4 Simulation

4.1 Introduction

Simulation by Monte Carlo experimentation is a useful and powerful methodology for investigating the properties of econometric estimators and tests. The power of the methodology derives from being able to define and control the statistical environment in which the investigator specifies the data-generating process (DGP) and generates data used in subsequent experiments.

Monte Carlo experiments can be used to verify that valid methods of statistical inference are being used. An obvious example is checking a new computer program or algorithm. Another example is investigating the robustness of an established estimation or test procedure to deviations from settings where the properties of the procedure are known.

Even when valid methods are used, they often rely on asymptotic results. We may want to check whether these provide a good approximation in samples of the size typically available to the investigators. Also asymptotically equivalent procedures may have different properties in finite samples. Monte Carlo experiments enable finite-sample comparisons.

This chapter deals with the basic elements common to Monte Carlo experiments: computer generation of random numbers that mimic the theoretical properties of realizations of random variables; commands for repeated execution of a set of instructions; and machinery for saving, storing, and processing the simulation output, generated in an experiment, to obtain the summary measures that are used to evaluate the properties of the procedures under study. We provide a series of examples to illustrate various aspects of Monte Carlo analyses.

The chapter appears early in the book. Simulation is a powerful pedagogic tool for exposition and illustration of statistical concepts. At the simplest level, we can use pseudorandom samples to illustrate distributional features of artificial data. The goal of this chapter is to use simulation to study the distributional and moment properties of statistics in certain idealized statistical environments. Another possible use of the Monte Carlo methodology is to check the correctness of computer code. Many applied studies use methods complex enough that it is easy to make mistakes. Often these mistakes could be detected by an appropriate simulation exercise. We believe that simulation is greatly underutilized, even though Monte Carlo experimentation is relatively straightforward in Stata.

4.2 Pseudorandom-number generators: Introduction

Suppose we want to use simulation to study the properties of the ordinary least-squares estimator (OLS) estimator in the linear regression model with normal errors. Then, at the minimum, we need to make draws from a specified normal distribution. The literature on (pseudo) random-number generation contains many methods of generating such sequences of numbers. When we use packaged functions, we usually do not need to know the details of the method. Yet the match between the theoretical and the sample properties of the draws does depend upon such details.

Stata introduced a new suite of fast and easy-to-use random-number functions (generators) in mid-2008. These functions begin with the letter r (from random) and can be readily installed via an update to version 10. The suite includes the uniform, normal, binomial, gamma, and Poisson functions that we will use in this chapter, as well as several others that we do not use. The functions for generating pseudorandom numbers are summarized in `help functions`.

To a large extent, these new functions obviate the previous methods of using one's own generators or user-written commands to generate pseudorandom numbers other than the uniform. Nonetheless, there can sometimes be a need to make draws from distributions that are not included in the suite. For these draws, the uniform distribution is often the starting point. The new `runiform()` function generates exactly the same uniform draws as `uniform()`, which it replaces.

4.2.1 Uniform random-number generation

The term random-number generation is an oxymoron. It is more accurate to use the term pseudorandom numbers. Pseudorandom-number generators use deterministic devices to produce long chains of numbers that mimic the realizations from some target distribution. For uniform random numbers, the target distribution is the uniform distribution from 0 to 1, for which any value between 0 and 1 is equally likely. Given such a sequence, methods exist for mapping these into sequences of nonuniform draws from desired distributions such as the normal.

A standard simple generator for uniform draws uses the deterministic rule $X_j = (kX_{j-1} + c) \bmod m$, $j = 1, \ldots, J$, where the modulus operator $a \bmod b$ forms the remainder when a is divided by b, to produce a sequence of J integers between 0 and m. Then $R_j = X_j/m$ is a sequence of J numbers between 0 and 1. If computation is done using 32-bit integer arithmetic, then $m = 2^{31} - 1$ and the maximum periodicity is $2^{31} - 1 \simeq 2.1 \times 10^9$, but it is easy to select poor values of k, c, and X_0 so that the cycle repeats much more often than that.

This generator is implemented using Stata function `runiform()`, a 32-bit KISS generator that uses good values of k and c. The initial value for the cycle, X_0, is called the seed. The default is to have this set by Stata, based on the computer clock. For reproducibility of results, however, it is best to actually set the initial seed by using `set seed`. Then, if the program is rerun at a later time or by a different researcher, the same results will be obtained.

To obtain and display one draw from the uniform, type

```
. * Single draw of a uniform number
. set seed 10101
. scalar u = runiform()
. display u
.16796649
```

This number is internally stored at much greater precision than the eight displayed digits.

The following code obtains 1,000 draws from the uniform distribution and then provides some details on these draws:

```
. * 1000 draws of uniform numbers
. quietly set obs 1000
. set seed 10101
. generate x = runiform()
. list x in 1/5, clean
            x
1.   .1679665
2.   .3197621
3.   .7911349
4.   .7193382
5.   .5408687
. summarize x
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           x |      1000    .5150332    .2934123    .0002845   .9993234
```

The 1,000 draws have a mean of 0.515 and a standard deviation of 0.293, close to the theoretical values of 0.5 and $\sqrt{1/12} = 0.289$. A histogram, not given, has ten equal-width bins with heights that range from 0.8 to 1.2, close to the theory of equal heights of 1.0.

The draws should be serially uncorrelated, despite a deterministic rule being used to generate the draws. To verify this, we create a time-identifier variable, `t`, equal to the observation number (`_n`), and we use `tsset` to declare the data to be time series with time-identifier `t`. We could then use the `corrgram`, `ac`, and `pac` commands to test whether autocorrelations and partial autocorrelations are zero. We more simply use `pwcorr` to produce the first three autocorrelations, where `L2.x` is the `x` variable lagged twice and the `star(0.05)` option puts a star on correlations that are statistically significantly different from zero at level 0.05.

```
. * First three autocorrelations for the uniform draws
. generate t = _n
. tsset t
        time variable:  t, 1 to 1000
                delta:  1 unit
```

```
. pwcorr x L.x L2.x L3.x, star(0.05)
```

	x	L.x	L2.x	L3.x
x	1.0000			
L.x	-0.0185	1.0000		
L2.x	-0.0047	-0.0199	1.0000	
L3.x	0.0116	-0.0059	-0.0207	1.0000

The autocorrelations are low, and none are statistically different from zero at the 0.05 level. Uniform random-number generators used by packages such as Stata are, of course, subjected to much more stringent tests than these.

4.2.2 Draws from normal

For simulations of standard estimators such as OLS, nonlinear least squares (NLS), and instrumental variables (IV), all that is needed are draws from the uniform and normal distributions, because normal errors are a natural starting point and the most common choices of distribution for generated regressors are normal and uniform.

The command for making draws from the standard normal has the following simple syntax:

```
generate varname = rnormal()
```

To make draws from $N(m,s^2)$, the corresponding command is

```
generate varname = rnormal(m,s)
```

Note that $s > 0$ is the standard deviation. The arguments m and s can be numbers or variables.

Draws from the standard normal distribution also can be obtained as a transformation of draws from the uniform by using the inverse probability transformation method explained in section 4.4.1; that is, by using

```
generate varname = invnormal(runiform())
```

where the new function `runiform()` replaces `uniform()` in the older versions.

The following code generates and summarizes three pseudorandom variables with 1,000 observations each. The pseudorandom variables have distributions uniform(0,1), standard normal, and normal with a mean of 5 and a standard deviation of 2.

```
* normal and uniform
clear
quietly set obs 1000
set seed 10101                          // set the seed
generate uniform = runiform()           // uniform(0,1)
```

```
. generate stnormal = rnormal()            // N(0,1)
. generate norm5and2 = rnormal(5,2)
. tabstat uniform stnormal norm5and2, stat(mean sd skew kurt min max) col(stat)
```

variable	mean	sd	skewness	kurtosis	min	max
uniform	.5150332	.2934123	-.0899003	1.818878	.0002845	.9993234
stnormal	.0109413	1.010856	.0680232	3.130058	-2.978147	3.730844
norm5and2	4.995458	1.970729	-.0282467	3.050581	-3.027987	10.80905

The sample mean and other sample statistics are random variables; therefore, their values will, in general, differ from the true population values. As the number of observations grows, each sample statistic will converge to the population parameter because each sample statistic is a consistent estimator for the population parameter.

For norm5and2, the sample mean and standard deviation are very close to the theoretical values of 5 and 2. Output from tabstat gives a skewness statistic of -0.028 and a kurtosis statistic of 3.051, close to 0 and 3, respectively.

For draws from the truncated normal, see section 4.4.4, and for draws from the multivariate normal, see section 4.4.5.

4.2.3 Draws from t, chi-squared, F, gamma, and beta

Stata's library of functions contains a number of generators that allow the user to draw directly from a number of common continuous distributions. The function formats are similar to that of the rnormal() function, and the argument(s) can be a number or a variable.

Let $t(n)$ denote Students' t distribution with n degrees of freedom, $\chi^2(m)$ denote the chi-squared distribution with m degrees of freedom, and $F(h,n)$ denote the F distribution with h and n degrees of freedom. Draws from $t(n)$ and $\chi^2(h)$ can be made directly by using the rt(df) and rchi2(df) functions. We then generate $F(h,n)$ draws by transformation because a function for drawing directly from the F distribution is not available.

The following example generates draws from $t(10)$, $\chi^2(10)$, and $F(10,5)$.

```
. * t, chi-squared, and F with constant degrees of freedom
. clear
. quietly set obs 2000
. set seed 10101
. generate xt = rt(10)              // result xt ~ t(10)
. generate xc = rchi2(10)           // result xc ~ chisquared(10)
. generate xfn = rchi2(10)/10       // result numerator of F(10,5)
. generate xfd = rchi2(10)/5        // result denominator of F(10,5)
. generate xf = xfn/xfd             // result xf ~ F(10,5)
```

```
. summarize xt xc xf
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xt	2000	.0295636	1.118426	-5.390713	4.290518
xc	2000	9.967206	4.530771	.7512587	35.23849
xf	2000	1.637549	2.134448	.0511289	34.40774

The $t(10)$ draws have a sample mean and a standard deviation close to the theoretical values of 0 and $\sqrt{10/(10-2)} = 1.118$; the $\chi^2(10)$ draws have a sample mean and a standard deviation close to the theoretical values of 10 and $\sqrt{20} = 4.472$; and the $F(10,5)$ draws have a sample mean close to the theoretical value of $5/(5-2) = 1.7$. The sample standard deviation of 2.134 differs from the theoretical standard deviation of $\sqrt{2 \times 5^2 \times 13/(10 \times 3^2 \times 1)} = 2.687$. This is because of randomness, and a much larger number of draws eliminates this divergence.

Using `rbeta(a,b)`, we can draw from a two-parameter beta with the shape parameters $a, b > 0$, mean $a/(a+b)$, and variance $ab/(a+b)^2(a+b+1)$. Using `rgamma(a,b)`, we can draw from a two-parameter gamma with the shape parameter $a > 0$, scale parameter $b > 0$, mean ab, and variance ab^2.

4.2.4 Draws from binomial, Poisson, and negative binomial

Stata functions also generate draws from some leading discrete distributions. Again the argument(s) can be a number or a variable.

Let Bin(n,p) denote the binomial distribution with positive integer n trials (`n`) and success probability p, $0 < p < 1$, and let Poisson(m) denote the Poisson distribution with the mean or rate parameter m. The `rbinomial(n,p)` function generates random draws from the binomial distribution, and the `rpoisson(m)` function makes draws from the Poisson distribution.

We demonstrate these functions with an argument that is a variable so that the parameters differ across draws.

Independent (but not identically distributed) draws from binomial

As illustration, we consider draws from the binomial distribution, when both the probability p and the number of trials n may vary over i.

```
. * Discrete rv's: binomial
. set seed 10101
. generate p1 = runiform()                  // here p1~uniform(0,1)
. generate trials = ceil(10*runiform()) // here # trials varies btwn 1 & 10
. generate xbin = rbinomial(trials,p1)  // draws from binomial(n,p1)
```

```
summarize p1 trials xbin
```

Variable	Obs	Mean	Std. Dev.	Min	Max
p1	2000	.5155468	.2874989	.0002845	.9995974
trials	2000	5.438	2.887616	1	10
xbin	2000	2.753	2.434328	0	10

The DGP setup implies that the number of trials n is a random variable with an expected value of 5.5 and that the probability p is a random variable with an expected value of 0.5. Thus we expect that xbin has a mean of $5.5 \times 0.5 = 2.75$, and this is approximately the case here.

Independent (but not identically distributed) draws from Poisson

For simulating a Poisson regression DGP, denoted $y \sim \text{Poisson}(\mu)$, we need to make draws that are independent but not identically distributed, with the mean μ varying across draws because of regressors.

We do so in two ways. First, let μ_i equal xb=4+2*x with x=runiform(). Then $4 < \mu_i < 6$. Second, let μ_i equal xb times xg where xg=rgamma(1,1), which yields a draw from the gamma distribution with a mean of $1 \times 1 = 1$ and a variance of $1 \times 1^2 = 1$. Then $\mu_i > 0$. In both cases, the setup can be shown to be such that the ultimate draw has a mean of 5, but the variance differs from 5 for the independent and identically distributed (i.i.d.) Poisson because in neither case are the draws from an identical distribution. We obtain

```
. * Discrete rv's: independent poisson and negbin draws
. set seed 10101
. generate xb= 4 + 2*runiform()
. generate xg = rgamma(1,1)          // draw from gamma;E(v)=1
. generate xbh = xb*xg               // apply multiplicative heterogeneity
. generate xp = rpoisson(5)          // result xp - Poisson(5)
. generate xp1 = rpoisson(xb)        // result xp1 - Poisson(xb)
. generate xp2 = rpoisson(xbh)       // result xp2 - NB(xb)
. summarize xg xb xp xp1 xp2
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xg	2000	1.032808	1.044434	.000112	8.00521
xb	2000	5.031094	.5749978	4.000569	5.999195
xp	2000	5.024	2.300232	0	14
xp1	2000	4.976	2.239851	0	14
xp2	2000	5.1375	5.676945	0	44

The xb variable lies between 4 and 6, as expected, and the xg gamma variable has a mean and variance close to 1, as expected. For a benchmark comparison, we make draws of xp from Poisson(5), which has a sample mean close to 5 and a sample standard deviation close to $\sqrt{5} = 2.236$. Both xp1 and xp2 have means close to 5. In the case of xp2, the model has the multiplicative unobserved heterogeneity term xg that is itself drawn from a gamma distribution with shape and scale parameter both set to 1. Introducing

this type of heterogeneity means that xp2 is drawn from a distribution with the same mean as that of xp1, but the variance of the distribution is larger. More specifically, Var(xp2|xb) = xb*(1+xb), using results in section 17.2.2, leading to the much larger standard deviation for xp2.

The second example makes a draw from the Poisson–gamma mixture, yielding the negative binomial distribution. The `rnbinomial()` function draws from a different parameterization of the negative binomial distribution. For this reason, we draw from the Poisson–gamma mixture here and in chapter 17.

Histograms and density plots

For a visual depiction, it is often useful to plot a histogram or kernel density estimate of the generated random numbers. Here we do this for the draws xc from $\chi^2(10)$ and xp from Poisson(5). The results are shown in figure 4.1.

```
. * Example of histogram and kernel density plus graph combine
. quietly twoway (histogram xc, width(1)) (kdensity xc, lwidth(thick)),
> title("Draws from chisquared(10)")
. quietly graph save mus04cdistr.gph, replace
. quietly twoway (histogram xp, discrete) (kdensity xp, lwidth(thick) w(1)),
> title("Draws from Poisson(mu) for 5<mu<6")
. quietly graph save mus04poissdistr.gph, replace
. graph combine mus04cdistr.gph mus04poissdistr.gph,
> title("Random-number generation examples", margin(b=2) size(vlarge))
```

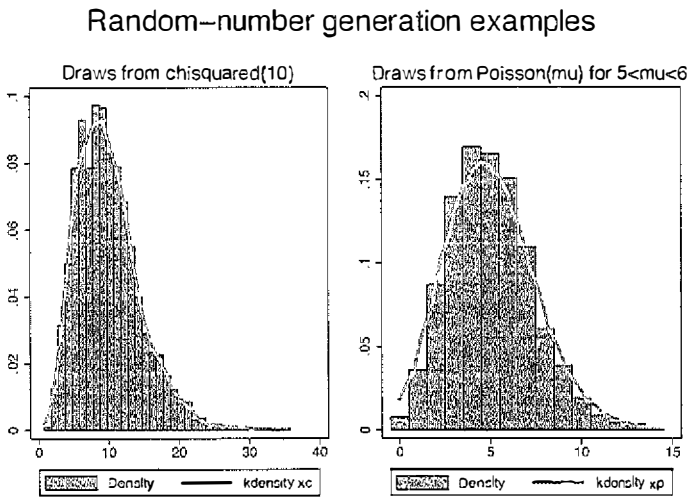

Figure 4.1. $\chi^2(10)$ and Poisson(5) draws

4.3 Distribution of the sample mean

As an introductory example of simulation, we demonstrate the central limit theorem result, $(\bar{x}_N - \mu)/(\sigma/\sqrt{N}) \rightarrow N(0,1)$; i.e., the sample mean is approximately normally distributed as $N \rightarrow \infty$. We consider a random variable that has the uniform distribution, and a sample size of 30.

We begin by drawing a single sample of size 30 of the random variable X that is uniformly distributed on $(0,1)$, using the `runiform()` random-number function. To ensure the same results are obtained in future runs of the same code or on other machines, we use `set seed`. We have

```
. * Draw 1 sample of size 30 from uniform distribution
. quietly set obs 30
. set seed 10101
. generate x = runiform()
```

To see the results, we use `summarize` and `histogram`. We have

```
. * Summarize x and produce a histogram
. summarize x
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
           x |         30   .5459987    .2803788   .0524637   .9983786

. quietly histogram x, width(0.1) xtitle("x from one sample")
```

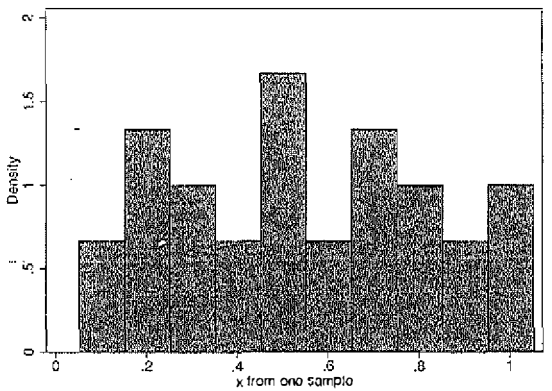

Figure 4.2. Histogram for one sample of size 30

The summary statistics show that 30 observations were generated and that for this sample $\bar{x} = 0.546$. The histogram for this single sample of 30 uniform draws, given in figure 4.2, looks nothing like the bell-shaped curve of a normal, because we are sampling from the uniform distribution. For very large samples, this histogram approaches a horizontal line with a density value of 1.

To obtain the distribution of the sample mean by simulation, we redo the preceding 10,000 times, obtaining 10,000 samples of size 30 and 10,000 sample means $\bar{x}$. These 10,000 sample means are draws from the distribution of the sample-mean estimator. By the central limit theorem, the distribution of the sample-mean estimator has approximately a normal distribution. Because the mean of a uniform(0, 1) distribution is 0.5, the mean of the distribution of the sample-mean estimator is 0.5. Because the standard deviation of a uniform(0, 1) distribution is $\sqrt{1/12}$ and each of the 10,000 samples is of size 30, the standard deviation of the distribution of the sample-mean estimator is $\sqrt{(1/12)/30} = 0.0527$.

4.3.1 Stata program

A mechanism for repeating the same statistical procedure 10,000 times is to write a program (see appendix A.2 for more details) that does the procedure once and use the `simulate` command to run the program 10,000 times.

We name the program `onesample` and define it to be r-class, meaning that the ultimate result, the sample mean for one sample, is returned in `r()`. Because we name this result `meanforonesample`, it will be returned in `r(meanforonesample)`. The program has no inputs, so there is no need for program arguments. The program drops any existing data on variable x, sets the sample size to 30, draws 30 uniform variates, and obtains the sample mean with `summarize`. The `summarize` command is itself an r-class command that stores the sample mean in `r(mean)`; see section 1.6.1. The last line of the program returns `r(mean)` as the result `meanforonesample`.

The program is

```
. * Program to draw 1 sample of size 30 from uniform and return sample mean
. program onesample, rclass
  1.      drop _all
  2.      quietly set obs 30
  3.      generate x = runiform()
  4.      summarize x
  5.      return scalar meanforonesample = r(mean)
  6. end
```

To check the program, we run it once, using the same seed as earlier. We obtain

```
. * Run program onesample once as a check
. set seed 10101
. onesample
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           x |        30    .5459987    .2803788   .0524637   .9983786

. return list
scalars:
   r(meanforonesample) =  .5459987225631873
```

The results for one sample are exactly the same as those given earlier.

4.3.2 The simulate command

The `simulate` command runs a specified command # times, where the user specifies #. The basic syntax is

`simulate` [*exp_list*], `reps(`#`)` [*options*]: *command*

where *command* is the name of the command, often a user-written program, and # is the number of simulations or replications. The quantities to be calculated and stored from *command* are given in *exp_list*. We provide additional details on `simulate` in section 4.6.1.

After `simulate` is run, the Stata dataset currently in memory is replaced by a dataset that has # observations, with a separate variable for each of the quantities given in *exp_list*.

4.3.3 Central limit theorem simulation

The `simulate` command can be used to run the `onesample` program 10,000 times, yielding 10,000 sample means from samples of size 30 of uniform variates. We additionally used options that set the seed and suppress the output of a dot for each of the 10,000 simulations. We have

```
. * Run program onesample 10,000 times to get 10,000 sample means
. simulate xbar = r(meanforonesample), seed(10101) reps(10000) nodots:
> onesample
      command:  onesample
         xbar:  r(meanforonesample)
```

The result from each sample, `r(meanforonesample)`, is stored as the variable `xbar`.

The `simulate` command overwrites any existing data with a dataset of 10,000 "observations" on $\bar{x}$. We summarize these values, expecting them to have a mean of 0.5 and a standard deviation of 0.0527. We also present a histogram overlaid by a normal density curve with a mean and standard deviation, which are those of the 10,000 values of $\bar{x}$. We have

```
. * Summarize the 10,000 sample means and draw histogram
. summarize xbar
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        xbar |     10000    .4995835    .0533809    .3008736   .6990562
. quietly histogram xbar, normal xtitle("xbar from many samples")
```

(*Continued on next page*)

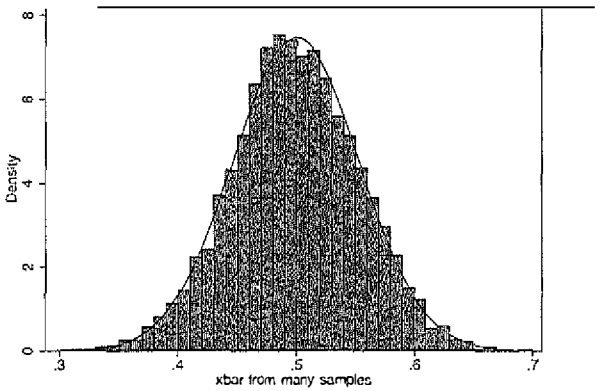

Figure 4.3. Histogram of the 10,000 sample means, each from a sample of size 30

The histogram given in figure 4.3 is very close to the bell-shaped curve of the normal.

There are several possible variations on this example. Different distributions for x can be used with different random-number functions in the `generate` command for x. As sample size (`set obs`) and number of simulations (`reps`) increases, the results become closer to a normal distribution.

4.3.4 The postfile command

In this book, we generally use `simulate` to perform simulations. An alternative method is to use a looping command, such as `forvalues`, and within each iteration of the loop use `post` to write (or post) key results to a file that is declared in the `postfile` command. After the loop ends, we then analyze the data in the posted file.

The `postfile` command has the following basic syntax:

`postfile` *postname newvarlist* `using` *filename* [`, every(#) replace`]

where *postname* is an internal filename, *newvarlist* contains the names of the variables to be put in the dataset, and *filename* is the external filename.

The `post` *postname* `(`*exp1*`)(`*exp2*`)...` command is used to write *exp1*, *exp2*, ... to the file. Each expression needs to be enclosed in parentheses.

The `postclose` *postname* command ends the posting of observations.

The `postfile` command offers more flexibility than `simulate` and, unlike `simulate`, does not lead to the dataset in memory being overwritten. For the examples in this book, `simulate` is adequate.

4.3.5 Alternative central limit theorem simulation

We illustrate the use of `postfile` for the central limit theorem example. We have

```
. * Simulation using postfile
. set seed 10101
. postfile sim_mem xmean using simresults, replace
. forvalues i = 1/10000 {
  2.      drop _all
  3.      quietly set obs 30
  4.      tempvar x
  5.      generate `x' = runiform()
  6.      quietly  summarize `x'
  7.      post sim_mem (r(mean))
  8. }
. postclose sim_mem
```

The `postfile` command declares the memory object in which the results are stored, the names of variables in the results dataset, and the name of the results dataset file. In this example, the memory object is named `sim_mem`, `xmean` will be the only variable in the results dataset file, and `simresults.dta` will be the results dataset file. (The `replace` option causes any existing `simresults.dta` to be replaced.) The `forvalues` loop (see section 1.8) is used to perform 10,000 repetitions. At each repetition, the sample mean, result `r(mean)`, is posted and will be included as an observation in the new `xmean` variable in `simresults.dta`.

To see the results, we need to open `simresults.dta` and `summarize`.

```
. * See the results stored in simresults
. use simresults, clear
. summarize
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xmean	10000	.4995835	.0533809	.3008736	.6990562

The results are identical to those in section 4.3.3 with `simulate` due to using the same seed and same sequence of evaluation of random-number functions.

The `simulate` command suppresses all output within the simulations. This is not the case for the `forvalues` loop, so the `quietly` prefix was used in two places in the code above. It can be more convenient to instead apply the `quietly` prefix to all commands in the entire `forvalues` loop.

4.4 Pseudorandom-number generators: Further details

In this section, we present further details on random-number generation that explain the methods used in section 4.2 and are useful for making draws from additional distributions.

Commonly used methods for generating pseudorandom samples include inverse-probability transforms, direct transformations, accept–reject methods, mixing and compounding, and Markov chains. In what follows, we emphasize application and refer the interested reader to Cameron and Trivedi (2005, ch. 12) or numerous other texts for additional details.

4.4.1 Inverse-probability transformation

Let $F(x) = \Pr(X \leq x)$ denote the cumulative distribution function of a random variable x. Given a draw of a uniform variate r, $0 \leq r \leq 1$, the inverse transformation $x = F^{-1}(r)$ gives a unique value of x because $F(x)$ is nondecreasing in x. If r approximates well a random draw from the uniform, then $x = F^{-1}(r)$ will approximate well a random draw from $F(x)$.

A leading application is to the standard normal distribution. Then the inverse of the cumulative distribution function (c.d.f.),

$$F(x) = \Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-z^2/2} dz$$

has no closed-form solution, and there is consequently no analytical expression for $\Phi^{-1}(x)$. Nonetheless, the inverse-transformation method is easy to implement because numerical analysis provides functions that calculate a very good approximation to $\Phi^{-1}(x)$. In Stata, the function is `invnormal()`. Combining the two steps of drawing a random uniform variate and evaluating the inverse c.d.f., we have

```
* Inverse probability transformation example: standard normal
quietly set obs 2000
set seed 10101
generate xstn = invnormal(runiform())
```

This method was presented in section 4.2.2 but is now superseded by the `rnormal()` function.

As another application, consider drawing from the unit exponential, with c.d.f. $F(x) = 1 - e^{-x}$. Solving $r = 1 - e^{-x}$ yields $x = -\ln(1 - r)$. If the uniform draw is, say, 0.640, then $x = -\ln(1 - 0.640) = 1.022$. With continuous monotonically increasing c.d.f., the inverse transformation yields a unique value of x, given r. The Stata code for generating a draw from the unit exponential illustrates the method:

```
. * Inverse probability transformation example: unit exponential
. generate xue = -ln(1-runiform())
```

For discrete random variables, the c.d.f. is a step function. Then the inverse is not unique, but it can be uniquely determined by a convention for choosing a value on the flat portion of the c.d.f., e.g., the left limit of the segment.

In the simplest case, we consider a Bernoulli random variable taking a value of 1 with a probability of p and a value of 0 with a probability of $1 - p$. Then we take a

uniform draw, u, and set $y = 1$ if $u \leq p$ and $y = 0$ if $u > p$. Thus, if $p = 0.6$, we obtain the following:

```
* Inverse probability transformation example: Bernoulli (p = 0.6)
generate xbernoulli = runiform() > 0.6   // Bernoulli(0.6)
summarize xstn xue xbernoulli
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xstn	2000	.0481581	1.001728	-3.445941	3.350993
xue	2000	.9829519	1.000921	.0003338	9.096659
xbernoulli	2000	.4055	.4911113	0	1

This code uses a logical operator that sets $y = 1$ if the condition is met and $y = 0$ otherwise; see section 2.4.7.

A more complicated discrete example is the Poisson distribution because then the random variable can potentially take an infinite number of values. The method is to sequentially calculate the c.d.f. $\Pr(Y \leq k)$ for $k = 0, 1, 2, \ldots$. Then stop when the first $\Pr(Y \leq k) > u$, where u is the uniform draw, and set $y = k$. For example, consider the Poisson with a mean of 2 and a uniform draw of 0.701. We first calculate $\Pr(y \leq 0) = 0.135 < u$, then calculate $\Pr(y \leq 1) = 0.406 < u$, then calculate $\Pr(y \leq 2) = 0.677 < u$, and finally calculate $\Pr(y \leq 3) = 0.857$. This last calculation exceeds the uniform draw of 0.701, so stop and set $y = 3$. $\Pr(Y \leq k)$ is computed by using the recursion $\Pr(Y \leq k) = \Pr(Y \leq k-1) + \Pr(Y = k)$.

4.4.2 Direct transformation

Suppose we want to make draws from the random variable Y, and from probability theory, it is known that Y is a transformation of the random variable X, say, $Y = g(X)$.

In this situation, the direct transformation method obtains draws of Y by drawing X and then applying the transformation $g(\cdot)$. The method is clearly attractive when it is easy to draw X and evaluate $g(\cdot)$.

Direct transformation is particularly easy to illustrate for well-known transforms of a standard normally distributed random variable. A $\chi^2(1)$ draw can be obtained as the square of a draw from the standard normal; a $\chi^2(m)$ draw is the sum of m independent draws from $\chi^2(1)$; an $F(m_1, m_2)$ draw is $(v_1/m_1)/(v_2/m_2)$, where v_1 and v_2 are independent draws from $\chi^2(m_1)$ and $\chi^2(m_2)$; and a $t(m)$ draw is $u/\sqrt{v/m}$ where u and v are independent draws from $N(0,1)$ and $\chi^2(m)$.

4.4.3 Other methods

In some cases, a distribution can be obtained as a mixture of distributions. A leading example is the negative binomial, which can be obtained as a Poisson–gamma mixture (see section 4.2.4). Specifically, if $y|\lambda$ is Poisson(λ) and $\lambda|\mu, \alpha$ is gamma with a mean of μ and a variance of $\alpha\mu$, then $y|\mu, \alpha$ is a negative binomial distributed with a mean

of μ and a variance of $\mu + \alpha\mu^2$. This implies that we can draw from the negative binomial by using a two-step method in which we first draw (say, ν) from the gamma distribution with a mean equal to 1 and then, conditional on ν, draw from Poisson($\mu\nu$). This example, using mixing, is used again in chapter 17.

More-advanced methods include accept–reject algorithms and importance sampling. Many of Stata's pseudorandom-number generators use accept–reject algorithms. Type `help random number functions` for more information on the methods used by Stata.

4.4.4 Draws from truncated normal

In simulation-based estimation for latent normal models with censoring or selection, it is often necessary to generate draws from a truncated normal distribution. The inverse-probability transformation can be extended to obtain draws in this case.

Consider making draws from a truncated normal. Then $X \sim TN_{(a,b)}(\mu, \sigma^2)$, where without truncation $X \sim N(\mu, \sigma^2)$. With truncation, realizations of X are restricted to lie between left truncation point a and right truncation point b.

For simplicity, first consider the standard normal case ($\mu = 0$, $\sigma = 1$) and let $Z \sim N(0,1)$. Given the draw u from the uniform distribution, x is defined by the solution of the inverse-probability transformation equation

$$u = F(x) = \frac{\Pr(a \leq Z \leq x)}{\Pr(a \leq Z \leq b)} = \frac{\Phi(x) - \Phi(a)}{\Phi(b) - \Phi(a)}$$

Rearranging, $\Phi(x) = \Phi(a) + \{\Phi(b) - \Phi(a)\}u$ so that solving for x we obtain

$$x = \Phi^{-1}[\Phi(a) + \{\Phi(b) - \Phi(a)\}\, u]$$

To extend this to the general case, note that if $Z \sim N(\mu, \sigma^2)$ then $Z^* = (Z - \mu)/\sigma \sim N(0,1)$, and the truncation points for Z^*, rather than Z, are $a^* = (a - \mu)/\sigma$ and $b^* = (b - \mu)/\sigma$. Then

$$x = \mu + \sigma\Phi^{-1}[\Phi(a^*) + \{\Phi(b^*) - \Phi(a^*)\}\, u]$$

As an example, we consider draws from $N(5, 4^2)$ for a random variable truncated to the range $[0, 12]$.

```
* Draws from truncated normal x ~ N(mu,sigma^2) in [a,b]
quietly set obs 2000
set seed 10101
scalar a = 0                          // lower truncation point
scalar b = 12                         // upper truncation point
scalar mu = 5                         // mean
scalar sigma = 4                      // standard deviation
generate u = runiform()
```

```
. generate w=normal((a-mu)/sigma)+u*(normal((b-mu)/sigma)-normal((a-mu)/sigma))
. generate xtrunc = mu + sigma*invnormal(w)
. summarize xtrunc
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xtrunc	2000	5.605522	2.944887	.005319	11.98411

Here there is more truncation from below, because a is 1.25σ from μ whereas b is 1.75σ from μ, so we expect the truncated mean to exceed the untruncated mean. Accordingly, the sample mean is 5.606 compared with the untruncated mean of 5. Truncation reduces the range and, for most but not all distributions, will reduce the variability. The sample standard deviation of 2.945 is less than the untruncated standard deviation of 4.

An alternative way to draw $X \sim TN_{(a,b)}(\mu, \sigma^2)$ is to keep drawing from untruncated $N(\mu, \sigma^2)$ until the realization lies in (a, b). This method will be very inefficient if, for example, $(a, b) = (-0.01, 0.01)$. A Poisson example is given in section 17.3.5.

4.4.5 Draws from multivariate normal

Making draws from multivariate distributions is generally more complicated. The method depends on the specific case under consideration, and inverse-transformation methods and transformation methods that work in the univariate case may no longer apply.

However, making draws from the multivariate normal is relatively straightforward because, unlike most other distributions, linear combinations of normals are also normal.

Direct draws from multivariate normal

The drawnorm command generates draws from $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ for the user-specified vector $\boldsymbol{\mu}$ and matrix $\boldsymbol{\Sigma}$. For example, consider making 200 draws from a standard bivariate normal distribution with means of 10 and 20, variances of 4 and 9, and a correlation of 0.5 (so the covariance is 3).

```
. * Bivariate normal example:
. * means 10, 20; variances 4, 9; and correlation 0.5
. clear
. quietly set obs 1000
. set seed 10101
. matrix MU = (10,20)                    // MU is 2 x 1
. scalar sig12 = 0.5*sqrt(4*9)
. matrix SIGMA = (4, sig12 \ sig12, 9)   // SIGMA is 2 x 2
. drawnorm y1 y2, means(MU) cov(SIGMA)
```

```
. summarize y1 y2
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          y1 |      1000    10.08618    2.082605    3.108118   16.40892
          y2 |      1000    20.20292    2.999583    10.12452   29.79675

. correlate y1 y2
(obs=1000)
             |       y1       y2
-------------+------------------
          y1 |   1.0000
          y2 |   0.5553   1.0000
```

The sample means are close to 10 and 20, and the standard deviations are close to $\sqrt{4} = 2$ and $\sqrt{9} = 3$. The sample correlation of 0.5553 differs somewhat from 0.50, though this difference disappears for much larger sample sizes.

Transformation using Cholesky decomposition

The method uses the result that if $\mathbf{z} \sim N(0, \mathbf{I})$ then $\mathbf{x} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z} \sim N(\boldsymbol{\mu}, \mathbf{L}\mathbf{L}')$. It is easy to draw $\mathbf{z} \sim N(0, \mathbf{I})$ because $\mathbf{z}$ is just a column vector of univariate normal draws. The transformation method to make draws of $\mathbf{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ evaluates $\mathbf{x} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$, where the matrix $\mathbf{L}$ satisfies $\mathbf{L}\mathbf{L}' = \boldsymbol{\Sigma}$. More than one matrix $\mathbf{L}$ satisfies $\mathbf{L}\mathbf{L}' = \boldsymbol{\Sigma}$, the matrix analog of the square root of $\boldsymbol{\Sigma}$. Standard practice is to use the Cholesky decomposition that restricts $\mathbf{L}$ to be a lower triangular matrix. Specifically, for the trivariate normal distribution, let $E(\mathbf{z}\mathbf{z}') = \boldsymbol{\Sigma} = \mathbf{L}\mathbf{z}\mathbf{z}'\mathbf{L}'$, where $\mathbf{z} \sim N(\mathbf{0}, \mathbf{I}_3)$ and

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}$$

Then the following transformations of $\mathbf{z}' = (z_1 \; z_2 \; z_3)$ yield the desired multivariate normal vector $\mathbf{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$:

$$\begin{aligned} x_1 &= \mu_1 + l_{11}z_1 \\ x_2 &= \mu_2 + l_{21}z_1 + l_{22}z_2 \\ x_3 &= \mu_3 + l_{31}z_1 + l_{32}z_2 + l_{33}z_3 \end{aligned}$$

4.4.6 Draws using Markov chain Monte Carlo method

In some cases, making direct draws from a target joint (multivariate) distribution is difficult, so the objective must be achieved in a different way. However, if it is also possible to make draws from the distribution of a subset, conditional on the rest, then one can create a Markov chain of draws. If one recursively makes draws from the conditional distribution and if a sufficiently long chain is constructed, then the distribution of the draws will, under some conditions, converge to the distribution of independent draws from the stationary joint distribution. This so-called Markov chain Monte Carlo method is now standard in modern Bayesian inference.

To be concrete, let $\mathbf{Y} = (Y_1, Y_2)$ have a bivariate density of $f(\mathbf{Y}) = f(Y_1, Y_2)$, and suppose the two conditional densities $f(Y_1|Y_2)$ and $f(Y_2|Y_1)$ are known and that it is possible to make draws from these. Then it can be shown that alternating sequential draws from $f(Y_1|Y_2)$ and $f(Y_2|Y_1)$ converge in the limit to draws from $f(Y_1, Y_2)$, even though in general $f(Y_1, Y_2) \neq f(Y_1|Y_2)f(Y_2|Y_1)$ (recall that $f(Y_1, Y_2) = f(Y_1|Y_2)f(Y_2)$). The repeated recursive sampling from $f(Y_1|Y_2)$ and $f(Y_2|Y_1)$ is called the Gibbs sampler.

We illustrate the Markov chain Monte Carlo approach by making draws from a bivariate normal distribution, $f(Y_1, Y_2)$. Of course, using the drawnorm command, it is quite straightforward to draw samples from the bivariate normal. So the application presented is illustrative rather than practical. The relative simplicity of this method comes from the fact that the conditional distributions $f(Y_1|Y_2)$ and $f(Y_2|Y_1)$ derived from a bivariate normal are also normal.

We draw bivariate normal data with means of 0, variances of 1, and a correlation of $\rho = 0.9$. Then $Y_1|Y_2 \sim N\{0, (1-\rho^2)\}$ and $Y_2|Y_1 \sim N\{0, (1-\rho^2)\}$. Implementation requires looping that is much easier using matrix programming language commands. The following Mata code implements this algorithm by using commands explained in appendix B.2.

```
. * MCMC example: Gibbs for bivariate normal mu's=0 v's=1 corr=rho=0.9
. set seed 10101
. clear all
. set obs 1000
obs was 0, now 1000
. generate double y1 =.
(1000 missing values generated)
. generate double y2 =.
(1000 missing values generated)
. mata:
------------------------------------------------- mata (type end to exit) -----
:   s0 = 10000              // Burn-in for the Gibbs sampler (to be discarded)
:   s1 = 1000               // Actual draws used from the Gibbs sampler
:   y1 = J(s0+s1,1,0)       // Initialize y1
:   y2 = J(s0+s1,1,0)       // Initialize y2
:   rho = 0.90              // Correlation parameter
:   for(i=2; i<=s0+s1; i++) {
>       y1[i,1] = ((1-rho^2)^0.5)*(rnormal(1, 1, 0, 1)) + rho*y2[i-1,1]
>       y2[i,1] = ((1-rho^2)^0.5)*(rnormal(1, 1, 0, 1)) + rho*y1[i,1]
>   }
:   y = y1,y2
:   y = y[(s0+1),1 \ (s0+s1),.]  // Drop the burn-ins
:   mean(y)                     // Means of y1, y2
                1              2
    ------------------------------------------
  1 | .0831308345   .0647158328 |
    ------------------------------------------
```

```
: variance(y)                    // Variance matrix of y1, y2
[symmetric]
                 1              2
    ┌──────────────────────────────┐
  1 │  1.104291499                 │
  2 │  1.005053494   1.108773741   │
    └──────────────────────────────┘

: correlation(y)                 // Correlation matrix of y1, y2
[symmetric]
                 1              2
    ┌──────────────────────────────┐
  1 │             1                │
  2 │  .9082927488                 1 │
    └──────────────────────────────┘

: end
```

Many draws may be needed before the chain converges. Here we assume that 11,000 draws are sufficient, and we discard the first 10,000 draws; the remaining 1,000 draws are kept. In a real application, one should run careful checks to ensure that the chain has indeed converged to the desired bivariate normal. For the example here, the sample means of Y_1 and Y_2 are 0.08 and 0.06, differing quite a bit from 0. Similarly, the sample variances of 1.10 and 1.11 differ from 1 and the sample covariance of 1.01 differs from 0.9, while the implied correlation is 0.91 as desired. A longer Markov chain or longer burn-in may be needed to generate numbers with desired properties for this example with relatively high ρ.

Even given convergence of the Markov chain, the sequential draws of any random variable will be correlated. The output below shows that for the example here, the first-order correlation of sequential draws of y_2 is 0.823.

```
mata:
                                        mata (type end to exit) ------
  :y2 = y[|2,2 \ s1,2|]
  :y2lag1 = y[|1,2 \ (s1-1),2|]
  :y2andlag1 = y2,y2lag1
  :correlation(y2andlag1,1)      // Correlation between y2 and y2 lag 1
[symmetric]
                 1         2
    ┌──────────────────────────┐
  1 │             1            │
  2 │  .822692407              1 │
    └──────────────────────────┘

  end
```

4.5 Computing integrals

Some estimation problems may involve definite or indefinite integrals. In such cases, the integral may be numerically calculated.

4.5.1 Quadrature

For one-dimensional integrals of the form $\int_a^b f(y)dy$, where possibly $a = -\infty$, $b = \infty$, or both, Gaussian quadrature is the standard method. This approximates the integral by a weighted sum of m terms, where a larger m gives a better approximation and often even $m = 20$ can give a good approximation. The formulas for the weights are quite complicated but are given in standard numerical analysis books.

One-dimensional integrals often appear in regression models with a random intercept or random effect. In many nonlinear models, this random effect does not integrate out analytically. Most often, the random effect is normal so that integration is over $(-\infty, \infty)$ and Gauss–Hermite quadrature is used. A leading example is the random-effects estimator for nonlinear panel models fitted using various `xt` commands. For Stata code, see, for example, the user-written command `rfprobit.do` for a random-effects probit package or file `gllamm.ado` for generalized linear additive models.

4.5.2 Monte Carlo integration

Suppose the integral is of the form

$$E\{h(Y)\} = \int_a^b h(y)g(y)dy$$

where $g(y)$ is a density function. This can be estimated by the direct Monte Carlo integral estimate

$$\widehat{E}\{h(Y)\} = S^{-1}\sum_{s=1}^{S} h(y^s)$$

where $y^1, \ldots, y^S$ are S independent pseudorandom numbers from the density $g(y)$, obtained by using methods described earlier. This method works if $E\{h(Y)\}$ exists and $S \to \infty$.

This method can be applied to both definite and indefinite integrals. It has the added advantage of being immediately applicable to multidimensional integrals, provided we can draw from the appropriate multivariate distribution. It has the disadvantage that it will always provide an estimate, even if the integral does not exist. For example, to obtain $E(Y)$ for the Cauchy distribution, we could average S draws from the Cauchy. But this would be wrong because the mean of the Cauchy does not exist.

As an example, we consider the computation of $E[\exp\{-\exp(Y)\}]$ when $y \sim N(0,1)$. This is the integral:

$$E[\exp\{-\exp(Y)\}] = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\{-\exp(y)\}\exp\left(-y^2/2\right) dy$$

It has no closed-form solution but can be proved to exist. We use the estimate

$$\widehat{E}[\exp\{-\exp(Y)\}] = \frac{1}{S}\sum_{s=1}^{S} \exp\{-\exp(y^s)\}$$

where y^s is the sth draw of S draws from the $N(0,1)$ distribution.

This approximation task can be accomplished for a specified value of S, say, 100, by using the following code.

```
* Integral evaluation by Monte Carlo simulation with S=100
clear all
quietly set obs 100
set seed 10101
generate double y = invnormal(runiform())
generate double gy = exp(-exp(y))
quietly summarize gy, meanonly
scalar Egy = r(mean)
display "After 100 draws the MC estimate of E[exp(-exp(x))] is " Egy
After 100 draws the MC estimate of E[exp(-exp(x))] is .3524417
```

The Monte Carlo estimate of the integral is 0.352, based on 100 draws.

4.5.3 Monte Carlo integration using different S

It is not known in advance what value of S will yield a good Monte Carlo approximation to the integral. We can compare the outcome for several different values of S (including $S = 100$), stopping when the estimates stabilize.

To investigate this, we replace the preceding code by a Stata program that has as an argument S, the number of simulations. The program can then be called and run several times with different values of S.

The program is named `mcintegration`. The number of simulations is passed to the program as a named positional argument, `numsims`. This variable is a local variable within the program that needs to be referenced using quotes. The call to the program needs to include a value for `numsims`. Appendix A.2 provides the details on writing a Stata program. The program is r-class and returns results for a single scalar, $E\{g(y)\}$, where $g(y) = \exp\{-\exp(y)\}$.

```
* Program mcintegration to compute Eg(y) numsims times
program mcintegration, rclass
1.      version 10.1
2.      args numsims        // Call to program will include value for numsims
3.      drop _all
4.      quietly set obs `numsims'
5.      set seed 10101
6.      generate double y = rnormal(0)
7.      generate double gy = exp(-exp(y))
8.      quietly summarize gy, meanonly
9.      scalar Egy = r(mean)
10.     display "#simulations: " %9.0g `numsims' ///
>       "  MC estimate of  E[exp(-exp(x))] is " Egy
11. end
```

The program is then run several times, for $S = 10, 100, 1000, 10000$, and 100000.

```
. * Run program mcintegration S = 10, 100, ...., 100000 times
. mcintegration 10
#simulations:        10  MC estimate of  E[exp-exp(x)] is .30979214
. mcintegration 100
#simulations:       100  MC estimate of  E[exp-exp(x)] is .3714466
. mcintegration 1000
#simulations:      1000  MC estimate of  E[exp-exp(x)] is .38146534
. mcintegration 10000
#simulations:     10000  MC estimate of  E[exp-exp(x)] is .38081373
. mcintegration 100000
#simulations:    100000  MC estimate of  E[exp-exp(x)] is .38231031
```

The estimates of $E\{g(y)\}$ stabilize as $S \rightarrow \infty$, but even with $S = 10^5$, the estimate changes in the third decimal place.

4.6 Simulation for regression: Introduction

The simplest use of simulation methods is to generate a single dataset and estimate the DGP parameter θ. Under some assumptions, if the estimated parameter $\widehat{\theta}$ differs from θ for a large sample size, the estimator is probably inconsistent. We defer an example of this simpler simulation to section 4.6.4.

More often, θ is estimated from each of S generated datasets, and the estimates are stored and summarized to learn about the distribution of $\widehat{\theta}$ for a given DGP. For example, this approach is necessary if one wants to check the validity of a standard error estimator or the finite-sample size of a test. This approach requires the ability to perform the same analysis S times and to store the results from each simulation. The simplest approach is to write a Stata program for the analysis of one simulation and then use `simulate` to run this program many times.

4.6.1 Simulation example: OLS with χ^2 errors

In this section, we use simulation methods to investigate the finite-sample properties of the OLS estimator with random regressors and skewed errors. If the errors are i.i.d., the fact that they are skewed has no effect on the large-sample properties of the OLS estimator. However, when the errors are skewed, we will need a larger sample size for the asymptotic distribution to better approximate the finite-sample distribution of the OLS estimator than when the errors are normal. This example also highlights an important modeling decision: when y is skewed, we sometimes choose to model $E(\ln y|\mathbf{x})$ instead of $E(y|\mathbf{x})$ because we believe the disturbances enter multiplicatively instead of additively. This choice is driven by the multiplicative way the error affects the outcome and is independent of the functional form of its distribution. As illustrated in this simulation, the asymptotic theory for the OLS estimator works well when the errors are i.i.d. from a skewed distribution.

We consider the following DGP:

$$y = \beta_1 + \beta_2 x + u; \quad u \sim \chi^2(1) - 1; \quad x \sim \chi^2(1)$$

where $\beta_1 = 1$, $\beta_2 = 2$, and the sample size $N = 150$. For this DGP, the error u is independent of the regressor x (ensuring consistency of OLS) and has a mean of 0, variance of 2, skewness of $\sqrt{8}$, and kurtosis of 15. By contrast, a normal error has a skewness of 0 and a kurtosis of 3.

We wish to perform 1,000 simulations, where in each simulation we obtain parameter estimates, standard errors, t-values for the t test of $H_0 : \beta_2 = 2$, and the outcome of a two-sided test of H_0 at level 0.05.

Two of the most frequently changed parameters in a simulation study are the sample size and the number of simulations. For this reason, these two parameters are almost always stored in something that can easily be changed. We use global macros. In the output below, we store the number of observations in the global macro numobs and the number of repetitions in the global macro numsims. We use these global macros in the examples in this section.

```
* defining global macros for sample size and number of simulations
global numobs 150              // sample size N
global numsims "1000"          // number of simulations
```

We first write the chi2data program, which generates data on y, performs OLS, and returns $\widehat{\beta}_2$, $s_{\widehat{\beta}_2}$, $t_2 = (\widehat{\beta}_2 - 2)/s_{\widehat{\beta}_2}$, a rejection indicator $r_2 = 1$ if $|t_2| > t_{0.025}(148)$, and the p-value for the two-sided t test. The chi2data program is an r-class program, so these results are returned in r() using the return command.

```
* Program for finite-sample properties of OLS
program chi2data, rclass
1.      version 10.1
2.      drop _all
3.      set obs $numobs
4.      generate double x = rchi2(1)
5.      generate y = 1 + 2*x + rchi2(1)-1      // demeaned chi^2 error
6.      regress y x
7.      return scalar b2 =_b[x]
8.      return scalar se2 = _se[x]
9.      return scalar t2 = (_b[x]-2)/_se[x]
10.     return scalar r2 = abs(return(t2))>invttail($numobs-2,.025)
11.     return scalar p2 = 2*ttail($numobs-2,abs(return(t2)))
12. end
```

Instead of computing the t statistic and p-value by hand, we could have used test, which would have computed an F statistic with the same p-value. We perform the computations manually for pedagogical purposes. The following output illustrates that test and the manual calculations yield the same p-value.

```
set seed 10101
quietly chi2data
```

```
. return list
scalars:
                  r(p2) =  .0419507319188174
                  r(r2) =  1
                  r(t2) =  2.051809742705663
                 r(se2) =  .0774765767688598
                  r(b2) =  2.15896719504583
. quietly test x=2
. return list
scalars:
               r(drop) =  0
               r(df_r) =  148
                  r(F) =  4.209923220261881
                 r(df) =  1
                  r(p) =  .0419507319188174
```

Below we use simulate to call chi2data $numsims times and to store the results; here $numsims = 1000. The current dataset is replaced by one with the results from each simulation. These results can be displayed by using summarize, where obs in the output refers to the number of simulations and not the sample size in each simulation. The summarize output indicates that 1) the mean of the point estimates is very close to the true value of 2, 2) the standard deviation of the point estimates is close to the mean of the standard errors, and 3) the rejection rate of 0.046 is very close to the size of 0.05.

```
. * Simulation for finite-sample properties of OLS
. simulate b2f=r(b2) se2f=r(se2) t2f=r(t2) reject2f=r(r2) p2f=r(p2),
> reps($numsims) saving(chi2datares, replace) nolegend nodots: chi2data
. summarize b2f se2f reject2f
```

Variable	Obs	Mean	Std. Dev.	Min	Max
b2f	1000	2.000502	.0842622	1.719513	2.40565
se2f	1000	.0839736	.0172607	.0415919	.145264
reject2f	1000	.046	.2095899	0	1

Below we use mean to obtain 95% confidence intervals for the simulation averages. The results for b2f and the rejection rate indicate that there is no significant bias and that the asymptotic distribution approximated the finite-sample distribution well for this DGP with samples of size 150. The confidence interval for the standard errors includes the sample standard deviation for b2f, which is another indication that the large-sample theory provides a good approximation to the finite-sample distribution.

```
. mean b2f se2f reject2f
Mean estimation                   Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2f	2.000502	.0026646	1.995273	2.005731
se2f	.0839736	.0005458	.0829025	.0850448
reject2f	.046	.0066278	.032994	.059006

Further information on the distribution of the results can be obtained by using the `summarize, detail` and `kdensity` commands.

4.6.2 Interpreting simulation output

We consider in turn unbiasedness of $\widehat{\beta}_2$, correctness of the standard-error formula for $s_{\widehat{\beta}_2}$, distribution of the t statistic, and test size.

Unbiasedness of estimator

The average of $\widehat{\beta}_2$ over the 1,000 estimates, $\overline{\widehat{\beta}_2} = (1/1000)\sum_{s=1}^{1000}\widehat{\beta}_s$, is the simulation estimate of $E(\widehat{\beta}_2)$. Here $\overline{\widehat{\beta}_2} = 2.001$ (see the mean of `b2f`) is very close to the DGP value $\beta_2 = 2.0$, suggesting that the estimator is unbiased. However, this comparison should account for simulation error. From the `mean` command, the simulation yields a 95% confidence interval for $E(\widehat{\beta}_2)$ of [1.995, 2.006]. This interval is quite narrow and includes 2.0, so we conclude that $E(\widehat{\beta}_2)$ is unbiased.

Many estimators, particularly nonlinear estimators, are biased in finite samples. Then exercises such as this can be used to estimate the magnitude of the bias in typical sample sizes. If the estimator is consistent, then any bias should disappear as the sample size N goes to infinity.

Standard errors

The variance of $\widehat{\beta}_2$ over the 1,000 estimates, $s^2_{\widehat{\beta}_2} = (1/999)\sum_{s=1}^{1000}(\widehat{\beta}_s - \overline{\widehat{\beta}_2})^2$, is the simulation estimate of $\sigma^2_{\widehat{\beta}_2} = \text{Var}(\widehat{\beta}_2)$, the variance of $\widehat{\beta}_2$. Similarly, $\overline{s_{\widehat{\beta}_2}} = 0.084$ (see the standard deviation of `b2f`) is the simulation estimate of $\sigma_{\widehat{\beta}_2}$. Here $\overline{\text{se}(\widehat{\beta}_2)} = 0.084$ (see the mean of `se2f`) and the 95% confidence interval for $\text{se}(\widehat{\beta}_2)$ is [0.083, 0.085]. Since this interval includes $s_{\widehat{\beta}_2} = 0.084$, there is no evidence that $\text{se}(\widehat{\beta}_2)$ is biased for $\sigma_{\widehat{\beta}_2}$, which means that the asymptotic distribution is approximating the finite-sample distribution well.

In general, that $\{\text{se}(\widehat{\beta}_2)\}^2$ is unbiased for $\sigma^2_{\widehat{\beta}_2}$ does not imply that upon taking the square root $\text{se}(\widehat{\beta}_2)$ is unbiased for $\sigma_{\widehat{\beta}_2}$.

t statistic

Because we impose looser restrictions on the DGP, t statistics are not exactly t distributed and z statistics are not exactly z distributed. However, the extent to which they diverge from the reference distribution disappears as the sample size increases. The output below generates the graph in figure 4.4, which compares the density of the t statistics with the $t(148)$ distribution.

```
. kdensity t2f, n(1000) gen(t2_x t2_d) nograph
. generate double t2_d2 = tden(148, t2_x)
. graph twoway (line t2_d t2_x) (line t2_d2 t2_x)
```

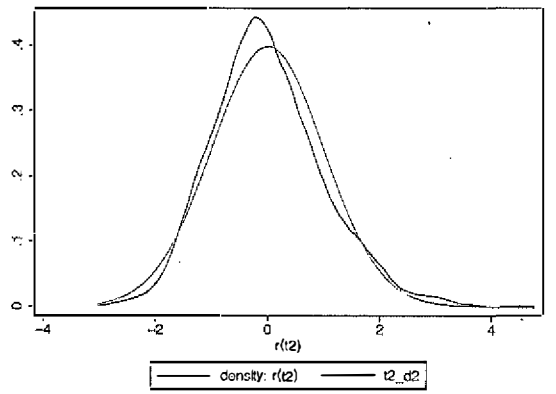

Figure 4.4. t statistic density against asymptotic distribution

Although the graph highlights some differences between the finite-sample and the asymptotic distributions, the divergence between the two does not appear to be great. Rather than focus on the distribution of the t statistics, we instead focus on the size of tests or coverage of confidence intervals based on these statistics.

Test size

The size of the test is the probability of rejecting H_0 when H_0 is true. Because the DGP sets $\beta_2 = 2$, we consider a two-sided test of H_0: $\beta_2 = 2$ against H_a: $\beta_2 \neq 2$. The level or nominal size of the test is set to 0.05, and the t test is used. The proportion of simulations that lead to a rejection of H_0 is known as the rejection rate, and this proportion is the simulation estimate of the true test size. Here the estimated rejection rate is 0.046 (see the mean of `reject2f`). The associated 95% confidence interval (from `mean reject2f`) is [0.033, 0.059], which is quite wide but includes 0.05. The width of this confidence interval is partially a result of having run only 1,000 repetitions, and partially an indication that, with 150 observations, the true size of the test can differ from the nominal size. When this simulation is rerun with 10,000 repetitions, the estimated rejection rate is 0.049 and the confidence interval is [0.044, 0.052].

The simulation results also include the variable `p2f`, which stores the p-values of each test. If the $t(148)$ distribution is the correct distribution for the t test, then `p2f` should be uniformly distributed on (0,1). A histogram, not shown, reveals this to be the case.

More simulations are needed to accurately measure test size (and power) than are needed for bias and standard-error calculations. For a test with estimated size a based on S simulations, a 95% confidence interval for the true size is $a \pm 1.96 \times \sqrt{a(1-a)/S}$. For example, if $a = 0.06$ and $S = 10{,}000$ then the 95% confidence interval is $[0.055, 0.065]$. A more detailed Monte Carlo experiment for test size and power is given in section 12.6.

Number of simulations

Ideally, 10,000 simulations or more would be run in reported results, but this can be computationally expensive. With only 1,000 simulations, there can be considerable simulation noise, especially for estimates of test size (and power).

4.6.3 Variations

The preceding code is easily adapted to other problems of interest.

Different sample size and number of simulations

Sample size can be changed by changing the global macro `numobs`. Many simulation studies focus on finite-sample deviations from asymptotic theory. For some estimators, most notably IV with weak instruments, such deviations can occur even with samples of many thousands of observations.

Changing the global macro `numsims` can increase the number of simulations to yield more-precise simulation results.

Test power

The power of a test is the probability that it rejects a false null hypothesis. To simulate the power of a test, we estimate the rejection rate for a test against a false null hypothesis. The larger the difference between the tested value and the true value, the greater the power and the rejection rate. The example below modifies `chi2data` to estimate the power of a test against the false null hypothesis that $\beta_2 = 2.1$.

```
. * Program for finite-sample properties of OLS: fixed regressors
. program chi2datab, rclass
  1.      version 10.1
  2.      drop _all
  3.      set obs $numobs
  4.      generate double x = rchi2(1)
  5.      generate y = 1 + 2*x + rchi2(1)-1      // demeaned chi^2 error
  6.      regress y x
  7.      return scalar b2  =_b[x]
  8.      return scalar se2 =_se[x]
  9.      test x=2.1
 10.      return scalar r2 = (r(p)<.05)
 11. end
```

Below we use simulate to run the simulation 1,000 times, and then we summarize the results.

```
. * Power simulation for finite-sample properties of OLS
. simulate b2f=r(b2) se2f=r(se2) reject2f=r(r2), reps($numsims)
> saving(chi2databres, replace) nolegend nodots: chi2datab
. mean b2f se2f reject2f
Mean estimation                     Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2f	2.001816	.0026958	1.996526	2.007106
se2f	.0836454	.0005591	.0825483	.0847426
reject2f	.241	.0135315	.2144465	.2675535

The sample mean of reject2f provides an estimate of the power. The estimated power is 0.241, which is not high. Increasing the sample size or the distance between the tested value and the true value will increase the power of the test.

A useful way to incorporate power estimation is to define the hypothesized value of β_2 to be an argument of the program chi2datab. This is demonstrated in the more detailed Monte Carlo experiment in section 12.6.

Different error distributions

We can investigate the effect of using other error distributions by changing the distribution used in chi2data. For linear regression, the t statistic becomes closer to t distributed as the error distribution becomes closer to i.i.d. normal. For nonlinear models, the exact finite-sample distribution of estimators and test statistics is unknown even if the errors are i.i.d. normal.

The example in section 4.6.2 used different draws of both regressors and errors in each simulation. This corresponds to simple random sampling where we jointly sample the pair (y, x), especially relevant to survey data where individuals are sampled, and we use data (y,x) for the sampled individuals. An alternative approach is that of fixed regressors in repeated trials, especially relevant to designed experiments. Then we draw a sample of x only once, and we use the same sample of x in each simulation while redrawing only the error u (and hence y). In that case, we create fixedx.dta, which has 150 observations on a variable, x, that is drawn from the $\chi^2(1)$ distribution, and we replace lines 2–4 of chi2data by typing use fixedx, clear.

4.6.4 Estimator inconsistency

Establishing estimator inconsistency requires less coding because we need to generate data and obtain estimates only once, with a large N, and then compare the estimates with the DGP values.

We do so for a classical errors-in-variables model of measurement error. Not only is it known that the OLS estimator is inconsistent, but in this case, the magnitude of the inconsistency is also known, so we have a benchmark for comparison.

The DGP considered is

$$y = \beta x^* + u; \quad x^* \sim N(0,9); \quad u \sim N(0,1)$$
$$x = x^* + v; \quad v \sim N(0,1)$$

OLS regression of y on x^* consistently estimates β. However, only data on x rather than x^* are available, so we instead obtain $\widehat{\beta}$ from an OLS regression of y on x. It is a well-known result that then $\widehat{\beta}$ is inconsistent, with a downward bias, $s\beta$, where $s = \sigma_v^2/(\sigma_v^2 + \sigma_{x^*}^2)$ is the noise–signal ratio. For the DGP under consideration, this ratio is $1/(1+9) = 0.1$, so plim $\widehat{\beta} = \beta - s\beta = 1 - 0.1 \times 1 = 0.9$.

The following simulation checks this theoretical prediction, with sample size set to 10,000. We use `drawnorm` to jointly draw (x^*, u, v), though we could have more simply made three separate standard normal draws. We set $\beta = 1$.

```
. * Inconsistency of OLS in errors-in-variables model (measurement error)
. clear
. quietly set obs 10000
. set seed 10101
. matrix mu = (0,0,0)
. matrix sigmasq = (9,0,0\0,1,0\0,0,1)
. drawnorm xstar u v, means(mu) cov(sigmasq)
. generate y = 1*xstar + u    // DGP for y depends on xstar
. generate x = xstar + v      // x is mismeasured xstar
. regress y x, noconstant
```

Source	SS	df	MS	Number of obs = 10000
				F(1, 9999) =42724.08
Model	31730.3312	1	81730.3312	Prob > F = 0.0000
Residual	19127.893	9999	1.9129806	R-squared = 0.8103
				Adj R-squared = 0.8103
Total	100858.224	10000	10.0858224	Root MSE = 1.3831

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x	.9001733	.004355	206.70	0.000	.8916366	.90871

The OLS estimate is very precisely estimated, given the large sample size. The estimate of 0.9002 clearly differs from the DGP value of 1.0, so OLS is inconsistent. Furthermore, the simulation estimate essentially equals the theoretical value of 0.9.

4.6.5 Simulation with endogenous regressors

Endogeneity is one of the most frequent causes of estimator inconsistency. A simple method to generate an endogenous regressor is to first generate the error u and then generate the regressor x to be the sum of a multiple of u and an independent component.

We adapt the previous DGP as follows:

$$y = \beta_1 + \beta_2 x + u; \quad u \sim N(0,1);$$
$$x = z + 0.5u; \quad z \sim N(0,1)$$

We set $\beta_1 = 10$ and $\beta_2 = 2$. For this DGP, the correlation between x and u equals 0.5. We let $N = 150$.

The following program generates the data:

```
. * Endogenous regressor
. clear
. set seed 10101
. program endogreg, rclass
  1.    version 10.1
  2.    drop _all
  3.    set obs $numobs
  4.    generate u = rnormal(0)
  5.    generate x = 0.5*u + rnormal(0)          // endogenous regressors
  6.    generate y = 10 + 2*x + u
  7.    regress y x
  8.    return scalar b2 =_b[x]
  9.    return scalar se2 = _se[x]
 10.    return scalar t2 = (_b[x]-2)/_se[x]
 11.    return scalar r2 = abs(return(t2))>invttail($numobs-2,.025)
 12.    return scalar p2 = 2*ttail($numobs-2,abs(return(t2)))
 13. end
```

Below we run the simulations and summarize the results.

```
. simulate b2r=r(b2) se2r=r(se2) t2r=r(t2) reject2r=r(r2) p2r=r(p2),
> reps($numsims) nolegend nodots: endogreg

. mean b2r se2r reject2r
Mean estimation                   Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2r	2.399301	.0020709	2.395237	2.403365
se2r	.0658053	.0001684	.0654747	.0661358
reject2r	1	0	.	.

The results from these 1,000 repetitions indicate that for $N = 150$, the OLS estimator is biased by about 20%, the standard error is about 32 times too small, and we always reject the true null hypothesis that $\beta_2 = 2$.

By setting N large, we could also show that the OLS estimator is inconsistent with a single repetition. As a variation, we could instead estimate by IV, with z an instrument for x, and verify that the IV estimator is consistent.

4.7 Stata resources

The key reference for random-number functions is `help functions`. This covers most of the generators illustrated in this chapter and several other standard ones that have not been used. Note, however, that the `rnbinomial(`k`,`p`)` function for making draws from the negative binomial distribution has a different parameterization from that used in this book. The key Stata commands for simulation are [R] **simulate** and [P] **postfile**. The `simulate` command requires first collecting commands into a program; see [P] **program**.

A standard book that presents algorithms for random-number generation is Press et al. (1992). Cameron and Trivedi (2005) discuss random-number generation and present a Monte Carlo study; see also chapter 12.7.

4.8 Exercises

1. Using the normal generator, generate a random draw from a 50–50 scale mixture of $N(1,1)$ and $N(1,3^2)$ distributions. Repeat the exercise with the $N(1,3^2)$ component replaced by $N(3,1)$. For both cases, display the features of the generated data by using a kernel density plot.
2. Generate 1,000 observations from the $F(5,10)$ distribution. Use `rchi2()` to obtain draws from the $\chi^2(5)$ and the $\chi^2(10)$ distributions. Compare the sample moments with their theoretical counterparts.
3. Make 1,000 draws from the $N(6,2^2)$ distribution by making a transformation of draws from $N(0,1)$ and then making the transformation $Y = \mu + \sigma Z$.
4. Generate 1,000 draws from the $t(6)$ distribution, which has a mean of 0 and a variance of 4. Compare your results with those from exercise 3.
5. Generate a large sample from the $N(\mu = 1, \sigma^2 = 1)$ distribution and estimate σ/μ, the coefficient of variation. Verify that the sample estimate is a consistent estimate.
6. Generate a draw from a multivariate normal distribution, $N(\boldsymbol{\mu}, \boldsymbol{\Sigma} = \mathbf{LL}')$, with $\boldsymbol{\mu}' = [0\ 0\ 0]$ and

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & \sqrt{3} & 0 \\ 0 & \sqrt{3} & \sqrt{6} \end{bmatrix}, \text{ or } \boldsymbol{\Sigma} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 4 & 3 \\ 0 & 3 & 9 \end{bmatrix}$$

   using transformations based on this Cholesky decomposition. Compare your results with those based on using the `drawnorm` command.
7. Let s denote the sample estimate of σ and $\bar{x}$ denote the sample estimate of μ. The coefficient of variation (CV) σ/μ, which is the ratio of the standard deviation to the mean, is a dimensionless measure of dispersion. The asymptotic distribution of the sample CV $s/\bar{x}$ is $N[\sigma/\mu, (N-2)^{-1/2}(\sigma/\mu)^2\ \{0.5 + (\sigma/\mu)^2\}]$; see Miller (1991). For $N = 25$, using either `simulate` or `postfile`, compare the Monte

Carlo and asymptotic variance of the sample CV with the following specification of the DGP: $x \sim N(\mu, \sigma^2)$ with three different values of CV $= 0.1$, 0.33, and 0.67.

8. It is suspected that making draws from the truncated normal using the method given in section 4.4.4 may not work well when sampling from the extreme tails of the normal. Using different truncation points, check this suggestion.

9. Repeat the example of section 4.6.1 (OLS with χ^2 errors), now using the `postfile` command. Use `postfile` to save the estimated slope coefficient, standard error, the t statistic for H_0: $\beta = 2$, and an indicator for whether H_0 is rejected at 0.05 level in a Stata file named `simresults`. The template program is as follows:

```
* Postfile and post example: repeat OLS with chi-squared errors example
clear
set seed 10101
program simbypost
    version 10.1
    tempname simfile
    postfile `simfile' b2 se2 t2 reject2 p2 using simresults, replace
    quietly {
       forvalues i = 1/$numsims {
         drop _all
         set obs $numobs
         generate x = rchi2(1)
         generate y = 1 + 2*x + rchi2(1) - 1   // demeaned chi^2 error
         regress y x
         scalar b2 = _b[x]
         scalar se2 = _se[x]
         scalar t2 = (_b[x]-2)/_se[x]
         scalar reject2 = abs(t2) > invttail($numobs-2,.025)
         scalar p2 = 2*ttail($numobs-2,abs(t2))
         post `simfile' (b2) (se2) (t2) (reject2) (p2)
       }
    }
    postclose `simfile'
end
simbypost
use simresults, clear
summarize
```