey of computer code we report in Appendix A.

Contents

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 recent proliferation of journal articles that employ complex estimation procedures applied to large scale datasets. In addition, it has become a requirement that data and computer programs be submitted as part of a submission of a research essay or dissertation. [2] 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 5 pages to define the policy for the American Economic Association publications but scant attention to the condition of the code submitted.[3] 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 proports to do. Frequently, the code is written in a manner that it 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 a clear style.[4]

There are few occasions when a computer code work or perform the task perfectly the first time they are run. 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, to speed it up, or borrow from it.[5] Good style helps to concentrate on parts of code elements that can be checked separately. Additionally, the availability of well written code has become a useful tool in the training of future researchers.

Another factor in the need for good code style is the rise in the advent of "Research Teams" in economics. Recently Jones (2021) has documented the rise of multiple author contributions in economics over solo author papers. When multiple authors work on a project it is not uncommon for different authors to share code to be used by others. This may necessitate that each element in the overall project be available for all the team members to contribute. It may also be the case that multiple projects are based on a common set of datasets or algorithms which requires that code be in a form that allows it to be reused for different projects.

---

[2] In this paper we will refer to programs, scripts and code interchangeably to mean a computer readable file that lists a series of instructions to be read by software designed to carry out data manipulations and statistical computations.

[3] For more information see: https://www.aeaweb.org/journals/data/faq#package.

[4] The code we discuss here is in the nature of "scratch programs" that intended to be written in high level languages such as *Stata, R* and *SAS* and may only be used a limited number of times. Production programs that are used repeatedly to perform large scale tasks require more detailed specifications that are outside the scope this study.

[5] Many of these details reflect longstanding and general advice (e.g., Kernighan and Plauger 1978).

For most applications, the construction of code involves assembling data into a form that can be used by computation programs (e.g. regressions) that have been written by professional programmers. Programs have been written to invert matrices, multiply matrices, find the eigenvalues of matrices, etc. are widely available. The algorithms for these computations have been the subject of intense investigation by many authors over the past 70+ years.[6] Many statistical packages have optional elements that allow the researcher to design a special purpose estimation procedure, however the focus in this paper is the construction of code to manipulate data in a transparent way 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.

In addition, we have also conducted a limited survey of current programming practice by examining the *Stata* code submitted for the replication of analysis conducted for the papers published in volume 110 (2020) of the *American Economic Review* and the Papers and Proceedings of the 2020 meeting of the American Economic Association. In this survey we also categorise code as their similarity to two diametrically different sets of code that perform the same tasks.

The paper proceeds as follows: first, we provide a style guide for some of the best practice in writing code with attention to the main elements of the code, then we discuss examples in *Stata*, *SAS* and *R Stata*, *SAS* and *R* code for performing some of the most common tasks, and finally we present the results of a computer generated survey of the code provided for the 2020 volume of the *American Economic Review* and the *Papers and Proceedings* of the 2020 meeting of the American Economic Association.

2. Style Guidelines

The history of computer coding is not long, the basic form of code was first written in the 1960s. These early forms of code still have an influence on current coding elements and conventions. Originally, code was 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). This language still exists and the conventions for the code have carried through to current computer languages of today many of which started out as *FORTRAN* programs. Basic rules for this code included: that there could be leading blanks and that the variable mnemonics must: start with a letter, are in capital letters, have no imbedded blanks and be no longer than 8 characters.

[6] See Press et al (2007) for details on many of these algorithms.

Due to the need to conserve storage in early main frame computers, (they may have had as little as 124k of memory) it was considered important to conserve as much space as possible. Memory 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 code process. The basic form of storage for code and data was on Hollerith punch 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 terseness 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 an 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.[7]

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, maintaining it, and to assist in reducing the likelihood of introducing errors.

2.1 Documentation Comments

It is always important to include a preliminary comment line or lines containing the name of the program file (so it can be found 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 (e.g. Cox 2005; Nagler 1995). For example, you may have written this program to perform an analysis done by another program but only on a sub-set of the data or using a modified specification or method of estimation (GMM versus 2SLS). In addition, if the code follows a specific article, it is important to include complete references to the literature and the source information for any data employed.

Program lines can be of any length. Originally, programmers prided themselves on their brevity. The days when every line of code was an additional card in a deck are over. With portable storage measured in terabytes, it is easy to error on the side of being verbose over being terse. Additional comments are especially useful when scanning code or may even help spot mistakes.

---

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

Comments should be used to explain what a piece of code does or what it is supposed to do. If the program is used to make another many of these comments will stay the same from program to program. While it may be cumbersome to write comments, it is the best way to avoid problems and when measured against the time wasted trying to figure out was done is often a time saver.

In both *Stata* and *SAS*, comments can start with a /* and end with a */ on as many lines as desired. There are other alternatives. In both *SAS* and *Stata*, a line 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* doesn't support multi-lined comments so if a comment runs over several lines there must be a # on every line of the comment. The software provided editors use different colour fonts for comments as opposed to the colour of the font used for commands in as an aid in determining which lines in code are non-operating.[8] Examples of writing a comment for each of these programs is given in Figure 1.

Figure 1: An Example of Comment Writing

SAS	***Stata***	***R***

`/*  program parallelassay_pokies.sas`
 	
`SAS 9.4  Hirschberg & Lye  22/12/2020`
 	
`Program to perform Assay analysis to estimate the equivalent value for`
`dummy variable. Use the pokies data for smoking bans near the border t`
`determine what change in the number of EGMs is equivalent to being on`
`border based on the data from Hirschberg J. & J. Lye (2010), 'The indi`
`impacts of smoking bans in gaming venues', Chapter 11, in Current Issu`
`Health Economics, Contributions to Economic Analysis, eds D. Slottje a`
`R. Tchernis, Emerald Press, Sydney.`
`*/`
	`/*  program parallelassay_pokies.do`
	
	`Stata17  Hirschberg & Lye  22/12/2020`
	
	`Program to perform Assay analysis to estimate the equivalent value fo`
	`dummy variable. Use the pokies data for smoking bans near the border`
	`determine what change in the number of EGMs is equivalent to being on`
	`border based on the data from Hirschberg J. & J. Lye (2010), 'The ind`
	`impacts of smoking bans in gaming venues', Chapter 11, in Current Iss`
	`Health Economics, Contributions to Economic Analysis, eds D. Slottje`
	`R. Tchernis, Emerald Press, Sydney.`
	`*/`
		`#  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`
		`#  dummy variable. Use the pokies data for smoking bans near the border t`
		`#  determine what change in the number of EGMs is equivalent to being on`
		`#  border based on the data from Hirschberg J. & J. Lye (2010), 'The indi`
		`#  impacts of smoking bans in gaming venues', Chapter 11, in Current Issu`
		`#  Health Economics, Contributions to Economic Analysis, eds D. Slottje a`
		`#  R. Tchernis, Emerald Press, Sydney.`
		`#`

2.2 Names/Mnemonics

It is important to choose meaningful descriptive names for programs, sub-routines, variables, and macros. Names chosen for a program should not conflict with anything that already exists although the use of sequential names may help aid in finding 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*.

[8] We use *R-studio* for the code examples to demonstrate the colour variation in the fonts.

B. Add long descriptive variable labels when possible. And the format of the responses in the case of a survey (i.e. 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 the 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.

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, eg. `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.

2.5 Code Width and Margins

Code width should be limited, and left-hand margins should be used 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) and to start lines of code with either blanks or tabs. The 80-column width harks back to the number of columns on a punch card, although with modern code editors one can read up to 200 on a screen shorter lines are generally easier to read than longer lines. The awkwardness of viewing (and understanding) long lines outweighs the awkwardness of splitting commands into two or more physical lines when possible.

In some languages, 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*, the default line delimiter is a carriage return that is an unprinted character in the code editor (although you can specify one - it is not commonly used in most *Stata* code). In some cases, this can lead to very long lines that are very hard to read and debug. Alternatively, *Stata* commands may 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.

Code on card images in FORTRAN started in column 7, the code automatically included a left-hand margin. Currently, beginning margins are not well defined in the code editors for *R*, *Stata* and *SAS*. It is physically very hard to print documents right on the edge of a page thus most all printed matter we read uses margins. For this reason, it is hard to read a script that starts on the first column of the file. In the past limits on storage space may have required that code be written as compactly as possible however current computer storage capability makes this saving inconsequential. To reduce the possibility of error by improving readability, left-hand margins are a simple but effective feature.

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)

2.6 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 eg. Wassberg 2020 and our example routines in Appendix B). Unless the routine is part of a very large scale simulation it is usually better to allow the routine to run 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. Another example where one can save is by allowing the size of a data set be larger than necessary by not worrying about storage space. Given the very large capacity of modern computers and storage devices, writing code to compress data may cause more effort than gain.

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 which 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 commonly used expression 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*. The single equal sign can result in errors when the computer assumes it is part of a conditional statement and instead of flagging an error where you have two equal signs in one expression and it evaluates the part of the expression as either a 0 or a 1.

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 while *SAS* allows an expression with multiple equal signs.

Figure 3: Construction of binary variables

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

In addition, we can also use a mix of the 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 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")`

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 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)`

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.

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 + x3, data = subset (df, df$firm == "Chrysler") )`

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 data set option for the first observation to be used and the total number to be considered for use. Thus, the number of observations used is listed and the first of those to be use is listed.

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,] )`

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.

Figure 8: Using Conditional Statements

SAS	*Stata*[1]	*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)`

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 *ifelse* 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 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 which 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 y1 to y5 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 y1 to y5. 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 straight forward.

Figure 9: Repeating a Regression with different dependent variables

SAS	*Stata*[9]	*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)))`

In Figure 10 we show how to construct 5 new variables of random uniformly distributed numbers. In *Stata* we use the *forvalues* 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 5 times and *assign* and *paste* commands to construct the 5 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*. We then use an *array* for 5 x's (this could be any size) called with the temporary name *xx*. Next, we cycle the creation of the 5 uniform RVs 100 times and write them out to the data set.

Figure 10: Constructing variables of random numbers

SAS	*Stata*[7]	*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) )` `}`

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 first line of the program "*capture program drop convert*" enables the program to be loaded and reloaded using the same name *convert*. The program itself begins with *program* followed by its name *convert*. The sequence of commands of the program follow 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*

[9] Note that the rules regarding the layout are the same as described under Conditional Evaluation.

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 in 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.

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)

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 setting the *seed*() option. The data for *x* is the same for each replication and is read in from a *Stata* data file prior to 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 is 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 in from an

external file. Within a *for* loop, *y* is generated and the OLS estimates are stored in the vectors b1 and b2.

Figure 12: Simulation for Regression

SAS

```
/*
    Define the macro rep_reg where num is the number
    of repetions
*/
    %macro rep_reg(num) ;
    %do ii = 1 %to &num ;
/*
    Run over the data set test
*/
    data temp ; set test ;
    y = 2 + x1 + rannor(0) ;
/*
    Run the regression on each case
*/
    proc reg noprint data=temp outest=est ;
    model y = x1 ;
/*
    Save the results in a data set
*/
    %if &ii = 1 %then %do; data total ; set est ; %en
    %else %do; data total ; set total est ; run ; %en

    %end ;
    %mend rep_reg ;
/*
    run the simulation 1000 times and compute the mea
*/
    ods graphics off ;
    %rep_reg(100) ; RUN;

    proc means data=total ; var x1 intercept ; run;
```

```
proc iml ;
use xdata ; read all into x ;
n = nrow(x) ; x = j(n,1,1) || x ;
ixpxx = sweep((x`*x),1:2)*x` ;
do ii = 1 to 5000 ;
        y = 2 + 5#x[,2] + rannor(j(n,1,0)) ;
        b = b // (ixpxx * y)`;
end;
print "Means of betas", (b[:,]) ;
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 OLS estima
*/
   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)
```

In *SAS* we can define a macro to perform four steps. First, we generate new data with a simulated value for the dependent variable. This assumes that there is a data set called *test* with a variable named *x1* using this variable and the random number generator we define the variable *y*. In the second step this data set is used to run a regression with the option to save the results in a data set called *est* and not to print any results or create the default plots. In the third step we add the estimates to the data set called *total* note the first time we need to start the process with total defined as the first estimation results. In the last step we compute the means and standard deviations of the intercept and the parameter for x1. Alternatively, a far more efficient method 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: $\left(\mathbf{X}'\mathbf{X}\right)^{-1}\mathbf{X}'$ is computed once and the only change for each simulation, is the generation of different a different vector of the dependent variable.

4. Coding practice

In this section we report on a survey of current practice in coding format of programs that have been provided for purposes of replication. Currently, many journals request that authors of papers published make their code and data available for readers to replicate their results. In this survey we report on the analysis of *Stata* code that accompany papers published in two journals of

the American Economic Association (AEA). We examine the code for *Stata* since it does not have a specialised editor that provides automatic style changes and it has been shown to be the predominant software used for economics analysis as shown by the surveys conducted by Lars Vilhuber, James Turitto, and Keesler Welch (2020) and Jon Fiva, Tuva Værøy and Federico Herrera (2021) (FVH) (see Appendix A for the results of the FVH survey). Another reason to examine *Stata* code is the ability to write *Stata* code in a wide variety of programming styles. This is demonstrated by two examples of equivalent code in Appendix B.

4.1 A survey of *Stata* coding practice.

We conducted a survey in which we downloaded all the code and data publicly available for the papers published in volume 110 for 2020 of *The American Economic Review* (*AER*) and American Economic Association *Papers and Proceedings* (*AEA P&P*) for 2020. Of the 235 papers published in the *AER* and the *AEA P&P* 175 provided a link to the code and data. The papers that did not were primarily papers that did not report the results of an econometric analysis or employed data that was not publicly available – thus the code and data are not available on-line. From these programs we found extracted 1,700 *Stata* scripts as identified with a ".do" last delimiter in the name.[10] The combined lines of code constituted a file with 528,191 lines of code. [11] We use this file as data to determine the practices of the authors in this journal[12]. In performing this analysis, we are not making judgements on coding quality of any specific program or paper, we only use this analysis to demonstrate that current coding practice often does not necessarily reflect the adherence to many of the guidelines that we propose.

Table 1 The program elements that appear at least once in a script

Program Feature	*% of scripts*
A Blank line	98.38
Equation is used	97.82
A Comment	91.09
An Equation that is spaced	82.80
Use of *,= , # for separator	75.52
Tab in a line	70.70
Start a line with either a tab or at least 2 spaces	68.68
There is a loop defined	49.92
Use of a continuation "///" or a delimiter	35.80
Definition of a variable label	22.86
At least one Line > 320 characters	13.95
Install special routine via "ssc install***"***	7.34
Make an assertion with "assert***"***	5.94
There is "global***" or a "***local***" defined***	1.74

[10] Other files used by *Stata* with last the delimiters ".mata" and ".ado" have been removed for this analysis.
[11] The analysis of this code was done via the use of a computer routine that examined each line of code.
[12] Because most of the papers in this journal have multiple authors, we do not report which code was from which paper.

Set the version of Stata	1.01

Table 1 provides the presence of characteristics observed in these *Stata* scripts. From this summary we find that all the scripts include at least one blank line, 98% includes an equation, 92% have at least one comment, 84% have at least one equation with spaces before and after an operator (ie =, +, *, /, <, >), 77% used lines with multiple characters (ie, *, =, #), 72% used at least one tab in a line, and 69% also started at least one line that started with a tab or more than 2 blanks. Other characteristics with lower frequencies of occurrence are listed as well. The presence of these good style features is indicative that these features are used quite widely, however in many cases they are only employed in a small portion of the script. Note that in over 12% of these scripts there was a line that was more than 320 characters long. This would imply that one could not read this line without moving the editor window from the start of the line of code. One way to avoid this is to use a continuation or an end of line delimiter and these are used in at least 35% of the scripts. Another way of limiting the width of code is to use global or local statements to allow for the reference to multiple variables with a single name. We find a very low use of this method with just a bit over 1% of the scripts using these. We found that almost 7% of code install third-party procedures via the use of the "ssc install" command. The most frequently used of these are listed in Appendix C. We also found that 6% of the scripts employ the "assert" command to check for consistency in variable definitions and other data characteristics. This command is a very useful command when working with complex data that may have been generated by other researchers in order to verify the construction of variables from other variables.

Table 2 Means of average values per script

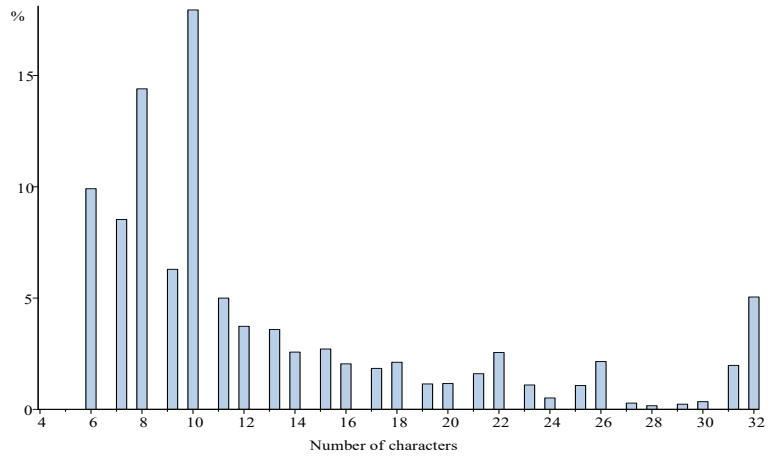
Characteristic	*Scripts*	*Mean*	*St D*	*Min*	*Max*
The line length	1785	35.07	15.11	2.00	268
Number of lines in program	1785	333.49	1303.98	1.00	42946
Number of words used	1669	92.66	949.17	1.00	36382
Number of words per line	1785	5.09	3.78	0.67	47.07
Word size	1669	10.55	3.20	6	31
# spaces to start	1785	0.20	0.74	0.00	13.85
% of equations w space	1746	48.47	34.67	0.00	100.00

In Table 2 we list the statistics on the means of these characteristics for each code script. From this table we note that on average the average line length in the scripts was 31.42 characters, there were 314 lines of code per script, there were 5.22 words per line, and the average word size was 10.56 characters. However, the average number of spaces to begin a line was only .15, this implies that although in Table 1 we note that more than 2/3 of the programs used spaces to start a

line at least once, on average there was only .17 spaces to start a line of code. This implies that most code started with no spaces and reading these programs is equivalent to reading a text with no left-hand margin.

To provide an overview of the size of the mnemonics or variable names used we provide Figure 14 that displays the distribution of the number of characters used to define the mnemonics. Note that there are predominant spikes at 8 and 10 characters. The 8-character mnemonics appear to be a throwback to the original character limit on mnemonic lengths imposed by early computer programs such as FORTRAN and earlier versions of *SAS*. The mnemonic size is limited in *Stata* to be no more than 32.[13] An alternative to long mnemonics is the use of global and local statements to provide shorthand values for collections of mnemonics and the definition of variable labels which we find are used in at least 20% of the scripts.

Figure 14 The distribution of the mnemonic size.



In Figure 15 we provide the distribution of the number of different mnemonics used by each program script. Note that in most cases there are fewer than 100 used although when using a typical census survey, it may be necessary to use files with as many as 10,000 variables.

[13] *SAS* also limits the variable names to 32, however *R* allows much larger names.

Figure 15 The number of mnemonics used by each script.

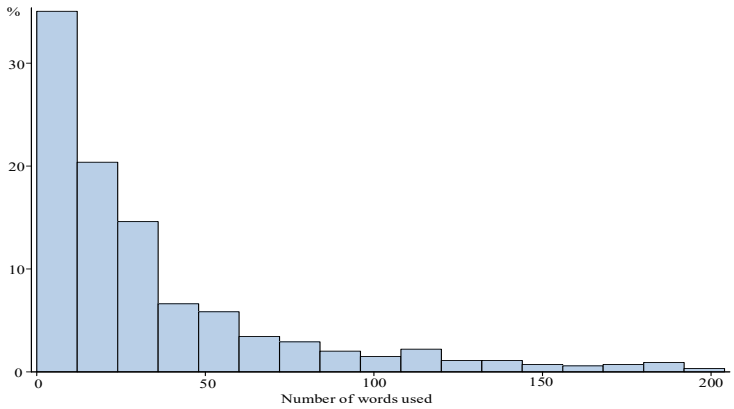
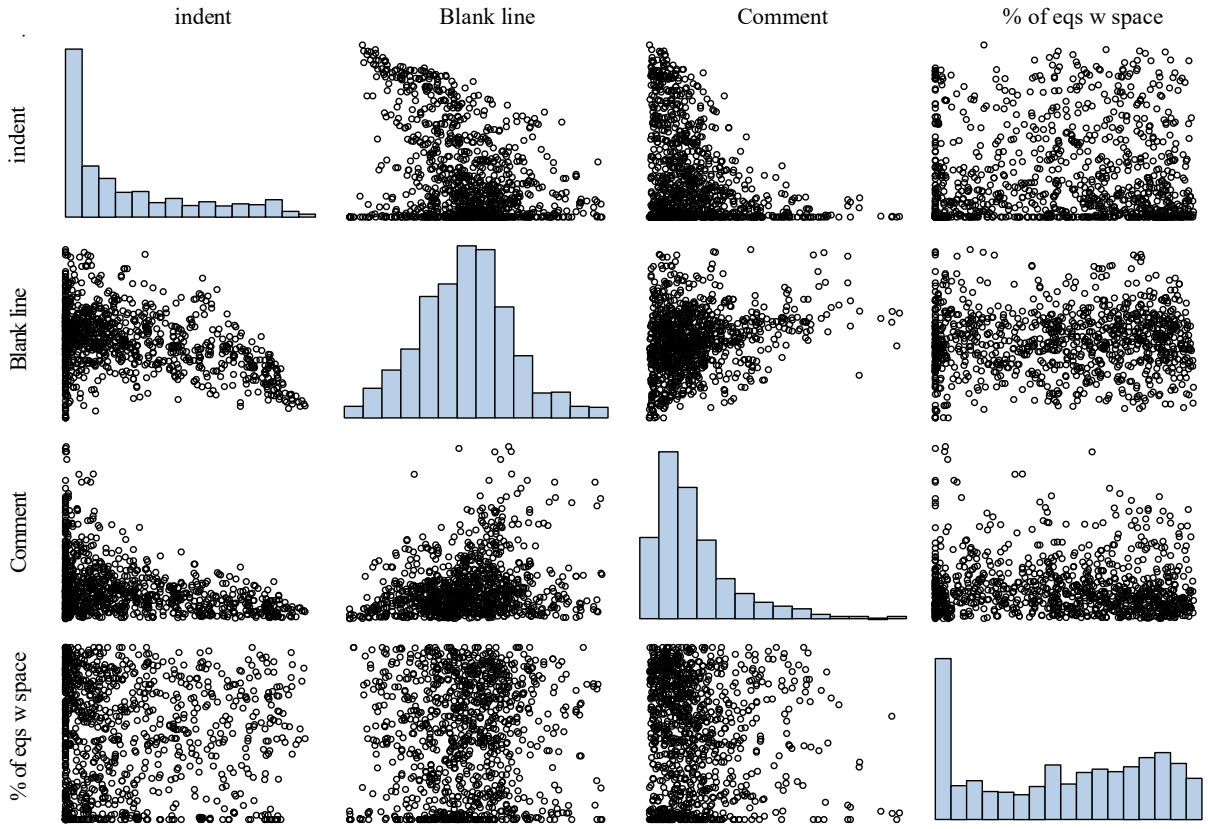


Figure 16 provides a multivariate plot of the proportion of the code that has these characteristics that we propose are aspects of code that improve readability. In this case we use the presence of indented lines, blank lines, comments, and equations that are spaced. The diagonal of the figure shows the histogram of these measures. The scatter plots for each of these measures appear not to show positive relationship between these characteristics if one might assume that these characteristics satisfy a common indication of a "good" programming style.

Figure 16 The use of: indents of lines, blank lines, comments, and spaces before and after equation operators in separate scripts.



Stata is often used in economic research for the estimation of regressions. One of the prominent innovative features of *Stata* is the estimation of the standard errors of the regression to account for unknown heteroskedasticity using clustered covariance structures. This feature has been incorporated in several estimation subroutines that are unique to *Stata*. In order to determine the prevalence of the use of these clustered standard error options we provide the frequency of the use of regression procedures, regression with clustered standard errors, and the use of clustered standard errors with estimations procedures that do not have the characters "reg" in them. From Table 3 we note that this option is very widely used and may be one of the primary reasons for the wide popularity of *Stata*.

Table 3 Econometric practice: the use of regression and clustered Standard error estimates.[14]

Proportion of all programs with at least single	
Use of a regression	63.81%
Use of a regression with clustered se	15.80%
Use of clustered se in any procedure	60.95%

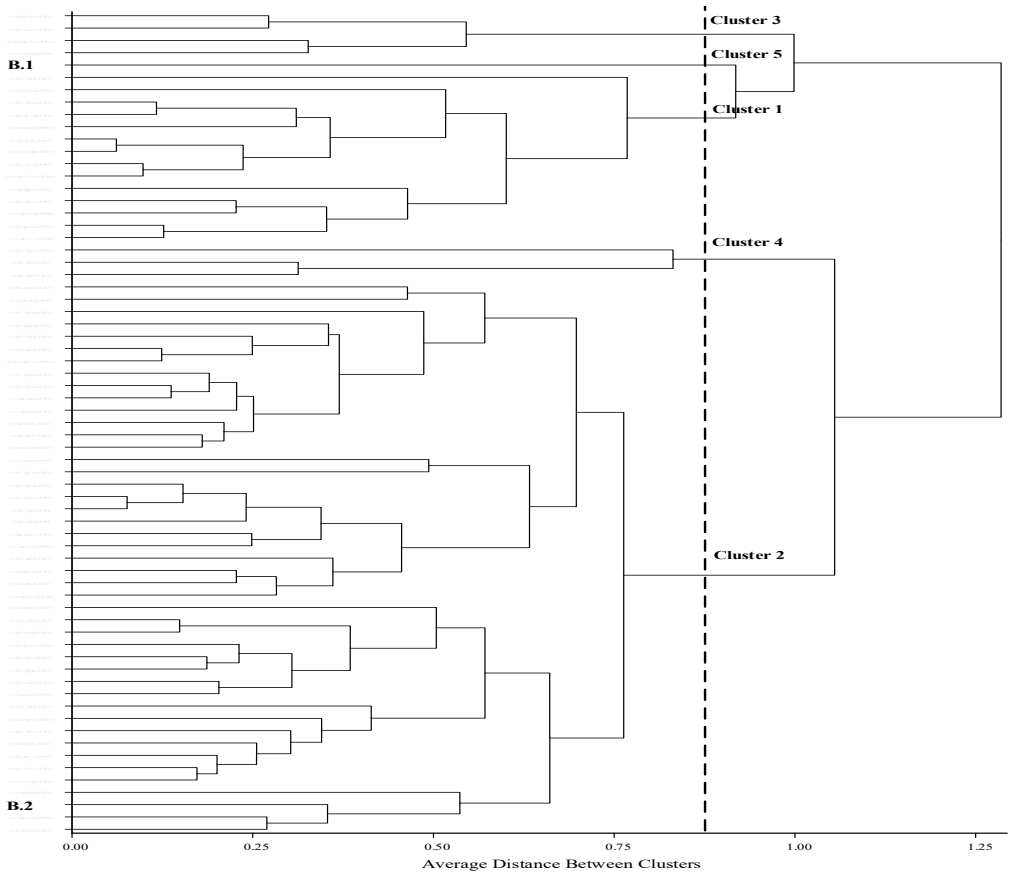
4.2 A categorization of code type

To compare these two scripts defined in Appendix B with the ones we observe for the 2020 *AEA P&P*, we performed an hierarchical cluster analysis based on the average number of comments per line, average the use of spaces in equations, the proportions of each line that is blanks (where the characters in each line are counted), the proportions of lines that have tabs, and the proportion of the code that has blank lines. In order to evaluate the cluster analysis, we present the dendrogram from this analysis where we identify the location of the two scripts given above in Figure 17.[15]

[14] Note not all scripts are used for estimation. Some are only designed to prepare data for analysis.

[15] For applications of cluster analysis in economics see Hirschberg et al (1991,2001) and Hirschberg and Lye (2020).

Figure 17 The Dendrogram for averages from the AEA P&P paper's code and the two example routines in Appendix B.[16]



We limit this analysis to only the P&P papers since the majority provide no more than two *Stata* scripts for replication and thus are more comparable to the single scripts in Appendix B. The cluster analysis we employ is a hierarchical cluster analysis that allows for the construction of a dendrogram in which the "distance" between the scripts can be measured to define the similarity between them based on selected characteristics. We use the average method which was first proposed by Sokal and Michener (1958). In this method the distance between two clusters is the square of the average Euclidean distance between pairs of observations, one in each cluster – when more than one paper is in a cluster, we use the average of the separate distances between the two clusters considered for combination. The hierarchical method begins with each observation as a separate cluster until all observations are allocated to one cluster. This is a one-way process – once an observation is allocated to a cluster it stays in that cluster and cannot move to another cluster. Because all the characteristics we use are in the same units we do not standardize their values.

From Figure 17 we note that the code for Stata scripts B.1 and B.2 are at almost opposite ends of the dendrogram. The B.1 script is not included in a cluster with any of the other scripts until

[16] The directory for each paper has been deliberately obscured.

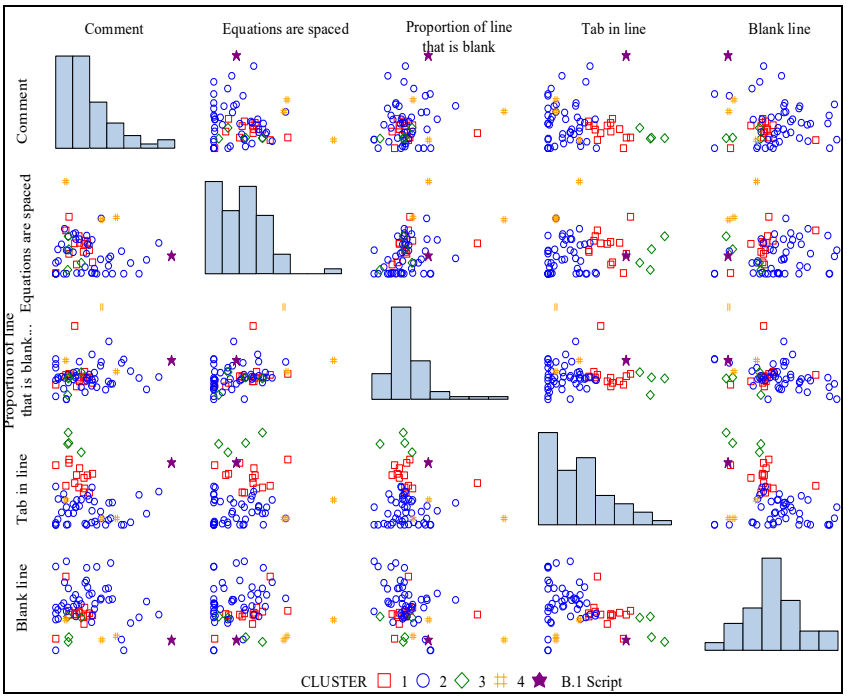
fewer than 5 clusters are defined. We have added a line where the distance between the clusters allows the definition of 5 clusters where script B.1 is the only member of its own cluster (# 5). Note that script B.2 is included in the most numerous cluster (# 2). The average characteristics across the programs used for each paper. These characteristics are: the proportion of the lines of code that are comments, the proportion of equations that use spaces between operators, the proportion of the characters in the lines that are blanks, the proportion of the lines that use tabs, and the proportion of the code that are blank lines.

Table 4 Average values of the characteristics by cluster.

CLUSTER	*Number of papers*	*% that are Comments*	*% of Equations with spaces*	*Blank % of line*	*% of lines with tabs*	*% of lines that are blank*
1	14	6.52	16.58	8.44	47.07	16.06
2	45	9.45	9.92	8.09	13.16	21.07
3	4	4.64	11.71	7.38	81.83	9.55
4	3	11.45	41.25	13.62	12.45	7.96
5	1	34.04	10.64	11.00	61.70	4.26
All	67	9.01	12.84	8.41	25.04	18.50

Table 4 provides the average percentage by cluster and over all the papers considered here. Note that cluster 5 includes more comments than other scripts but does not dominate in any of the other characteristics.

Figure 18 The scatter plot matrix of the values of the characteristics used in the cluster analysis with the allocation of cluster.



A scatter plot matrix of these measures is listed in Figure 18 along with the indicator of cluster membership for each paper's code. Note that the B.1 script has been identified with a filled in star.

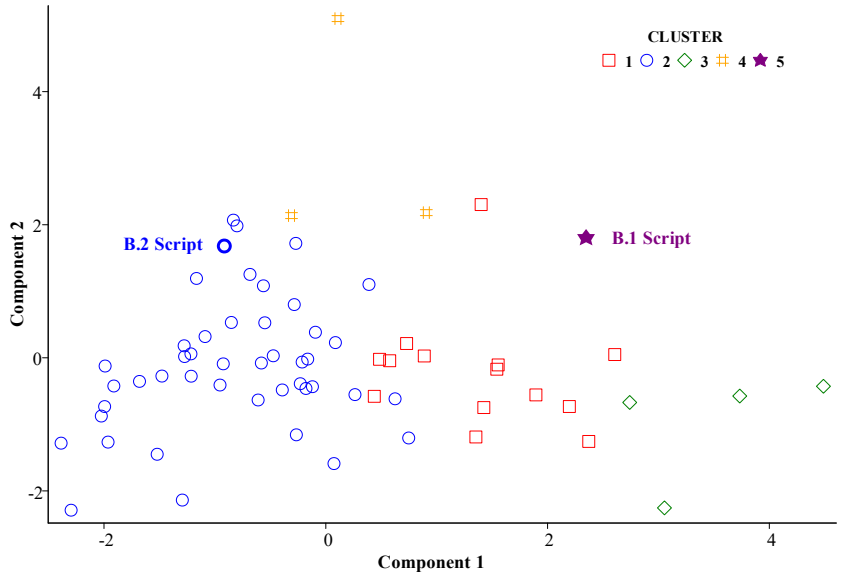
The correlation matrix of these characteristics listed in Table 5. Note that these characteristics are not highly correlated with each other. As an alternative to the cluster analysis we have computed the principle components of these characteristics based on the correlation matrix. Figure 19 plots the scatter plot of the first two components, that explain 57% of the variation in the data, with the identification of the cluster membership as another way of demonstrating the categorisation of these scripts that can be done with these measures of the code characteristics.

Table 5 The correlation matrix of the characteristics.

	% that is Comments	*% of Equations with spaces*	*Blank % of line*	*% of lines with tabs*	*% of lines that are blank*
comment	1.00	-0.07	0.16	-0.12	-0.12
eq_space	-0.07	1.00	0.37	0.11	-0.01
p_b_line	0.16	0.37	1.00	-0.07	-0.38
tab	-0.12	0.11	-0.07	1.00	-0.35
b_line	-0.12	-0.01	-0.38	-0.35	1.00

From Figure 19 we can identify that the B.1 script is portrayed as an outlier in this plot as we have seen in Figure 18 as well. This demonstrates that by use of the quantitative values we can compute from the code scripts we can identify which scripts are more likely to conform to a particular style of code.

Figure 19 The scatter plot of the first two principle components.



This analysis is not presented as a definitive categorisation of code scripts. We present it here as an example of the type of analysis that can be conducted to provide indications of differing

quality code. Cluster analysis is influenced by a myriad of assumptions that will influence the resulting grouping. The establishment of a rigorous method for the categorisation of code scripts is outside the scope of this study.

4.3 Software combinations

In addition to the analysis of the scripts of the *Stata* routines submitted for replication to the *American Economic Review* and *the AEA Papers and Proceedings* for 2020, we also replicated the approach taken by research conducted by Jon Fiva, Tuva Værøy and Federico Herrera (FVH) the results of which has been reported in tweets posted in 2019 and 2021 where they reported the distribution of the software used for the analysis in all papers published from January 2009 to July 2021 in the *American Economic Journal: Applied Economics* with programs and data available for replication. (see Appendix A for these results).

When conducting econometric research, it is important to keep in mind that different software have strengths for different tasks, and it is often important to be able to employ software that best suits the needs for the research. This may mean that data preparation may be conducted in a different program than the program used for estimation. Or that estimation of specific models may be best performed with specific programs available in one software package, while the graphic display of the results may be best performed in another. Much of the decision as to which software to use can come down to which software is most familiar to the researcher, has the better documentation, and/or has better examples one can adapt for the analysis. Most current software has the facility to read and write files from and to other format structures so that they may be read by one another. In our survey of the 2020 *AER* and the *AEA P&P* along with the FVH survey of over 10 years there appears to be a predominant use of *Stata*.

Table 6. The distribution of software in the 3,681 scripts for the *AER* and *AEA P&P* for 2020.

Software	%
Julia	1.11
Matlab[17]	27.19
Python	6.57
R	10.49
SAS	2.34
Stata	52.30
Total	100.00

[17] Many of the Matlab scripts are very short and are used to define small routines designated as functions called by other programs.

From Table 6 we note that of the 3,681 software scripts that we can identify in the replication folders for the 2020 *AER* and *AEA Papers and Proceedings* over 52% are *Stata* code with over 27% in *Matlab* and approximately 10.5% for *R*. With scripts for *Python*, *SAS* and *Julia* as the next most numerous.[18]

Table 7. The software employed for 232 paper/software combinations.[19]

Software	*Papers that use this software*	*Number of papers* *Papers that use other software as well*	*Papers that use this software alone*
Julia	3	2	1
Matlab	44	31	12
Python	16	14	2
R	28	20	8
SAS	6	6	0
Stata	135	46	86
Total	232	119	109

In Table 7 we provide the breakdown of software use by each paper for which we were able to download the replication files. From this table we can see that of the 170 papers for which we can download the corresponding data and code, 135 use *Stata*, 44 use *Matlab*, 28 use *R*, 16 use *Python*, 6 use *SAS* and 3 use *Julia*. Interestingly, we note that all papers that used *SAS* used other software as well. This table shows that approximately ⅔ of the papers that use *Stata* use it alone.

In Table 8 we show combinations of software employed in the 170 papers in both the journal and the proceedings that provided a folder of replication materials. The diagonal elements provide the total number of papers that use the software. The off-diagonal elements indicate the number of papers that use a combination of software. For example, of the 28 papers that have scripts for *R*, 16 also use at least one script for *Stata*. From Table 5 *Matlab* is the sole software used in only about ¼ of the applications in which it is used and from Table 6 we see that *Stata* was used in 26 cases of the 30 cases where other software was used with *Matlab*. Also, from Table 5, 14 of the 16 papers that used Python used other software as well and in Table 6 we see that in 11 cases the other routines were in *Stata*. In all 6 cases where *SAS* was used, it is used for data access with *Stata* as well as for one application in *R* and two in *Matlab*. We found no papers that used *SAS* with either *Julia* or

[18] This analysis was conducted by an analysis of the directories of the folders made available for the replication of results. We searched for the last delimiter of the files where: the delimiter *do*, *ado*, and *mata* were categorized as *Stata* files, the delimiter *R* was used for *R*, the delimiter *sas* was used for *SAS*, the delimiter *.m* was used for *Matlab*, the delimiter *jl* was used for *Julia*, and the delimiter *py* was used for *Python*.

[19] Of the 170 papers many use a combination of software scripts.

Python. From this table, we can conclude that besides being the most popular software overall, *Stata* is also the software most used with other software.

Table 8. Distribution of multiple software used by paper.[20]

Use	**Other software used**					
	Stata	***R***	***SAS***	***Matlab***	***Julia***	***Python***
Stata	**135**	16	6	26	1	11
R	16	**28**	1	7	1	6
SAS	6	1	**6**	2	0	0
Matlab	26	7	2	**44**	1	4
Julia	1	1	0	1	**3**	2
Python	11	6	0	4	2	**16**

Note that in addition to these software packages we found 74 papers that included files in *Microsoft Excel* format in their replication folders. Because the data manipulation and coding commands in these files are not stored in a separate script, we have not tabulated its use. We found one case in the *AEA Papers and Proceedings* where *Excel* was the only software used for data exposition but in all other cases it was used in conjunction with other software as a means of data storage. It is outside the scope of this paper to discuss the use of spreadsheet programs for replication of results. However, researchers should be aware of the potential pitfalls in their use. We strongly urge the reader to consult Broman and Woo (2018) for recommendations for best practice in the use of spreadsheets in statistical applications.

We also investigated the number of programs included for each paper. Table 7 provides the number of programs found for the 2020 *AER* and the number for the much shorter papers that appeared in the 2020 *AEA Papers and Proceedings*. Figure 20 provides the histograms of the number of programs by type of issue.

Table 9. The number of all programs used by papers in 2020 issues.

Issue	*Number*	*Mean*	*Median*	*Max*	*Min*
AEA P&P	78	6.69	3	78	1
AER	87	39.61	24	270	1

From Table 9 we can conclude that in more than 50% cases the 2020 *AER* papers use at least 24 scripts for the analysis and for seven papers there are over 100 separate scripts. Given that in some cases separate program scripts are combined by a master script that calls these as subprograms, this still requires a significant effort to establish what these elements of the analysis. Furthermore,

[20] Note that 32% of papers used more than one additional software routine and 8% used two or more.

for other researchers and students to be able to replicate or build on these results, well documented scripts are most important considering the scale of the code employed.

Table 10 The number of lines of *Stata* code over the 135 papers that uses *Stata* in either the *AER* or the *AEA P&P*.

Issue	*Number*	*Mean*	*Median*	*Max*	*Min*
AEA P&P	65	846.72	378	6,650	6
AER	70	6,801.99	4,419	42,661	84

Table 10 provides a count of the total number of lines of *Stata* code used over the 135 papers that used *Stata* in either the 2020 issues of the *AER* or the *AEA Papers and Proceedings*. Note that the average number of lines of code for papers in the *AER* is over 6,800 lines with a median of over 4,400 and a maximum of line that is almost 10 times the median. Even for the much shorter papers appearing in the *AEA P&P* the average number of lines is close to 850. To be able to check these massive amounts of code for errors as well as allow for replication requires significant documentation as well as the use of a clear coding style.

5. Recommendations

To summarize some of the rules to use in preparing code we recommend the following:

a. Only write a new 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 "reuse" code from other programs that perform a similar function. Try searching for other programs by googling for it first. However, when using code from other programs make sure you work through the code and understand every line. When borrowing significant parts of code make sure to reference the author.

b. Begin by getting a version of your program working first and then concern yourself about making it your ideal program (e.g. 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. One technique used by professional programmers is to get a routine to work then throw it out and start all over again now that you know where you are going. This may be more appropriate for "production routines" that are designed to be used over and over again.

d. Write a program that only does what it needs to do. Do not make it more complicated than is required. This often only results in errors being introduced into the code. In many cases it is worthwhile writing multiple smaller scripts that do specific tasks that are then called from a master file.

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, 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 show in Figure 12 with the *R* code. Try breaking large amounts of code up into logically arranged sections by grouping related lines of code (e.g. 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 out that can be performed sequentially into 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) and Orozco et al. (2020).

g. Do not sacrifice clarity in your program over optimization for speed or memory use. It is more important that it is clearly, cleanly written and easily understood. Most routines do not provide the appropriate results the first time. Be prepared to consider major rewrites.

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 prior to running the regressions and then this list or group used in each specification as was demonstrated in Section 3.4 on loops. 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 than 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 e.g. Gentzkow and Shapiro (2014) and Orozco et al (2020). Put the name as the first comment in the program so that when it is printed you can see which code you are viewing.

j. Be aware of the possibilities for the use of combinations of software. As we have documented, modern economic applications often require the use of a variety of software options.

We have found that the use of multiple software routines in the same project is very common and although the most common software is *Stata*, there are many instances where other software is used as well.

From our survey of program scripts submitted to the 2020 American Economic Review and the Papers and Proceedings of the American Economic Association we found that the number of scripts used varies from 1 to 270. That the in many cases multiple software were used although the predominant estimation software was *Stata*, there were many cases where multiple software were used for the same project. This implies that with the growing trend for papers to be written by multiple authors (as noted by Jones 2021), with multiple programs written in multiple software languages along with the need for replicable results, there is a necessity for the use of a clear and well documented programming style.

References

Broman, K. and K. Woo (2018), "Data Organization in Spreadsheets", *The American Statistician*, 72, 2-10.

Cox, N. (2005), "Suggestions on *Stata* programming style", *The Stata Journal*, 5, 560-566.

Cox, N. (2020), "Speaking *Stata*: Loops, again and again", *The Stata Journal*, 20, 999-1015.

Fiva, J., and T. Værøy (2019, Oct 11), "For applied economics @*Stata* is still the only game in town", https://twitter.com/JFiva/status/1182293282195460097?s=20

Fiva, J., T. Værøy and F. Herrera (2021, Aug 9), "For applied economics @*Stata* is still the only game in town", https://twitter.com/JFiva/status/1424679980538241025?s=20.

Gentzkow, M. and J. Shapiro,(2014), "Code and Data for the Social Sciences: A Practitioner's Guide", https://web.stanford.edu/~gentzkow/research/CodeAndData.pdf, downloaded 11/1/2021.

Hirschberg, J. and J. Lye, (2020) "Grading Journals in Economics: The ABCs Of The ABDC, *The Journal of Economic Surveys,* 34, 876-921.

Hirschberg, J, E. Maasoumi, and D. Slottje, (1991)"Cluster Analysis and the Quality of Life Across Countries", *Journal of Econometrics*, 50, 131-150.

_____, (2001), "Cluster of Attributes and Well-Being in the USA", *Journal of Applied Econometrics,* 16, 445-460.

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

Jones, B. (2021), "The Rise of Research Teams: Benefits and Costs in Economics", *Journal of Economic Perspectives*, 35, 191-216.

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

Nagler, J. (1995), "Coding style and good computing practices", *Political Science and Politics*, 28, 488–92.

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

Press, W., S. Teukolsky, W. Vetterling, B. Flannery, and M. Metcalf, (2007), *Numerical Recipes: The Art of Scientific Computing,* 3rd ed, Cambridge University Press.

Sokal, R. and Michener, C. (1958), "A Statistical Method for Evaluating Systematic Relationships", *University of Kansas Science Bulletin,* 38,1409–1438.

Vilhuber, L., Turrito, J., & Welch, K. (2020), "Report by the AEA data editor", *AEA Papers and Proceed-ings*, 110, 764–75. https://doi.org/10.1257/pandp.110.764

Vilhuber, L. (2020). "Reproducibility and replicability in economics", *Harvard Data Science Review*, 2(4).https://doi.org/10.1162/99608f92.4f6b9e67

_____ (2021), "AEA Data and Code Availability Policy", *AEA Papers and Proceedings,* 111, 818-823.

Wassberg, J. (2020), *Computer Programming for Absolute Beginners: Learn essential computer science concepts and coding techniques to kick-start your programming career*, Packet Publishing.

Appendix A. A survey of programs in *American Economic Journal: Applied Economics*

In a series of surveys conducted by Jon Fiva, Tuva Værøy and Federico Herrera (FVH) they collected information as to which software was used for the analysis in all papers published from January 2009 to July 2021 in the *American Economic Journal: Applied Economics* with programs and data available for replication. The results of this survey were first reported in a tweet Fiva and Værøy (2019) and updated in a subsequent tweet Fiva, Værøy, and Herrera (2021). The plots of the data reported in the 2021 tweet have been reproduced in Figure A.1. [21] They provide an indication of the frequency with which the four most used software was employed. From this plot we note that *Stata* was by far the most widely used of the four software alternatives used in these papers.

Figure A.1 Fraction of papers in *American Economic Journal: Applied Economics* using software by year.[22]

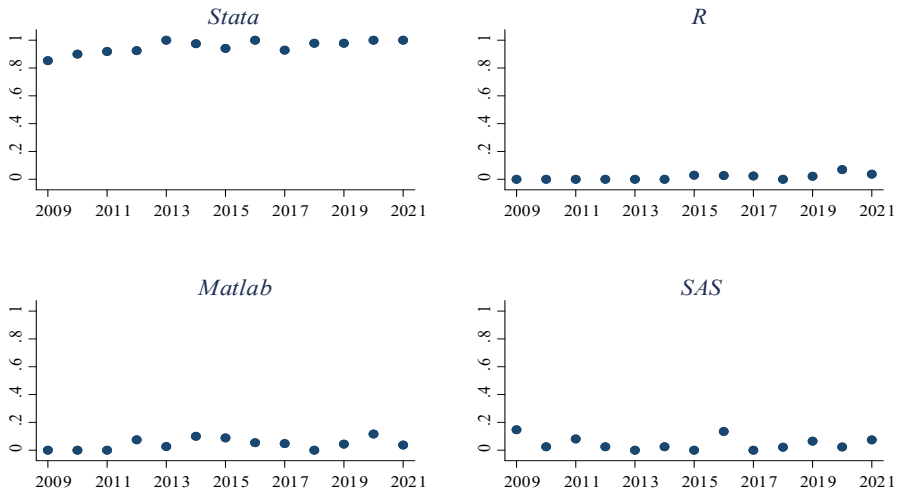


Table A.1 The distribution of software use in conjunction with other software in 504 papers.

Software	*Number of papers*		
	Papers that use this software	***Papers that use other software as well***	***Papers that use this software alone***
Excel	51	49	2
Matlab	23	23	0
Other	3	3	0
Python	1	1	0
R	8	6	2
SPSS	1	0	1
SAS	23	20	3
Stata	481	87	394

[21] These data were made available on Oct 3, 2021 from a private communication from Jon Fiva.
[22] Data from *AEJ:Applied* 2009-2021 (n=504). Graph by @JFiva & @FedeHerrera0.

Using the data from this analysis we constructed Table A.1. From this table it can be noted that we find a similar pattern of software usage to those papers we examine from the *AER* and *Papers and Proceedings* for 2020 in Table 5. Most of the papers use *Stata* and approximately 82% of these papers use only *Stata*. Also, in the majority of cases the uses of *Excel, R*, and *SAS* are also used in concert with *Stata*.

Appendix B Two Contrasting Examples of *Stata* code:

To demonstrate the variability in the appearance of equivalent *Stata* code we have included two scripts of *Stata* code below perform the same task. However, the first script sets out the code in a readable (some might say verbose) style while the second performs the same analysis using code that is abbreviated to make it more compact but also difficult to follow by employing the "write a little – get a lot" *Stata* mantra. Thus, a routine of 4 lines of code is equivalent to a 46-line routine and the number of characters (including spaces) used in the second is less than 15% the number used in the first.

Script B.1. Readable style code (1,213 characters including spaces)

```
/*
        name.do
        Do-file to run a regression and to test the hypothesis that
        the price parameters sum to zero.
*/
/*
        Make sure that there are no data sets in the memory
*/
        clear all
/*
        Read the comma delimitated data set beer.txt and label the variables
*/
        insheet using https://www.XXX/beer.txt

                label data "Beer demand and prices in logs"
                label variable q "Log of Beer consumed"
                label variable m "Log of income"
                label variable pb "Log of price of beer"
                label variable pl "Log of price of liquor"
                label variable pr "Log of price of other goods"
/*
        Make a scatter plot matrix of the data to check if there are outliers or other
        potential problems.
*/
        graph matrix q pb pl pr m
/*
        Run a regression on the log quantity
*/
        regress q pb pl pr m
/*
        Test if the demand equation is homogeneous of degree 1
*/
        test pb + pl + pr + m = 0
/*
        We can alternatively test the hypothesis by transforming the data so
        that a parameter is defined as the sum of the parameters - then the t-test
        if it is equal to zero should be equivalent to the test performed above.
*/
        gen z1 = pb
        gen z2 = pl - pb
        gen z3 = pr - pb
        gen z4 = m - pb
/*
        The coefficient for z1 will be the test statistic for the sum of the parameters
*/
        regress q z1 z2 z3 z4
```

Script B.2. Obtuse style code (165 characters including spaces)

```
#delimit;
clear all;insheet using "https://www.XXX/beer.txt";
gr mat q pb pl pr m;reg q pb pl pr m;te pb+pl+pr+m=0;
g z1=pb;g z2=pl-pb;g z3=pr-pb;g z4=m-pb;reg q z1-z4;
```

Table B.1 The beer.txt data used in Scripts B.1 and B.2

m	*pb*	*pl*	*pr*	*q*
10.13	0.58	1.94	0.10	4.40
10.19	0.82	1.99	-0.40	4.04
10.15	0.79	1.94	-0.19	4.16
10.21	0.77	1.97	-0.29	4.18
10.21	0.82	2.01	0.06	4.16
10.22	0.91	2.01	0.10	4.06
10.25	0.92	2.06	0.09	4.12
10.29	0.90	2.06	0.17	4.18
10.27	0.93	2.08	-0.13	4.06
10.31	1.00	2.07	0.26	4.15
10.33	0.96	2.09	0.16	4.19
10.32	1.05	2.11	-0.06	3.88
10.35	1.10	2.07	-0.09	4.02
10.35	1.17	2.12	0.10	3.87
10.39	1.13	2.09	0.41	4.04
10.39	1.13	2.13	0.16	3.94
10.42	1.13	2.17	0.17	3.99
10.43	1.21	2.18	0.31	3.95
10.45	1.20	2.18	0.42	4.02
10.49	1.23	2.15	0.14	3.95
10.46	1.28	2.18	0.33	3.96
10.49	1.27	2.18	0.47	3.79
10.53	1.31	2.19	0.55	4.06
10.51	1.31	2.21	0.30	3.94
10.55	1.31	2.20	0.31	3.99
10.51	1.34	2.22	0.34	3.91
10.56	1.35	2.23	0.48	3.84
10.57	1.38	2.25	0.52	3.85

Appendix C *Stata* routines added by the "ssc install" command.

Stata users often employ new or novel routines that have been written by third-parties. To install these programs, one uses a "ssc install" command.[23] By searching the scripts for each paper from the 2020 *AER* and the *AEA Papers and Proceedings* we tabulated which of these programs were loaded at least once in any script for each paper. The list below counts the number of papers where a routine was installed in scripts for at least 2 papers. Note some of these are installed in multiple scripts for the same paper. Interestingly, three of the top four routines (*estout, outreg2*, and *coefplot*) are programs designed to provide enhanced displays of regression results.

Table C.1 The most commonly added programs to *Stata* via "ssc install"[24]

Rank	Name	# of Papers
1	*estout*	18
2	*reghdfe*	12
3	*outreg2*	10
4	*coefplot*	8
5	*ftools*	7
5	*ivreg2*	7
6	*egenmore*	6
6	*ranktest*	6
7	*binscatter*	5
8	*carryforward*	4
8	*parmest*	4
8	*unique*	4
8	*winsor*	4
9	*ivreghdfe*	3
9	*moremata*	3
9	*regsave*	3
9	*spmap*	3
9	*xtivreg2*	3
10	*blindschemes*	2
10	*cdfplot*	2
10	*freduse*	2
10	*geodist*	2
10	*grstyle*	2
10	*gtools*	2
10	*mat2txt*	2
10	*palettes*	2
10	*psacalc*	2
10	*rdrobust*	2
10	*shp2dta*	2

[23] There are alternative methods for the addition of third-party routines (via *.ado* files) that we have not included here.

[24] Note that in several cases programs install multiple third-party *Stata* programs via a loop where a number of routines were listed in a global or local statement.

Script C.1. Stata script to install the programs listed in Table C.1

```
/*
        Program to install many of the more commonly added ados
        List the ados to add in local called required_ados
*/
        local required_ados ///
        estout reghdfe outreg2 coefplot ftools ivreg2 egenmore ranktest binscatter ///
        carryforward parmest unique winsor ivreghdfe moremata regsave spmap xtivreg2 ///
        blindschemes cdfplot freduse geodist grstyle gtools mat2txt palettes psacalc ///
        rdrobust shp2dta
/*
        Loop over the ados and install when not found
*/
        foreach x of local required_ados {
                capture findfile `x'.ado
                if _rc ssc install `x'
        }
```