MONTE CARLO EXPERIMENTS USING STATA: A PRIMER WITH EXAMPLES

LEE C. ADKINS AND MARY N. GADE

Abstract. Monte Carlo simulations are a very powerful way to demonstrate the basic sampling properties of various statistics in econometrics. The commercial software package **Stata** makes these methods accessible to a wide audience of students and practitioners. The purpose of this paper is to present a self-contained primer for conducting Monte Carlo exercises as part of an introductory econometrics course. More experienced econometricians that are new to Stata may find this useful as well. Many examples are given that can be used as templates for various exercises. Examples include linear regression, confidence intervals, the size and power of t-tests, lagged dependent variable models, heteroskedastic and autocorrelated regression models, instrumental variables estimators, binary choice, censored regression, and nonlinear regression models. Stata do-files for all examples are available from the author's website `http://learneconometrics.com/pdf/MCstata/`.

1. Introduction–Using Monte Carlo Simulations in Econometrics

Kennedy (2003, p. 24) defines a Monte Carlo study as "a simulation exercise designed to shed light on the small-sample properties of competing estimators for a given estimating problem." More importantly, it can provide "a thorough understanding of the repeated sample and sampling distribution concepts, which are crucial to an understanding of econometrics." The basic idea is to model the data generation process, generate sets of artificial data from that process, estimate the parameters of the model using one or more estimators, use summary statistics or plots to compare or study the performance of the estimators. Monte Carlo methods are used extensively by econometricians to study the finite sample performance of statistics, compare the power of tests, and to determine the effects of various statistical designs on the statistical properties of estimators that are commonly used.

Davidson and MacKinnon (2004, p. ix) are enthusiastic proponents of simulation methods in econometrics. In 2004 they state:

Date: July 25, 2012.

Key words and phrases. Monte Carlo, econometrics, Stata, simulations.

> Ten years ago, one could have predicted that [personal computers] would make the practice of econometrics a lot easier, and of course that is what has happened. What was less predictable is that the ability to perform simulations easily and quickly would change many of the directions of econometric theory as well as econometric practice.

It has also changed the way we teach econometrics to undergraduates and graduate students (e.g., see Barreto and Howland, 2006; Day, 1987; Judge, 1999). Kiviet (2012) has written a thorough discussion of Monte Carlo methods that is suitable for classroom use, employing EViews as the computing vehicle. In this paper, the use of Monte Carlo simulations to learn about the sampling properties of estimators in econometrics will be discussed and the usefulness of **Stata** will be demonstrated using a number of examples. As such, this paper can serve a tutorial suitable for classroom use.

Stata 12 (StataCorp, 2011) provides a powerful means of conducting Monte Carlo studies. **Stata** contains many estimators and its programming structure allows functions to be looped rather easily, which greatly expands the types of things that can be studied via simulation. For the most part, programming simulations in **Stata** is not difficult, but there are a few tricks and idiosyncracies to contend with. Many of these are documented below.

The best introduction to simulations using **Stata** is found in Cameron and Trivedi (2009, ch. 4). They concentrate on the use of **Stata**'s `simulate` command due to its simplicity. **Stata**'s `simulate` command runs a specified command or user written function (which **Stata** refers to as a *program*) a given number of times. The results can be output to another dataset. However, when the simulation finishes, the **Stata** dataset that is currently in memory is overwritten with the results of the simulation. This may not always be desirable, for instance you may want to conduct additional loops of the simulation using different model parameters and collect the simulated statistics into a single dataset.

In this paper, the slightly more basic, but versatile `postfile` command is used. The `postfile` command in **Stata** allows users to write the value of a computed statistic (or set of statistics) at each iteration of a simulation to a dataset. This allows one to loop through various experimental designs and collect the results in a single data file (see section 6).

There are some advantages of doing it this way rather than with `simulate`. Although `simulate` allows one to write the statistics from a Monte Carlo to a specified file, it does not allow that file to be appended with the results of subsequent runs of the simulation. That means if you change the value of one or more parameters in the simulation and rerun it, the results have to be written to a new dataset; joint analysis of the different parameterizations requires manually changing filenames and parameters, rerunning the simulation, and then merging the differently named datasets. That requires a lot of user intervention. With `postfile` no user intervention is required until the joint results are analyzed.

In addition, the syntax of `postfile` routines are more transparent. In most cases you do not have to write separate functions as you do with `simulate`; this allows you to avoid creating *return class objects*[1] and to minimize the use of **Stata**'s *macro* structure. In **Stata**, a macro permits the transport of data and results in and out of functions (and loops); inexperienced programmers find their use confusing and programming errors within them are more difficult to diagnose.[2] It should be said, however, their use cannot be avoided altogether because they are the principal way **Stata** moves objects (data and computations) in and out of functions.

We find the `postfile` and loop method no more difficult to use than `simulate` for most problems, and it is more versatile and slightly faster in execution. Evidence of this is presented in section 7. In order to compare the two ways of simulating in **Stata**, the first example below is developed using both methods.

In the next section, we briefly review the concept of a fully specified statistical model as laid out by Davidson and MacKinnon (2004) and present their basic recipe for using it in a Monte Carlo exercise. In section 3 we review some of the basics of **Stata**. In section 4 the basics of Monte Carlo simulations are summarized. Then in section 5 we go through a complete example of using **Stata** to study the coverage probability of confidence intervals centered at the least squares estimator. In subsequent subsections, we outline how to generate simulated models for some of the estimators used in a first econometrics course. In section 6 an example is presented that automates the changing of model parameterizations. Finally, our conclusion includes a few remarks on speed and a comparison of **Stata** to the free software **gretl**, version 1.9.9 (Cottrell and Lucchetti, 2012).

2. Fully Specified Statistical Model

The first step in a Monte Carlo exercise is to model the data generation process. This requires what Davidson and MacKinnon (2004) refer to as a fully specified statistical model. A **fully specified parametric model** "is one for which it is possible to simulate the dependent variable once the values of the parameters are known" (Davidson and MacKinnon, 2004, p.

[1]These are functions that return the result of a computation.

[2]Experience programmers love them because they conserve computer memory and when properly used, reduce the chances of making inadvertent programming errors, especially in very long programs.

19). First you need a regression function, for instance:

$$E(y_t|\Omega_t) = \beta_1 + \beta_2 x_t \tag{1}$$

where y_t is your dependent variable, x_t is the dependent variable, Ω_t is the current information set, and β_1 and β_2 are the parameters of interest. The information set Ω_t contains x_t as well as other potential explanatory variables that determine the average of y_t. The conditional mean of y_t given the information set could represent a linear regression model (e.g., as in equation (1)) or a discrete choice model (e.g., as in $E(y_t|\Omega_t) = F(\beta_1 + \beta_2 x_t)$ where F is a cumulative distribution function).

The actual values of $y_t|\Omega_t$ will differ from the mean by some amount, u_t, i.e., $y_t|\Omega_t - E[y_t|\Omega_t] = u_t$.

$$y_t|\Omega_t = \beta_1 + \beta_2 x_t + u_t \tag{2}$$

To complete the model requires a description of how the unobserved or excluded factors, u_t, behave.

A fully specified model requires an "unambiguous recipe" for simulating the model on a computer (Davidson and MacKinnon, 2004, p. 17). This means one needs to specify a probability distribution for the unobserved components of the model, u_t, and then use a pseudo-random number generator to generate samples of the desired size. Again following Davidson and MacKinnon (2004) the recipe is:

- Set the sample size, n.
- Choose the parameter values β_1 and β_2 for the deterministic conditional mean function (1).
- Obtain n successive values x_t, $t = 1, 2, \ldots, n$, for the explanatory variable. You can use real data or generate them yourself.
- Compute $\beta_1 + \beta_2 x_t$ for each observation.
- Choose a probability distribution for the error terms, u_t and choose any parameters that it may require (e.g., the normal requires a mean, usually zero, and a variance, σ^2).
- Use a pseudo-random number generator to get the n successive values of the errors, u_t.
- Add these to your computed values of $\beta_1 + \beta_2 x_t$ to get y_t for each observation.
- Estimate the model using the random sample of y, the given values of x, and the desired estimator.
- Save the statistics of interest.
- Repeat this a large number of times.

- Print out the summary statistics from the preceding step.

In the last step it is common to evaluate the mean of the sampling distribution, bias, variance, and the mean square error. Kennedy (2003, pp. 26-27) gives explicit formulae for these.

3. **Stata** Basics

Stata is a statistical package for managing, analyzing, and graphing data. It is available for a variety of computing platforms. It can be used either as a point-and-click application or as a command-driven package. **Stata**'s system of windows and dialog boxes provides an easy interface for newcomers to **Stata** and for experienced **Stata** users who wish to execute a command that they seldom use. The command language provides a fast way to communicate with **Stata** and to communicate more complex ideas. For Monte Carlo studies, the command language is a necessity.

3.1. Common Conventions.

In the few instances when the pull-down menus are employed the convention used will be to refer to menu items as `A>B>C` which indicates that you are to click on option A on the menu bar, then select B from the pull-down menu and further select option C from B's pull-down menu.

An important fact to keep in mind when using **Stata** is that its language is **case sensitive**. This means that lower case and capital letters have different meanings in **Stata**. The practical implication of this is that you need to be very careful when using the language. Since **Stata** considers x to be different from X, it is easy to make programming errors. If **Stata** gives you a programming error statement that you cannot decipher, make sure that the variable or command you are using is in the proper case.

A very powerful, but often confusing, feature of the **Stata** programming language is the macro concept. In **Stata**, macros can be thought of as the variables in a **Stata** program. A *macro* is a string of characters, called the *macroname*, that stands for another string of characters, called the *macro contents*. These can either be defined locally (exist only temporarily within a **Stata** program) or globally (permanently defined within the entire set of commands, which **Stata** calls a do-file). To access the contents of a macro, special syntax is required that identifies the name of the macro and tells **Stata** whether it is locally or globally defined. To substitute the macro contents of a global macro name, the macro name is typed (punctuated) with a dollar sign ($) in front. To substitute the macro contents of a local macro name, the macro name is typed (punctuated) with surrounding left and right single quotes (‘’). The main problem inexperienced users have with **Stata** macros is knowing when it is necessary to use them and then figuring out how to get the syntax correct–the ability to nest macros can make getting it right quite a trick. Since our paper is intended to serve the less-experienced user, we try to limit their use in this paper (though it is not entirely possible).

3.2. **Ways to Work in Stata.** There are several different ways to work in **Stata**. One can use the program through its built-in graphical user interface (GUI) where **Stata** collects input from the user through dialogs boxes, delivered by mouse clicks and a few keystrokes, to generate computer code that is executed in the background. The output is sent to the **Results Window**. After estimating a model one can access a set of *postestimation* commands that will perform various functions on the output of your estimation The postestimation commands give one access to various graphs, tests, analyses and a means of storing the results. Commands executed through the GUI are listed in the **Review window**, where they can be debugged and rerun using a few mouse clicks.

Stata commands can be collected and put into a file that can be executed at once and saved to be used again. **Stata** refers to these collections of code as *do-files* because they have .do filename extensions (e.g., MC.do). Even though you can run do-files from the command line, the **do-file editor** is the easiest way used to do Monte Carlo simulations in **Stata**. Start by opening a new do-file from the file menu. Select `Window>Do-file Editor>New Do-file Editor` from the pull-down menu to open the do-file editor. You can use “CRTL+8” as a keyboard shortcut.

One of **Stata**’s great strengths is the consistency of its command syntax. Most of **Stata**’s commands share the following syntax, where square brackets mean that something is optional and a `varlist` is a list of variables.

```
 [prefix:] command [varlist][if][in][weight][, options]
```

Some general rules:

- Most commands accept prefix commands that modify their behavior. One of the more common prefix commands is `by`, which tells **Stata** to repeat the given command on subsets of the data. See the example in section 6.
- If an optional `varlist` is not specified, all the variables are used.
- `if` and `in` restrict the observations on which the command is run.
- `options` modify what the command does.

- Each command's syntax is found in the online help and the reference manuals.

Type the commands you want to execute in the tabbed editor box using one line for each command. If you have a very long command that exceeds one line, use the triple backslash (\\\) as a **continuation command**.[3] Then, to save the file, use the "CRTL+S." If this is a new file, you will be prompted to provide a name for it.

4. Monte Carlo Basics

There are two main tools that can be used to do Monte Carlo simulation in **Stata**. It includes a special `simulate` command that will execute a program a given number of times and collect the outcomes in a dataset. It is specifically designed to make model simulation of the type suggested here very easy to do. The other method uses the `postfile` set of commands, which opens and posts observations to a dataset one at a time. Thus, with `postfile` one computes a statistic at each iteration of the Monte Carlo and adds it to a specified dataset. Used in conjunction with **Stata**'s flexible loop constructs it is both versatile and powerful. We find the `postfile` and loop method just as easy to use as `simulate` and it is slightly more versatile; it is the main method used below in the series of examples. Cameron and Trivedi (2009) provide a very nice chapter on the use of simulate and the discussion of its use here is limited. For purposes of comparison, though, two examples are repeated using `simulate`.

Using the `postfile` command to run Monte Carlo experiments requires the programming of loops. There are three ways to loop in Stata. The fastest uses the syntax:

```
1           forvalues lname = range {
2                   commands referring to ‘lname’
3           }
```

The braces that appear in lines 1 and 3 must be specified in the `forvalues` loop (as well as in each of the methods used for looping), and

(1) the open brace must appear on the same line as `forvalues`;

[3]If you are used to using software that requires explicit punctuation to indicate the end of a line of computer code, then you can set **Stata** to operate this way using the `#delimit` command. SAS programmers will be familiar with the convention that a line ends with a semi-colon. To accomplish this in **Stata**, use `#delimit ;` on the first line of your do-file and end each line with `;`.

(2) nothing may follow the open brace except, of course, comments; the first command to be executed must appear on a new line;
(3) the close brace must appear on a line by itself.

Notice that `‘lname’` in line 2 is enclosed in single quotes. It is a local macro that contains the values of `lname` in the proceeding line.

The `forvalues` loop has its limitations. If you need to create loops within loops (referred to as nested loops), you have to use the `while` looping construct.

```
        while lname = range {
                commands referring to ‘lname’
        }
```

There is also a `foreach` loop construct that can be used to loop over the elements of a list.

```
        foreach var of local varlist {
                do something based on ‘var’
}
```

You can use lists of numbers rather than variables (`numlist`), which proves to be especially useful in section 6 below.

Each simulation follows a similar recipe. The basic structure is:

(1) Open a dataset to use as a basis for the simulation or create values of the regressors using random numbers.
(2) Set a seed
(3) Define global variables for the number of monte carlo trials (`nmc`) and sample size (`nobs`). For instance, using a samples of size 100 and doing a 1000 simulations
```
   global nobs = 100
   global nmc = 1000
```
(4) Set the values of the parameters
(5) Initialize data that may need starting values in the loop, e.g., errors and conditional mean
(6) Create a temporary name, in this case we use the local macro name `sim`, using the `tempname` command. sim will hold statistics as they accumulate in the simulation.
```
  tempname sim
```

(7) Initiate the `postfile` command. Put the macro name `‘sim’` in single quotes. Assign variable names to the statistics that are being posted to `sim` and give a file name for placing the entire contents of `sim` once the simulation finishes. The `replace` option permits the `results.dta` file to be overwritten each time the simulation is executed.

```
postfile ‘sim’ vname1 vname2 .... vnamek using results, replace
```

(8) 8 Start a **quietly** block to enclose the loop. This suppresses output to the screen and speeds things up.

```
quietly {
```

(9) start the **forvalues** loop, indicating the desired number of samples (`$nmc`).

```
forvalues i = 1/$nmc {
```

(10) Replace random errors

(11) Replace conditional mean, y

(12) Estimate the model

(13) Compute the desired statistics (e.g., `b`, `se`, `pv`)

(14) Post the statistics into the local macro `‘sim’` using variable names. The name of each computed statistic that is to be posted is contained within parentheses.

```
post ‘sim’ (b) (se) (pv)
```

(15) Close the loop with a right bracket

```
}
```

(16) Close the temporary storage, `‘sim’`, using the `postclose` command

```
postclose ‘sim’
```

(17) Close the `quietly` block with a right bracket

```
}
```

The `quietly` command suppresses the printing results to the screen at each iteration of the loop. In the `forvalues` command that appears in step 9, notice that the macro `$nmc` has a dollar sign in front. Assuming that the number of Monte Carlo simulation to use is stored in the global macro `nmc` as in step 3, it must be referred to by its global macro name, `$nmc`.

Once the samples of your statistics from the Monte Carlo experiments have been obtained, what do you do with them? For instance how can you tell whether an estimator is biased? The simple answer is to use statistics. The Monte Carlo mean, $\bar{x}$, for a statistic should be approximately normally distributed, e.g., $\bar{x} \overset{a}{\sim} N(\mu, \sigma^2/n)$. Therefore $z = \sqrt{\text{NMC}}(\bar{x} - \mu)/\hat{\sigma}$ should be a standard normal. The statistic $\hat{\sigma}$ is simply the standard deviation that **Stata** prints for you when you use the `summarize` command; now, compute z and compare it to the desired critical value from the standard normal.

5. Examples

In this section of the paper, a series of examples is given. Each example illustrates important features of model specification and estimation and is typical of models taught in introductory econometrics courses. The first example is based on the classical normal linear regression model. One thousand (NMC=1000) samples are generated using Engel's food expenditure and income data (Koenker and Bassett, 1982) included with **gretl** (Cottrell and Lucchetti, 2012) and available from `http://www.learneconometrics.com/pdf/MCstata/engel.dta`. The slope and intercept parameters with each simulated set of data are computed using least squares, and 95% confidence intervals are constructed. The number of times the actual values of the parameters falls within the interval is counted. We expect that approximately 95% will fall within the computed intervals.

Subsequent examples are given, though with less explanation. These include estimating a lagged dependent variable model using least squares (which is biased but consistent). Autocorrelated errors are then added, making OLS inconsistent. A Breusch-Godfrey test to detect first order autocorrelation is simulated and can be studied for both size and power. Heteroskedasticity Autocorrelation Consistent (HAC) standard errors are compared to the inconsistent least squares standard errors in a separate example.

Issues associated with heteroskedastic models are studied as well. The simple simulation shows that least squares standard errors are estimated consistently using the usual formula when the heteroskedasticity is unrelated to the model's regressors. The final examples demonstrate the versatility of the exercises. An instrumental variables example is used to study the error-in-variables problem, instrument strength, and other issues associated with this estimator. A binary choice example is given and censored regression are given. The final example explores the properties of a nonlinear least squares estimator.

Although the errors in each of the examples are generated from the normal distribution, **Stata** offers other choices. These include uniform, Student's t, chi-square, beta, binomial, hypergeometric, gamma and Poisson.

5.1. **Classical Normal Linear Regression and Confidence Intervals.** We start with the linear model:

$$ln(y_t) = \beta_1 + \beta_2 ln(x_t) + u_t \tag{3}$$

where y_t is total food expenditure for the given time period and x_t is income, both of which are measured in Belgian francs. Let $\beta_1 = .5$ and $\beta_2 = .5$ and assume that the error, u_t iid $N(0, 1)$.

The model's errors take into account the fact that food expenditures are sure to vary for reasons other than differences in family income. Some families are larger than others, tastes and preferences differ, and some may travel more often or farther making food consumption more costly. For whatever reasons, it is impossible for us to know beforehand exactly how much any household will spend on food, even if we know how much income it earns. All of this uncertainty is captured by the error term in the model.

Stata is used to generate sequences of random normals to represent these unknown errors. Distributions other than the normal could be used to explore the effect on coverage probabilities of the intervals when this vital assumption is violated by the data. Also, it must be said that computer generated sequences of random numbers are not actually random in the true sense of the word; they can be replicated exactly if you know the mathematical formula used to generate them and the 'key' that initiates the sequence. This key is referred to as a **seed**. In most cases, these numbers *behave as if* they randomly generated by a physical process.[4]

A total of 1000 samples of size 235 are created using the fully specified parametric model by appending the generated errors to the parameterized value of the regression. The model is estimated by least squares for each sample and the summary statistics are used to determine whether least squares is biased and whether $1 - \alpha$ confidence intervals centered at the least squares estimator contain the known values of the parameters the desired proportion of the time.

The $(1 - \alpha)$ confidence interval is

$$P[b_2 - t_c se(b_2) \leq \beta_2 \leq b_2 + t_c se(b_2)] = 1 - \alpha \tag{4}$$

where b_2 is the least squares estimator of β_2, and that $se(b_2)$ is its estimated standard error. The constant t_c is the $\alpha/2$ critical value from the t-distribution and α is the total desired probability associated with the "rejection" area (the area outside of the confidence interval).

The complete do-file is found below. Start by opening a dataset. In the second line, a seed is chosen to initialize the stream of pseudo-random numbers. This ensures that the results can be replicated as shown here. In the next part of the do-file, the values of the parameters are set and then the natural logarithm of income, which is used as the independent variable

[4]If a seed is not specified, then **Stata** initializes it to be 123456789 each time the software is started.

in the simulation, is taken. In line 12 the systematic portion of the model is created. In lines 13 and 14 the error terms and the dependent variable are initialized and filled with missing values. This will allow us to use the `replace` statements in lines 20 and 21 to generate each new sample of `y` to use in the simulation. A temporary name called `sim` is created in line 15. This is the name that will hold the results posted in line 28 by the `post` command.

Performance of Confidence Intervals

```
 1  use "http://www.learneconometrics.com/pdf/MCstata/engel.dta", clear
 2  set seed 3213799
 3
 4  * Set the values of the parameters
 5  scalar constant = .5
 6  scalar slope = .5
 7  scalar sigma = 1
 8  scalar alpha = .05
 9
10  * Take the natural log of income to use as x
11  gen x = log(income)
12  gen Ey = constant + slope*x  /* mean function */
13  gen u =.          /* initiate u -- all missing values */
14  gen y =.          /* initiate y -- all missing values */
15  tempname sim
16
17  postfile ‘sim’ b se coverage using results, replace
18    quietly {
19      forvalues i = 1/1000 {
20      replace u = rnormal(0,sigma)
21      replace y = Ey + u
22      reg y x
23      scalar b = _b[x]
24      scalar se = _se[x]
25      scalar lb = b - se*invttail(e(df_r),alpha/2)
26      scalar ub = b + se*invttail(e(df_r),alpha/2)
27      scalar pv = slope<ub & slope>lb
28    post ‘sim’ (b) (se) (pv)
29    }
30  }
31    postclose ‘sim’
32
33  use results, clear
34  summarize
```

The heart of the simulation begins in line 17 with the `postfile` command. The set of `postfile` commands[5] was created to assist **Stata** programmers in performing Monte Carlo-type experiments. The `postfile` command declares the variable names and the filename of

[5] `postfile`, `post`, and `postclose`

a (new) **Stata** dataset where results will be stored. In order to use the `postfile` construct, a temporary name has to be created that will store the returns generated by the program. In line 17 the `postfile` command tells **Stata** the names to be given to the computations being posted to the file in line 28 and provides the filename into which the temporary file will be written.

The `postclose` command in line 31 declares an end to the posting of observations. After `postclose`, the new dataset contains the posted results and may be loaded using `use`.

The loop itself uses the `forvalues` loop construct. The counter is given the name `i` which is instructed to loop from 1 to 1000 using the syntax `1/1000`. Any subsequent references to the counter `i` must be via its (local) macro name `‘i’` (that is it must be enclosed in quotes). As a local macro the name `‘i’` is temporary and exists only as long as the loop is executing. It becomes empty once the loop finishes. This allows you to reuse the name `‘i’` in other loops. In order to suppress the output from all of the iterations, the loop is actually initiated using the `quietly` command. Just as in the template given in the preceding section, notice how the loop actually sits within the `quietly` command.

The remainder of the program is very simple. The samples are created, the regression is estimated and the desired results are computed using `scalar` computations. The logical statement in line 27 takes the value of 1 if the statements to the right of the equality are true. The ampersand (`&`) is the union of the two sets given on both of its sides. Therefore to be true, the slope must be less than the upper bound **and** greater than the lower. If not, `pv` is zero. Thus, the number of times the parameter falls within the interval can be counted and later averaged to estimate the coverage probability.

Running the do-file loads the data, initializes the parameters and simulates the model. The results are stored in the dataset called `results`. Be sure to issue the `clear` option or the current dataset won't be cleared from memory and the new one containing the simulated values cannot be loaded. The `summarize` command will give you summary statistics. At this point you could do additional analysis, like test for normality or plot a kernel density.

When writing a do-file, it is always a good idea to add comments. In **Stata** the star sign (`*`) at the beginning of a line can be used to make comments, e.g., lines 4 and 10. To insert comments after a **Stata** command, you can enclose it in `\* *\`, e.g., lines 12-14.

The results from the simulation appear below:

```
. summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           b |      1000    .5023573    .1497425  -.0332597   1.021356
          se |      1000    .1488163    .0069646   .1289464   .1706571
    coverage |      1000        .956    .2051977          0          1
```

Note that the average value of the slope is about 0.502, which is very close to the true value set in line 6. If you were to repeat the experiments with larger numbers of Monte Carlo iterations, you will find that these averages get closer to the values of the parameters used to generate the data. This is what it means to be unbiased. Unbiasedness only has meaning within the context of repeated sampling. In your experiments, you generated many samples and averaged results over those samples to get closer to the truth. In actual practice, you do not have this luxury; you have one sample and the proximity of your estimates to the true values of the parameters is always unknown.

The bias of the confidence interval (or coefficients) can be tested using $z = \sqrt{\text{NMC}}(\bar{x} - \mu)/\hat{\sigma}$ which in this case of `p1` is $\sqrt{1000}(.956 - .95)/0.2052 = 0.9246$. This is less than 1.96 and not significantly different from zero at 5%. The two-sided p-value is 0.36, which just confirms this. Increasing the number of Monte Carlo samples to 4000 produced $z = \sqrt{4000}(0.9485 - .95)/0.2205 = -0.3584$; this shows that increasing the number of samples increases the precision of the Monte Carlo results. Of course, we could let **Stata** do the arithmetic using:

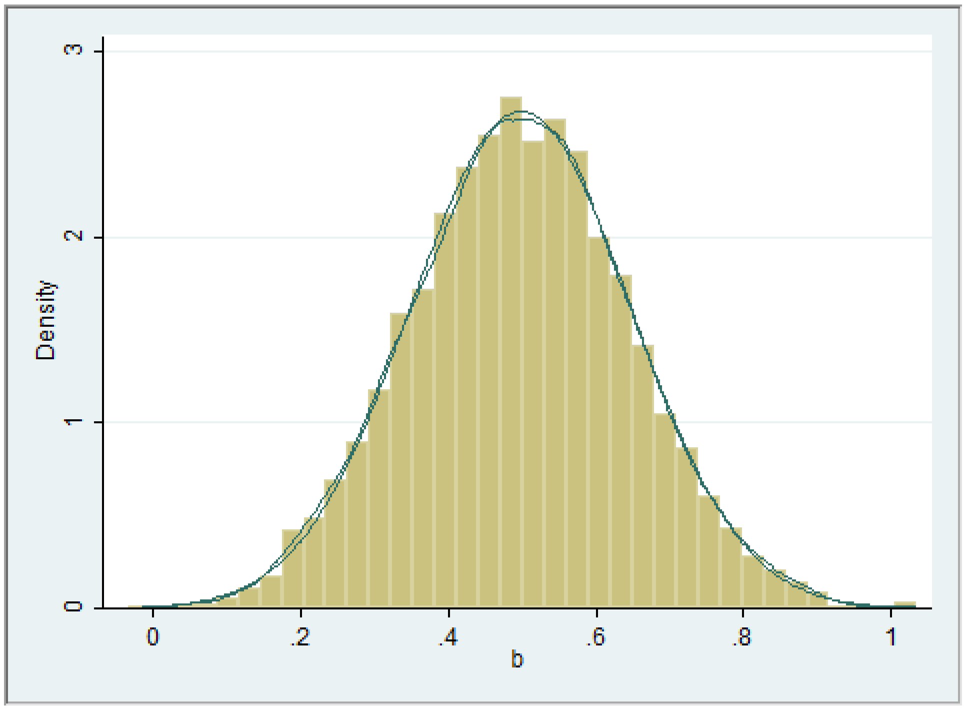
Monte Carlo test for size distortion of the 95% confidence interval

```
summarize coverage
scalar t = (r(mean)-.95)/(r(sd)/sqrt(r(N)))
scalar pval = 2*ttail(r(N),abs(t))
scalar list t pval
```

The saved results can be analyzed statistically or visualized graphically. From the command line typing

```
histogram b, normal kdensity
```

produces a histogram with a normal density plot. The `kdensity` option also adds a kernel density plot.



The kernel density plot and the normal plot are virtually indistinguishable.

Large sample properties like consistency and convergence in distribution can likewise be studied by increasing the number of observations, `nobs`.

5.2. **Using `simulate`.** In many cases you can use the `simulate` command to make things a bit easier. With `simulate`, you do not have to create loops and simple `return` statements are used to write results to the desired dataset. Otherwise, the two ways of working are very similar. Consider the confidence interval example done using `simulate`:

```
program drop _all
use "http://www.learneconometrics.com/pdf/MCstata/engel.dta", clear
set seed 3213799

* Set the values of the parameters
scalar constant = .5
scalar slope = .5
scalar sigma = 1
scalar alpha = .05
global scalar nmc = 1000

gen x = log(income)
gen Ey = constant + slope*x  /* mean function */
gen u =.           /* initiate u -- all missing values */
```

```
15 gen y =.          /* initiate y -- all missing values */
16
17 program regCI, rclass
18     replace u = rnormal(0,sigma)
19     replace y = Ey + u
20     reg y x
21     return scalar b = _b[x]
22     return scalar se = _se[x]
23     scalar lb = _b[x] - _se[x]*invttail(e(df_r),alpha/2)
24     scalar ub = _b[x] + _se[x]*invttail(e(df_r),alpha/2)
25     return scalar pv = (slope<ub) & (slope>lb)
26 end
27
28 simulate b=r(b) s=r(se) pv = r(pv), reps($nmc) \\\
29    saving(results, replace) nolegend nodots: regCI
30
31 use results, clear
32 summarize
33
34 summarize pv
35 scalar t = (r(mean)-.95)/(r(sd)/sqrt(r(N)))
36 scalar pval = 2*ttail(r(N),abs(t))
37 scalar list t pval
```

The top part of the program is unchanged. The data are loaded, a seed chosen, and parameters for the simulation set. We also initialized the variables that will be used in the simulation. The `simulate` command requires a program to host the set of commands to be simulated. In this example we create a program called `regCI`. The loop is no longer needed and there are no `postfile` and `post` commands. The loop is handled by `simulate` when the `regCI` function is called.

The `post` command is replaced by a series of `return scalar` commands in the body of the function. Each statistic that is to be sent to the dataset must be specified by a `return` statement. One thing requires caution, however. Notice that in lines 23 and 24 the accessors `_b[x]` and `_se[x]` were used rather than `b` and `se` that had been returned as scalars in the previous two lines. This is not accidental and it must be done this way. **The command `return scalar b` does not leave a copy of `b` in memory**; subsequent uses of `b` must refer to the accessor (or you need a separate line to define the scalar and then return it).

Running the simulation requires special syntax that can be tricky. Each of the statistics you want to collect in the simulation must be referred to by its `rclass` name; for instance, the `return` command in line 21 creates a return class object, a scalar named `b`, that is stored in `r(b)`. The `simulate` command in line 28 writes the `r(b)` object to a series called `b`. This is done similarly for the objects returned in `r(se)` and `r(pv)`. The options to `simulate` include the number of repetitions, `reps()`, a dataset name to which results are saved, a couple of options to reduce output to the screen, and finally a colon followed by the program name that is to be simulated.

Whether `simulate` is easier to use or more transparent then `postfile` is debateable. The fussiness of the `postfile` commands, which requires defining a temporary filename, `postfile`, `post`, the formal setup of a loop, and `postclose`, is replaced by another fussiness. With `simulate` one must write a program, include the properly named returns, call the program properly with `simulate` and its options, and populate the desired returns to match those in the program. The command `simulate` essentially collects decisions made when setting up a `postfile` and loop and puts those into the `simulate` calling function.

The `postfile` and `loop` commands can also be set up to use functions and in the examples below this is done. As you can see from the preceding examples, this is not necessary. For repeated use of the loop portion of the do-file, this can make things easier though.

5.3. **Antithetic Variates.** For the estimation of means and regression coefficients, antithetic draws from the desired distribution can reduce the number of simulations needed to demonstrate unbiasedness. Some care has to be used though as Davidson and MacKinnon (1992) point out, antithetics can be harmful if one's interest lies elsewhere, e.g., to study the variance of an estimator. See Train (2003, pp. 219-221) for an excellent discussion of antithetic variates and why they work.

Antithetic draws are perfectly negatively correlated with one another. For a symmetric distribution centered at zero like the standard normal, one simply draws a set of normal errors, u, and use these once to generate y. Then, reverse their signs, and use them again to generate another sample of y. The residuals in successive odd and even draws, `u` and `-u`, will be perfectly negatively correlated. This ensures that the regression errors are symmetrically centered around zero.

To use antithetic draws in the study of the least squares estimator of intercept and slope, replace line 22 of the preceding example with

Generating Antithetic Normal Samples

```
22 if mod(`i',2) != 0 {
23     replace u = rnormal(0,sigma)
24     }
25     else {
    replace u = -u
    }
```

First, notice that a new `if` command is used. This use of `if` should not be confused with the qualifier of the same name that is used in model statements (e.g., as in the syntax example in section 3.2). In this use, `if` evaluates the expression that appears to its right. If the result is true (nonzero), the commands inside the braces are executed. If the result is false (zero), those statements are ignored, and the statement (or statements if enclosed in braces) following the else is executed. The statement that is being checked is `mod(‘i’,2) != 0`. It uses the modulus command to produce the remainder of the division of the index `i` by 2. For odd integers the statement is true and the errors are generated and placed in `u`. When false (even integers), `u` is replaced by its negative value. `u` and `-u` are perfectly negatively correlated and hence make perfect antithetic variates for this example.

```
. summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           b |     10000          .5    .1492158  -.0806312   1.080631
          se |     10000     .148489    .0068318   .1252936    .174046
    coverage |     10000       .9486    .2208233          0          1
```

You can see that the least squares estimator has a mean equal to its theoretical value, 0.5.

5.4. **Autocorrelation and Lagged Dependent Variables.** In a linear regression with first order autocorrelation among the residuals, least squares is consistent provided there are no lags of the dependent variable used as regressors. Now, if the model contains a lagged dependent variable, least squares is inconsistent if there is autocorrelation. This is easy to demonstrate using a simple simulation.

Let the model be

$$y_t = \beta_1 + \beta_2 x_t + \delta y_{t-1} + u_t \qquad t = 1, 2, \ldots, n \tag{5}$$

$$u_t = \rho u_{t-1} + e_t \tag{6}$$

where ρ is a parameter and e_t is random error with mean zero and variance, σ_e^2. This model encompasses several possibilities. If $\delta = 0$ the model is the usual AR(1) model. If $\rho = 0$ then it is a lagged dependent variable model. If both $\delta = 0$ and $\rho = 0$ the model reduces to the classical linear regression implied by equation (2). Stability requires $|\rho| < 1$. If $|\delta| = 1$, then the dependent variable is nonstationary and should be modeled as a change $(y_t - y_{t-1})$ rather than a level. A partial adjustment model is implied when $|\delta| < 1$.

The complete model can be written as an ARDL(2,1):

$$y_t = \beta_1(1-\rho) + \beta_2 x_t + (\delta+\rho)y_{t-1} - (\rho\beta_2)x_{t-1} - (\rho\delta)y_{t-2} + e_t \tag{7}$$

There are many questions that can be answered by simulating this model. Among them, how biased is least squares when the LDV model is autocorrelated? Does this depend on the degree of autocorrelation? Suppose the model is autocorrelated but δ is small. Is least squares badly biased? Does HAC work effectively using standard bandwidth computations in various parameterizations? How powerful is the Breusch-Godfrey test of autocorrelation? How accurate is the approximation implied by use of the Delta theorem? and so on.

We have also introduced the ability to autocorrelate the independent variable, $x_t = \theta x_{t-1} + v_t$. This is useful because otherwise, x_t and y_{t-1} will not be contemporaneously correlated; hence, omitting the lagged dependent variable in the model will not create inconsistency in estimation of β_2. This can be verified quite easily by setting $\theta = 0$ in the simulation. As we have done in most of these simulations, we have set the value of the intercept paramter to be zero in the simulation, which can be done without losing any generality.

The do-file in **Stata** is shown below.

generating samples from LDV model with Autocorrelation

```
global nobs = 200
global nmc = 1000
set seed 10101
set obs $nobs
gen time = _n
tsset time

scalar theta = .8    /* autocorrelation in x      */
scalar beta  = 10    /* slope for x               */
scalar sigma = 20    /* variance of y             */
scalar delta = .5    /* coeff for lagged y        */
scalar rho   = .8    /* autocorrelation in errors */

gen x = rnormal()
replace x = theta*L.x + rnormal() in 2/$nobs
gen u = 0
gen y = 0

program regLDV, rclass
tempname sim
postfile `sim' b1 b2 b3 b4 se1 se2 se3 se4 using results, replace
quietly {
```

```
23     forvalues i = 1/$nmc {
24     * generate errors and samples of y
25       replace u = rho*L.u + rnormal(0,sigma) in 2/$nobs
26       replace y = beta*x+delta*L.y + u in 2/$nobs
27     * run the regression
28       reg y x                   /* b1 OLS, w/o LDV  */
29     * save the estimated slopes and its std error of b
30       scalar b1 = _b[x]
31       scalar se1 = _se[x]
32     * repeat for other estimators
33       reg L(0/1).y x            /* b2 LDV w/o Prais */
34       scalar b2 = _b[x]
35       scalar se2 = _se[x]
36       prais L(0/1).y x, twostep /* b3 LDV w/Prais   */
37       scalar b3 = _b[x]
38       scalar se3 = _se[x]
39       reg L(0/2).y L(0/1).x     /* b4 ARDL(2,1)     */
40       scalar b4 = _b[x]
41       scalar se4 = _se[x]
42  post ‘sim’ (b1) (b2) (b3) (b4) (se1) (se2) (se3) (se4)
43        }
44  }
45    postclose ‘sim’
46  end
47
48  regLDV
49  use results, clear
50  summarize
```

There are a couple of new things in this example. First, on lines 1 and 2 global macro variables are created; `nobs` allows you to set the number of observations each sample will contain and `nmc` the number of simulated samples to draw. Recall that reference to a global variable contained in a macro requires the $ prefix as shown. In line 4 the `set obs $nobs` command opens an empty dataset with room for `nobs` observations. The autoregressive independent variable is created in lines 14 and 15. Rather than initialize the errors and dependent variables using missing observations, they are populated with zeros in lines 16 and 17. Essentially, this allows the initial values of the time series to be set to zero.

A program is created named `regLDV` and given a return classification using the `rclass` option; this allows something calculated within the `regLDV` program to be carried outside of `regLDV`. It is required in order to post the calculations of the coefficients and standard errors to the designated dataset.

The `replace` command makes it easy to construct the sequences of time series without resorting to recursion. However, some care must be exercised to limit the observations to those available. Hence in lines 24 and 25 the `in 2/$nobs` qualifier is required. In this example $u_1 = 0$ and $y_1 = 0$ due to the way these series were initialized above. Subsequent values will be computed using the formulae in 25 and 26. It is simple and elegant.

The rest of the example is straightforward. A model is estimated and statistics collected and posted to a temporary memory location called `sim`. The model is estimated in several ways: 1) least squares with y_{t-1} omitted 2) least squares with y_{t-1} included, but ingnoring the autocorrelation in the errors 3) the LDV model with a two-step Prais-Winsten transformation and 4) the correctly specified ARDL(2,1) model. Even though we did not include a constant in the data generation, one is estimated (the estimate of which should be not statistically significant from zero). When finished the contents of `sim` are written to permanent storage in the file *results.dta*, which is available for subsequent analysis.

The simulation reveals

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          b1 |      1000    15.01713    9.167224  -12.40116   44.45594
          b2 |      1000    5.307263     1.72269  -.9130358   10.61727
          b3 |      1000    8.235005    1.635338   2.534937   12.85917
          b4 |      1000     9.96528    1.445586   5.661244   14.54532
```

Based on 1000 simulations, it is pretty clear that only the ARDL(2,1) estimator (`b4`) is getting anywhere near the correct value of 10. The high variance associated with `b1` is due to the high value of ρ in the simulation.

5.4.1. *Breusch-Godfrey Test.* To study the size or power of the Breusch-Godfrey test a few more statements have to be added. These are shown in below. Note, **Stata** can perform this test using `estat bgodfrey, lags(1)`, which is an `rclass` function. One of the results written to memory is the p-value of the test statistic and it is held in a return class object `r(p)`. The return, `r(p)`, is classified as a `matrix` rather than a `scalar` because additional lags may be specified when the function is called. If so, these will appear in a vector of results. The logical statement simply takes the value 1 when the p-value is less than α and zero otherwise. The mean will give you the overall rejection rate of the test. When $\rho = 0$ this should be close to α.

Size and Power of the Breusch-Godfrey Test for AR(1)

```
      reg y x L.y
      estat bgodfrey, lags(1)
          matrix p1 = r(p)
          scalar rej1 = (p1[1,1]<alpha)
      reg L(0/2).y L(0/1).x
          estat bgodfrey, lags(1)
          matrix p2 = r(p)
          scalar rej2 = (p2[1,1]<alpha)
post ‘sim’ (p1[1,1]) (rej1) (p2[1,1]) (rej2)
```

The errors of the ARDL(2,1) model are not autocorrelated, and the second use of the test verifies this. Letting $\rho = .4$ the rejection rate for the test for the regression $y_t = \beta_1 + \beta_2 x_t + \delta y_{t-1} + u_t$ is compared to that of the ARDL(2,1). The result is:

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          p1 |      1000    .0192006    .0658037   4.69e-10   .8686352
        rej1 |      1000        .913    .2819761          0          1
          p2 |      1000    .4883838    .2947545   .0012266   .9998515
        rej2 |      1000        .057    .2319586          0          1
```

When the residuals are autocorrelated, the null hypothesis is rejected 91.3% of the time. The power is quite high. The residuals of the correctly specified ARDL(2,1) are not autocorrelated and the rejection rate is 5.7%, which is very close to the nominal level of the test.

5.4.2. *HAC Standard Errors.* The experimental design of Sul et al. (2005) is used to study the properties of HAC. They propose a model $y_t = \beta_1 + \beta_2 x_t + u_t$, where $u_t = \rho u_{t-1} + e_t$ and $x_t = \rho x_{t-1} + v_t$ with $\beta_1 = 0$ and $\beta_2 = 1$ and the innovation vector (v_t, e_t) is independently and identically distributed (i.i.d.) $N(0, I_2)$ for $n = 10000$. The do-file appears below.

Coverage probabilities of HAC and the usual OLS confidence intervals

```
global nobs = 10000
global nmc = 1000
set seed 10101
set obs $nobs

* Set the values of the parameters
scalar beta = 1
scalar sigma = 1
scalar rho = .8
scalar alpha = .05

* generate n observations on x and time.  Set as time-series
gen x = rnormal()
```

```
14 gen u = 0
15 gen y = 0
16 gen time = _n
17 tsset time
18 global nwlag = ceil(4*($nobs/100)^(2/9))
19
20 program regLDV_hac, rclass
21 tempname sim
22 postfile ‘sim’ b1 se1 b2 se2 c_ls c_nw using results, replace
23 quietly {
24     forvalues i = 1/$nmc {
25     * generate errors and samples of y
26       replace u = rho*L.u + rnormal(0,sigma) in 2/$nobs
27       replace x = rho*L.x + rnormal(0,1) in 2/$nobs
28       replace y = beta*x + u
29     * run the regression
30       reg y x
31     * save the estimated slope and its std error of b
32       scalar b1 = _b[x]
33       scalar se1 = _se[x]
34     * Confidence Interval
35       scalar lb = b1 - se1*invttail(e(df_r),alpha/2)
36       scalar ub = b1 + se1*invttail(e(df_r),alpha/2)
37       scalar pv = beta<ub & beta>lb
38     * run the regression
39       newey y x, lag($nwlag)
40     * save the estimated slope and HAC std error of b
41       scalar b2 = _b[x]
42       scalar se2 = _se[x]
43     * HAC Confidence Interval
44       scalar lb2 = b2 - se2*invttail(e(df_r),alpha/2)
45       scalar ub2 = b2 + se2*invttail(e(df_r),alpha/2)
46       scalar pv2 = beta<ub2 & beta>lb2
47 post ‘sim’ (b1) (se1) (b2) (se2) (pv) (pv2)
48       }
49 }
50 postclose ‘sim’
51 end
52
53 regLDV_hac
54 use results, clear
55 summarize
```

The commands are very similar to the ones in the preceding example. A notable difference is the creation of the global constant in line 18 to specify the desired lag length for the HAC kernel. The `replace` command is used as before, and once again the independent variable is autoregressive. Also, the `newey` command is used to estimate the model by least squares with the HAC standard errors in line 39.

Although some patience is required while the simulation runs, the results based on samples of size 10,000 are revealing.

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          b1 |      1000    1.000417    .0215846   .9309496   1.068051
         se1 |      1000    .0100025    .0002108   .0094095   .0108052
          b2 |      1000    1.000417    .0215846   .9309496   1.068051
         se2 |      1000    .0194082    .0007425   .0173165   .0216077
        c_ls |      1000        .633    .4822277          0          1
-------------+--------------------------------------------------------
        c_nw |      1000        .924    .2651307          0          1
```

Even with a very large sample ($n = 10,000$), the confidence interval based on the HAC standard error is still too small on average to cover the slope at the nominal rate. It only covers the parameter 92.4% of the time. The usual least squares confidence interval is terrible, only covering the slope in 63.3% of the simulated samples. You can also see that the standard deviation of the least squares estimator, `b1` or `b2`, is 0.0215846. The average value of the HAC standard error is slightly smaller, 0.0194982, which is consistent with coverage at less than the nominal rate. When n=1000, still a large sample in most people's minds, the coverage rate is only 0.9. The HAC standard errors are certainly an improvement over the usual ones, but one needs very large samples for them to closely approximate the actual LS standard error. This is essentially the same finding as Sul et al. (2005).

5.5. Heteroskedasticity.

The data generation process is modeled $y_t = \beta_1 + \beta_2 x_t + u_t$, where u_t iid $N(0, \sigma_t^2)$ with $\sigma_t^2 = \sigma^2 \exp(\gamma z_t)$, $z_t = \rho_{xz} x_t + e_t$, e_t iid $N(0,1)$ and $\beta_1 = 0$ and $\beta_2 = 1$ and the innovation vector (u_t, e_t) is independently distributed for $n = 100$.

In this example one can demonstrate the fact that heteroskedasticity is only a 'problem' for estimating least squares standard errors when the model's error variances are correlated with the regressors. Setting $\rho_{xz} = 0$ leaves the errors heteroskedastic, but not with respect to the regressor, x. One can verify that the standard errors of OLS are essentially correct. As ρ_{xz} deviates from zero, the usual least squares standard errors become inconsistent and the heteroskedastic consistent ones are much closer to the actual standard errors.

Standard Errors Using HCCME

```
* set numbers of observations and number of MC samples
global nobs = 200
global nmc = 1000
```

```
set obs $nobs
set seed 10101

* Set the values of the parameters
scalar slope = .5
scalar sigma = 1
scalar rho_xz = .99
scalar gam = .5
scalar alpha = .05

* generate n observations on x and a correlated variable z
gen x = 10*runiform()
gen z = rho_xz*x+rnormal(0,sigma)
gen y =.
gen sig =.

program regHET, rclass
tempname sim
   postfile `sim' b se se_r p_ls p_r using results, replace
   quietly {
    forvalues i = 1/$nmc {
    replace sig = exp(gam*z)
        summarize sig
        replace sig = (1/(r(mean)))*sig
        replace y = slope*x + rnormal(0,sig)
    reg y x
          scalar b = _b[x]
          scalar se = _se[x]
          scalar lb = b - se*invttail(e(df_r),alpha/2)
          scalar ub = b + se*invttail(e(df_r),alpha/2)
          scalar pv = slope<ub & slope>lb

          reg y x, vce(hc3)
          scalar ser = _se[x]
          scalar lb = b - ser*invttail(e(df_r),alpha/2)
          scalar ub = b + ser*invttail(e(df_r),alpha/2)
          scalar pvr = slope<ub & slope>lb
        post `sim' (b) (se) (ser) (pv) (pvr)
           }
  }
  postclose `sim'
end

regHET
use results, clear
summarize
```

The only trick used here comes in line 27. As the parameter, γ changes, the overall variance of the model will change. Hence in this line the overall variance in the model is normalized around the mean.

Stata has options for the different versions of the Heteroskedasticity Consistent Covariance Matrix Estimator (HCCME). In line 36, `vce(hc3)` scales the least squares residuals $\hat{u}_t/(1-h_t)^2$.

The parameter ρ_{xz} controls the degree of correlation between the regressor, `x`, and the variable `z` that causes heteroskedasticity. If $\rho_{xz} = 0$ then the degree of heteroskedasticity (controlled by γ) has no effect on estimation of least squares standard errors. That is why in practice `z` does not have to be observed. One can simply use the variables in `x` that are correlated with it to estimate the model by feasible GLS. This would make an interesting extension of this experiment. Whether this is more precise in finite samples than least squares with HCCME could be studied.

5.6. **Instrumental Variables.** The statistical model contains five parameters: β, σ, σ_x, γ, and ρ. The data generation process is modeled $y_t = \theta + \beta x_t + u_t$, where u_t iid $N(0, \sigma_t^2)$, $x_t = \gamma z_t + \rho u_t + e_t$, e_t *iid* $N(0, \sigma_e^2)$ and the innovation vector (u_t, e_t) is independently distributed for $n = 100$. Without loss of generality, the intercept is set to zero, $\theta = 0$. The γ controls the strength of the instrument z_t, ρ controls the amount of correlation between x_t and the errors of the model u_t, σ_e^2 can be used to control the relative variability of x_t and u_t as in an error-in-variables problem.

Instrumental Variables

```
global nobs = 200
global nmc = 1000
set seed 10101
set obs $nobs

scalar slope = 1     /* regression slope */
scalar sigma = 10    /* measurement error in y */
scalar sige = .1     /* amount of measurement error in e */
scalar gam = 1       /* instrument strength */
scalar rho = .5      /* correlation between errors  */
scalar alpha=.05     /* test size */

gen z = 5*runiform()
gen y = .
gen x = .
gen u = .

program regIV, rclass
```

```
tempname sim
  postfile ‘sim’ b biv se se_iv p_ls p_iv using results, replace
  quietly {

    forvalues i = 1/$nmc {
        replace u = rnormal()
        replace x = gam*z+rho*u+rnormal(0,sige)
        replace y = slope*x + u
        reg y x
          scalar b = _b[x]
          scalar se = _se[x]
          scalar lb = b - se*invttail(e(df_r),alpha/2)
          scalar ub = b + se*invttail(e(df_r),alpha/2)
          scalar pv = slope<ub & slope>lb

        ivreg y (x=z)
          scalar biv = _b[x]
          scalar seiv = _se[x]
          scalar lb = biv - seiv*invttail(e(df_r),alpha/2)
          scalar ub = biv + seiv*invttail(e(df_r),alpha/2)
          scalar pvr = slope<ub & slope>lb

        post ‘sim’ (b) (biv) (se) (seiv) (pv) (pvr)
        }
  }
  postclose ‘sim’
end

regIV
use results, clear
summarize
```

5.7. **Binary Choice.** The statistical model contains only two parameters: β and σ. The latent variable is modeled $y_t^* = \mu + \beta x_t + u_t$, where u_t iid $N(0, \sigma_t^2)$. The observed indicator variable $y_t = 1$ if $y_t^* > 0$ and is 0 otherwise. By changing the parameters μ and β one can alter the proportion of 1s and 0s in the samples. Changing σ should have no effect on the substance of the results since it is not identified when the other two parameters are.

This simulation is designed to illustrate two things. First, parameter significance tests from ML estimation of a probit and least squares estimation of the the linear probability model usually agree. The exceptions occur, as Horrace and Oaxaca (2006) point out, if the least squares estimated index for any observation lies outside of the 0/1 interval. This is easy to check. If the only goal is to determine whether x affects the probability of observing $y = 1$, then the least squares estimator of the linear probability model may have better small sample properties than the probit MLE. Second, if estimating the average marginal effect is of interest, least squares of the LPM offers a close approximation, is much easier to produce, and unambiguous in interpretation. The following experiment demonstrates their similarity.

The do-file is:

Probit and Linear Probability Model

```
global nobs = 200
global nmc = 500
set seed 10101
set obs $nobs

scalar slope = 0    /* regression slope */
scalar sigma = 1    /* measurement error in y */
scalar alpha=.05    /* test size */

gen x = runiform()
gen u = .
gen y = .

program regBC, rclass
tempname sim
  postfile `sim' t t_p pv pv_p b ame prop using results, replace
  quietly {
    forvalues i = 1/$nmc {
        replace u = rnormal(0,sigma)
        replace y = slope*x + u
        replace y = (y>0)
        summarize y
        scalar prop = r(mean)
        reg y x, vce(hc3)
          scalar b = _b[x]
          scalar t = _b[x]/_se[x]
          scalar pv = abs(t)>invttail(e(df_r),alpha/2)
        probit y x
          scalar tp = _b[x]/_se[x]
          scalar pv_p = abs(tp) > invnormal(1-alpha/2)
          margins, dydx(x)
          matrix m = r(b)
      scalar m = m[1,1]
        post `sim' (t) (tp) (pv) (pv_p) (b) (m) (prop)
        }
  }

  postclose `sim'
end

```

```
41 regBC
42 use results, clear
43 summarize
```

In this exercise the data generation process is handled easily in lines 19, 20, and 21. The proportions of 1s to 0s is saved as a scalar in line 23 after computing its summary statistics in line 22. Then both least squares and the probit MLE are used to estimate the parameters of the binary choice model. In most econometrics courses great effort is made to discuss the fact that marginal effects in probit models are different at each observation. This creates angst for many. In lines 31-32 the average marginal effects are computed and saved for the regressor, x. The results show that in fact the AME of the probit model (`ame`) and the slope from least squares (`b`) are nearly identical.

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           t |       500    .0205409    .9701433  -3.010698   2.827217
         t_p |       500    .0209067    .9692976  -2.921262   2.780452
          pv |       500        .044    .2053005          0          1
        pv_p |       500        .044    .2053005          0          1
           b |       500    .0026487     .123441  -.3727096   .3533991
-------------+--------------------------------------------------------
         ame |       500    .0026391    .1229774  -.3691716   .3499526
        prop |       500      .49916    .0348859         .4        .61
```

In 500 samples the size of the t-test is the same for the two estimators. The marginal effect in the LPM is 0.00265 and for probit it is 0.00264. There is slightly more variation in the t-ratio for the LPM than for probit, but the difference is very small. For this design, it is a toss-up. Increasing the slope to equal 1 increases the proportion of 1s in the sample to 0.69. The power of the t-test is the same to two decimal places, 0.87. The slope and AME are 0.3497 and 0.3404, respectively. The more unbalanced the sample, the larger the divergence in these estimates of the marginal effect. Still, with nearly 70% of the sample being observed as a 1, the two are very close (as are their standard deviations and range). Also, one could confirm the superiority of using the MLE when the index of the LPM strays outside of the logical (0,1) interval.

5.8. **Tobit.** To generate samples for tobit estimation requires an additional line in the probit code. Recall that in the tobit model, all of the latent variables that are less than zero are censored at zero. The continuous values above the threshold are actually observed. Therefore the probit do-file can serve as a basis for this model. Lines 19-21 of the probit example are replaced by:

```
19      replace ystar = const + slope*x + rnormal(0,sigma)
20          replace y = (ystar>const)
21          replace y = y*ystar
```

The constant in this model serves as the threshold for tobit. Computing the proportions of 1s in this model enables you to keep track of how much censoring occurs. Experiments could be conducted by changing the distance between actual threshold and zero and the tobit estimator could be compared to least squares. This is another poorly understood feature of this model. Choosing the proper threshold is critical for proper performance of the MLE. For instance, add a constant of 20 to the right-hand side of the equation in line 19, which generates the latent variable y_t^* (`ystar`), and change the actual threshold in line 20 from 0 to 20. Increase the value of `sigma`, to 10 for instance, to create more variation in `y`. Re-run the simulation and see that least squares is now far superior to the tobit MLE, which erroneously assumes that the threshold is 0.

5.9. **Nonlinear Least Squares.** **Stata** can also be used to estimate more complicated models in a simulation. It includes generic routines for estimating nonlinear least squares and maximum likelihood estimators, the properties of which can also be studied in this way. In the following example, nonlinear least squares is placed within a loop and the estimates are collected in matrices and output to a data set. This simple example could be used as a rough template for more advanced problems.

Consider the nonlinear consumption function

$$C_t = \alpha + \beta Y_t^{\gamma} + u_t \tag{8}$$

where C_t is consumption and Y_t is output. Using data from Greene (2000), the model is estimated to obtain starting values, parameters are set, and simulated samples are drawn. In the first snippet of code, the data are opened in line 1, the parameter values set in lines 3-6, starting values are acquired by least squares (12-15). The `nl` command is used in lines 23 and 24 (using the `///` line continuation). Parameters are enclosed in braces { } and given starting values using the assignment operator, `=`. The parameters in nonlinear models are accessible in postestimation, but the syntax is a little different. The `_b[varname]` convention used in linear models has been replaced by `_b[paramname:_cons]`. The `coeflegend` option can be used after the `nl` command to find the proper names for the parameters. To verify that you have identified the parameters correctly, run the nonlinear least squares regression again using the `coeflegend` option. This suppresses much of the output that you ordinarily want, but it does produce a legend that identifies **Stata**'s names for each of the parameters.

```
use "http://www.learneconometrics.com/pdf/MCstata/greene11_3.dta", clear

scalar A = 180
scalar B = .25
scalar G = 1
scalar sigma = 2

global nmc = 1000
gen c0 = A + B*y^G

* Starting values
reg c y
scalar alpha0 = _b[_cons]
scalar beta0 = _b[y]
scalar gamma0 = 1

program regNLS, rclass
tempname sim
  postfile `sim' alpha beta gamma using results, replace
   quietly {
    forvalues i = 1/$nmc {
    replace c = c0 + rnormal(0,sigma)
    nl (c = {alpha=alpha0} + {beta=beta0} * y^{gamma=gamma0}), ///
    variables(c y)
    post `sim' (_b[alpha:_cons]) (_b[beta:_cons]) (_b[gamma:_cons])
        }
  }
  postclose `sim'
end

regNLS
use results, clear
summarize
```

The results of 1000 iterations from the simulation are:

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       alpha |      1000    179.8951    6.768631   157.7997   199.7664
        beta |      1000    .2527278    .0389655   .1565982   .4028383
       gamma |      1000    1.000128    .0183172   .9431589   1.056078
```

The results indicate that the nonlinear least squares estimator has done quite well, on average. It is statistically unbiased.

6. Changing the design using loops

In some circumstances one wants to study the properties of an estimator or test statistic under different parameterizations of the model. The methods discussed here can easily be adapted to this and this is where the benefits of using the `postfile` commands can really be seen. Consider the instrumental variables example in section 5.6 and suppose one wants to examine how instrument strength affects the bias of the IV estimator. One design consideration is to keep the overall variability in the endogenous regressor constant. This is desirable because it keeps the overall fit between y and x relatively constant as instruments get stronger. Note, $var(x) = \gamma^2 var(z) + \rho^2 var(u) + var(e)$. The revised program now includes a extra loop over which the simulation will execute.

In this version we add a computation for the F-statistic used to test the strength of the instruments, saving it as a scalar in line 11, and save the overall fit of the reduced form regression, R^2, as a scalar in line 14. The program follows.

```
 1 program regIV, rclass
 2 tempname sim
 3   postfile ‘sim’ gam r2 F b biv se se_iv p_ls p_iv using results, replace
 4   quietly {
 5   foreach gam of numlist 0.025 0.0375 0.05 0.1 0.15 0.2 0.5 {
 6     forvalues i = 1/$nmc {
 7         replace u = rnormal()
 8         replace x = ‘gam’*z+rho*u+rnormal(0,sige)
 9         replace y = slope*x + u
10         reg x z                      /* reduced form        */
11         scalar F = (_b[z]/_se[z])^2  /* instrument strength */
12 
13         reg y x
14           scalar r2=e(r2)            /* overall fit         */
15           scalar b = _b[x]
16           scalar se = _se[x]
17           scalar lb = b - se*invttail(e(df_r),alpha/2)
18           scalar ub = b + se*invttail(e(df_r),alpha/2)
19           scalar pv = slope<ub & slope>lb
20 
21         ivreg y (x=z)
22           scalar biv = _b[x]
23           scalar seiv = _se[x]
24           scalar lb = biv - seiv*invttail(e(df_r),alpha/2)
25           scalar ub = biv + seiv*invttail(e(df_r),alpha/2)
26           scalar pvr = slope<ub & slope>lb
27 
28         post ‘sim’ (‘gam’) (r2) (F) (b) (biv) (se) (seiv) (pv) (pvr)
29          }
30    }
31 }
32   postclose ‘sim’
33 end
```

The parameter `gam` changes instrument strength and we add it to the `postfile` statement in line 3; the reason for this will become apparent below. A `foreach` loop is initiated and populated in line 5. This type of loop is quite handy for this use. It will loop over the elements of the `numlist` (the numbers that follow `numlist`, i.e., 0.025, 0.0375, 0.05, 0.1, 0.15, 0.2 , and 0.5). The loop is closed on line 30. Any reference to the contents of the local macro `gam` in the program must be enclosed in the now familiar single quotes. Also, note that line 28 now includes `(‘gam’) (r2)` and `(F)` which matches the declaration in the `postfile` command and ensures that the current values related to instrument strength are written to the *results.dta* file.

First, the output from the simulation can be summarized by the design parameter, γ.

```
regIV
use results, clear
by gam, sort: summarize r2 F biv b se_iv p_iv
by gam, sort: summarize p_ls p_iv
```

This produces the following set of summary statistics. Recall, that $\beta = 1$ and that the nominal coverage rate for the confidence interval is 95%.

```
-----------------------------------------------------------------------
-> gam = .025
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          r2 |      1000    .7551756    .0127816   .7209752   .8055412
           F |      1000    2.027093    2.507469   4.73e-06   18.04113
         biv |      1000    .6928968     14.1146  -275.4101   126.6003
           b |      1000    1.401108    .0254189    1.30706   1.468426

-----------------------------------------------------------------------
-> gam = .0375
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          r2 |      1000    .7539033    .0136158   .7106823   .7929032
           F |      1000    3.646357    3.672428   .0000266   23.84322
         biv |      1000    .7281356    6.266472  -108.6397   41.78983
           b |      1000    1.400052    .0254246   1.323858   1.478802
```

```
-----------------------------------------------------------------------
-> gam = .05
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          r2 |      1000    .7538432    .0138031    .712468    .790554
           F |      1000    5.195976    4.309673   .0000954   32.33922
         biv |      1000     1.02068    4.401356  -37.72792   92.09246
           b |      1000    1.398645    .0251742   1.310623    1.47523

-----------------------------------------------------------------------
-> gam = .1
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          r2 |      1000    .7535283    .0142405   .6980119   .7988308
           F |      1000    18.21511    8.546679   .7164295   55.45287
         biv |      1000     .969141    .2596705  -.4665824   1.800038
           b |      1000     1.39171    .0252075   1.280399   1.468121

-----------------------------------------------------------------------
-> gam = .15
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          r2 |      1000    .7551459    .0131145   .7074705   .7903711
           F |      1000    39.52843    12.62963   8.662003    95.4943
         biv |      1000    .9862381    .1514629   .2889033   1.502262
           b |      1000    1.385531    .0236149   1.313096   1.467276

-----------------------------------------------------------------------
-> gam = .2
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          r2 |      1000    .7560955    .0132708   .7115782   .7941926
           F |      1000     70.0239    16.88823   22.88008   136.5724
         biv |      1000    .9928045     .113901   .4058512   1.355679
           b |      1000    1.375369     .024529   1.297485   1.458363

-----------------------------------------------------------------------
-> gam = .5
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          r2 |      1000    .7723736    .0122313   .7272431   .8121825
           F |      1000    432.9301    45.91504   304.8695   603.1799
         biv |      1000    .9993733    .0428512   .8498048    1.14667
           b |      1000     1.27941    .0201706   1.218947   1.355316
```

It is clear that one design goal has been met; `r2` is not changing as γ increases in value. The mean of the instrumental variable estimator is getting closer to 1 as the instrument strengthens. We can also see that least squares actually outperforms the instrumental variable estimator in terms of bias when instruments are very weak. Also, notice the very high standard deviation associated with the instrumental variable estimator when instruments are weak ($F < 18$). This is also supported by theory.

The coverage rate of the confidence intervals could likewise be studied. Below is a table that contains the coverage rates of 95% confidence intervals centered at the least squares and IV estimators, respectively.

```
Coverage rates for 95% confidence intervals
-----------------------------------------------------------
                                gam
   Estimator |0.025 |.0375|0.05 |0.10 |0.15 |0.20 |0.50
-------------+------+-----+-----+-----+-----+-----+----
        OLS  |   0  |   0 |   0 |   0 |   0 |   0 |   0
        IV   |.983  |.966 |.974 |.965 |.959 |.945 |.956
-----------------------------------------------------------
n=1000 and 1000 simulations per design point
```

Bias and precision of least squares may be lower when instruments are weak, but that does not translate into acceptable performance in the coverage rate of confidence intervals. Least squares never covers the parameter. The IV estimator may be erratic, but it's confidence interval is doing a better job of covering the parameter. As instrument strength improves, it gets very close to the 95% nominal coverage rate.

Finally, we can explore the relationship between instrument strength and bias via the mechanism suggested by (Stock and Watson, 2011, p. 464). They suggest a rule of thumb that bias of and instrumental variable estimator is roughly proportional to the inconsistency of least squares scaled by the average F statistic minus 1, e.g., $E[\hat{\beta}_{IV}] - \beta \approx [plim(\hat{\beta}_{OLS}) - \beta]/(E(F) - 1)$, where we take F to be the average for a given design.

```
use results, clear
by gam: egen Fbar = mean(F)      /* Average F by gamma   */
by gam: egen tslsbar = mean(biv) /* Average biv by gamma */
by gam: egen olsbar = mean(b)    /* Average b by gamma   */
by gam: egen r2bar = mean(r2)    /* Average r2 by gamma  */

gen tsls_bias = tslsbar-1         /* IV bias             */
gen ols_bias = olsbar-1           /* OLS bias            */
gen relb = tsls_bias/ols_bias     /* relative bias       */
gen rot = 1/(Fbar-1)              /* rule of thumb       */
```

This requires the use of the extended generation functions `egen` to populate the dataset with the average values of the F-statistic, R^2, and the averages of the IV and OLS estimators. With 1000 observations in our sample, the average of the OLS estimator is hopefully close to its probability limit as required by the Stock and Watson result; increasing sample size will improve the approximation.

Based on our design with sample size $n = 1000$ and on 1000 simulations for each design, the results are:

```
                                              F
    Variable |    2.03    3.64    5.19     18.21    39.52    70.02   432.9
    -----------------------------------------------------------------------
rule-of-thumb|  .97362   .37787  .23832   .05809   .02595   .01449  .00232
relative bias| -.76564  -.67957  .05188  -.07878  -.03569  -.01917 -.00224
```

Keep in mind that the rule-of-thumb is always positive; the bias can be negative as is the case here. So, the magnitudes of the two indicators is what we are comparing. The approximations are reasonable, though the rule-of-thumb is much better when $F > 10$ it appears. A regression of relative bias onto the rule-of-thumb should yield a coefficient of -1. There are only 7 actual design points, therefore we need to eliminate all of the redundant observations.

```
gen t = _n                    /* obtain observation numbers        */
keep if mod(t,1000) == 0      /* keep 1 observation per design     */
```

A regression that includes the rule-of-thumb squared is used to test whether the relationship is actually linear. The coefficient on the squared term is not significant ($t = 0.72$) and we conclude that the relationship is linear. Next, we test whether the proportional relationship is one-to-one; the simple regression *relative bias* $= \beta($*rule-of-thumb*$) +$ *residual* is estimated and $\beta = -1$ is tested against $\beta \neq -1$. The t-ratio is approximately equal to 0.8, which is not significant at any reasonable level of significance.

```
. reg relb rot, noconst
                                                 Number of obs =       7
                                                 R-squared     =  0.8131
------------------------------------------------------------------------
relative bias:   Coef.   Std. Err.      t    P>|t|      [95% Conf. Interval]
----------+-------------------------------------------------------------
```

```
rule-o-tmb|   -.864443   .1692116    -5.11   0.002    -1.278489    -.450397
------------------------------------------------------------------------------
```

7. Conclusion

In this primer the basics of conducting a Monte Carlo experiment are discussed and the concepts are put to use in **Stata**. As many authors have noted, Monte Carlo methods illustrate quite clearly what sampling properties are all about and reinforce the concepts of relative frequency in classical econometrics. The **Stata** software is well-suited for this purpose. The `postfile` construct makes it relatively easy to collect statistics of interest and to output them to a dataset. Once there, they can be recalled for further analysis without having to rerun the simulation.

A series of examples is given. These include linear regression, confidence intervals, the size and power of t-tests, lagged dependent variable models, heteroskedastic and autocorrelated regression models, instrumental variables estimators, binary choice, censored regression, and nonlinear regression models. Do-files and data for all examples are provided in the paper and are available from the author's website, the address for which can be found in the references.

The large library of built-in estimators and tests makes **Stata** a rich platform for Monte Carlo analysis. The biggest drawback is that it can be relatively slow (see the discussion and table below), at least in the single processor versions of the software. For serious simulation work, one would want to consider a multiprocessor version. However, as demonstrated in the last example, Stata provides a fine tool for developing prototypes for eventual translation to a faster computing platform. The huge library of built-in estimators and test statistics makes preliminary analysis a breeze.

Stata's `simulate` command allows you to put whatever you want to compute within a program and makes simulating the contents quite simple. The desired results are output to a **Stata** dataset, allowing for subsequent analysis. The advantages of using the `postfile` construction is that it gives you more control over what gets computed and saved. It also makes it easier to loop over different designs and have all of the output placed into the resulting dataset.

Below we compare time required to compute the first confidence interval example using three methods. The computations were performed using Windows 7 64-bit, equipped with an Intel i7 2600K processor (not overclocked). The system has 16 gigabytes of RAM. A total of 4000 Monte Carlo iterations were used in each example. These times were based on version 11.2

of **Stata**.[6] For purposes of comparison, we also used **gretl** (a native 32-bit program) to run comparable simulations. The **gretl** code appears in the appendix and examples similar to the ones in this paper can be found at Adkins (2011*a*,*b*). The simulation was repeated using the `postfile` commands and then using `simulate`. The numbers reported are cpu seconds. Smaller is faster.

TABLE 1. Speed comparison of **Stata**'s `simulate` and `postfile` commands to **gretl** measured in cpu seconds based on 4000 Monte Carlo samples.

		Stata[a]	
Program	**gretl** [b]	`postfile`	`simulate`
Confidence Interval	0.608	2.86	3.26
Nonlinear Least Squares	1.81	50.01	52.35[c]

[a] **Stata** 11.2
[b] **gretl** 1.9.9
[c] The do-file for the `simulate` version can be found at `http://www.learneconometrics.com/pdf/MCstata/index.htm`.

As one can easily see, **gretl** is 4 times faster than **Stata** 11.2 in simulating the regression model and confidence interval coverage rates. The `simulate` command adds extra overhead that costs about a second. This extra time is not proportional to the number of iterations performed; the gap remains fairly constant as the number of simulations increases. If speed is important, **gretl** is an excellent choice. For nonlinear least squares, things get worse for **Stata**. **Gretl** manages to run 4000 iterations of the simulation in less than 2 seconds while **Stata** takes over 50. **Stata** may not be very quick, but it contains a vast number of routines that can be easily simulated either via the `postfile` method or via the `simulate` command. Because of its wide availability and ease of use, it is an excellent vehicle for teaching sampling theory in econometrics and for exploring the properties of a wide variety of estimators.

References

Adkins, Lee C. (2011*a*), *Monte Carlo experiments using gretl: A review with examples.*
URL: *http://www.learneconometrics.com/pdf/MCgretl*

[6]The timings of the programs in Stata 12.1 were also taken. Inexplicably, the do-files are much slower to execute in version 12.1. The `postfile` version of the confidence interval do-file takes 14.18 seconds and the `simulate` version 14.88. This suggests a potential problem with version 12.1 that is likely to be fixed in later updates. Hence, we have chosen to report the version 11.2 results in Table 1.

Adkins, Lee C. (2011*b*), 'Using gretl for Monte Carlo experiments', *Journal of Applied Econometrics* **26**(5), 880–885.

Barreto, Humberto and Frank M. Howland (2006), *Introductory Econometrics using Monte Carlo Simulation with Microsoft Excel*, Cambridge University Press, New York.

Cameron, A. Colin and Pravin K. Trivedi (2009), *Microeconometrics Using Stata*, Stata Press, College Station, TX.

Cottrell, Allin and Riccardo "Jack" Lucchetti (2012), *Gretl Users Guide.*
URL: *http://sourceforge.net/projects/gretl/files/manual/*

Davidson, Russell and James G. MacKinnon (1992), 'Regression-based methods for using control variates in Monte Carlo experiments', *Journal of Econometrics* **54**, 203–222.

Davidson, Russell and James G. MacKinnon (2004), *Econometric Theory and Methods*, Oxford University Press, New York.

Day, Edward (1987), 'A note on simulation models in the economics classroom', *Journal of Economic Education* **18**(4), 351–356.

Greene, William (2000), *Econometric Analysis*, 4th edn, Prentice-Hall.

Horrace, William C. and Ronald L. Oaxaca (2006), 'Results on the bias and inconsistency of ordinary least squares for the linear probability model', *Economics Letters* **90**(3), 321–327.

Judge, Guy (1999), 'Simple monte carlo studies on a spreadsheet', *Computers in Higher Education Economics Review* **13**(2).
URL: *http://www.economicsnetwork.ac.uk/cheer/ch13_2/ch13_2p12.htm*

Kennedy, Peter E. (2003), *A Guide to Econometrics, Fifth Edition*, MIT Press.

Kiviet, Jan F. (2012), 'Monte Carlo simulation for econometricians', *Foundations and Trends in Econometrics* **5**(1-2), 1 – 181.

Koenker, R. and G Bassett (1982), 'Robust tests of heteroscedasticity based on regression quantiles', *Econometrica* **50**, 43 – 61.

StataCorp (2011), *Stata Statistical Software: Release 12*, StataCorp LP, College Station, TX.

Stock, James H. and Mark W. Watson (2011), *Introduction to Econometrics*, 3rd edn, Addison Wesley, Boston.

Sul, Donggyu, Peter C. B. Phillips and Chi-Young Choi (2005), 'Prewhitening bias in HAC estimation', *Oxford Bulletin Of Economics And Statistics* **67**(4), 517–546.

Train, Kenneth E. (2003), *Discrete Choice Methods with Simulation*, Cambridge University Press, Cambridge, UK.

Appendix A. *hansl* Scripts

To give the reader an idea about the speed of **Stata**, we compared it to a free software called **gretl**. **Gretl**'s matrix language is called *hansl* and *hansl* refers to a collection of code as a *script*. The *hansl* scripts for studying confidence intervals and nonlinear least squares are found below in sections A.1 and A.2, respectively.

A.1. **Confidence Intervals.**

```
open engel.gdt
# set a seed if you want to get same results each time you run this
set seed 3213799

# Set the values of the parameters
scalar constant = .5
scalar slope = .5
scalar sigma = 1
scalar alpha = .025 # (size of one tail of a 2 sided CI)

# Take the natural log of income to use as x
genr x = log(income)

# start the loop, indicating the desired number of samples.
# --progressive is a special command for doing MC simulations (necessary)
# --quiet tells gretl not to print the iterations (highly recommended)
set stopwatch
loop 4000 --progressive --quiet
   # generate normal errors
   genr u = normal(0,sigma)
   # generate y, call it y1
   genr y1 = constant + slope*x + u
   # run the regression
   ols y1 const x
   # save the estimated coefficients
   genr b2 = $coeff(x)
   # save the estimated standard errors
   genr s2 = $stderr(x)
   # generate the lower and upper bounds for the confidence interval
   genr c2L = b2 - critical(t,$df,alpha)*s2
   genr c2R = b2 + critical(t,$df,alpha)*s2
   # count the number of instances when coefficient is inside inverval
   genr p2 = (slope>c2L && slope<c2R)
   # print the results
   print b2 p2
   # store the results in a dataset for future analysis is desired
   store cicoeff.gdt b2 s2 c2L c2R
```

```
endloop
scalar t = $stopwatch
```

A.2. Nonlinear Least Squares.

```
open "greene11_3.gdt"
setobs 1 1 --cross-section
# Set the actual values of the parameters
scalar A = 180
scalar B = .25
scalar G = 1

# Starting values
ols C 0 Y
genr alpha0 = $coeff(0)
genr beta0 = $coeff(Y)
genr gamma0 = 1


# Set the number of Simulated Samples
scalar NMC = 4000

# Create an empty matrix to store results
matrix coeffs = zeros(NMC, 3)
matrix vcvs   = zeros(NMC, 6)

# Create systematic portion of the model and log(y)
series C0 = A + B*Y^G
series lY = log(Y)

set stopwatch
# The loop
set warnings off
loop i=1..NMC --quiet
    # generate new sample
    genr C = C0 + normal(0,10)
    # Initialize parameters
    alpha = alpha0
    beta  = beta0
    gamma = gamma0

    # Estimate parameters via NLS
    nls C = alpha + beta * Y^gamma
        deriv alpha = 1
        deriv beta = Y^gamma
        deriv gamma = beta * Y^gamma * lY
    end nls --quiet

    # Collect the coefficients into a vector and matrix
    matrix coeffs[i,] = {alpha, beta, gamma}
    matrix vcvs[i,] = vech($vcv)'
endloop

# open a new, empty dataset
nulldata NMC --preserve

# Convert the columns of matrix to data
series a = coeffs[,1]
series b = coeffs[,2]
series c = coeffs[,3]

# Print the summary Statistics
summary a b c
scalar t = $stopwatch
printf "Monte Carlo vcv vs average estimated vcv\n"

MCV = mcov(coeffs)
AEV = unvech(meanc(vcvs)')

print MCV AEV
```

Lee C. Adkins, Professor of Economics, College of Business Administration, Oklahoma State University, Stillwater OK 74078

E-mail address: lee.adkins@okstate.edu

Mary N. Gade, Associate Professor of Economics, College of Business Administration, Oklahoma State University, Stillwater OK 74078

E-mail address: mary.gade@okstate.edu