MONTE CARLO EXPERIMENTS USING GRETL: A PRIMER

LEE C. ADKINS

Abstract. Monte Carlo simulations are a very powerful way to demonstrate the basic sampling properties of various statistics in econometrics. The free software package **gretl** makes these methods accessible to a wide audience of students and practitioners. In this paper I will discuss the basic principles of Monte Carlo simulations and demonstrate how easy they are to do in **gretl**. 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, and binary choice models. Scripts for all examples are available from the website `http://learneconometrics.com/pdf/MCgretl/`.

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.

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

> 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.

Date: January 14, 2011.

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

Thanks to Allin Cottrell, Jack Lucchetti, Lars Pålsson Syll, and James MacKinnon for reading prior versions of this paper and correcting several errors. Any mistakes that remain are mine alone.

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]. 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 **gretl** software will be demonstrated using a number of examples.

Gretl [Cottrell and Lucchetti, 2010] is particularly useful for several reasons. First, it works well with all major microcomputer operating systems and therefore should be available to anyone with a computer, not just Microsoft Windows users. Second, it is distributed as a free download. It is also fast and accurate (see Yalta and Yalta [2007]); from a numerical standpoint, it is the equal or better of many commercial products. More importantly for our purposes, it is very easy to use. The amount of programming needed to perform a Monte Carlo simulation is minimal. It essentially amounts to generating new variables, using built-in estimators, and saving the statistics that you want to study. **Gretl**'s very versatile loop function includes an option that is specifically designed to collect the simulated values of statistics from a Monte Carlo experiment.

There are other works that take a similar approach. Barreto and Howland [2006] use Microsoft Excel in their introductory textbook. They provide worksheets that accomplish the simulations, but there is not information on how they are constructed. Their implementation is excellent, but different from the one used here. Murray [2006] is also a strong proponent of using Monte Carlo methods to illustrate sampling. He provides a unique web application with his book to illustrate important concepts. Again, the application is very useful, but it doesn't illustrate how to build a Monte Carlo experiment or allow much ability to vary many of the parameters of interest in a study of sampling properties. Hill et al. [2008] uses the results from simulations to illustrate important principles, but actual code to the underlying experiments is not given.

In the next section, the concept of a fully specified statistical model as laid out by Davidson and MacKinnon [2004] is summarized and the basic recipe for using it in a Monte Carlo exercise is presented. In section 3 some of the basics of **gretl** are discussed. In section 4 the basics of Monte Carlo simulations will be summarized. Then in section 5 I'll carefully go through a complete example of using **gretl** to study the coverage probability of confidence intervals centered at the least squares estimator. In subsequent subsections, how to generate simulated models for some of the estimators used in a first econometrics course is outlined.

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. 19]. Consider a regression function:

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

where y_t is the dependent variable, x_t the dependent variable, Ω_t the current information set, and β_1 and β_2 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 or a discrete choice model. However, equation (1) is not complete; it requires some description of how the unobserved or excluded factors affect $y_t|\Omega_t$.

To complete the the specification one needs to specify an "unambiguous recipe" for simulating the model on a computer [Davidson and MacKinnon, 2004, p. 17]. This means that a specific probability distribution for the unobserved components of the model must be chosen and a pseudo-random number generator used 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 and choose any parameters that it may require (e.g., the normal requires the variance, σ^2)
- Use a pseudo-random number generator to get the n successive and mutually independent 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, but as you shall see shortly, they are not needed with **gretl** .

3. Gretl Basics

Gretl is an acronym for Gnu Regression, Econometrics and Time-series Library. It is a software package for doing econometrics that is easy to use and reasonably powerful. **Gretl** is distributed as free software that can be downloaded from `http://gretl.sourceforge.net` and installed on your personal computer. Unlike software sold by commercial vendors (SAS, Eviews, Shazam to name a few) you can redistribute and/or modify **gretl** under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation. Baiocchi and Distaso [2003], Mixon and Smith [2006], and Rosenblad [2008] provide interesting observations on using **gretl** to teach basic econometrics. Also, Adkins [2010] has written a free textbook that contains more examples of how **gretl** can be used in the classroom.

Gretl comes with many sample data files and provides access to a database server that houses many useful macro and microeconomic data sets.

Gretl can be used to compute least-squares, weighted least squares, nonlinear least squares, instrumental variables least squares and LIML, logit, probit, tobit, GMM, and a number of time series estimators. The list of procedures available in **gretl** continues to grow thanks to tireless efforts of its primary programmers Allin Cottrell and Jack Lucchetti [Cottrell and Lucchetti, 2010]. **Gretl** uses a separate Gnu program called ***gnuplot*** to generate graphs and is capable of generating output in LaTeX format [see Racine, 2006]. As of this writing **gretl** is in version 1.9.0 and is actively under development.

For reference below, the main **gretl** window appears in Figure **??**. At the top is the **menu bar**, which includes options for File, Tools, Data, View, Add, Sample, Variable, Model, and Help. At the bottom of the main window is a row of icons; this is referred to as the **toolbar**. Some of the functions of the toolbar are shown in Figure 2 and are discussed below.

3.1. Ways to Work in Gretl.

There are several different ways to work in **gretl**. One can use the program through its built in graphical user interface (GUI) where **gretl** 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 opened in

a new window that gives additional access to various graphs, tests, analysis and means of storing the results. Commands executed through the GUI are saved in the `command log`, which can be accessed for later review by clicking on `Tools>Command log` on the menu bar.

The two other ways of working in **gretl** are better suited for Monte Carlo analysis, though the GUI can be useful for finding commands and using the documentation. These methods are: the **console** or **command line** and **scripts**.

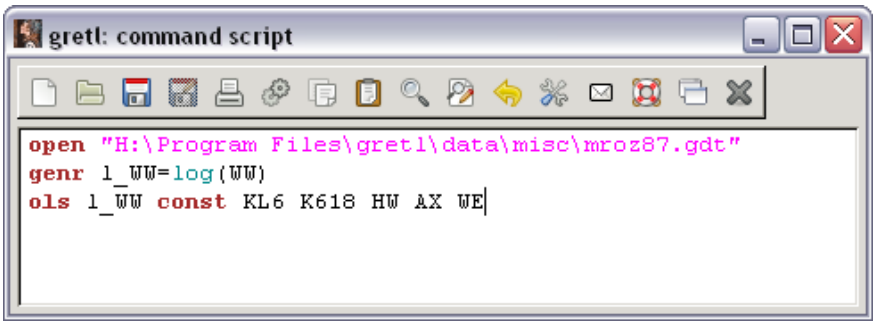
3.1.1. ***Gretl**'s Console and Command Line Interface.* **Gretl** offers two ways to issue commands directly, bypassing the pull-down menus and dialog boxes. In one mode a separate **console** window is opened where single line **gretl** commands can be given with the output returned to the console window.

There is also command line version of **gretl** that skips windows and dialogs altogether. The command line version is launched by executing `gretlcli` in a command window. In MS Windows, click `Start>Run` and browse for the directory that contains **gretl**. To carry out a substantial Monte Carlo analysis with many thousands of repetitions, memory capacity and processing time may be an issue. To minimize the use of computer resources, run the script using `gretlcli`, with output redirected to a file.[1]

3.1.2. *Scripts.* **Gretl** commands can be collected and put into a file that can be executed at once and saved to be used again. **Gretl** refers to these collections of code as a `script`. Even though you can run scripts from the command line or the console, the `script window` is the easiest way used to do Monte Carlo simulations in **gretl**. Start by opening a new **command script** from the file menu. The command `File>Script files>New script` from the pull-down menu opens the command script editor shown in Figure 1. Type the commands you want to execute in the box using one line for each command. Notice that in the first line of the script, `"H:\Program Files\gretl\data\misc\mroz87.gdt"`, the complete file and path name are enclosed in double quotes, `" "`. This is necessary because the `Program Files` directory in the pathname includes a space. If you have **gretl** installed in a location that does not include a space, then these can be omitted.

[1]The syntax for which is `gretlcli -b scriptfile > outputfile`. The argument `scriptfile` refers to the ascii text file that contains your script; and, only the output file requires redirection (`>`). Dont forget the `-b` (batch) switch, otherwise the program will wait for user input after executing the script. [see Cottrell and Lucchetti, 2010, ch. 31].

FIGURE 1. The Command Script editor is used to collect a series of commands into what **gretl** calls a **script**. The script can be executed as a block, saved, and rerun at a later time.



For a very long command that exceeds one line, use the backslash (\) as a **continuation command**. Then, to save the file, use the "save" button at the top of the box (third one from the left). If this is a new file, you'll be prompted to provide a name for it.

In the first line of the figure shown, the `mroz87.gdt` **gretl** data file is opened. The `genr` commands are used to take the logarithm of the variable `WW`, and the `ols` command discussed in section 5 is used to estimate a simple linear regression model that has `l_WW` as its dependent variable and `KL6`, `K618`, `HW`, `AX`, and `WE` as the independent variables. Note, the model also includes constant. Executing this script will open another window that contains the output where further analysis can be done. To run the program, click the mouse on the "gear" button (6th from the left in Figure 1).

A new script file can also be opened from the toolbar by mouse clicking on the "new script" button–second from left and looks like a pad of paper–or by using the keyboard command, Ctrl+N.[2]

One of the handy features of the command script window is how the help function operates. At the top of the window there is an icon that looks like a lifesaver. Click on the help button and the cursor changes into a question mark. Move the question mark over the command you want help with and click. This will either produce an error message or take you to the topic from the command reference.

Although scripts will be the main way to program and run simulations in **gretl**, the console and the pull-down menus also have their uses. As will be seen shortly, various statistics produced in the simulation can be saved to **gretl** data sets. Once the simulation finishes, one can open these and subject the statistics to further analysis, either from the console of

[2] "Ctrl+N" means press the "Ctrl" key and, while holding it down, press "N".

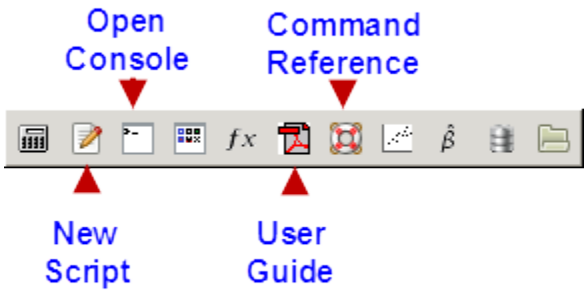
by using the dialog boxes that open when commands are chosen from the pull-down menus. A few examples will be given in later sections of the paper.

3.2. **Common Conventions.** Monte Carlo simulations require a series of commands that are best executed from **gretl** scripts; basically you write a simple programs in its entirety, store them in a file, and then execute all of the commands in a single batch. In this paper, scripts for all of the exercises are given but only a few snippets of output are included. Since the experimental results depend on how the models are parameterized, its best to use the scripts to generate your own output for the conditions you want to study. Also, 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. All of this is fairly standard practice, but if you don't know what this means, ask your instructor now.

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

Nearly anything that can be done with the pull-down menus can also be done through the console. The command reference can be accessed from the toolbar by clicking the button that looks like a lifesaver. It's the fourth one from the right hand side of the toolbar (Figure 2):

FIGURE 2. The Toolbar appears at the bottom of the main **gretl** window.



3.3. **Importing Data.** Obtaining data in econometrics and getting it into a format that can be used by software can be challenging. There are dozens of different pieces of software and many use proprietary data formats that make transferring data between applications difficult. Most textbook authors provide data in several formats for your convenience. **Gretl** is able to read Stata files directly. Simply drag a Stata data set onto the main **gretl** window and it will open. From the menu bar click (`File>Open data>Import`) to read data from EViews, SAS, SPSS, various spreadsheets, Octave, Gnumeric, and text. **Gretl** is also web aware and you can download actual economic data from many sources including Penn World tables, the St. Louis Fed, Bureau of Economics Analysis, and Standard and Poors. Click on `File>Databases>On database server` you will be taken to a web site (provided your computer is connected to the internet) that these and other high quality data sets. In addition, data sets (and in some cases replication scripts) are available for several popular econometrics textbooks (see `http://gretl.sourceforge.net/gretl_data.html` for details).

4. Monte Carlo Basics

The main programming tool used to run Monte Carlo experiments is **gretl**'s flexible `loop` function. The syntax for it is:

```
loop control-expression [ --progressive | --verbose | --quiet ]
  loop body
endloop
```

The loop is initiated with the command `loop`. This is followed by the `control-expression`, which essentially instructs the loop how to perform. This is followed by several options, the most important of which for our purposes is `--progressive`.

There are actually five forms of control-expression that are available and these are explained in section 9.2 of the Gretl Users Guide [see Cottrell and Lucchetti, 2010]. In the examples that follow, only two of these are used. In most of the examples that follow, the control-expression is simply a number indicating how may Monte Carlo samples to take (`NMC`). This is referred to as a *count loop*.

By default, the `genr` command operates quietly in the context of a loop (without printing information on the variable generated). To force the printing of feedback from `genr` you may specify the `--verbose` option to loop. This can be helpful when debugging. The `--quiet` option suppresses the usual printout of the number of iterations performed, which may be desirable when loops are nested. The `--progressive` option to loop modifies the behavior

of the commands print and store, and certain estimation commands, in a manner that may be useful with Monte Carlo analysis.

When the `--progressive` option is used the results from each individual iteration of the estimator are not printed. Instead, after the loop is completed you get a printout of (a) the mean value of each estimated coefficient across all the repetitions, (b) the standard deviation of those coefficient estimates, (c) the mean value of the estimated standard error for each coefficient, and (d) the standard deviation of the estimated standard errors. This makes sense only if there is some random input at each step.

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

Basic Recipe with the loop code

```
# open a data set to use as a basis for the simulation
                        or
# create values of the regressors using random numbers
# set a seed to get same results each time you run this
# Set the values of the parameters
# Initialize data that may need starting
                    values in the loop (occasionally necessary)
# start the loop, indicating the desired number of samples (NMC).
#   use --progressive to do MC simulations (necessary)
#   use --quiet to suppress printing the iterations (highly recommended)
loop NMC --progressive --quiet
   # generate random errors
   # generate conditional mean, y
   # estimate the model
   # compute and save the desired statistics
   # print results
   # store results in a data set for future analysis if desired
endloop
```

Also, when the `print` command is used, the `progressive` option suppresses the printing results of the estimation at each round the loop. Instead, when the loop is terminated you get a printout of the mean and standard deviation of the variable, across the repetitions of the loop. This mode is intended for use with variables that have a scalar value at each iteration, for example the error sum of squares from a regression. Data series cannot be printed in this way.

The `store` command writes out the values of the specified scalars, from each time round the loop, to a specified file. Thus it keeps a complete record of their values across the iterations. For example, coefficient estimates could be saved in this way so as to permit subsequent

examination of their frequency distribution. Only one such store can be used in a given loop.

Once the samples of your statistics from the Monte Carlo experiments have been obtained, what to do with them? For instance how can one judge whether an estimator is biased? The simple answer is to use more 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 **gretl** prints for you; now, compare that to the desired critical value from the standard normal.

Finally, there is a trick one can use to significantly improve the accuracy of some studies: use antithetic draws from the desired distribution. Antithetic draws are perfectly negatively correlated with one another. For a symmetric distribution like the 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. See Train [2003, pp. 219-221] for an excellent discussion of antithetic variates.

5. Examples

In this section, a series of examples is given. Each example illustrates important features of model specification and estimation. The first example is based on the classical normal linear regression model. Four hundred (NMC=400) samples based on Engel's food expenditure and income data included with **gretl** are generated. The slope and intercept parameters with each simulated data set are computed using least squares, and then 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. What becomes clear is the outcome from any single sample is a poor indicator of the true value of the parameters. Keep this in mind whenever you estimate a model with what is invariably only one sample or instance of the true (but always unknown) data generation process.

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 three 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. Next, a binary choice example is studied. Finally, a example is given for a procedure that involves a block of commands–nonlinear least squares.

Although the errors in each of the examples are generated from the normal distribution, **gretl** offers other choices. These include uniform, Student's t, Chi-square, Snedecor's F, Gamma, Binomial, Poisson and Weibull.

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

$$\ln(y_t) = \beta_1 + \beta_2 \ln(x_t) + u_t \tag{2}$$

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.

The computer 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.

A total of 400 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

$$P[b_2 - t_c se(b_2) \leq \beta_2 \leq b_2 + t_c se(b_2)] = 1 - \alpha \tag{3}$$

where b_2 is the least squares estimator of β_2 and $se(b_2)$ is its estimator of 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 script is found in section 6 below. In what follows, the components of it will be discussed. First, open the script window in **gretl** by clicking on the open script button on the toolbar shown in Figure 2. The first line of your script opens the data set

Performance of Confidence Intervals

```
1 open engel.gdt
```

When writing a program, it is always a good idea to add comments. **Gretl** will ignore anything to the right of #, making it perfect for adding comments. In line 2 of the completed script (Figure 6), I suggest setting a seed for the random number generator. Line 3 uses `set seed 3213799` to initialize the stream of pseudo-random normals that will be called later.

The next section, lines 6-9, set the values of the parameters used in the Monte Carlo simulation. Once the simulation is up and working to your satisfaction, these can be changed to study the effect on the estimators.

In Monte Carlo experiments loops are used to estimate a model using many different samples that the experimenter generates and to collect the results. As discussed in Section 4, the loop construct in **gretl** should begin with the command `loop NMC --progressive --quiet` and ends with `endloop`. `NMC` in this case is the desired number of Monte Carlo samples. The option `--progressive` is a command that invokes special behavior for certain commands, namely, `print`, `store` and simple estimation commands. This option will work with any of the single equation estimators in **gretl** , but not with any that use blocks of code (e.g., `nls`, `gmm`, `mle`). For these types of procedures, a little more programming will be required to simulate samples and collect results.

To reproduce the results below, the sequence of random numbers is initiated using a key, called the *seed*, with the command `set seed 3213799`. Basically, this ensures that the stream of pseudo random numbers will start at the same place each time you run your

program. Try changing the value of the seed (3213799) or the number of Monte Carlo iterations (400) to see how your results are affected.

Within the body of the loop, **gretl** is told how to generate each sample and then what to do with that sample. The data generation is accomplished here as

Performance of Confidence Intervals continued

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

Then, start the loop with the progressive and quiet options, generate errors and samples of the dependent variable:

Performance of Confidence Intervals continued

```
17 loop 400 --progressive --quiet
18    # generate normal errors
19    genr u = normal(0,1)
20    # generate y
21    genr y = constant + slope*x + u
```

The `genr` command is used to generate new variables. In the first instance the natural logarithm of income is generated and the second `u` is created as a standard normal random variable. The **gretl** command `normal(0,1)` produces a computer generated standard normal random variable. The first argument is the desired mean (0) and the second is the standard error (1). The final instance of `genr` adds this random element to the systematic portion of the model to generate a new sample for food expenditures (using the known values of income in x).

Next, the model is estimated using least squares. Then, the coefficients are stored internally in variables named `b1` and `b2`. The `genr` command provides a means of retrieving various values calculated by the program in the course of estimating models or testing hypotheses. The variables that can be retrieved in this way are listed in the *Gretl Command Reference*, but include the coefficients from the preceding regression and the standard errors. With no arguments, `$coeff` returns a column vector containing the estimated coefficients for the last model. With the optional string argument it returns a scalar, namely the estimated parameter associated with the variable named `x`. The estimated constant is referred to as `const`. The `$stderr` works likewise, holding the estimated standard error.

Performance of Confidence Intervals continued

```
22    # run the regression
23    ols y const x
24    save the estimated coefficients
25    genr b1 = $coeff(const)
```

```
26    genr b2 = $coeff(x)
27    # save the estimated standard errors
28    genr s1 = $stderr(const)
29    genr s2 = $stderr(x)
```

It is important to know how precise your knowledge of the parameters is after sampling. One way of doing this is to look at the least squares parameter estimate along with a measure of its precision, i.e., its estimated standard error.

The confidence interval serves a similar purpose, though it is much more straightforward to interpret because it gives you upper and lower bounds between which the unknown parameter will fall with a given frequency.

In **gretl** you can obtain confidence intervals either through a dialog or by manually building them using saved regression results. In the 'manual' method I will use the `genr` command to generate upper and lower bounds based on regression results that are saved in **gretl**'s memory, letting **gretl** do the arithmetic.

The critical value t_c can be obtained from a statistical table, the `Tools>Statistical tables` dialog contained in the program, or using the **gretl** command `critical`. The `critical` command is quite versatile. It can be used to obtain critical values from the standard normal, Student's t, Chi square, Snedecor's F, or the Binomial. In this case the `t($df)` is used, where `$df` returns the degrees of freedom of the last model estimated by **gretl**.

Performance of Confidence Intervals continued

```
30    # generate the lower and upper bounds for the confidence interval
31    genr c1L = b1 - critical(t,$df,alpha)*s1
32    genr c1R = b1 + critical(t,$df,alpha)*s1
33    genr c2L = b2 - critical(t,$df,alpha)*s2
34    genr c2R = b2 + critical(t,$df,alpha)*s2
35    # count the number of instances when coefficient is inside interval
36    genr p1 = (constant>c1L && constant<c1R)
37    genr p2 = (slope>c2L && slope<c2R)
```

The last two lines use logical operators that equal 1 if the statement in parentheses is true and 0 otherwise.[3] The average of these measures the proportion of the intervals that contain the actual values of the slope and intercept.

[3] && is the logical "AND" and || is the logical "OR".

Finally, several of the computed variables are stored to a data set `cicoeff.gdt`. Once the simulation finishes, the data set can be opened and other computations can be made or graphs constructed using the results.

After executing the script, **gretl** prints out some summary statistics to the screen. These appear below in Figure 3. Note that the average value of the intercept is about 0.448 and

FIGURE 3. The summary results from 400 random samples of the Monte Carlo experiment.

```
OLS estimates using the 235 observations 1-235
Statistics for 400 repetitions
Dependent variable: y

                    mean of      std. dev. of     mean of     std. dev. of
                   estimated      estimated      estimated      estimated
      Variable    coefficients   coefficients   std. errors    std. errors

          const      0.448355       0.935386        1.01143      0.0488781
              x      0.507090       0.136915       0.148732     0.00718759

Statistics for 400 repetitions
       Variable     mean         std. dev.
             p1      0.960000       0.195959
             p2      0.957500       0.201727

store: using filename H:\Documents and Settings\My Documents\gretl\cicoeff.gdt
Data written OK.
```

he average value of the slope is 0.507, both of these are reasonably close to the true values set in lines 6 and 7. If the experiments were repeated with larger numbers of Monte Carlo iterations, one 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 the experiment, many samples are generated and averaged results over those samples to get closer to the true values, both as sample size and as the number of Monte Carlo samples increases. In actual practice, one does not have this luxury; there is only one sample of a given size and the proximity of the estimates to the true values of the parameters is always unknown.

The bias of the estimator can be evaluated using $z = \sqrt{\text{NMC}}(\bar{x} - \mu)/\hat{\sigma}$ which in this case of `p1` is $\sqrt{400}(.96 - .95)/0.195959 = 1.02$. This is less than the 5% two-sided critical value of 1.96 so we cannot reject the hypothesis that the coverage rate of the confidence interval is equal to its nominal value. Increasing the number of Monte Carlo samples to 4000 produced

$z = \sqrt{4000}(0.9495 - .95)/0.218974 = -0.144413$; this shows that increasing the number of samples increases the precision of the Monte Carlo results.

After executing the script, open the `cicoeff.gdt` data file that **gretl** has created and view the data. From the menu bar this is done using `File>Open data>user file` and selecting `coeff.gdt` from the list of available data sets. From the example this yields the output in Figure 4. Notice that even though the actual value of $\beta_1 = 0.5$ there is considerable variation

FIGURE 4. The results from the first 20 sets of estimates from the 400 random samples of the Monte Carlo experiment.

	b1	b2
1	-0.751753	0.6875541
2	0.358194	0.5348383
3	2.902894	0.1340339
4	1.279589	0.3897894
5	-0.561178	0.6502060
6	-0.310848	0.6191087
7	0.680899	0.4656558
8	1.278022	0.3884723
9	0.279857	0.5174504
10	-0.744204	0.6741758
11	-0.246119	0.6114379
12	-1.162340	0.7302507
13	1.086522	0.4145859
14	1.214361	0.3938145
15	-0.047051	0.5941038
16	-0.651301	0.6641363
17	0.284105	0.5407113
18	-0.385047	0.6177857
19	0.145503	0.5452904
20	-1.867144	0.8470560

in the estimates. In sample 3 it was estimated to be 2.903 and in sample 20 it was -1.867. Likewise, β_2 also varies around its true value of 0.5. Notice that for any given sample, an estimate is never equal to the true value of the parameter.

The saved results can be analyzed statistically or visualized graphically. Opening a console window and typing `summary` at the prompt will produce summary statistics for all of the variables. To plot a frequency distribution and test against a normal, type

```
freq b1 --normal
```

to produce Figure 5. Notice in Figure 5 that the p-value for the test of normality of the least squares estimator is .71, well above any reasonable significance level.

The performance of the Monte Carlo can be improved by using antithetic variates to generate the samples.[4] To do that replace lines 17-19 with

Generating Antithetic Normal Samples

```
17 loop i=1..400 --progressive --quiet
18 # generate antithetic normal errors
19     if i%2 != 0
20     series u = normal(0,sigma)
21     else
22     series u = -u
23     endif
```

First, notice that a new loop control condition is used. The command `loop i=1..400`, referred to as the *index loop*, replaces the simpler *count loop* `loop 400`. This structure sets a counter called `i` that will loop from the first number after the `=` sign to the number listed

[4]Antithetic variates should be used with care as they are not always helpful. For instance they should not be used to study the properties of variance estimators. See Davidson and MacKinnon [1992].

FIGURE 5. Using the command `freq b1 --normal` yields this histogram with a normal pdf superimposed.

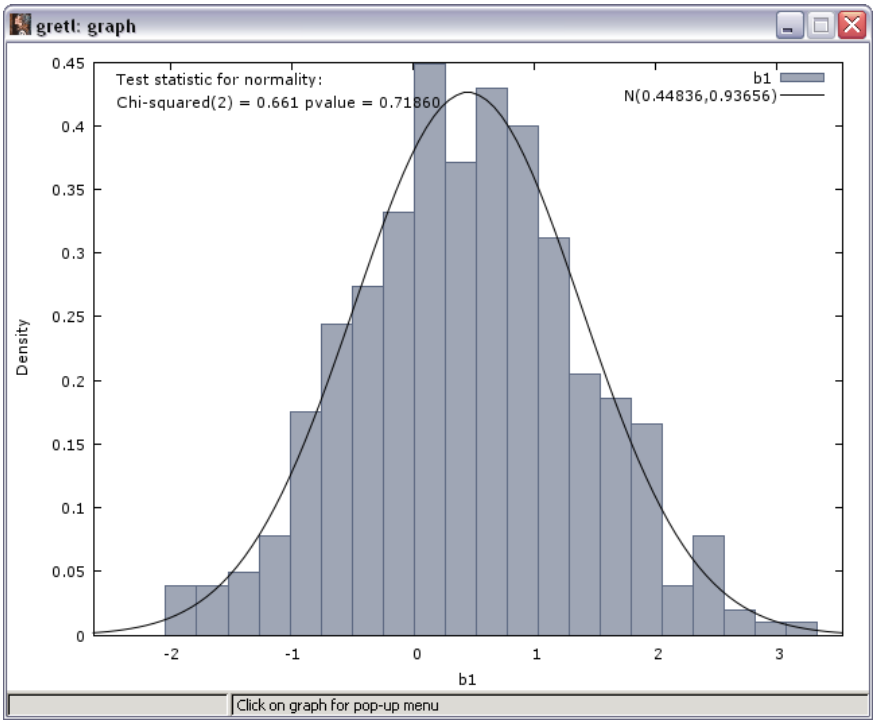


FIGURE 6.

Performance of Confidence Intervals

```
open engel.gdt
# set a seed to get same results each time this runs
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 all the iterations (highly recommended)
loop 400 --progressive --quiet
   # generate normal errors
   genr u = normal(0,sigma)
   # generate y
   genr y = constant + slope*x + u
   # run the regression
   ols y const x
   # save the estimated coefficients
   genr b1 = $coeff(const)
   genr b2 = $coeff(x)
   # save the estimated standard errors
   genr s1 = $stderr(const)
   genr s2 = $stderr(x)
   # generate the lower and upper bounds for the confidence interval
   genr c1L = b1 - critical(t,$df,alpha)*s1
   genr c1R = b1 + critical(t,$df,alpha)*s1
   genr c2L = b2 - critical(t,$df,alpha)*s2
   genr c2R = b2 + critical(t,$df,alpha)*s2
   # proportion of instances when coefficient is inside interval
   genr p1 = (constant>c1L && constant<c1R)
   genr p2 = (slope>c2L && slope<c2R)
   # print the results
   print p1 p2
   # store the results in a data set for future analysis is desired
   store cicoeff.gdt b1 b2 s1 s2 c1L c1R c2L c2R
endloop
```

after the two periods, in this case `400`. So, `i=1` for the first sample in the loop and increases by 1 with each iteration. This variable is used in line 19 along with the modulus operator (%) to determine whether the experiment is on an odd or even number. The statement `i%2 != 0` divides i by 2 and checks to see whether the remainder is equal to zero or not. If not, a draw from the normal is made in line 20. If it is, then line 20 is skipped and line 22 is executed, which uses the negative values of `u` in the previous iteration of the Monte Carlo. The results of the first script using antithetic draws appear in Figure 7 below.

FIGURE 7. The summary results from 400 random samples of the Monte Carlo experiment using antithetic variates.

```
OLS estimates using the 235 observations 1-235
Statistics for 400 repetitions
Dependent variable: y

                    mean of      std. dev. of     mean of     std. dev. of
                  estimated      estimated      estimated      estimated
     Variable     coefficients   coefficients   std. errors    std. errors

         const      0.500000        1.01016        1.01981      0.0509399
             x      0.500000       0.148667       0.149964     0.00749077

Statistics for 400 repetitions
       Variable     mean         std. dev.
             p1      0.950000       0.217945
             p2      0.940000       0.237487

store: using filename H:\Documents and Settings\My Documents\gretl\cicoeff.gdt
Data written OK.
```

Comparing the results from Figure 3 to those in Figure 7 that the use of antithetics makes it easy to see that least squares is unbiased for the constant and slope.

In the following sections, the basic data generation process for the fully specified model is given and simple **gretl** scripts are given to generate experiments based on these. To illustrate their usefulness, some suggestions are made in how to use them. To keep the scripts as simple and transparent as possible I have opted to forgo the use of antithetic variates in the examples that follow.

5.2. **Autocorrelation and Lagged Dependent Variables.** In a linear regression with first order autocorrelation among the residuals, least squares is unbiased provided there are no lags of the dependent variable used as regressors. In this case, least squares is consistent

only. 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

$$y_t = \beta_1 + \beta_2 x_t + \delta y_{t-1} + u_t \qquad t = 1, 2, \ldots, N \tag{4}$$

$$u_t = \rho u_{t-1} + e_t \tag{5}$$

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 equation 1. Stability requires $|\rho| < 1$. Also, partial adjustment models lead us to believe that $|\delta| < 1$ as well. $|\delta| = 1$ suggests that the dependent variable be modeled as a change $(y_t - y_{t-1})$ rather than a level.

There are many questions that can be answered by simulating this ARDL(1,1) 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? And so on.

The basic script in **gretl** is shown in Figure 8. There are a couple of new things in this script. First, on line 2 the `nulldata=100` command is used to open an empty data set and to set the number of observations, $n = 100$. Increasing and decreasing this number is used to explore the properties of the estimators as the sample size changes. Also, new is the generation of x. Rather than use the regressors from a data set, an artificial set is created here. Then on lines 17-18 two series are intialized. Basically, they are place holders that will provide starting values for `y` and `u` for the first round of the simulation. In line 25 the lag operator is used for the first time to get lagged values of u_t, using `u(-1)`. The *variable name* with `-1` in parentheses lags the named variable 1 period. The same is used in line 27 and 29. Notice that in line 29 a new variable for the lagged value of `y` (i.e., `y(-1)`) did not have to be created separately and stored in the data set to use in the estimation command. This saves a lot of time and clutter in the main gretl window and in the programs.

To explore the properties of least squares, one can change the sample size in line 2 (Figure 8), the coefficient on `y(-1)` in line 13, or the degree of autocorrelation (`rho`) in line 14. The standard theoretical results are easy to verify numerically, though with some surprises.

FIGURE 8.

generating samples from LDV model with Autocorrelation

```
# Set the sample size and save it in n
  nulldata 100
  scalar n = $nobs

# generate n observations on x
  series x = uniform()

set seed 3213799 # set a seed to get same results

# Set the values of the parameters
  scalar slope = 10
  scalar sigma = 20
  scalar delta = .7
  scalar rho = .9

# initialize variables u and y
  series u = normal()
  series y = normal()

# start the loop, indicating the desired number of samples.
loop 400 --progressive --quiet
   # generate normal errors
   series e = normal(0,sigma)
   # generate autocorrelated errors
   series u=rho*u(-1)+e
   # generate sample of y
   series y = slope*x + delta*y(-1) + u
   # run the LDV regression
   ols y const x y(-1)
   # save the estimated coefficients
   genr b2_LDV = $coeff(x)
   genr d = $coeff(y_1)
   # run the non-LDV regression
   ols y const x
   genr b2 = $coeff(x)
   # store the results in a data set for future analysis is desired
   store arcoeff.gdt b2 b2_LDV d
endloop
```

5.2.1. *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 Figure 9. Note, that although **gretl** can perform this test using `modtest 1 --autocorrelation`, the `modtest` commands do not yet work in the loops. The auxiliary regressions manually must be run manually. This is a good exercise anyway since it makes the mechanics of the test very clear. Basically, one

estimates the model using ordinary least squares, obtains the residuals, and then estimates an auxiliary regression of the residuals on the full set of original regressors and p lagged residuals. The statistic nR^2 is distributed as a Chi-square with p degrees of freedom. After the regressions in lines 29 (or 34, which omits the lagged value of y) save the residuals, add the auxiliary regression, and save the desired statistic or p-value. Since the p-value computation is not one of the model parameters, a `print` statement is necessary to send the summary statistic to the screen.

FIGURE 9.

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

```
ols y const x y(-1)
genr uhat = $uhat
ols uhat const uhat(-1) x y(-1)
genr nr =$trsq
genr pval = (nr>critical(X,1,.05))
print pval
```

5.2.2. *HAC Standard Errors.* The experimental design of Sul et al. [2005] to study the properties of HAC. They propose a model $y_t = \beta_1 + \beta_2 x_t + u_t$, where $u_t = \rho x_{t-1} + e_t$ 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 $n = 100$. The script appears below.

Coverage probabilities of HAC and the usual OLS confidence intervals

```
1  # Set the sample size and save it in n
2  set hac_lag nw1
3  nulldata 300
4  scalar n = $nobs
5  setobs 1 1 --time-series
6
7  # generate n observations on x
8  series x = uniform()
9
10 set seed 3213799 # set a seed to get same results
11
12 # Set the values of the parameters
13 scalar slope = 1
14 scalar sigma = 1
15 scalar rho = .880
16 scalar alpha = .05
17
18 # initialize variables u
19 series u = normal()
20
21 # start the loop, indicating the desired number of samples.
```

```
22 loop 400 --progressive --quiet
23    # generate normal errors
24    series e = normal(0,sigma)
25    # generate autocorrelated errors and x
26    series u=rho*u(-1)+e
27    series x=rho*x(-1)+normal()
28    # generate sample of y
29    series y = x + u
30    # Estimate the model using OLS, save the slope estimates
31    ols y const x
32    genr b2 = $coeff(x)
33    genr s = $stderr(x)
34    # generate the lower and upper bounds for the confidence interval
35    genr c2L = b2 - critical(t,$df,alpha)*s
36    genr c2R = b2 + critical(t,$df,alpha)*s
37    # count the number of instances when coefficient is inside interval
38    genr p2 = (slope>c2L && slope<c2R)
39    # Repeat for the HAC covariance estimator --robust
40    ols y const x --robust
41    genr sr = $stderr(x)
42    # generate the lower and upper bounds for the confidence interval
43    genr c2Lr = b2 - critical(t,$df,alpha)*sr
44    genr c2Rr = b2 + critical(t,$df,alpha)*sr
45    # count the proportion of instances when coefficient is inside inverval
46    genr p2r = (slope>c2Lr && slope<c2Rr)
47    print p2 p2r
48 endloop
```

This example is very similar to the LDV model, except the independent variable is lagged in the DGP (line 27), but this lagged value is omitted from the model estimated in line 29. The `set hac_lag nw1` command in line 2 sets the routine used to compute the number of lags to use in the computation of the HAC standard errors. The data have to be declared as time series in order for the `--robust` option in line 40 to compute HAC standard errors. This is accomplished using `setobs 1 1 --time-series` in line 5, the numbers `1 1` indicate annual observations that increment by 1 year. Otherwise, this script combines elements from the previous ones and will not be discussed further.

As Sul et al. [2005] note, this usual variant of HAC does better than least squares unassisted, but leaves much to be desired. This can be confirmed by running the script.

5.3. **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' when the error variances are correlated with the regressors. By 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 the sample size and save it in n
set hc_version 3
nulldata 100
scalar n = $nobs

# set a seed to get same results each time you run this
set seed 3213799

# Set the values of the parameters
scalar slope = 1
scalar sigma = 1
scalar rho_xz = .8
scalar gam = .4

# generate n observations on x and a correlated variable z
series x = 10*uniform()
series z = rho_xz*x + normal(0,sigma)

# initialize variables u and y
series u = normal()
series y = normal()

# start the loop, indicating the desired number of samples.
loop 400 --progressive --quiet
   # generate variances that depend on z
   genr sig = exp(gam*z)
   # generate normal errors
   series u = normal(0,sig)
   # generate sample of y
   series y = slope*x + u
   # run the regression with usual error
   ols y const x
   # save the estimated coefficients
   genr b2 = $coeff(x)
   genr se2 = $stderr(x)
   # run OLS regression with HC std errors
   ols y const x --robust
   genr se2r = $stderr(x)
endloop
```

Gretl has options for the different versions of the Heteroskedasticity Consistent Covariance Matrix Estimator (HCCME). In this example `set hc_version 3` 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's 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.

5.4. **Instrumental Variables.** The statistical model contains five parameters: β, σ, σ_x, γ, and ρ. The data generation process is modeled $y_t = \beta x_t + u_t$, where u_t iid $N(0, \sigma_t^2)$, $x_t = \gamma z_t + \rho u_t + e_t$, e_t iid $N(0, \sigma_x^2)$ and the innovation vector (u_t, e_t) is independently distributed for $n = 100$. 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_x^2 can be used to control the relative variability of x_t and u_t as in an error-in-variables problem.

Instrumental Variables

```
# Set the sample size and save it in n
nulldata 100
scalar n = $nobs

# generate n observations on x
series z = 5*uniform()
# set a seed if to get same results each time
set seed 3213799
# Set the values of the parameters
scalar slope = 1  # regression slope
scalar sigma = 10 # measurement error in y
scalar sigx = .1  # amount of measurement error in x
scalar gam = 1    # instrument strength
scalar rho = .0   # correlation between error
scalar n = $nobs

# start the loop, indicating the desired number of samples.
loop 400 --progressive --quiet
   series u = normal(0,sigma)
   # generate sample of x and y
   series x=gam*z+rho*u+normal(0,sigx)
   series y = slope*x + u
   # run the ols regression
   ols y const x
   # save the estimated coefficients
   genr b2 = $coeff(x)
   genr s2 = $stderr(x)
```

```
   # run the tsls regression and save the coefficient and se
   tsls y const x; const z
   genr biv2 = $coeff(x)
   genr siv2 = $stderr(x)
   # store the results in a data set for future analysis is desired
   store arcoeff.gdt b2 biv2 s2 siv2
   print b2 biv2 s2 siv2
endloop
```

5.5. **Binary Choice.** The statistical model contains only two parameters: β and σ. The latent variable is modeled $y_t^* = \beta x_t + u_t$, 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 parameter 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 β is.

Probit and Linear Probability Model

```
# Set the sample size and save it in n
nulldata 100
scalar n = $nobs

# generate n observations on x
series x = uniform()
# set a seed to get same results each time
set seed 3213799
# Set the values of the parameters
scalar slope = .5  # regression slope
scalar sigma = 1 # measurement error in y

# start the loop, indicating the desired number of samples.
loop 400 --progressive --quiet
   series u = normal(0,sigma)
   # generate samples of ystar and y
   series ystar=slope*x+u
   series y = (ystar>0)
   genr prop = mean(y)
   # run the ols regression
   ols y const x --robust
   # save the estimated coefficients
   genr b2 = $coeff(x)
   genr s2 = $stderr(x)
   # run the probit regression
   probit y const x --p-values
   genr bp2 = $coeff(x)
   genr sp2 = $stderr(x)
   genr ind = $coeff(const)+$coeff(x)*x
   genr d = dnorm(ind)
   genr ame = mean(d)*bp2
```

```
32    print b2 bp2 ame prop
33 endloop
```

In this exercise the data generation process is handled easily in lines 15, 17, and 18. The proportions of 1s to 0s is computed in line 19 and its summary statistics printed out via line 32. Then both least squares and the probit MLE are used to estimate the parameters of the binary choice model. In most courses great effort is made to discuss the fact that marginal effects in probit models are different at each observation. This creates great angst for many. In lines 29-31 the average marginal effects are computed for the regressor, x. The results show that in fact the AME of the probit model and the slope from least squares are nearly identical.

5.6. **Tobit.** To generate samples for tobit estimation requires an additional line in the probit script. 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. Thus, add the line

```
series y = y*ystar
```

after line 20 in the probit script. The generation of `y` becomes:

```
18    series ystar=slope*x+u
19    series y = (ystar>0)
20    genr prop = mean(y)
21    series y = y*ystar
```

Adding this after computing the proportions of 1s enables one 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 18, which generates the latent variable `ystar`, and change the actual threshold in line 19 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.

5.7. **Nonlinear Least Squares. Gretl** can also be used to estimate more complicated models in a simulation. It includes generic routines for estimating nonlinear least squares, maximum likelihood, and generalized method of moments estimators, the properties of which can also be studied in this way. In the following example, nonlinear least 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

$$C_t = \alpha + \beta Y_t^{\gamma} + u_t \tag{6}$$

where C_t is consumption and Y_t is output. Using data from Greene [1999] that is included in **gretl**, 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 4-6, starting values are acquired by least squares (9-12) and a few matrices are created in which results can be stored in lines 18 and 19.

Nonlinear least squares

```
1  open "greene11_3.gdt"
2  setobs 1 1 --cross-section
3  # Set the actual values of the parameters
4  scalar A = 180
5  scalar B = .25
6  scalar G = 1
7
8  # Starting values
9  ols C 0 Y
10 genr alpha0 = $coeff(0)
11 genr beta0 = $coeff(Y)
12 genr gamma0 = 1
13
14 # Set the number of Simulated Samples
15 scalar NMC = 1000
16
17 # Create an empty matrix to store results
18 matrix coeffs = zeros(NMC, 3)
19 matrix vcvs   = zeros(NMC, 6)
20
21 # Create systematic portion of the model and log(y)
22 series C0 = A + B*Y^G
23 series lY = log(Y)
```

In the last two lines, series are created for the systematic portion of the model and for the natural logarithm of Y, both of which only need to be generated once. Putting these outside of the loop reduces computations.

In the next section, the loop is created. The `nls` function will not work with the progressive option. This makes it marginally more difficult to accumulate results and to preserve them for further analysis. The easiest way to proceed is to use the index loop structure, as in line 26. The index loop makes it easy to accumulate results in a matrix as the **gretl** iterates.

```
24 # The loop
25 set warnings off
26 loop i=1..NMC --quiet
27     # generate new sample
28     genr C = C0 + normal(0,10)
29     # Initialize parameters
30     alpha = alpha0
31     beta  = beta0
32     gamma = gamma0
33
34     # Estimate parameters via NLS
35     nls C = alpha + beta * Y^gamma
36         deriv alpha = 1
37         deriv beta = Y^gamma
38         deriv gamma = beta * Y^gamma * lY
39     end nls --quiet
40
41     # Collect the coefficients into a vector and matrix
42     matrix coeffs[i,] = {alpha, beta, gamma}
43     matrix vcvs[i,] = vech($vcv)'
44 endloop
```

The nonlinear least squares estimator of the consumption function is found in lines 35-39. To speed computations, the analytical derivatives are given. The loop ends with the `--quiet` option in order to supress the output of 1000 `nls` iterations. In lines 42 and 43 the coefficients and the estimated variances are placed into the matrices `coeffs` and `vcvs` that were initialized in the first section of the program. As the index loop iterates, the coefficients and variances are placed in the ith row of these two matrices. Later, they will be written to data and analyzed.

In line 46 a new empty data set is opened that contains the number of Monte Carlo observations. If the `--preserve` option is not used, then the contents of all existing matrices that are held in memory will be emptied when the new data set is opened. In lines 49-51, each column of the saved coefficients matrix is written to a data series. The summary statistics are printed by line 54. Finally, in line 58 the Monte Carlo variance-covariance matrix is obtained. To compare these to the average value of the estimated variances, line 59 is included.

```
# 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

printf "Monte Carlo vcv vs average estimated vcv\n"

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

print MCV AEV
```

6. Conclusion

In this primer the basics of conducting a Monte Carlo experiment are discussed and the concepts are put to use in **gretl**. 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 **gretl** software is particularly well-suited for this purpose for several reasons, the main of which are its `--progressive` loop option and its transparent programming language. Through a series of examples the ease with which The examples includes linear regression, confidence intervals, the size and power of t-tests, lagged dependent variable models, heteroskedastic and autocorrelated regression models, instrumental variables estimators, and binary choice models. Scripts for all examples are provided in the paper and are available from the author's website.

Gretl can also be used to study the properties of more complicated estimators. This is demonstrated for nonlinear least squares, but the result could apply to method of moments or maximum likelihood estimation as well. For classroom use **gretl** is very highly recommended since nearly every estimator one is likely to use in a one year introductory econometrics course can be modeled and studied. Add to that that it is free and will work on any platform and **gretl** is hard to beat for learning econometrics. Because of its numerical accuracy, excellent scripting language, and ability to work with other open source software, it can also serve as an excellent platform for research.

References

Adkins, Lee C. [2010], *Using Gretl for Principles of Econometrics, Third Edition*.
URL: *http://www.learneconometrics.com/gretl/ebook.pdf*

Baiocchi, Giovanni and Walter Distaso [2003], 'Gretl: Econometric software for the gnu generation', *Journal of Applied Econometrics* **18**(1), 105–110.

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

Cottrell, Allin and Riccardo "Jack" Lucchetti [2010], *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* .

Greene, William [1999], *Econometric Analysis*, 4th edn, Prentice Hall.

Hill, R. Carter, William E. Griffiths and Guay C. Lim [2008], *Principles of Econometrics, Third Edition*, John Wiley & Sons, New York.

Judge, Guy [1999], 'Simple monte carlo studies on a spreadsheet', *Computers in Higher Education Economics Review* **13**(2).

Kennedy, Peter E. [2003], *A Guide to Econometrics, Fifth Edition*, MIT Press.

Mixon, J. Wilson Jr. and Ryan J. Smith [2006], 'Teaching undergraduate econometrics with gretl', *Journal of Applied Econometrics* **21**(7), 1103–1107.

Murray, Michael P. [2006], *Econometrics: A Modern Introduction*, Pearson Education, Boston.

Racine, J. [2006], '**gnuplot** 4.0: A portable interactive plotting utility', *Journal of Applied Econometrics* **21**, 133–141.

Rosenblad, Andreas [2008], 'gret1 1.7.3', *Journal of Statistical Software, Software Reviews* **25**(1), 1–14.

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.

Yalta, A. Talha and A. Yasemin Yalta [2007], 'Gretl 1.6.0 and its numerical accuracy', *Journal of Applied Econometrics* **22**(4), 849 854.

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

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