Simulation for estimation and testing

Christopher F Baum

ECON 8823: Applied Econometrics

Boston College, Spring 2015

Monte Carlo simulation is a useful and powerful tool for investigating the properties of econometric estimators and tests. The power is derived from being able to define and control the statistical environment in which you fully specify the data generating process (DGP) and use those data in controlled experiments.

Many of the estimators we commonly use only have an asymptotic justification. When using a sample of a particular size, it is important to verify how well estimators and postestimation tests are likely to perform in that environment. Monte Carlo simulation may be used, even when we are confident that the estimation techniques are appropriate, to evaluate their performance: for instance, their empirical rate of convergence when some of the underlying assumptions may not be satisfied.

Monte Carlo simulation is a useful and powerful tool for investigating the properties of econometric estimators and tests. The power is derived from being able to define and control the statistical environment in which you fully specify the data generating process (DGP) and use those data in controlled experiments.

Many of the estimators we commonly use only have an asymptotic justification. When using a sample of a particular size, it is important to verify how well estimators and postestimation tests are likely to perform in that environment. Monte Carlo simulation may be used, even when we are confident that the estimation techniques are appropriate, to evaluate their performance: for instance, their empirical rate of convergence when some of the underlying assumptions may not be satisfied.

In many situations, we must write a computer program to compute an estimator or test. Simulation is a useful tool in that context to check the validity of the code in a controlled setting, and verify that it handles all plausible configurations of data properly. For instance, a routine that handles panel, or longitudinal, data should be validated on both balanced and unbalanced panels if it is valid to apply that procedure in the unbalanced case.

Simulation is perhaps a greatly underutilized tool, given the ease of its use in Stata and similar econometric software languages. When conducting applied econometric studies, it is important to assess the properties of the tools we use, whether they are 'canned' or user-written. Simulation can play an important role in that process.

In many situations, we must write a computer program to compute an estimator or test. Simulation is a useful tool in that context to check the validity of the code in a controlled setting, and verify that it handles all plausible configurations of data properly. For instance, a routine that handles panel, or longitudinal, data should be validated on both balanced and unbalanced panels if it is valid to apply that procedure in the unbalanced case.

Simulation is perhaps a greatly underutilized tool, given the ease of its use in Stata and similar econometric software languages. When conducting applied econometric studies, it is important to assess the properties of the tools we use, whether they are 'canned' or user-written. Simulation can play an important role in that process.

Pseudo-random number generators

A key element in Monte Carlo simulation and bootstrapping is the pseudo-random number (PRN) generator. The term random number generator is an oxymoron, as computers with a finite number of binary bits actually use deterministic devices to produce long chains of numbers that *mimic* the realizations from some target distribution. Eventually, those chains will repeat; we cannot achieve an infinite periodicity for a PRNG.

All PRNGs are based on transformations of draws from the uniform (0,1) distribution. A simple PRNG uses the deterministic rule

$$X_j = (kX_{j-1} + c) \mod m, \; j = 1, \ldots, J$$

where mod is the modulo operator, to produce a sequence of integers between 0 and $(m-1)$. The sequence $R_j = X_j/m$ is then a sequence of J values between 0 and 1.

Pseudo-random number generators

A key element in Monte Carlo simulation and bootstrapping is the pseudo-random number (PRN) generator. The term random number generator is an oxymoron, as computers with a finite number of binary bits actually use deterministic devices to produce long chains of numbers that *mimic* the realizations from some target distribution. Eventually, those chains will repeat; we cannot achieve an infinite periodicity for a PRNG.

All PRNGs are based on transformations of draws from the uniform (0,1) distribution. A simple PRNG uses the deterministic rule

$$X_j = (kX_{j-1} + c) \mod m, \; j = 1, \ldots, J$$

where mod is the modulo operator, to produce a sequence of integers between 0 and $(m-1)$. The sequence $R_j = X_j/m$ is then a sequence of J values between 0 and 1.

Using 32-bit integer arithmetic, as is common, $m = 2^{31} - 1$ and the maximum periodicity is that figure, which is approximately 2.1×10^9. That maximum will only be achieved with optimal choices of k, c and X_0; with poor choices, the sequence will repeat more frequently than that.

These values are not truly random: if you start the PRNG with the same X_0, known as the *seed* of the PRNG, you will receive exactly the same sequence of pseudo-random draws. That is an advantage when validating computer code, as you will want to ensure that the program generates the same deterministic results when presented with a given sequence of pseudo-random draws. In Stata, you may

```
set seed nnnnnnnn
```

before any calls to a PRNG to ensure that the starting point is fixed.

Using 32-bit integer arithmetic, as is common, $m = 2^{31} - 1$ and the maximum periodicity is that figure, which is approximately 2.1×10^9. That maximum will only be achieved with optimal choices of k, c and X_0; with poor choices, the sequence will repeat more frequently than that.

These values are not truly random: if you start the PRNG with the same X_0, known as the *seed* of the PRNG, you will receive exactly the same sequence of pseudo-random draws. That is an advantage when validating computer code, as you will want to ensure that the program generates the same deterministic results when presented with a given sequence of pseudo-random draws. In Stata, you may

```
set seed nnnnnnnn
```

before any calls to a PRNG to ensure that the starting point is fixed.

If you do not specify a seed value, the seed is chosen from the time of day to millisecond precision, so even if you rerun the program at 10:00:00 tomorrow, you will not be using the same seed value. Stata's basic PRNG is `runiform()`, which takes no arguments (but the parentheses must be typed). Its maximum value is $1 - 2^{-32}$.

As mentioned, all other PRNGs are transformations of that produced by the uniform PRNG. To draw uniform values over a different range: e.g., over the interval $[a, b)$,

```
gen double varname = a+(b-a)*runiform()
```

and to draw (pseudo-)random integers over the interval (a, b),

```
gen double varname = a+int((b-a+1)*runiform())
```

If you do not specify a seed value, the seed is chosen from the time of day to millisecond precision, so even if you rerun the program at 10:00:00 tomorrow, you will not be using the same seed value. Stata's basic PRNG is `runiform()`, which takes no arguments (but the parentheses must be typed). Its maximum value is $1 - 2^{-32}$.

As mentioned, all other PRNGs are transformations of that produced by the uniform PRNG. To draw uniform values over a different range: e.g., over the interval $[a, b)$,

```
gen double varname = a+(b-a)*runiform()
```

and to draw (pseudo-)random integers over the interval (a, b),

```
gen double varname = a+int((b-a+1)*runiform())
```

If we draw using the `runiform()` PRNG, we see that its theoretical values of $\mu = 0.5$, $\sigma = \sqrt{1/12} = 0.28867513$ appear as we increase sample size:

```
. qui set obs 1000000
. set seed 10101
. g double x1k = runiform() in 1/1000
(999000 missing values generated)
. g double x10k = runiform() in 1/10000
(990000 missing values generated)
. g double x100k = runiform() in 1/100000
(900000 missing values generated)
. g double x1m = runiform()
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
x1k	1000	.5150332	.2934123	.0002845	.9993234
x10k	10000	.4969343	.288723	.000112	.999916
x100k	100000	.4993971	.2887694	7.72e-06	.999995
x1m	1000000	.4997815	.2887623	4.85e-07	.9999998

The sequence is deterministic: that is, if we rerun this do-file, we will get exactly the same draws every time, as we have set the seed of the PRNG. However, the draws should be serially uncorrelated. If that condition is satisfied, then the autocorrelations of this series should be negligible:

```
. g t = _n
. tsset t
        time variable:  t, 1 to 1000000
                delta:  1 unit
. pwcorr L(0/5).x1m, star(0.05)
```

	x1m	L.x1m	L2.x1m	L3.x1m	L4.x1m	L5.x1m
x1m	1.0000					
L.x1m	-0.0011	1.0000				
L2.x1m	-0.0003	-0.0011	1.0000			
L3.x1m	0.0009	-0.0003	-0.0011	1.0000		
L4.x1m	0.0009	0.0009	-0.0003	-0.0011	1.0000	
L5.x1m	0.0007	0.0009	0.0009	-0.0003	-0.0011	1.0000

```
. wntestq x1m
Portmanteau test for white noise
---------------------------------------------------------------------------
 Portmanteau (Q) statistic =    39.7976
 Prob > chi2(40)           =     0.4793
```

Both `pwcorr`, which computes significance levels for pairwise correlations, and the Ljung–Box–Pierce *Q* test, or portmanteau test, fail to detect any departure from serial independence in the uniform draws produced by the `runiform()` PRNG.

Draws from the normal distribution

To consider a more useful task, we may want to draw from the normal distribution, By default, the `rnormal()` function produces draws from the standard normal, with $\mu = 0, \sigma = 1$. If we want to draw from $N(m, s^2)$,

```
gen double varname = rnormal(m, s)
```

The function can also be used with a single argument, the desired mean, with the standard deviation set to 1.

Draws from other continuous distributions

Similar functions exist in Stata for Student's t with n d.f. and $\chi^2(m)$ with m d.f.: the functions `rt(n)` and `rchi2(m)`, respectively. There is no explicit function for the $F(h, n)$ for the F distribution with h and n d.f., so this can be done as the ratios of draws from the $\chi^2(h)$ and $\chi^2(n)$ distributions:

```
. set obs 100000
obs was 0, now 100000
. set seed 10101
. gen double xt = rt(10)
. gen double xc3 = rchi2(3)
. gen double xc97 = rchi2(97)
. gen double xf = ( xc3 / 3 ) / (xc97 / 97 )  // produces F[3, 97]
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xt	100000	.0064869	1.120794	-7.577694	8.765106
xc3	100000	3.002999	2.443407	.0001324	25.75221
xc97	100000	97.03116	13.93907	45.64333	171.9501
xf	100000	1.022082	.8542133	.0000343	8.679594

In this example, the *t*-distributed RV should have mean zero; the $\chi^2(3)$ RV should have mean 3.0; the $\chi^2(97)$ RV should have mean 97.0; and the $F(3, 97)$ should have mean 97/(97-2) = 1.021. We could compare their higher moments with those of the theoretical distributions as well.

We may also draw from the two-parameter Beta(a,b) distribution, which for $a, b > 0$ yields $\mu = a/(a+b)$, $\sigma^2 = ab/((a+b)^2(a+b+1))$, using `rbeta(a,b)`. Likewise, we can draw from a two-parameter Gamma(a,b) distribution, which for $a, b > 0$ yields $\mu = ab$ and $\sigma^2 = ab^2$. Many other continuous distributions can be expressed in terms of the Beta and Gamma distributions; note that the latter is often called the generalized factorial function.

In this example, the *t*-distributed RV should have mean zero; the $\chi^2(3)$ RV should have mean 3.0; the $\chi^2(97)$ RV should have mean 97.0; and the $F(3, 97)$ should have mean 97/(97-2) = 1.021. We could compare their higher moments with those of the theoretical distributions as well.

We may also draw from the two-parameter Beta(a,b) distribution, which for $a, b > 0$ yields $\mu = a/(a+b)$, $\sigma^2 = ab/((a+b)^2(a+b+1))$, using `rbeta(a,b)`. Likewise, we can draw from a two-parameter Gamma(a,b) distribution, which for $a, b > 0$ yields $\mu = ab$ and $\sigma^2 = ab^2$. Many other continuous distributions can be expressed in terms of the Beta and Gamma distributions; note that the latter is often called the generalized factorial function.

Draws from discrete distributions

You may also produce pseudo-random draws from several discrete probability distributions. For the binomial distribution $Bin(n, p)$, with n trials and success probability p, use `binomial(n,p)`. For the Poisson distribution with $\mu = \sigma^2 = m$, use `poisson(m)`.

```
. set obs 100000
obs was 0, now 100000
. set seed 10101
. gen double xbin = rbinomial(100, 0.8)
. gen double xpois = rpoisson(5)
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xbin	100000	79.98817	3.991282	61	94
xpois	100000	4.99788	2.241603	0	16

```
. di r(Var)  // variance of the last variable summarized
5.0247858
```

The means of these two variables are close to their theoretical values, as is the variance of the Poisson-distributed variable.

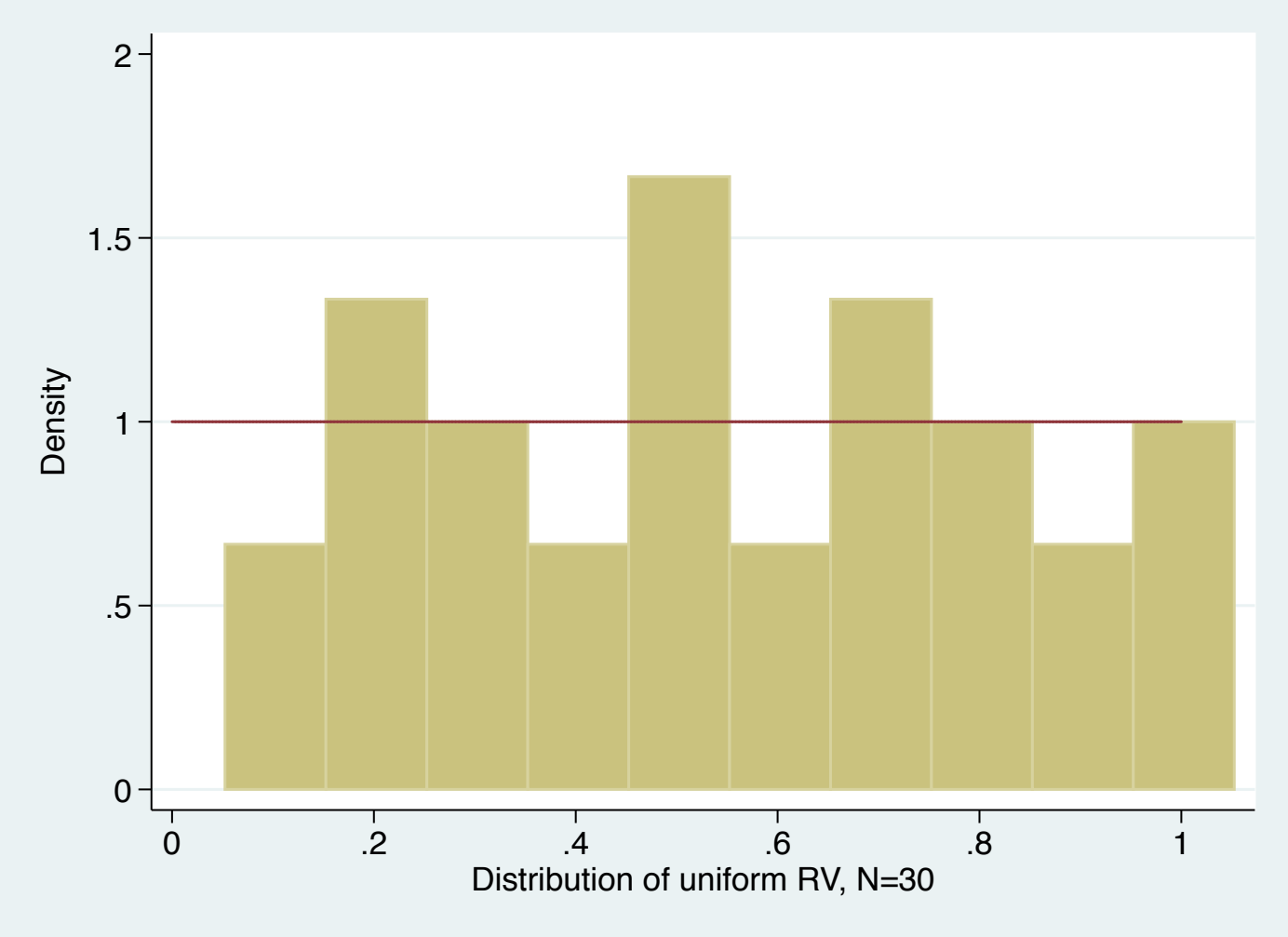
A first illustration

As a first illustration of Monte Carlo simulation in Stata, we demonstrate the central limit theorem result that in the limit, a standardized sample mean, $(\bar{x}_N - \mu)/(\sigma/\sqrt{N})$, has a standard normal distribution, $N(0,1)$, so that the sample mean is approximately normally distributed as $N \rightarrow \infty$. We first consider a single sample of size 30 drawn from the uniform distribution.

```
. set obs 30
obs was 0, now 30
. set seed 10101
. gen double x = runiform()
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
x	30	.5459987	.2803788	.0524637	.9983786

We see that the mean of this sample, 0.546, is quite far from the theoretical value of 0.5, and the resulting values do not look very uniformly distributed when viewed as a histogram. For large samples, the histogram should approach a horizontal line at density = 1.



To illustrate the features of the distribution of sample mean for a fixed sample size of 30, we conduct a Monte Carlo experiment using Stata's `simulate` prefix. As with other prefix commands in Stata such as `by`, `statsby`, or `rolling`, the `simulate` prefix can execute a single Stata command repeatedly.

Using Monte Carlo, we usually must write the ad hoc Stata command, or `program`, that produces the desired result. That program will be called repeatedly by `simulate`, which will produce a new dataset of simulated results: in this case, the sample mean from each sample of size 30.

To illustrate the features of the distribution of sample mean for a fixed sample size of 30, we conduct a Monte Carlo experiment using Stata's `simulate` prefix. As with other prefix commands in Stata such as `by`, `statsby`, or `rolling`, the `simulate` prefix can execute a single Stata command repeatedly.

Using Monte Carlo, we usually must write the ad hoc Stata command, or `program`, that produces the desired result. That program will be called repeatedly by `simulate`, which will produce a new dataset of simulated results: in this case, the sample mean from each sample of size 30.

In Stata terms, what we must write is an *ado-file*: a file containing a Stata `program` of the same name that adds a new verb to the Stata language. In the case of `simulate`, this is quite straightforward, as the program's structure is formulaic, focusing on the results to be produced and returned in the stored results.

The same methodology and programming constructs will be relevant if you are using Stata's maximum likelihood commands, `ml`, for which you must write a program containing the (log-)likelihood function.

Serious uses of the generalized method of moments command, `gmm`, require you to write a program containing the moment conditions, or orthogonality conditions. The same techniques may be used for Stata's nonlinear least squares commands (`nl` and `nlsur`).

In Stata terms, what we must write is an *ado-file*: a file containing a Stata `program` of the same name that adds a new verb to the Stata language. In the case of `simulate`, this is quite straightforward, as the program's structure is formulaic, focusing on the results to be produced and returned in the stored results.

The same methodology and programming constructs will be relevant if you are using Stata's maximum likelihood commands, `ml`, for which you must write a program containing the (log-)likelihood function.

Serious uses of the generalized method of moments command, `gmm`, require you to write a program containing the moment conditions, or orthogonality conditions. The same techniques may be used for Stata's nonlinear least squares commands (`nl` and `nlsur`).

In Stata terms, what we must write is an *ado-file*: a file containing a Stata `program` of the same name that adds a new verb to the Stata language. In the case of `simulate`, this is quite straightforward, as the program's structure is formulaic, focusing on the results to be produced and returned in the stored results.

The same methodology and programming constructs will be relevant if you are using Stata's maximum likelihood commands, `ml`, for which you must write a program containing the (log-)likelihood function.

Serious uses of the generalized method of moments command, `gmm`, require you to write a program containing the moment conditions, or orthogonality conditions. The same techniques may be used for Stata's nonlinear least squares commands (`nl` and `nlsur`).

The `simulate` command has the syntax

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

Per the usual notation for Stata syntax, the [bracketed] items are optional, and those in *italics* are to be filled in. All options for `simulate`, including the 'required option' `reps()`, appear before the colon `(:)`, while any options for *command* appear after a comma in the *command*. The quantities to be calculated and stored by your *command* are specified in *exp_list*.

We will employ the `saving()` option of `simulate`, which will create a new Stata dataset from the results produced in the *exp_list*. If successful, it will have *n* observations, one for each of the replications.

The `simulate` command has the syntax

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

Per the usual notation for Stata syntax, the [bracketed] items are optional, and those in *italics* are to be filled in. All options for `simulate`, including the 'required option' `reps()`, appear before the colon `(:)`, while any options for *command* appear after a comma in the *command*. The quantities to be calculated and stored by your *command* are specified in *exp_list*.

We will employ the `saving()` option of `simulate`, which will create a new Stata dataset from the results produced in the *exp_list*. If successful, it will have *n* observations, one for each of the replications.

We illustrate a program which may be called by `simulate`:

```
. prog drop _all
. prog onesample, rclass
  1.      version 12
  2.      drop _all
  3.      qui set obs 30
  4.      g double x = runiform()
  5.      su x, meanonly
  6.      ret sca mu = r(mean)
  7. end
```

The program is named `onesample` and declared `rclass`, which is necessary for the program to return stored results as `r()`. We have hard-coded the sample size of 30 observations, specifying that the program should create a uniform RV, compute its mean, and return it as a numeric scalar to `simulate` as `r(mu)`.

For future use, the program should be saved in `onesample.ado` on the `adopath`, preferably in your `PERSONAL` directory. Use `adopath` to locate that directory.

We illustrate a program which may be called by `simulate`:

```
. prog drop _all
. prog onesample, rclass
  1.      version 12
  2.      drop _all
  3.      qui set obs 30
  4.      g double x = runiform()
  5.      su x, meanonly
  6.      ret sca mu = r(mean)
  7. end
```

The program is named `onesample` and declared `rclass`, which is necessary for the program to return stored results as `r()`. We have hard-coded the sample size of 30 observations, specifying that the program should create a uniform RV, compute its mean, and return it as a numeric scalar to `simulate` as `r(mu)`.

For future use, the program should be saved in `onesample.ado` on the `adopath`, preferably in your `PERSONAL` directory. Use `adopath` to locate that directory.

We illustrate a program which may be called by `simulate`:

```
. prog drop _all
. prog onesample, rclass
  1.      version 12
  2.      drop _all
  3.      qui set obs 30
  4.      g double x = runiform()
  5.      su x, meanonly
  6.      ret sca mu = r(mean)
  7. end
```

The program is named `onesample` and declared `rclass`, which is necessary for the program to return stored results as `r()`. We have hard-coded the sample size of 30 observations, specifying that the program should create a uniform RV, compute its mean, and return it as a numeric scalar to `simulate` as `r(mu)`.

For future use, the program should be saved in `onesample.ado` on the `adopath`, preferably in your `PERSONAL` directory. Use `adopath` to locate that directory.

When you write a simulation program, you should always run it once as a check that it performs as it should, and returns the item or items that are meant to be used by `simulate`:

```
 . set seed 10101
. onesample
. return list
scalars:
                  r(mu) =   .5459987206074098
```

Note that the mean of the series that appears in the `return list` is the same as that which we computed earlier from the same seed.

Executing the simulation

We are now ready to invoke `simulate:` to produce the Monte Carlo results:

```
 . loc srep 10000
. simulate xbar = r(mu), seed(10101) reps(`srep´) nodots ///
> saving(muclt, replace) : onesample
      command:  onesample
         xbar:  r(mu)
```

We expect that the variable `xbar` in the dataset we have created, `muclt.dta`, will have a mean of 0.5 and a standard deviation of $\sqrt{(1/12)/30} = 0.0527$.

Executing the simulation

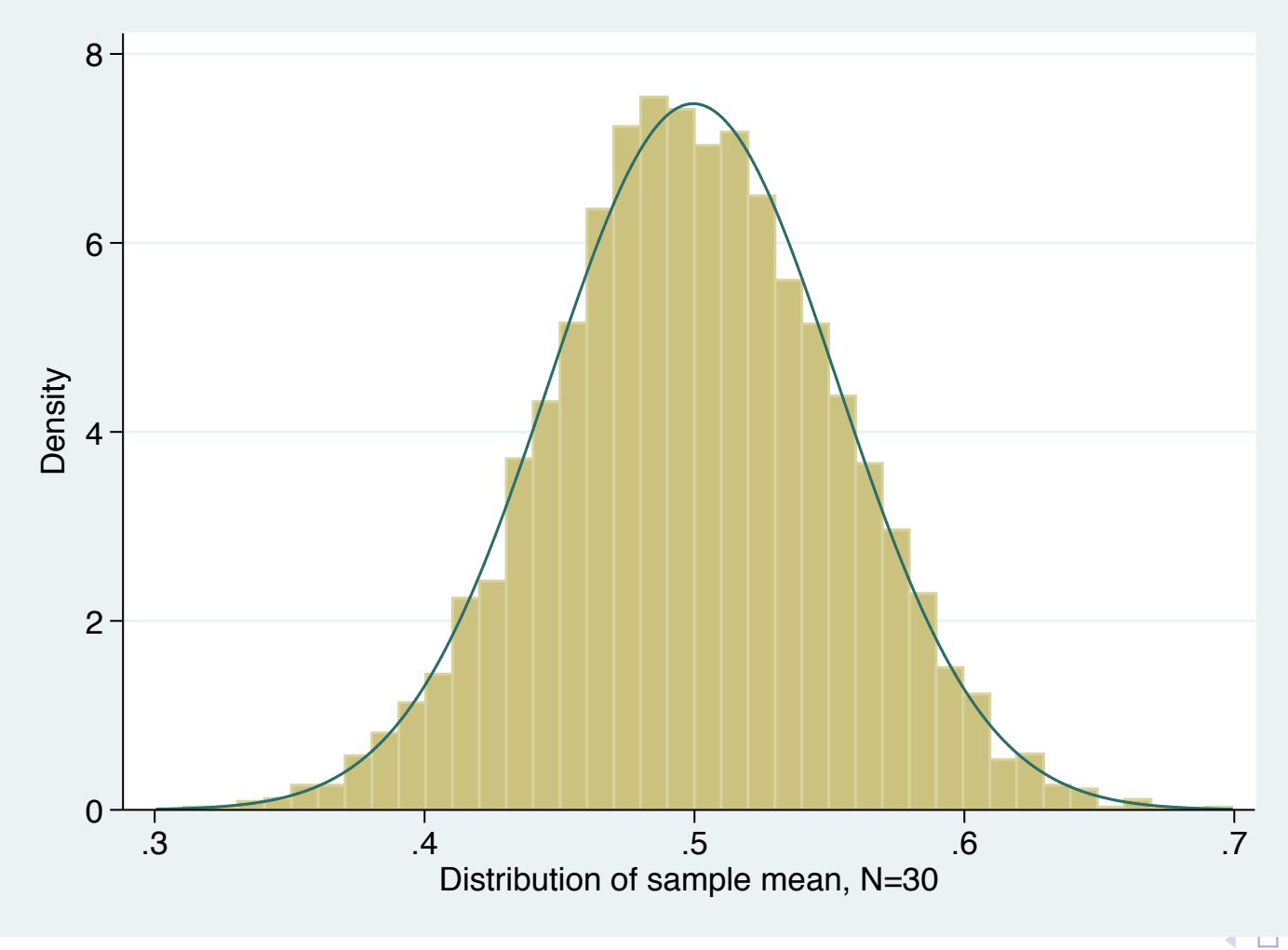
We are now ready to invoke `simulate:` to produce the Monte Carlo results:

```
 . loc srep 10000
. simulate xbar = r(mu), seed(10101) reps(`srep´) nodots ///
> saving(muclt, replace) : onesample
      command:  onesample
         xbar:  r(mu)
```

We expect that the variable `xbar` in the dataset we have created, `muclt.dta`, will have a mean of 0.5 and a standard deviation of $\sqrt{(1/12)/30} = 0.0527$.

```
. use muclt, clear
(simulate: onesample)
. su
```

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



Although the mean and standard deviation of the simulated distribution are not exactly in line with the theoretical values, they are quite close, and the empirical distribution of the 10,000 sample means is quite close to that of the overlaid normal distribution.

We might want to make our program more general by allowing for other sample sizes:

```
. prog drop _all
. prog onesamplen, rclass
  1.      version 12
  2.      syntax [, N(int 30)]
  3.      drop _all
  4.      qui set obs `n´
  5.      g double x = runiform()
  6.      su x, meanonly
  7.      ret sca mu = r(mean)
  8. end
```

We have added an n() option that allows `onesamplen` to use a different sample size if specified, with a default of 30.

Although the mean and standard deviation of the simulated distribution are not exactly in line with the theoretical values, they are quite close, and the empirical distribution of the 10,000 sample means is quite close to that of the overlaid normal distribution.

We might want to make our program more general by allowing for other sample sizes:

```
. prog drop _all
. prog onesamplen, rclass
  1.      version 12
  2.      syntax [, N(int 30)]
  3.      drop _all
  4.      qui set obs `n´
  5.      g double x = runiform()
  6.      su x, meanonly
  7.      ret sca mu = r(mean)
  8. end
```

We have added an n() option that allows `onesamplen` to use a different sample size if specified, with a default of 30.

Again, we should check to see that the program works properly with this new feature, and produces the same result as we could manually:

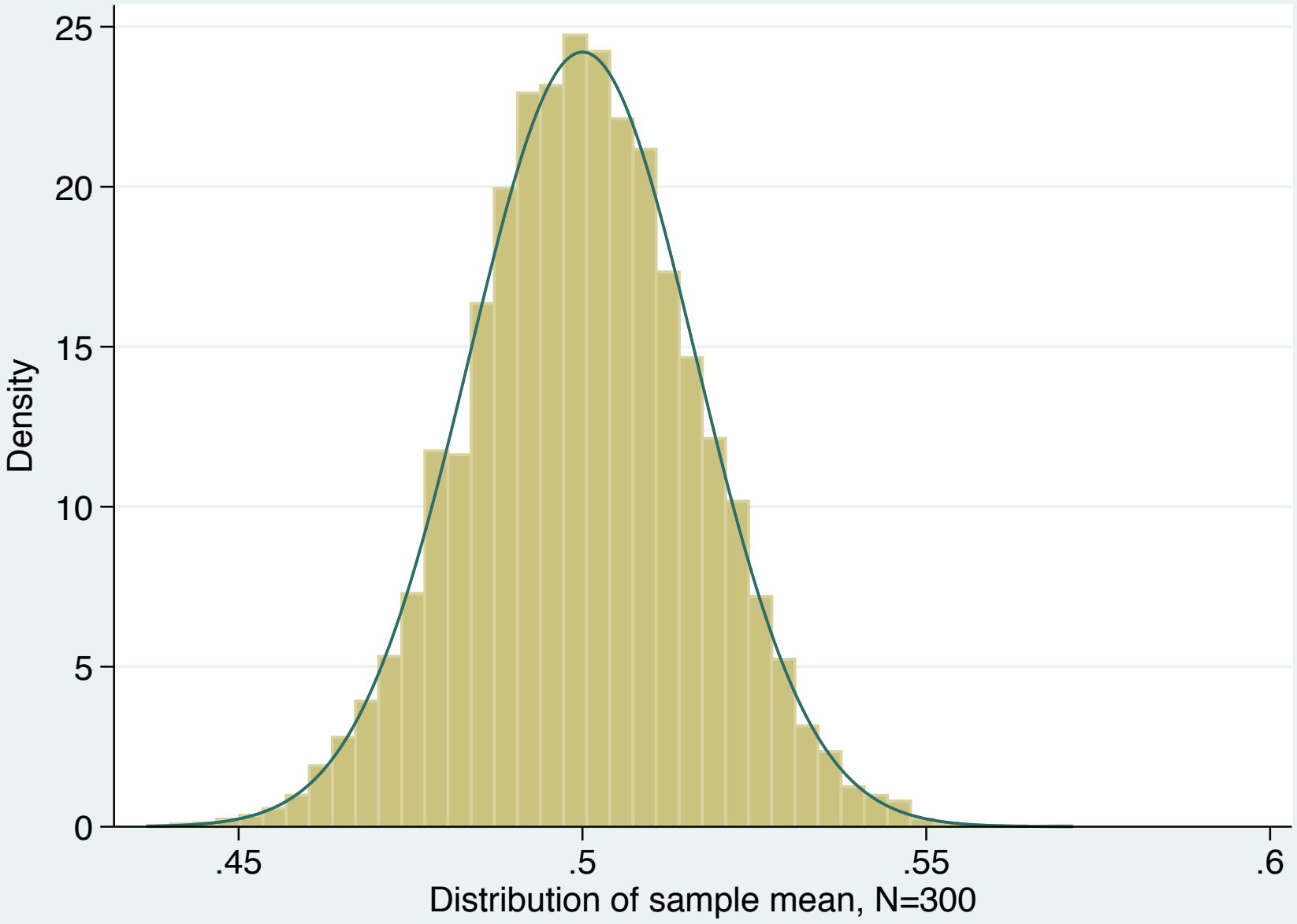
```
. set seed 10101
. set obs 300
obs was 0, now 300
. gen double x = runiform()
. su x
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
           x |        300    .5270966    .2819105    .0010465   .9983786
. set seed 10101
. onesamplen, n(300)
. return list
scalars:
                  r(mu) =  .527096571639025
```

We can now execute our new version of the program with a different sample size. Notice that the option is that of `onesamplen`, not that of `simulate`. We expect that the variable `xbar` in the dataset we have created, `muclt300.dta`, will have a mean of 0.5 and a standard deviation of $\sqrt{(1/12)/300} = .01667$.

```
 . loc srep 10000
. loc sampn  300
. simulate xbar = r(mu), seed(10101) reps(`srep´) nodots ///
> saving(muclt300, replace) : onesamplen, n(`sampn´)
      command:  onesamplen, n(300)
         xbar:  r(mu)
(note: file muclt300.dta not found)
. use muclt300, clear
(simulate: onesamplen)
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xbar	10000	.5000151	.0164797	.4367322	.5712539

The results are quite close to the theoretical values.


25
20
15
10
5
0
Density
.45
.5
.55
.6
Distribution of sample mean, N=300

More details on PRNGs

Bill Gould's entries in the Stata blog, Not Elsewhere Classified, discuss several ways in which the `runiform()` PRNG can be useful:

- shuffling observations in random order: generate a uniform RV and `sort` on that variable
- drawing a subsample of *n* observations without replacement: generate a uniform RV, sort on that variable, and `keep in 1/n`; see `help sample`
- drawing a *p*% random sample without replacement: `keep if runiform() <= P/100`; see `help sample`
- drawing a subsample of *n* observations with replacement, as needed in bootstrap methods; see `help sample`

More details on PRNGs

Bill Gould's entries in the Stata blog, Not Elsewhere Classified, discuss several ways in which the `runiform()` PRNG can be useful:

- shuffling observations in random order: generate a uniform RV and `sort` on that variable
- drawing a subsample of *n* observations without replacement: generate a uniform RV, sort on that variable, and `keep in 1/n`; see `help sample`
- drawing a *p*% random sample without replacement: `keep if runiform() <= P/100`; see `help sample`
- drawing a subsample of *n* observations with replacement, as needed in bootstrap methods; see `help sample`

More details on PRNGs

Bill Gould's entries in the Stata blog, Not Elsewhere Classified, discuss several ways in which the `runiform()` PRNG can be useful:

- shuffling observations in random order: generate a uniform RV and `sort` on that variable
- drawing a subsample of *n* observations without replacement: generate a uniform RV, sort on that variable, and `keep in 1/n`; see `help sample`
- drawing a *p*% random sample without replacement: `keep if runiform() <= P/100`; see `help sample`
- drawing a subsample of *n* observations with replacement, as needed in bootstrap methods; see `help sample`

More details on PRNGs

Bill Gould's entries in the Stata blog, Not Elsewhere Classified, discuss several ways in which the `runiform()` PRNG can be useful:

- shuffling observations in random order: generate a uniform RV and `sort` on that variable
- drawing a subsample of *n* observations without replacement: generate a uniform RV, sort on that variable, and `keep in 1/n`; see `help sample`
- drawing a *p*% random sample without replacement: `keep if runiform() <= P/100`; see `help sample`
- drawing a subsample of *n* observations with replacement, as needed in bootstrap methods; see `help sample`

Inverse-probability transformations

Let $F(x) = \Pr(X \leq x)$ denote the cdf of RV x. Given a random draw of a uniformly distributed RV $r, 0 \leq r \leq 1$, the inverse transformation $x = F^{-1}(r)$ provides a unique value of x, which will be a good approximation of a random draw from $F(x)$.

This *inverse-probability transformation* method allows us to generate pseudo-RVs for any distribution for which we can provide the inverse CDF. Although the normal distribution lacks a closed form, there are good numerical approximations to its inverse CDF. That allows a method such as

```
gen double xn = invnormal(runiform())
```

and until recently, that was the way in which one produced pseudo-random normal variates in Stata.

Inverse-probability transformations

Let $F(x) = \Pr(X \leq x)$ denote the cdf of RV x. Given a random draw of a uniformly distributed RV $r, 0 \leq r \leq 1$, the inverse transformation $x = F^{-1}(r)$ provides a unique value of x, which will be a good approximation of a random draw from $F(x)$.

This *inverse-probability transformation* method allows us to generate pseudo-RVs for any distribution for which we can provide the inverse CDF. Although the normal distribution lacks a closed form, there are good numerical approximations to its inverse CDF. That allows a method such as

```
gen double xn = invnormal(runiform())
```

and until recently, that was the way in which one produced pseudo-random normal variates in Stata.

We might want to draw from the unit exponential distribution, $F(x) = 1 - e^{-x}$, which has analytical inverse $x = -\log(1 - r)$. So the method yields

```
gen double xexp = -log(1-runiform())
```

One can also apply this method to a discrete CDF, with the convention that the left limit of a flat segment is taken as the x value.

We might want to draw from the unit exponential distribution, $F(x) = 1 - e^{-x}$, which has analytical inverse $x = -\log(1 - r)$.
So the method yields

```
gen double xexp = -log(1-runiform())
```

One can also apply this method to a discrete CDF, with the convention that the left limit of a flat segment is taken as the x value.

Direct transformations

When we want draws from $Y = g(X)$, then the direct transformation method involves drawing from the distribution of X and applying the transformation $g(\cdot)$. This in fact is the method used in common PRNG functions:

- a $\chi^2(1)$ draw is the square of a draw from $N(0,1)$
- a $\chi^2(m)$ is the sum of m independent draws from $\chi^2(1)$
- a $F(m_1, m_2)$ draw is $(v_1/m_1)/(v_2/m_2)$, where v_1, v_2 are independent draws from $\chi^2(m_1), \chi^2(m_2)$
- a $t(m)$ draw is $u = \sqrt{v/m}$, where u, v are independent draws from $N(0,1), \chi^2(m)$

Direct transformations

When we want draws from $Y = g(X)$, then the direct transformation method involves drawing from the distribution of X and applying the transformation $g(\cdot)$. This in fact is the method used in common PRNG functions:

- a $\chi^2(1)$ draw is the square of a draw from $N(0,1)$
- a $\chi^2(m)$ is the sum of m independent draws from $\chi^2(1)$
- a $F(m_1, m_2)$ draw is $(v_1/m_1)/(v_2/m_2)$, where v_1, v_2 are independent draws from $\chi^2(m_1), \chi^2(m_2)$
- a $t(m)$ draw is $u = \sqrt{v/m}$, where u, v are independent draws from $N(0,1), \chi^2(m)$

Direct transformations

When we want draws from $Y = g(X)$, then the direct transformation method involves drawing from the distribution of X and applying the transformation $g(\cdot)$. This in fact is the method used in common PRNG functions:

- a $\chi^2(1)$ draw is the square of a draw from $N(0,1)$
- a $\chi^2(m)$ is the sum of m independent draws from $\chi^2(1)$
- a $F(m_1, m_2)$ draw is $(v_1/m_1)/(v_2/m_2)$, where v_1, v_2 are independent draws from $\chi^2(m_1), \chi^2(m_2)$
- a $t(m)$ draw is $u = \sqrt{v/m}$, where u, v are independent draws from $N(0,1), \chi^2(m)$

Direct transformations

When we want draws from $Y = g(X)$, then the direct transformation method involves drawing from the distribution of X and applying the transformation $g(\cdot)$. This in fact is the method used in common PRNG functions:

- a $\chi^2(1)$ draw is the square of a draw from $N(0,1)$
- a $\chi^2(m)$ is the sum of m independent draws from $\chi^2(1)$
- a $F(m_1, m_2)$ draw is $(v_1/m_1)/(v_2/m_2)$, where v_1, v_2 are independent draws from $\chi^2(m_1), \chi^2(m_2)$
- a $t(m)$ draw is $u = \sqrt{v/m}$, where u, v are independent draws from $N(0,1), \chi^2(m)$

Mixtures of distributions

A widely used discrete distribution is the negative binomial, which can be written as a Poisson–Gamma mixture. If $y/\lambda \sim$Poisson(λ) and $\lambda/\mu, \alpha \sim \Gamma(\mu, \alpha\mu)$, then $y/\mu, \alpha \sim NB2(\mu, \mu + \alpha\mu^2)$. The NB2 can be seen as a generalization of the Poisson, which would impose the constraint that $\alpha = 0$.[1]

Draws from the NB2(1,1) distribution can be achieved by a two-step method: first draw ν from $\Gamma(1, 1)$, then draw from Poisson(ν).
To draw from NB2(μ,1), first draw ν from $\Gamma(\mu, 1)$.

[1] An alternative parameterization of the variance is known as the NB1 distribution.

Mixtures of distributions

A widely used discrete distribution is the negative binomial, which can be written as a Poisson–Gamma mixture. If $y/\lambda \sim$Poisson(λ) and $\lambda/\mu, \alpha \sim \Gamma(\mu, \alpha\mu)$, then $y/\mu, \alpha \sim NB2(\mu, \mu + \alpha\mu^2)$. The NB2 can be seen as a generalization of the Poisson, which would impose the constraint that $\alpha = 0$.[1]

Draws from the NB2(1,1) distribution can be achieved by a two-step method: first draw ν from $\Gamma(1, 1)$, then draw from Poisson(ν).
To draw from NB2(μ,1), first draw ν from $\Gamma(\mu, 1)$.

[1] An alternative parameterization of the variance is known as the NB1 distribution.

Draws from the truncated normal

In censoring or truncation models, we often encounter the truncated normal distribution. With truncation, realizations of X are constrained to lie in (a, b), one of which could be $\pm\infty$. Given $X \sim TN_{a,b}(\mu, \sigma^2)$, the μ, σ^2 parameters describe the untruncated distribution of X.

Given draws from a uniform distribution u, define $a^* = (a - \mu)/\sigma$, $b^* = (b - \mu)/\sigma$:

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

where $\Phi(\cdot)$ is the CDF of the normal distribution.

Draws from the truncated normal

In censoring or truncation models, we often encounter the truncated normal distribution. With truncation, realizations of X are constrained to lie in (a, b), one of which could be $\pm\infty$. Given $X \sim TN_{a,b}(\mu, \sigma^2)$, the μ, σ^2 parameters describe the untruncated distribution of X.

Given draws from a uniform distribution u,
define $a^* = (a - \mu)/\sigma$, $b^* = (b - \mu)/\sigma$:

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

where $\Phi(\cdot)$ is the CDF of the normal distribution.

```
 . qui set obs 10000
. set seed 10101
. sca a = 0
. sca b = 12   // draws from N(5, 4^2) truncated [0,12]
. sca mu = 5
. sca sigma = 4
. sca astar = (a - mu) / sigma
. sca bstar = (b - mu) / sigma
. g double u = runiform()
. g double w = normal(astar) + (normal(bstar) - normal(astar)) * u
. g double xtrunc = mu + sigma * invnormal(w)
. su xtrunc
```

Variable	Obs	Mean	Std. Dev.	Min	Max
xtrunc	10000	5.436194	2.951024	.0022294	11.99557

Note that `normal()` is the normal CDF, with `invnormal()` its inverse. This double truncation will increase the mean, as *a* is closer to μ than is *b*. With the truncated normal, the variance always declines: in this case $\sigma = 2.95$ rather than 4.0.

Draws from the multivariate normal

Draws from the multivariate normal are simpler to implement than draws from many multivariate distributions because linear combinations of normal RVs are also normal.

Direct draws can be made using the `drawnorm` command, specifying mean vector μ and covariance matrix Σ. For instance, to draw two RVs with means of (10,20), variances (4,9) and covariance = 3 (correlation 0.5):

Draws from the multivariate normal

Draws from the multivariate normal are simpler to implement than draws from many multivariate distributions because linear combinations of normal RVs are also normal.

Direct draws can be made using the `drawnorm` command, specifying mean vector μ and covariance matrix Σ. For instance, to draw two RVs with means of (10,20), variances (4,9) and covariance = 3 (correlation 0.5):

```
 . qui set obs 10000
. set seed 10101
. mat mu = (10,20)
. sca cov = 0.5 * sqrt(4 * 9)
. mat sigma = (4, cov \ cov, 9)
. drawnorm double y1 y2, means(mu) cov(sigma)
. su y1 y2
```

Variable	Obs	Mean	Std. Dev.	Min	Max
y1	10000	9.986668	1.9897	2.831865	18.81768
y2	10000	19.96413	2.992709	8.899979	30.68013

```
. corr y1 y2
(obs=10000)
```

	y1	y2
y1	1.0000	
y2	0.4979	1.0000

Simulation applied to regression

In using Monte Carlo simulation methods in a regression context, we usually compute parameters, their VCE or summary statistics for each of S generated datasets, and evaluate their empirical distribution.

As an example, we evaluate the finite-sample properties of the OLS estimator with random regressors and a skewed error distribution. If the errors are $i.i.d.$, then this skewness will have no effect on the asymptotic properties of OLS. In comparison to non-skewed error distributions, we will need a larger sample size for the asymptotic results to hold.

Simulation applied to regression

In using Monte Carlo simulation methods in a regression context, we usually compute parameters, their VCE or summary statistics for each of S generated datasets, and evaluate their empirical distribution.

As an example, we evaluate the finite-sample properties of the OLS estimator with random regressors and a skewed error distribution. If the errors are $i.i.d.$, then this skewness will have no effect on the asymptotic properties of OLS. In comparison to non-skewed error distributions, we will need a larger sample size for the asymptotic results to hold.

We consider the DGP

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

where $\beta_1 = 1$, $\beta_2 = 2$, $N = 150$. The error is independent of x, ensuring consistency of OLS, with a mean of zero, variance of 2, skewness of $\sqrt{8}$ and kurtosis of 15, compared to the normal error with a skewness of 0 and kurtosis of 3.

For each simulation, we obtain parameter estimates, standard errors, t-values for the test that $\beta_2 = 2$ and the outcome of a two-tailed test of that hypothesis at the 0.05 level.

We store the sample size in a global macro, as we may want to change it without revising the program.

We consider the DGP

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

where $\beta_1 = 1$, $\beta_2 = 2$, $N = 150$. The error is independent of x, ensuring consistency of OLS, with a mean of zero, variance of 2, skewness of $\sqrt{8}$ and kurtosis of 15, compared to the normal error with a skewness of 0 and kurtosis of 3.

For each simulation, we obtain parameter estimates, standard errors, t-values for the test that $\beta_2 = 2$ and the outcome of a two-tailed test of that hypothesis at the 0.05 level.

We store the sample size in a global macro, as we may want to change it without revising the program.

We consider the DGP

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

where $\beta_1 = 1$, $\beta_2 = 2$, $N = 150$. The error is independent of x, ensuring consistency of OLS, with a mean of zero, variance of 2, skewness of $\sqrt{8}$ and kurtosis of 15, compared to the normal error with a skewness of 0 and kurtosis of 3.

For each simulation, we obtain parameter estimates, standard errors, t-values for the test that $\beta_2 = 2$ and the outcome of a two-tailed test of that hypothesis at the 0.05 level.

We store the sample size in a global macro, as we may want to change it without revising the program.

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

The regression returns its coefficients and standard errors to our program in the _b[ ] and _se[ ] vectors. Those quantity are used to produce the *t* statistic, its *p*-value, and a scalar `r2`: a binary rejection indicator which will equal 1 if the computed *t*-statistic exceeds the tabulated value for the appropriate sample size.

We test the program by executing it once and verifying that the stored results correspond to those which we compute manually:

```
. set seed 10101
. glo numobs = 150
. chi2data
obs was 0, now 150
```

Source	SS	df	MS		Number of obs =	150
					F(1, 148) =	776.52
Model	1825.65455	1	1825.65455		Prob > F =	0.0000
Residual	347.959801	148	2.35107974		R-squared =	0.8399
					Adj R-squared =	0.8388
Total	2173.61435	149	14.5880158		Root MSE =	1.5333

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x	2.158967	.0774766	27.87	0.000	2.005864	2.31207
_cons	.9983884	.1569901	6.36	0.000	.6881568	1.30862

```
 . set seed 10101
. qui chi2data
. ret li
scalars:
                  r(r2) =  1
                  r(p2) =  .0419507116911909
                  r(t2) =  2.05180994793611
                 r(se2) =  .0774765768836093
                  r(b2) =  2.158967211181826
. di r(t2)^2
4.2099241
. test x = 2
 ( 1)  x = 2
       F(  1,   148) =    4.21
            Prob > F =    0.0420
```

As the results are appropriate, we can now proceed to produce the simulation.

```
. set seed 10101
. glo numsim = 1000
. simulate b2f=r(b2) se2f=r(se2) t2f=r(t2) reject2f=r(r2) p2f=r(p2),  ///
>              reps($numsim) saving(chi2errors, replace) nolegend nodots: ///
>              chi2data

. use chi2errors, clear
(simulate: chi2data)

. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
b2f	1000	2.000506	.08427	1.719513	2.40565
se2f	1000	.0839776	.0172588	.0415919	.145264
t2f	1000	.0028714	.9932668	-2.824061	4.556576
reject2f	1000	.046	.2095899	0	1
p2f	1000	.5175819	.2890326	.0000108	.9997773

The mean of simulated `b2f` is very close to 2.0, implying the absence of bias. The standard deviation of simulated `b2f` is close to the mean of `se2f`, suggesting that the standard errors are unbiased as well. The mean rejection rate of 0.046 is close to the size of the test, 0.05.

In order to formally evaluate the simulation results, we use `mean` to obtain 95% confidence intervals for the simulation averages:

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

	Mean	Std. Err.	[95% Conf.	Interval]
b2f	2.000506	.0026649	1.995277	2.005735
se2f	.0839776	.0005458	.0829066	.0850486
reject2f	.046	.0066278	.032994	.059006

The 95% CI for the point estimate is [1.995, 2.006], validating the conclusion of its unbiasedness. The 95% CI for the standard error of the estimated coefficient is [0.083, 0.085], which contains the standard deviation of the simulated point estimates. We can also compare the empirical distribution of the t statistics with the theoretical distribution of t_{148}.

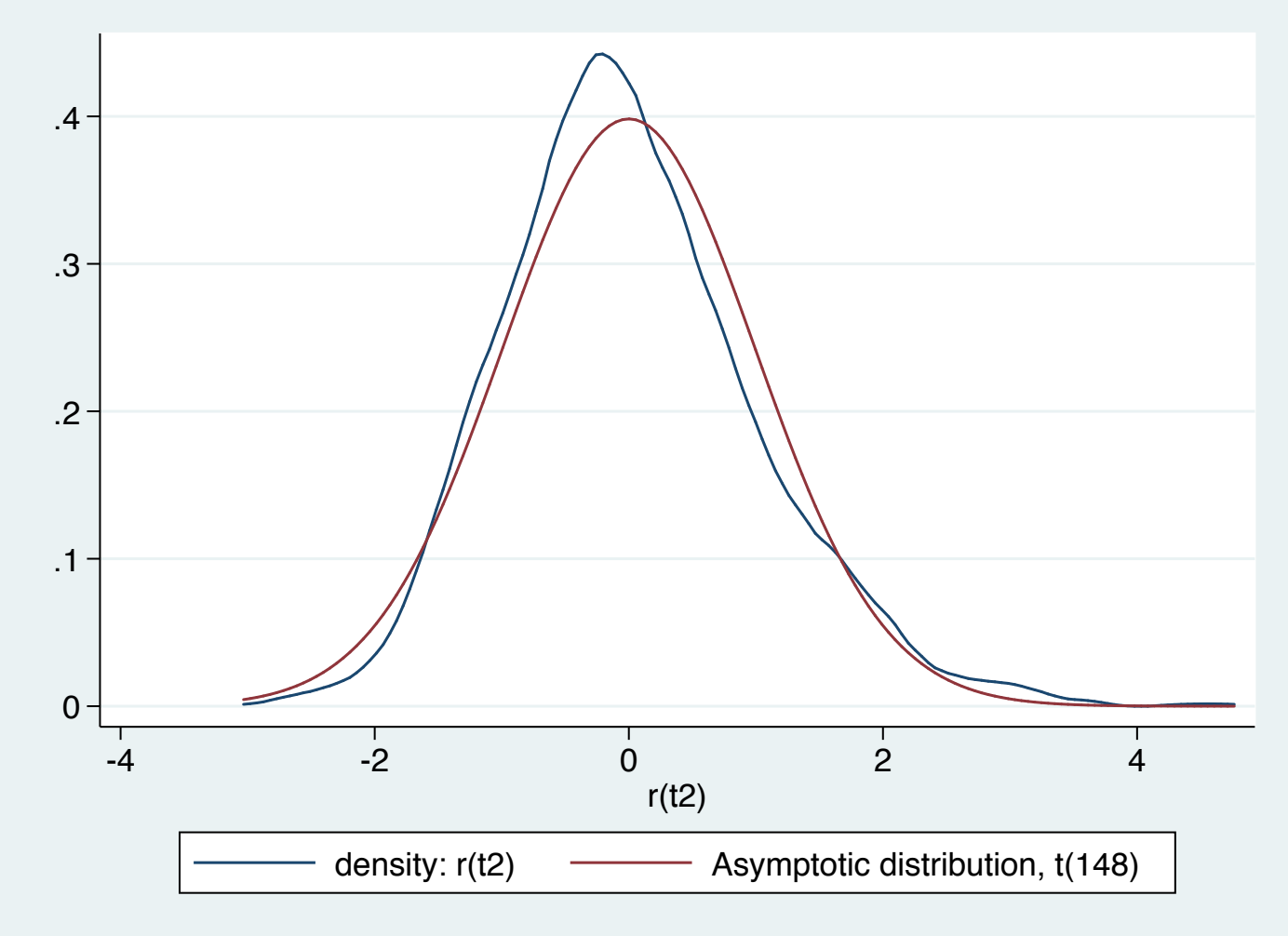
In order to formally evaluate the simulation results, we use `mean` to obtain 95% confidence intervals for the simulation averages:

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

	Mean	Std. Err.	[95% Conf.	Interval]
b2f	2.000506	.0026649	1.995277	2.005735
se2f	.0839776	.0005458	.0829066	.0850486
reject2f	.046	.0066278	.032994	.059006

The 95% CI for the point estimate is [1.995, 2.006], validating the conclusion of its unbiasedness. The 95% CI for the standard error of the estimated coefficient is [0.083, 0.085], which contains the standard deviation of the simulated point estimates. We can also compare the empirical distribution of the t statistics with the theoretical distribution of t_{148}.

```
. kdensity t2f, n($numobs) gen(t2_x t2_d) nograph
. qui gen double t2_d2 = tden(148, t2_x)
. lab var t2_d2 "Asymptotic distribution, t(148)"
. gr tw (line t2_d t2_x) (line t2_d2 t2_x, ylab(,angle(0)))
```



Size of the test

To evaluate the *size* of the test, the probability of rejecting a true null hypothesis: a Type I error, we can examine the rejection rate, `r2` above.

The estimated rejection rate from 1000 simulations is 0.046, with a 95% confidence interval of (0.033, 0.059): wide, but containing 0.05. With 10,000 replications, the estimated rejection rate is 0.049 with a confidence interval of (0.044, 0.052).

We computed the *p*-value of the test as `p2f`. If the *t*-distribution is the correct distribution, then *p*2 should be uniformly distributed on (0,1).

Size of the test

To evaluate the *size* of the test, the probability of rejecting a true null hypothesis: a Type I error, we can examine the rejection rate, `r2` above.

The estimated rejection rate from 1000 simulations is 0.046, with a 95% confidence interval of (0.033, 0.059): wide, but containing 0.05. With 10,000 replications, the estimated rejection rate is 0.049 with a confidence interval of (0.044, 0.052).

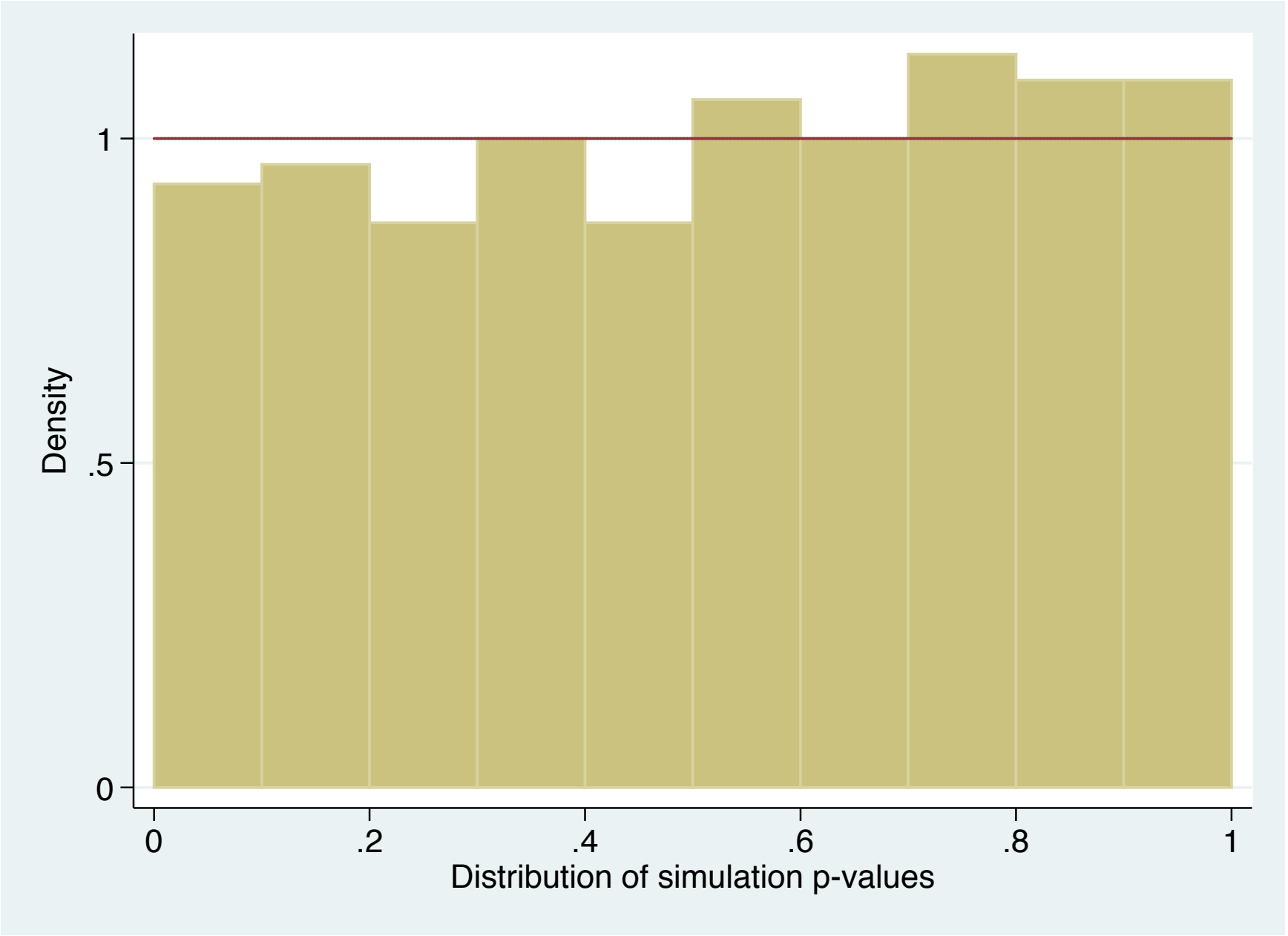
We computed the *p*-value of the test as `p2f`. If the *t*-distribution is the correct distribution, then *p*2 should be uniformly distributed on (0,1).

Size of the test

To evaluate the *size* of the test, the probability of rejecting a true null hypothesis: a Type I error, we can examine the rejection rate, `r2` above.

The estimated rejection rate from 1000 simulations is 0.046, with a 95% confidence interval of (0.033, 0.059): wide, but containing 0.05. With 10,000 replications, the estimated rejection rate is 0.049 with a confidence interval of (0.044, 0.052).

We computed the *p*-value of the test as `p2f`. If the *t*-distribution is the correct distribution, then *p*2 should be uniformly distributed on (0,1).


Density
0
.5
1
0
.2
.4
.6
.8
1
Distribution of simulation p-values

Using the computed set of *p*-values, we can evaluate the test size at any level of α:

```
 . qui count if p2f < 0.10
. di _n "Nominal size: 0.10" _n "For $numsim simulations: " _n "Test size    : "
>  r(N)/$numsim
Nominal size: 0.10
For 1000 simulations:
Test size    : .093
```

We see that the test is slightly undersized, corresponding to the histogram falling short of unity for lower levels of the *p*-value.

Power of the test

We can also evaluate the *power* of the test: its ability to reject a false null hypothesis. If we fail to reject a false null, we commit a Type II error. The power of the test is the complement of the probability of Type II error. Unlike the size, which can be evaluated for any level of α from a single simulation experiment, power must be evaluated for a specific null and alternative hypothesis.

We estimate the rejection rate for the 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. This modified version of the `chi2data` program estimates the power of a test against the false null hypothesis $\beta_x = 2.1$. We create a global macro to hold the hypothesized value so that it may be changed without revising the program.

Power of the test

We can also evaluate the *power* of the test: its ability to reject a false null hypothesis. If we fail to reject a false null, we commit a Type II error. The power of the test is the complement of the probability of Type II error. Unlike the size, which can be evaluated for any level of α from a single simulation experiment, power must be evaluated for a specific null and alternative hypothesis.

We estimate the rejection rate for the 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. This modified version of the `chi2data` program estimates the power of a test against the false null hypothesis $\beta_x = 2.1$. We create a global macro to hold the hypothesized value so that it may be changed without revising the program.

```
. capt prog drop chi2datab
. program chi2datab, rclass
  1.      version 12
  2.      drop _all
  3.      set obs $numobs
  4.      gen double x = rchi2(1)
  5.      gen y = 1 + 2*x + rchi2(1)-1
  6.      reg y x
  7.      ret sca b2  =_b[x]
  8.      ret sca se2 =_se[x]
  9.      test x = $hypbx
 10.      ret sca p2 = r(p)
 11.      ret sca r2 = (r(p)<.05)
 12. end
```

In this case, all we need do is invoke the `test` command and make use of one of its stored results, `r(p)`. The scalar `r2` is an indicator variable which will be 1 when the *p*-value of the test is below 0.05, 0 otherwise.

We run the program once to verify its functioning:

```
. set seed 10101
. glo hypbx = 2.1
. chi2datab
obs was 0, now 500
```

Source	SS	df	MS	Number of obs =	500
Model	5025.95627	1	5025.95627	F(1, 498) =	3368.07
Residual	743.13261	498	1.49223416	Prob > F =	0.0000
				R-squared =	0.8712
Total	5769.08888	499	11.5613004	Adj R-squared =	0.8709
				Root MSE =	1.2216

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x	1.981912	.0341502	58.04	0.000	1.914816	2.049008
_cons	.9134554	.0670084	13.63	0.000	.7818015	1.045109

```
 ( 1)  x = 2.1
       F(  1,   498) =   11.96
            Prob > F =    0.0006
. ret li
scalars:
                 r(r2) =  1
                 r(p2) =  .00059104547771
                r(se2) =  .03415021735296
                 r(b2) =  1.981911861267608
```

We proceed to run the simulation of test power:

```
 . set seed 10101
. glo numobs = 150
. glo numsim = 1000
. simulate b2f=r(b2) se2f=r(se2) reject2f=r(r2) p2f=r(p2),  ///
>              reps($numsim) saving(chi2errors, replace) nolegend nodots: ///
>              chi2datab

. use chi2errors, clear
(simulate: chi2datab)
. mean b2f se2f reject2f
Mean estimation                   Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2f	2.000506	.0026649	1.995277	2.005735
se2f	.0839776	.0005458	.0829066	.0850486
reject2f	.235	.0134147	.2086757	.2613243

We see that the test has quite low power, rejecting the false null hypothesis in only 23.5% of the simulations. Let's see how this would change with a larger sample size.

We see that with 1500 observations rather than 150, the power is substantially improved:

```
 . set seed 10101
. glo numsim = 1000
. glo numobs = 1500
. simulate b2f=r(b2) se2f=r(se2) reject2f=r(r2) p2f=r(p2),  ///
>              reps($numsim) saving(chi2errors, replace) nolegend nodots: ///
>              chi2datab

. use chi2errors, clear
(simulate: chi2datab)
. mean b2f se2f reject2f
Mean estimation                   Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2f	1.999467	.000842	1.997814	2.001119
se2f	.0258293	.0000557	.02572	.0259385
reject2f	.956	.0064889	.9432665	.9687335

The presence of skewed errors has weakened the ability of the estimates to reject the false null at smaller sample sizes.

The other dimension which we may explore is to hold sample size fixed and plot the *power curve*, which expresses the power of the test for various values of the false null hypothesis.

We can produce this set of results by using Stata's `postfile` facility, which allows us to create a new Stata dataset from within the program. The `postfile` command is used to assign a *handle*, list the scalar quantities that are to be saved for each observation, and the name of the file to be created. The `post` command is then called within a loop to create the observations, and the `postclose` command to close the resulting data file.

The other dimension which we may explore is to hold sample size fixed and plot the *power curve*, which expresses the power of the test for various values of the false null hypothesis.

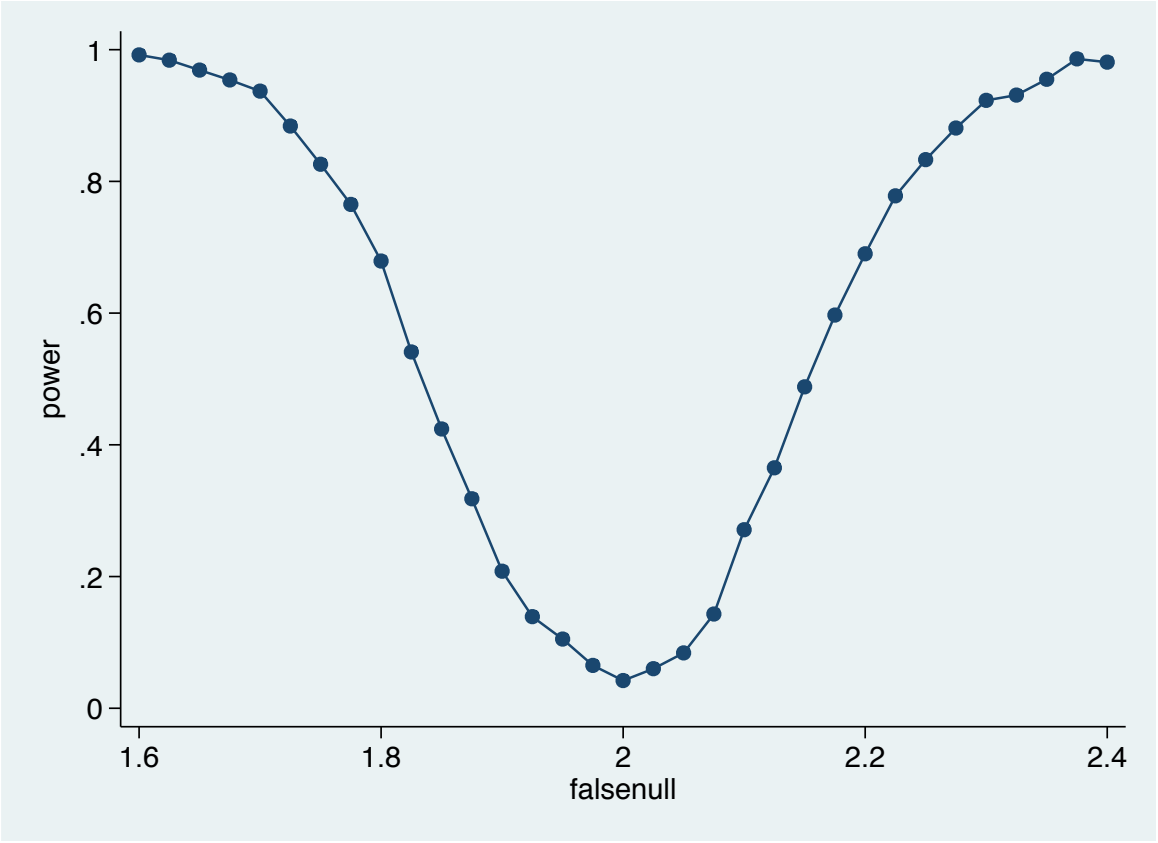
We can produce this set of results by using Stata's `postfile` facility, which allows us to create a new Stata dataset from within the program. The `postfile` command is used to assign a *handle*, list the scalar quantities that are to be saved for each observation, and the name of the file to be created. The `post` command is then called within a loop to create the observations, and the `postclose` command to close the resulting data file.

```
. glo numobs = 150
. tempname pwrcurve
. postfile `pwrcurve´ falsenull power using powercalc, replace
. forv i=1600(25)2400 {
  2.         glo hypbx = `i´/1000
  3.         qui simulate b2f=r(b2) se2f=r(se2) reject2f=r(r2) p2f=r(p2),  ///
>             reps($numsim) nolegend nodots: chi2datab
  4.     qui count if p2f < 0.05
  5.     loc power = r(N) / $numsim
  6.     qui post `pwrcurve´ ($hypbx) (`power´)
  7. }
. postclose `pwrcurve´
```

```
. use powercalc, clear
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
falsenull	33	2	.2417385	1.6	2.4
power	33	.5999394	.3497309	.042	.992

```
. tw (connected power falsenull, yla(,angle(0))), plotregion(style(none))
```



Evaluating coverage with simpplot

An excellent tool for examining the coverage of a statistical test is the `simpplot` routine, written by Maarten Buis of WZB and available from `ssc`. From the routine's description, "simpplot describes the results of a simulation that inspects the coverage of a statistical test. simpplot displays by default the deviations from the nominal significance level against the entire range of possible nominal significance levels. It also displays the range (Monte Carlo region of acceptance) within which one can reasonably expect these deviations to remain if the test is well behaved."

In this example, adapted from the help file, we consider the performance of a *t*-test when the data are not Gaussian, but rather generated by a $\chi^2(2)$, with a mean of 2.0. A *t*-test of the null that $\mu = 2$ is a test of the true null hypothesis. We want to evaluate how well the *t*-test performs at various sample sizes: *N* and $N/10$.

```
. capt program drop sim
. program define sim, rclass
  1.     drop _all
  2.     qui set obs $numobs
  3.     gen x = rchi2(2)
  4.     loc frac = $numobs / 10
  5.     ttest x=2 in 1/`frac´
  6.     ret sca pfrac = r(p)
  7.     ttest x=2
  8.     ret sca pfull = r(p)
  9.     end
```

In this example, adapted from the help file, we consider the performance of a *t*-test when the data are not Gaussian, but rather generated by a $\chi^2(2)$, with a mean of 2.0. A *t*-test of the null that $\mu = 2$ is a test of the true null hypothesis. We want to evaluate how well the *t*-test performs at various sample sizes: *N* and $N/10$.

```
. capt program drop sim
. program define sim, rclass
  1.     drop _all
  2.     qui set obs $numobs
  3.     gen x = rchi2(2)
  4.     loc frac = $numobs / 10
  5.     ttest x=2 in 1/`frac´
  6.     ret sca pfrac = r(p)
  7.     ttest x=2
  8.     ret sca pfull = r(p)
  9.     end
```

We choose $N = 500$ and produce the p-values for the full sample (`pfull`) and for $N = 50$ (`pfrac`):

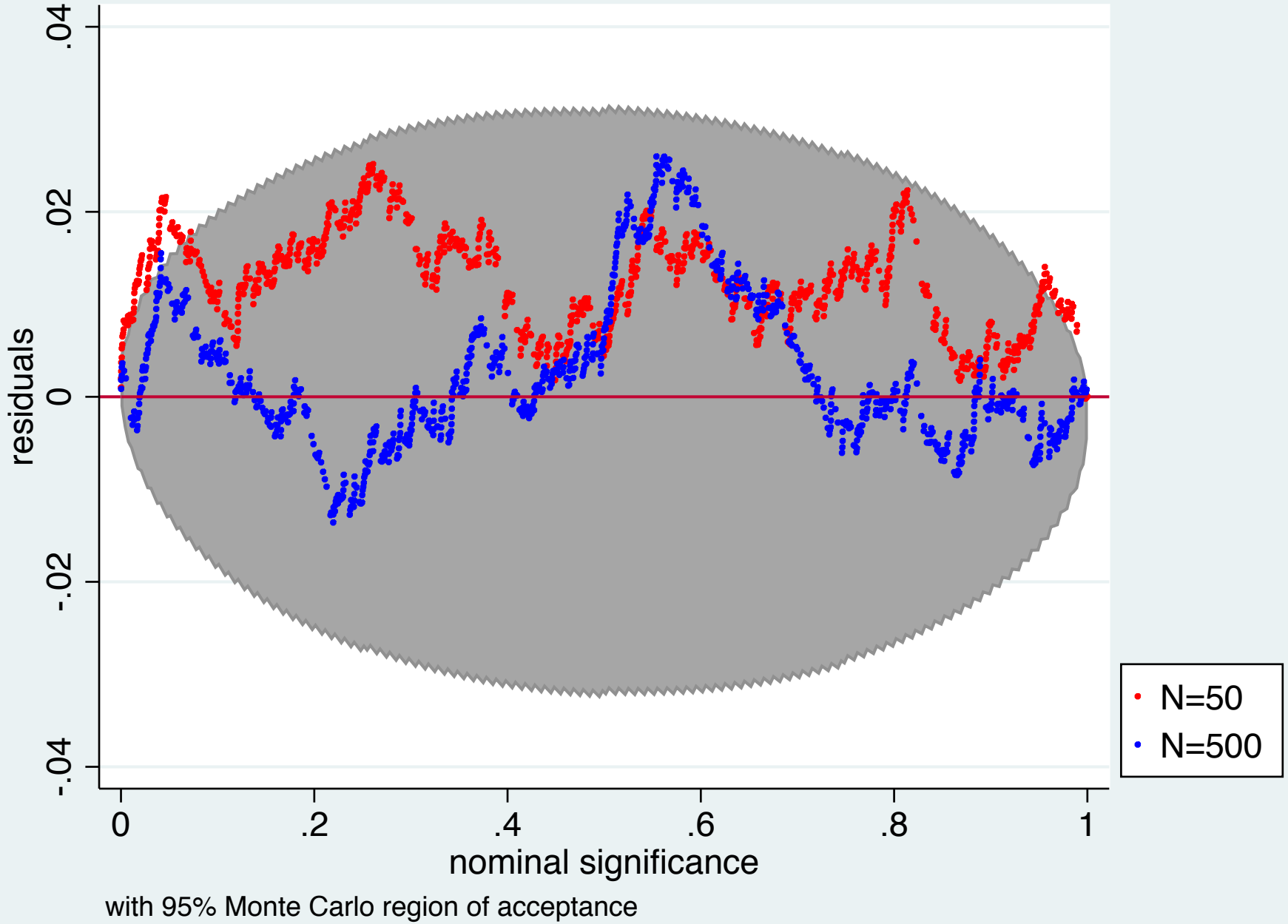
```
 . glo numobs = 500
. glo numrep = 1000
. set seed 10101
. simulate pfrac=r(pfrac) pfull=r(pfull), ///
>     reps($numrep) nolegend nodots : sim
. loc nfull = $numobs
. loc nfrac = `nfull´ / 10
. lab var pfrac "N=`nfrac´"
. lab var pfull "N=`nfull´"
. simpplot pfrac pfull, main1opt(mcolor(red) msize(tiny)) ///
>                       main2opt(mcolor(blue) msize(tiny)) ///
>                       ra(fcolor(gs9) lcolor(gs9))
```

By default, `simpplot` graphs the deviations from the nominal significance level across the range of significance levels. The shaded area is the region where these deviations should lie if the test is well behaved.

We choose $N = 500$ and produce the p-values for the full sample (`pfull`) and for $N = 50$ (`pfrac`):

```
 . glo numobs = 500
. glo numrep = 1000
. set seed 10101
. simulate pfrac=r(pfrac) pfull=r(pfull), ///
>     reps($numrep) nolegend nodots : sim
. loc nfull = $numobs
. loc nfrac = `nfull´ / 10
. lab var pfrac "N=`nfrac´"
. lab var pfull "N=`nfull´"
. simpplot pfrac pfull, main1opt(mcolor(red) msize(tiny)) ///
>                       main2opt(mcolor(blue) msize(tiny)) ///
>                       ra(fcolor(gs9) lcolor(gs9))
```

By default, `simpplot` graphs the deviations from the nominal significance level across the range of significance levels. The shaded area is the region where these deviations should lie if the test is well behaved.


residuals
.04
.02
0
-.02
-.04
0
.2
.4
.6
.8
1
nominal significance
N=50
N=500
with 95% Monte Carlo region of acceptance

We can see that for a sample size of 500, the test stays within bounds for almost all nominal significance levels. For the smaller sample of $N = 50$, there are a number of values 'out of bounds' for both low and high nominal significance levels, showing that the test rejects the true null too frequently at that limited sample size.

Simulating a spurious regression model

In the context of time series data, we can demonstrate Granger's concept of a *spurious regression* with a simulation. We create two independent random walks, regress one on the other, and record the coefficient, standard error, *t*-ratio and its tail probability in the saved results from the program. We use a global macro, `trcoef`, to allow the program to be used to model both pure random walks and random walks with drift.

```
. capt prog drop irwd
. prog irwd, rclass
 1.         version 12
 2.         drop _all
 3.         set obs $numobs
 4.         g double x = 0 in 1
 5.         g double y = 0 in 1
 6.         replace x = x[_n - 1] + $trcoef * 2 + rnormal() in 2/l
 7.         replace y = y[_n - 1] + $trcoef * 0.5 + rnormal() in 2/l
 8.         reg y x
 9.         ret sca b = _b[x]
10.         ret sca se = _se[x]
11.         ret sca t = _b[x]/_se[x]
12.         ret sca r2 = abs(return(t)) > invttail($numobs - 2, 0.025)
13. end
```

We simulate the model with pure random walks for 10000 observations:

```
. set seed 10101
. glo numsim = 1000
. glo numobs = 10000
. glo trcoef = 0
. simulate b=r(b) se=r(se) t=r(t) reject=r(r2), reps($numsim) ///
>          saving(spurious, replace) nolegend nodots: irwd
. use spurious, clear
(simulate: irwd)
. mean b se t reject
Mean estimation                   Number of obs    =       1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b	-.0305688	.019545	-.0689226	.0077851
se	.0097193	.0001883	.0093496	.0100889
t	-1.210499	2.435943	-5.990652	3.569653
reject	.979	.0045365	.9700979	.9879021

The true null is rejected in 97.9% of the simulated samples.

We simulate the model of random walks with drift:

```
. set seed 10101
. glo numsim = 1000
. glo numobs = 10000
. glo trcoef = 1
. simulate b=r(b) se=r(se) t=r(t) reject=r(r2), reps($numsim) ///
>          saving(spurious, replace) nolegend nodots: irwd
. use spurious, clear
(simulate: irwd)
. mean b se t reject
Mean estimation                   Number of obs    =       1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b	.2499303	.0001723	.249592	.2502685
se	.0000445	4.16e-07	.0000437	.0000453
t	6071.968	53.17768	5967.615	6176.321
reject	1	0	.	.

The true null is rejected in 100% of the simulated samples, clearly indicating the severity of the spurious regression problem.

Simulating an errors-in-variables model

In order to demonstrate how measurement error may cause OLS to produce biased and inconsistent results, we generate data from an errors-in-variables model:

$$\begin{aligned} y &= \alpha + \beta x^* + u,\ x^* \sim N(0,9),\ u \sim N(0,1) \\ x &= x^* + v,\ v \sim N(0,1) \end{aligned}$$

In the true DGP, y depends on x^*, but we do not observe x^*, only observing the mismeasured x. Even though the measurement error is uncorrelated with all other RVs, this still causes bias and inconsistency in the estimate of β.

We do not need `simulate` in this example, as a single dataset meeting these specifications is sufficient.

Simulating an errors-in-variables model

In order to demonstrate how measurement error may cause OLS to produce biased and inconsistent results, we generate data from an errors-in-variables model:

$$\begin{aligned} y &= \alpha + \beta x^* + u,\ x^* \sim N(0,9),\ u \sim N(0,1) \\ x &= x^* + v,\ v \sim N(0,1) \end{aligned}$$

In the true DGP, y depends on x^*, but we do not observe x^*, only observing the mismeasured x. Even though the measurement error is uncorrelated with all other RVs, this still causes bias and inconsistency in the estimate of β.

We do not need `simulate` in this example, as a single dataset meeting these specifications is sufficient.

Simulating an errors-in-variables model

In order to demonstrate how measurement error may cause OLS to produce biased and inconsistent results, we generate data from an errors-in-variables model:

$$\begin{aligned} y &= \alpha + \beta x^* + u,\ x^* \sim N(0,9),\ u \sim N(0,1) \\ x &= x^* + v,\ v \sim N(0,1) \end{aligned}$$

In the true DGP, y depends on x^*, but we do not observe x^*, only observing the mismeasured x. Even though the measurement error is uncorrelated with all other RVs, this still causes bias and inconsistency in the estimate of β.

We do not need `simulate` in this example, as a single dataset meeting these specifications is sufficient.

```
. set seed 10101
. qui set obs 10000
. mat mu = (0,0,0)
. mat sigmasq = (9,0,0 \ 0,1,0 \ 0,0,1)
. drawnorm xstar u v, means(mu) cov(sigmasq)
. g double y = 5 + 2 * xstar + u
. g double x = xstar + v      // mismeasured x
. reg y x
```

Source	SS	df	MS
Model	320512.118	1	320512.118
Residual	45636.9454	9998	4.56460746
Total	366149.064	9999	36.6185682

```
Number of obs =   10000
F(  1,  9998) =70216.80
Prob > F      =  0.0000
R-squared     =  0.8754
Adj R-squared =  0.8753
Root MSE      =  2.1365
```

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x	1.795335	.0067752	264.98	0.000	1.782054	1.808616
_cons	5.005169	.021366	234.26	0.000	4.963288	5.047051

We see a sizable attenuation bias in the estimate of β, depending on the noise-signal ratio $\sigma_v^2/(\sigma_v^2 + \sigma_{x*}^2) = 0.1$, implying an estimate of 1.8.

```
. set seed 10101
. qui set obs 10000
. mat mu = (0,0,0)
. mat sigmasq = (9,0,0 \ 0,1,0 \ 0,0,1)
. drawnorm xstar u v, means(mu) cov(sigmasq)
. g double y = 5 + 2 * xstar + u
. g double x = xstar + v      // mismeasured x
. reg y x
```

Source	SS	df	MS
Model	320512.118	1	320512.118
Residual	45636.9454	9998	4.56460746
Total	366149.064	9999	36.6185682

```
Number of obs =   10000
F(  1,  9998) =70216.80
Prob > F      =  0.0000
R-squared     =  0.8754
Adj R-squared =  0.8753
Root MSE      =  2.1365
```

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x	1.795335	.0067752	264.98	0.000	1.782054	1.808616
_cons	5.005169	.021366	234.26	0.000	4.963288	5.047051

We see a sizable attenuation bias in the estimate of β, depending on the noise-signal ratio $\sigma_v^2/(\sigma_v^2 + \sigma_{x^*}^2) = 0.1$, implying an estimate of 1.8.

If we increase the measurement error variance, the attenuation bias becomes more severe:

```
. set seed 10101
. qui set obs 10000
. mat mu = (0,0,0)
. mat sigmasq = (9,0,0 \ 0,1,0 \ 0,0,4)  // larger measurement error variance
. drawnorm xstar u v, means(mu) cov(sigmasq)
. g double y = 5 + 2 * xstar + u
. g double x = xstar + v      // mismeasured x
. reg y x
```

Source	SS	df	MS		
Model	246636.774	1	246636.774	Number of obs	= 10000
Residual	119512.29	9998	11.9536197	F(1, 9998)	=20632.81
				Prob > F	= 0.0000
				R-squared	= 0.6736
Total	366149.064	9999	36.6185682	Adj R-squared	= 0.6736
				Root MSE	= 3.4574

y	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
x	1.378317	.0095956	143.64	0.000	1.359508	1.397126
_cons	5.007121	.0345763	144.81	0.000	4.939344	5.074897

With a noise-signal ratio of 4/13, the coefficient that is 9/13 of the true value.

Simulating a model with endogenous regressors

In order to simulate how a violation of the zero conditional mean assumption, $E[u|X] = 0$, causes inconsistency, we simulate a DGP in which that correlation is introduced:

$$\begin{aligned} y &= \alpha + \beta x + u,\ u \sim N(0,1) \\ x &= z + \rho u,\ z \sim N(0,1) \end{aligned}$$

and then estimate the regression of y on x via OLS.

```
. capt prog drop endog
. prog endog, rclass
  1.          version 12
  2.          drop _all
  3.          set obs $numobs
  4.          g double u = rnormal(0)
  5.          g double z = rnormal(0)
  6.          g double x = z + $corrxu * u
  7.          g double y = 10 + 2 * x + u
  8.          if ($ols) {
  9.                  reg y x
 10.          }
 11.          else {
 12.                  ivreg2 y (x = z)
 13.          }
 14.          ret sca b2 = _b[x]
 15.          ret sca se2 = _se[x]
 16.          ret sca t2 = (_b[x] - 2) / _se[x]
 17.          ret sca p2 = 2 * ttail($numobs - 2, abs(return(t2)))
 18.          ret sca r2 = abs(return(t2) > invttail($numobs - 2, 0.025))
 19. end
```

The program returns the *t*-statistic for a test of β_x against its true value of 2.0, as well as the *p*-value of that test and an indicator of rejection at the 95% level.

Setting ρ, the correlation between regressor and error to 0.5, we find a serious bias in the estimated coefficient:

```
 . set seed 10101
. glo numobs = 150
. glo numrep = 1000
. glo corrxu = 0.5
. glo ols = 1
. simulate b2r=r(b2) se2r=r(se2) t2r=r(t2) p2r=r(p2) r2r=r(r2), ///
> reps($numrep) noleg nodots saving(endog, replace): endog
. mean b2r se2r r2r
Mean estimation                   Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2r	2.397172	.0021532	2.392946	2.401397
se2r	.0660485	.0001693	.0657163	.0663807
r2r	1	0	.	.

A smaller value of $\rho = 0.2$ reduces the bias in the estimated coefficient:

```
. set seed 10101
. glo numobs = 150
. glo numrep = 1000
. glo corrxu = 0.2
. glo ols = 1
. simulate b2r=r(b2) se2r = r(se2) t2r=r(t2) p2r=r(p2) r2r=r(r2), ///
> reps($numrep) noleg nodots saving(endog, replace): endog
. mean b2r se2r r2r
Mean estimation                   Number of obs    =    1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2r	2.187447	.0025964	2.182352	2.192542
se2r	.0791955	.0002017	.0787998	.0795912
r2r	.645	.0151395	.6152911	.6747089

The upward bias is still about 10% of the DGP value, and rejection of the true null still occurs in 64.5% of the simulations.

We can also demonstrate the inconsistency of the estimator by using a much larger sample size:

```
. set seed 10101
. glo numobs = 15000
. glo numrep = 1000
. glo corrxu = 0.2
. glo ols = 1
. simulate b2r=r(b2) se2r = r(se2) t2r=r(t2) p2r=r(p2) r2r=r(r2), ///
> reps($numrep) noleg nodots saving(endog, replace): endog
. mean b2r se2r r2r
Mean estimation                   Number of obs    =      1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2r	2.19204	.0002448	2.19156	2.19252
se2r	.0078569	2.04e-06	.0078529	.0078609
r2r	1	0	.	.

With N=15,000, the rejection of the true null occurs in every simulation.

By setting the global macro `ols` to 0, we can simulate the performance of the instrumental variables estimator of this exactly identified model, which should be consistent:

```
. set seed 10101
. glo numobs = 150
. glo numrep = 1000
. glo corrxu = 0.5
. glo ols = 0
. simulate b2r=r(b2) se2r=r(se2) t2r=r(t2) p2r=r(p2) r2r=r(r2), ///
> reps($numrep) noleg nodots saving(endog, replace): endog
. mean b2r se2r r2r
Mean estimation                   Number of obs    =    1000
```

	Mean	Std. Err.	[95% Conf.	Interval]
b2r	1.991086	.0026889	1.985809	1.996362
se2r	.0825012	.000302	.0819086	.0830939
r2r	.029	.0053092	.0185816	.0394184

The rejection frequency of the true null is only 2.9%, indicating that the IV estimator is consistently estimating β_x.