EC771: Econometrics, Spring 2009

Greene, Econometric Analysis (6th ed, 2008)

Chapter 17: Simulation Based Estimation and Inference

We often want to evaluate the properties of estimators, or compare a proposed estimator to another, in a context where analytical derivation of those properties is not feasible. In that case, econometricians resort to **Monte Carlo studies**: simulation methods making use of (pseudo-)random draws from an error distribution and multiple replications over a set of known parameters. This methodology is particularly relevant in situations where the only analytical findings involve asymptotic, large–sample results. Applied researchers need to understand how a particular estimation strategy will perform in small samples: for instance, when working with macro data on the national aggregates, we have no more than 150–200 quarterly observations available for many series. Where only annual data are available, the problem becomes even more striking. In that case, we require an understanding of the performance of estimation techniques, test statistics, etc. in a very small sample. Monte Carlo studies, although they do not generalize to cases beyond those performed in the experiment, may be useful in these situations. They also are useful in modelling quantities for which no analytical results have yet been derived: for instance, the critical values for many unit-root test statistics have been derived by simulation experiments, in the absence of closed-form expressions for the sampling distributions of the statistics.

Most econometric software provide some facilities for Monte Carlo experiments. Although one can write the code to generate an experiment in any programming language, it is most useful to do so in a context where one may readily save the results of each replication for further analysis. The quality of the pseudo-random number generators available is also an important concern. Recent studies published in the *Journal of Applied Econometrics* have compared many software packages' performance on a standard set of benchmarks for randomness. Although most packages meet these criteria, all but the most recent versions of GAUSS fail miserably—casting considerable doubt on those many published studies making use of GAUSS software. State-of-the-art pseudo-random number generators do exist, and you should use a package that implements them. You will also want a package with a full set of statistical functions, permitting random draws to be readily made from a specified distribution-not merely normal or t, but from a number of additional distributions, depending upon the experiment.

Stata version 10.1 provides a useful environment for Monte Carlo simulations. Setting up a simulation requires that you write a Stata program: not merely a "do-file" containing a set of Stata commands, but a sequence of commands beginning with the `program define` statement. This program sets up the simulation experiment and specifies what is to be done in one replication; you then invoke it with the `simulate` prefix to execute a specified number of replications. *Cameron and Trivedi, Microeconomics Using Stata, 2009*

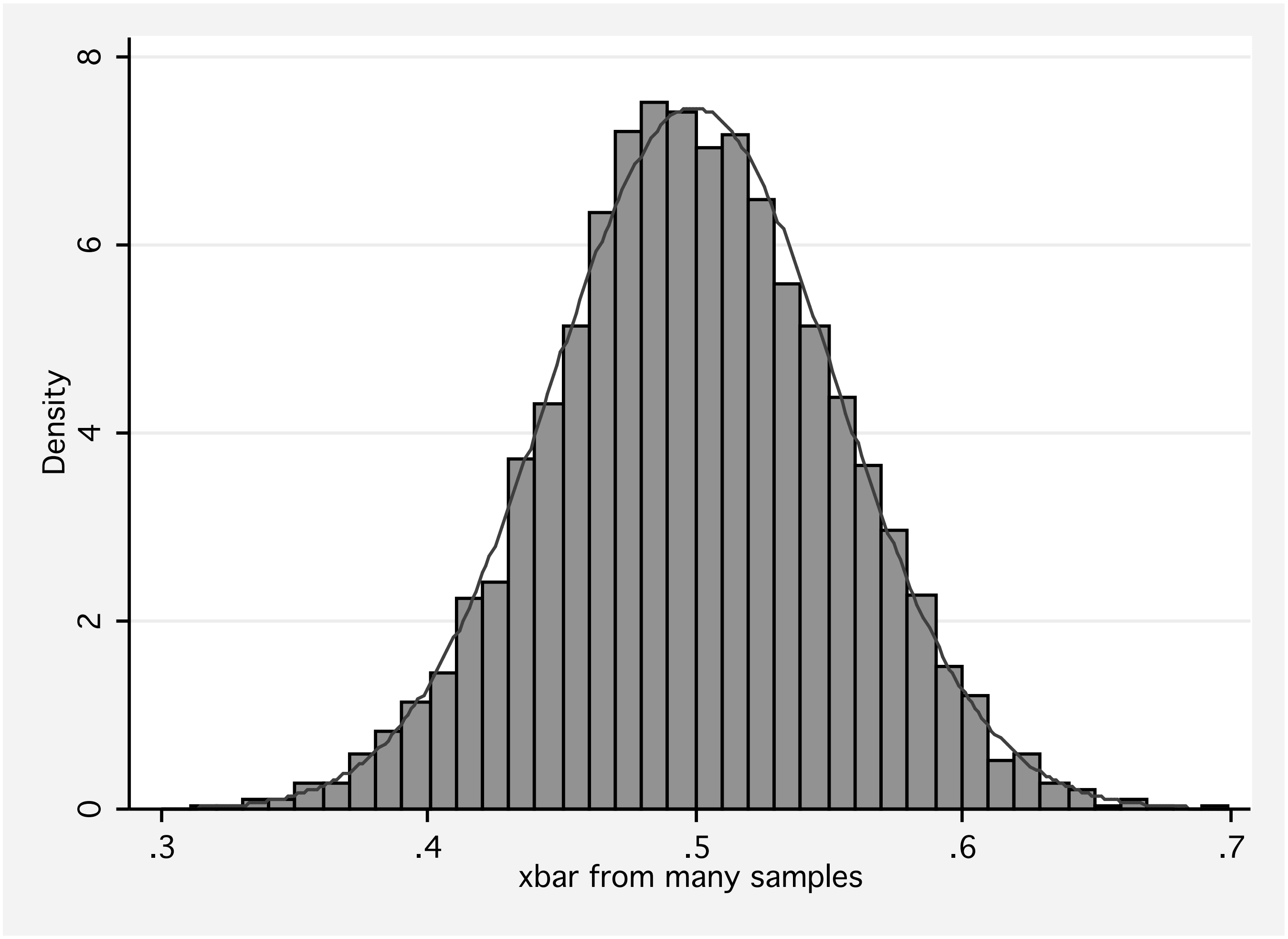
We first consider a very simple program in which we demonstrate the central limit their result that sample mean is approximately normally distributed as $N \to \infty$. We consider a random variable that has the uniform distribution and a sample size of 30. The `simulate` command runs a specified command a number of times, where the command will often be a user-written program. A number of expressions are returned by the command, and saved in a new Stata dataset by `simulate`. As an example:

```
// Program to draw sample of size 30 from uniform
// and return sample mean
program onesample, rclass
    drop _all
    quietly set obs 30
    generate x = runiform()
    summarize x
    return scalar meanforonesample = r(mean)
end
```

We can run the program with `simulate`, returning the one result as `xbar`:

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

The `set seed` ensures that the same sequence of pseudo-random numbers will be used every time the simulation is run. This is useful when debugging the program to ensure that it is coded properly. The results of `simulate` can then be summarized and/or graphed (see `771mcsim1.html`).



As a more interesting example of Monte Carlo simulation, let us consider simulation methods to investigate the finite-sample properties of the OLS estimator with random regressors and skewed errors. If errors are $i.i.d.$, skewness will have no effect on the large-sample properties of the OLS estimator. But with skewed errors, we will need a larger sample size for the asymptotic distribution to approximate the finite-sample distribution of the OLS estimator than when errors are normal.

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 and the number of simulations in global macros, as we often may want to change them.

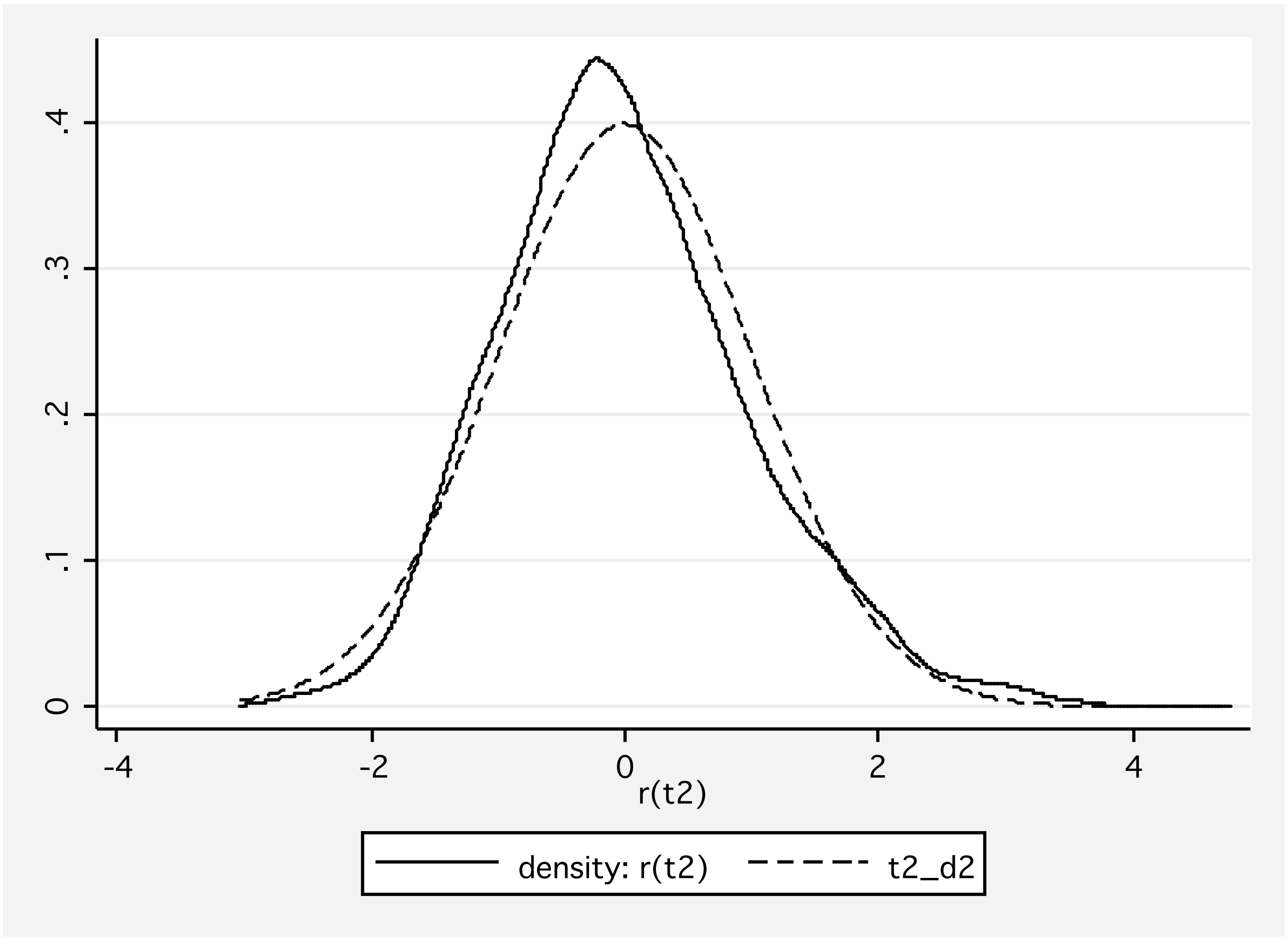
Our simulation program becomes

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

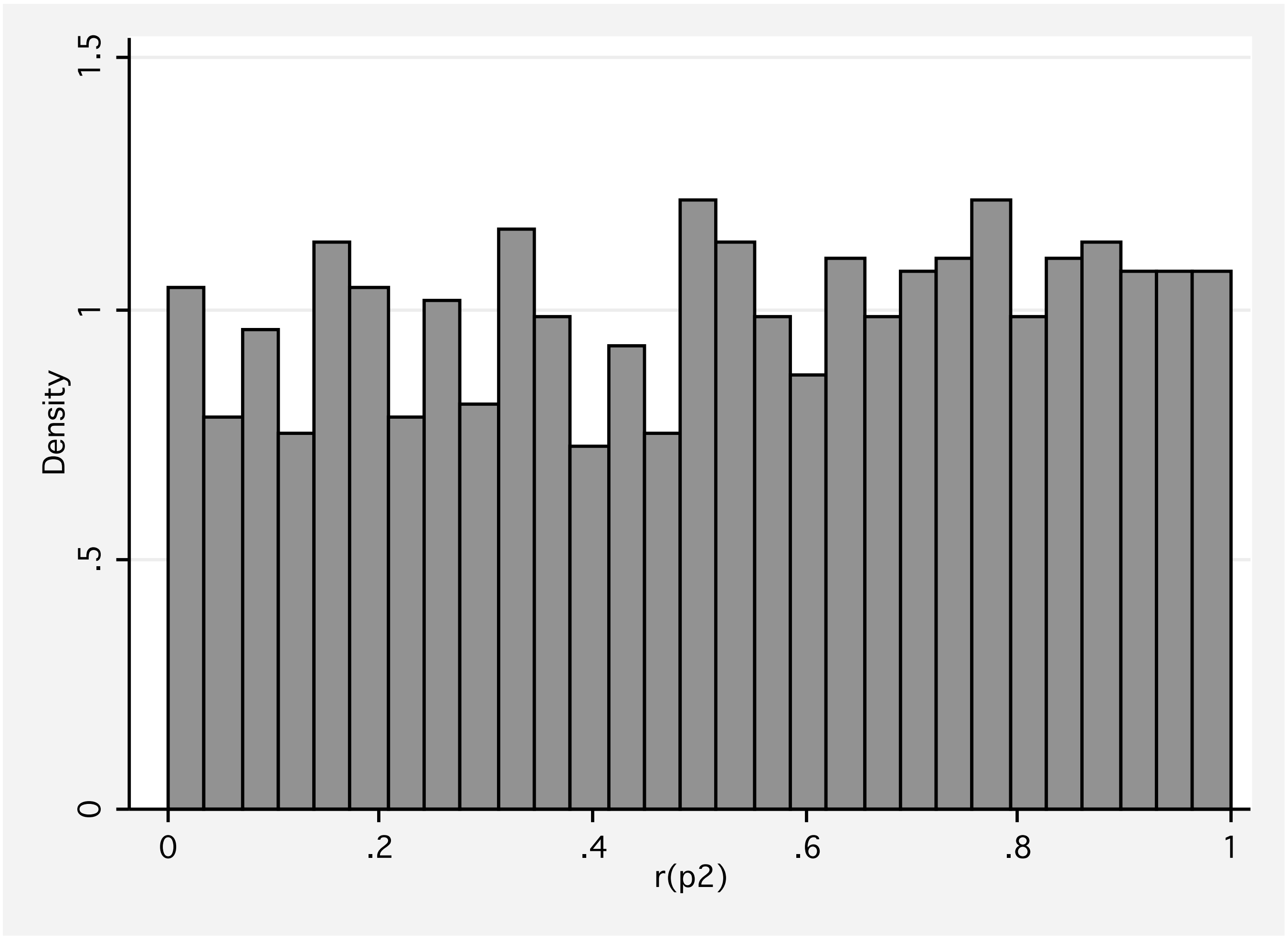
We can now run this program with `simulate`, producing a dataset containing the five scalars listed above for each simulation (see `771mcsim2.html`).

```
set seed 10101
* 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
```

We can conclude that our estimator of β_2 is unbiased, as the quite narrow 95% confidence interval from 1000 simulations contains the true value of 2.0. We can consider how closely the distribution of t-statistics from the program approximate the asymptotic distribution of a t_{148}:



To evaluate the *size* of the test (the probability of rejecting a true null hypothesis), 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). The last item computed is the p-value of the test. If the t-distribution is the correct distribution, then $p2$ should be uniformly distributed on (0,1).



We can also evaluate the *power* of the test: its ability to reject a false null 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 $\beta_2 = 2.1$.

```
* Finite-sample properties of OLS: power
program chi2datab, rclass
    version 10.1
    drop _all
    set obs $numobs
    generate double x = rchi2(1)
 // demeaned chi^2 error
    generate y = 1 + 2*x + rchi2(1)-1
    regress y x
    return scalar b2  =_b[x]
    return scalar se2 =_se[x]
    test x=2.1
    return scalar r2 = (r(p)<.05)
end
```

In this case, the power is not high, with a mean of 0.241. Using a larger sample size or increasing the distance between the true and false values would increase the power of the test (see `771mcsim2.html`).

Simulating a spurious regression

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 returns from the program:

```
* spurious regression: independent random walks
prog irwd, rclass
version 10.1
drop _all
set obs $N
local drift 2
```

```
g double x = 0 in 1
g double y = 0 in 1
replace x = x[_n - 1] + $trcoef * `drift' ///
                   + rnormal() in 2/l
replace y = y[_n - 1] + $trcoef * `drift' ///
                   + rnormal() in 2/l
reg y x
return scalar b = _b[x]
return scalar se = _se[x]
return scalar t = _b[x]/_se[x]
return scalar r2 = ///
abs(return(t)) > invttail($N - 2, 0.025)
end
```

We use a global, `trcoef`, to allow the program to be used for both pure random walks and random walks with drift. The handout `771irwd.html` illustrates that for pure random walks with $N = 100$, the true null hypothesis that $\partial y/\partial x = 0$ is rejected in over 75% of 10,000 simulations. For random walks with drift, the null is rejected in every simulation.

Bootstrapping

A closely related topic to Monte Carlo simulation is that of the technique of **bootstrapping,** developed by Efron (1979). A key difference: whereas Monte Carlo simulation is designed to utilize purely random draws from a specified distribution (which with sufficient sample size will follow that theoretical distribution) bootstrapping is used to obtain a description of the sampling properties of empirical estimators, using the empirical distribution of sample data. If we derive an estimate θ_{obs} from a sample $X = (x_1, x_2, ..., x_N)$, we can derive a bootstrap estimate of its precision by generating a sequence of bootstrap estimators $\left(\widehat{\theta}_1, \widehat{\theta}_2, ..., \widehat{\theta}_B\right)$, with each estimator generated from an $m-$observation sample from X, *with replacement*. The size of the bootstrap sample m may be larger, smaller or equal to N. The estimated asymptotic variance of θ may then be computed from this sequence of bootstrap estimates and the original estimator, θ_{obs} (for observed):

$$Est.Asy.Var[\theta] = B^{-1} \sum_{b=1}^{B} \left[\widehat{\theta}_b - \theta_{obs}\right] \left[\widehat{\theta}_b - \theta_{obs}\right]'$$

where the formula has been written to allow $\widehat{\theta}$ to be a vector of estimated parameters. The square roots of this variance-covariance matrix are known as the **bootstrap standard errors** of $\widehat{\theta}$. They will often prove useful when doubt exists regarding the appropriateness of the conventional estimates of the precision matrix, as well as in cases where no analytical expression for that matrix is available, e.g., in the context of a highly nonlinear estimator for which the numerical Hessian may not be computed.

After bootstrapping, we have the mean of the estimated statistic—e.g. $\widehat{\theta}$—which may be compared with the point estimate of the statistic computed from the original sample, θ_{obs} (for observed). The difference $\widehat{\theta}-\theta_{obs}$ is an estimate of the bias of the statistic; in the presence of a biased point estimate, this bias may be non-trivial. However we cannot use that difference to construct an unbiased estimate, since the bootstrap estimate contains an indeterminate amount of random error.

Why do we bootstrap quantities for which asymptotic measures of precision exist? All measures of precision come from the statistic's sampling distribution, which is in turn determined by the distribution of the population and the formula used to estimate the statistic from a sample of size N. In some cases, analytical estimates of the sampling distribution are difficult or infeasible to compute, such as those relating to the means from non-normal populations. Bootstrapping estimates of precision rely on the notion that the observed distribution in the sample is a good approximation to the population distribution.

The `bootstrap` command specifies a single estimation command, the results to be retained from that command, and the number of bootstrap samples (B) to be drawn. You may optionally specify the size of the bootstrap samples (m); if you do not, it defaults to $_N$ (the currently defined sample size). This is very useful, since it makes estimating bootstrap standard errors no more difficult than performing the estimation itself. If you are trying to construct a bootstrap distribution for a set of statistics which are forthcoming from a single Stata command, this may be done without further programming.

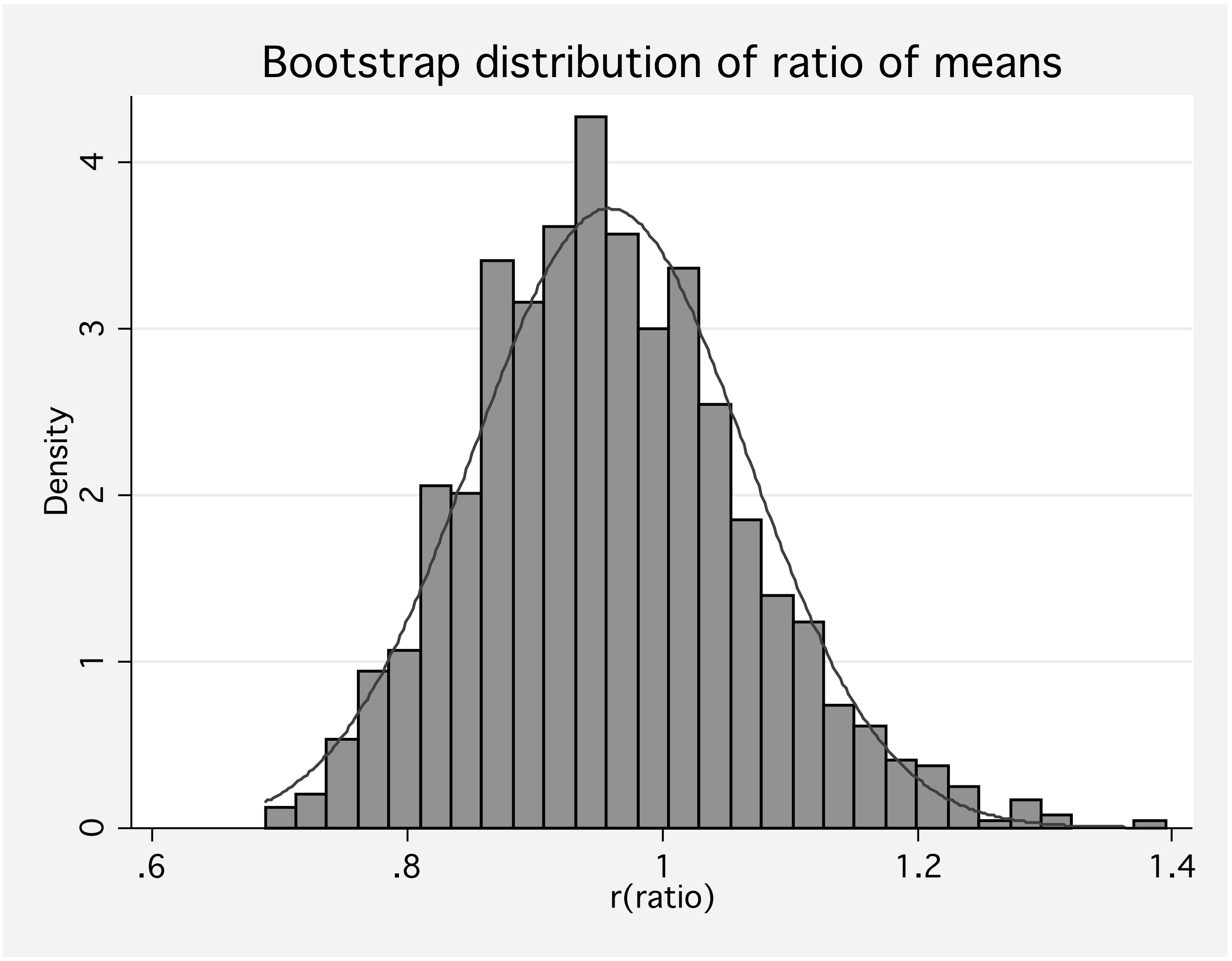
In the output from the `bootstrap` command (technically, in the output from the `bstat` command, which is automatically invoked by `bootstrap`) the bias is presented as the difference above. The first confidence interval (labelled (N)) is based on the assumption of approximate normality of the sampling (and bootstrap) distribution, and will be reasonable if that assumption is so. The percentile (P) and bias-corrected (BC) bootstrap confidence intervals are computed without making the assumption of approximate normality, and demonstrate the sensitivity of the bootstrap estimates to that feature of the empirical distribution (for instance, those confidence intervals need not be symmetric around θ_{obs}). Note on the graph that for the estimates of the mean, the bootstrap distribution diverges to some degree from normality, with considerably more mass in the center of the distribution.

The **bootstrap** command is not limited to generating bootstrap estimates from a single Stata command. To compute a bootstrap distribution for more complicated quantities, you must write a Stata program (just as with the `simulate` command) that specifies the estimation to be performed in the bootstrap sample. One may then execute `bootstrap`, specifying the name of your program, and the number of bootstrap samples to be drawn. For instance, if we wanted to generate a bootstrap estimate of the ratio of two means, we could not do so with a single Stata command. We could do so by writing a program that returned that ratio:

```
capture program drop muratio
program define muratio, rclass
           version 10.1
           syntax varlist(min=2 max=2)
           tempname ymu
           summarize ‘1’, meanonly
           scalar ‘ymu’ = r(mean)
           summarize ‘2’, meanonly
           return scalar ratio = ‘ymu’/r(mean)
end
```

We can now execute this program to compute the ratio of the average price of a domestic car vs. the average price of a foreign car, and generate a bootstrap confidence interval for the ratio (see `771bstrap1.html`).

```
set seed 10101
local reps 1000
bootstrap  r(ratio), reps(‘reps’) ///
   saving(771bs2_9,replace): ///
   muratio p_dom p_for
```



Note in the histogram that the empirical distribution is quite visibly skewed; this corresponds to the percentile and bias-corrected confidence intervals being wider than that derived from the assumption of approximate normality.

Combining simulation and bootstrapping

These commands are very flexible; one may combine both techniques in a single Stata program. The example in Stata's Reference Manual [S-Z] article on `simulate` illustrates an application where a random sample is generated, and `bootstrap` is used to generate a dataset of medians calculated by bootstrap sampling from the random sample. This procedure is called within a `simulate` program which calculates the standard deviation of these bootstrap standard errors, repeated over a number of Monte Carlo draws. The `simulate` program thus generates a point and interval estimate of the median of these simulated data, where the precision of the median estimates is derived from a bootstrapped standard error.