Ado-file programming

Christopher F Baum

Boston College and DIW Berlin

NCER, Queensland University of Technology, August 2015

Ado-file programming: a primer

We begin with a discussion of ado-file programming.

A Stata *program* adds a command to Stata's language. The name of the program is the command name, and the program must be stored in a file of that same name with extension `.ado`, and placed on the *adopath*: the list of directories that Stata will search to locate programs.

A program begins with the `program define` *progname* statement, which usually includes the option `, rclass`, and a `version 14` statement.

The *progname* should not be the same as any Stata command, nor for safety's sake the same as any accessible user-written command. If `search progname` does not turn up anything, you can use that name.

Programs (and Stata commands) are either *r-class* or *e-class*. The latter group of programs are for estimation; the former do everything else. Most programs you write are likely to be r-class.

The syntax statement

The `syntax` statement will almost always be used to define the command's format. For instance, a command that accesses one or more variables in the current data set will have a `syntax varlist` statement.

With specifiers, you can specify the minimum and maximum number of variables to be accepted; whether they are numeric or string; and whether time-series operators are allowed. Each variable name in the `varlist` must refer to an existing variable. Alternatively, you could specify a `newvarlist`, the elements of which must refer to new variables.

One of the most useful features of the `syntax` statement is that you can specify `[if]` and `[in]` arguments, which allow your command to make use of standard `if` *exp* and `in` *range* syntax to limit the observations to be used.

Later in the program, you use `marksample touse` to create an indicator (dummy) temporary variable identifying those observations, and an `if ‘touse’` qualifier on statements such as `generate` and `regress`, as we will illustrate.

The `syntax` statement may also include a `using` qualifier, allowing your command to read or write external files, and a specification of command options.

Option handling

Option handling includes the ability to make options optional or required; to specify options that change a setting (such as `regress, noconstant`); that must be integer values; that must be real values; or that must be strings.

Options can specify a *numlist* (such as a list of lags to be included), a *varlist* (to implement, for instance, a `by(`*varlist*`)` option); a *namelist* (such as the name of a matrix to be created, or the name of a new variable).

Essentially, any feature that you may find in an official Stata command, you may implement with the appropriate `syntax` statement. See `[P] syntax` for full details and examples.

tempvars and tempnames

Within your own command, you do not want to reuse the names of existing variables or matrices. You may use the `tempvar` and `tempname` commands to create “safe” names for variables or matrices, respectively, which you then refer to as local macros. That is, `tempvar eps1 eps2` will create temporary variable names which you could then use as `generate double ‘eps1’ = ....`

These variables and temporary named objects will disappear when your program terminates (just as any local macros defined within the program will become undefined upon exit).

So after doing whatever computations or manipulations you need within your program, how do you return its results? You may include `display` statements in your program to print out the results, but like official Stata commands, your program will be most useful if it also returns those results for further use. Given that your program has been declared `rclass`, you use the `return` statement for that purpose.

You may return scalars, local macros, or matrices:

```
    return scalar teststat = `testval'
    return local df = `N' - `k'
    return local depvar "`varname'"
    return matrix lambda = `lambda'
```

These objects may be accessed as `r(`*name*`)` in your do-file: e.g. `r(df)` will contain the number of degrees of freedom calculated in your program.

A sample program from `help return`:

```
program define mysum, rclass
version 14
syntax varname
return local varname `varlist'
tempvar new
quietly {
  count if `varlist'!=.
  return scalar N = r(N)
  gen double `new' = sum(`varlist')
  return scalar sum = `new'[_N]
  return scalar mean = return(sum)/return(N)
}
end
```

This program can be executed as `mysum` *varname*. It prints nothing, but places three scalars and a macro in the `return list`. The values `r(mean)`, `r(sum)`, `r(N)`, and `r(varname)` can now be referred to directly.

With minor modifications, this program can be enhanced to enable the `if` *exp* and `in` *range* qualifiers. We add those optional features to the `syntax` command, use the `marksample` command to delineate the wanted observations by `touse`, and apply `` if `touse' `` qualifiers on two computational statements:

```
program define mysum2, rclass
version 14
syntax varname [if] [in]
return local varname `varlist'
tempvar new
marksample touse
quietly {
  count if `varlist' !=. & `touse'
  return scalar N = r(N)
  gen double `new' = sum(`varlist') if `touse'
  return scalar sum = `new'[_N]
  return scalar mean = return(sum) / return(N)
}
end
```

Examples of ado-file programming

As a first example of ado-file programming, we consider that the `rolling:` prefix (see `help rolling`) will allow you to save the estimated coefficients (`_b`) and standard errors (`_se`) from a moving-window regression. What if you want to compute a quantity that depends on the full variance-covariance matrix of the regression (VCE)? Those quantities cannot be saved by `rolling:`.

For instance, the regression

```
. regress y L(1/4).x
```

estimates the effects of the last four periods' values of `x` on `y`. We might naturally be interested in the sum of the lag coefficients, as it provides the *steady-state* effect of `x` on `y`.

This computation is readily performed with `lincom`. If this regression is run over a moving window, how might we access the information needed to perform this computation?

A solution is available in the form of a *wrapper program* which may then be called by `rolling:`. We write our own `r-`class program, `myregress`, which returns the quantities of interest: the estimated sum of lag coefficients and its standard error.

The program takes as arguments the *varlist* of the regression and two required options: `lagvar()`, the name of the distributed lag variable, and `nlags()`, the highest-order lag to be included in the `lincom`. We build up the appropriate expression for the `lincom` command and return its results to the calling program.

```
 . type myregress.ado
*! myregress v1.0.0  CFBaum 20aug2013
program myregress, rclass
version 13
syntax varlist(ts) [if] [in], LAGVar(string) NLAGs(integer)
regress `varlist´ `if´ `in´
local nl1 = `nlags´ - 1
forvalues i = 1/`nl1´ {
        local lv "`lv´ L`i´.`lagvar´ + "
}
local lv "`lv´  L`nlags´.`lagvar´"
lincom `lv´
return scalar sum = `r(estimate)´
return scalar se = `r(se)´
end
```

As with any program to be used under the control of a prefix operator, it is a good idea to execute the program directly to test it to ensure that its results are those you could calculate directly with `lincom`.

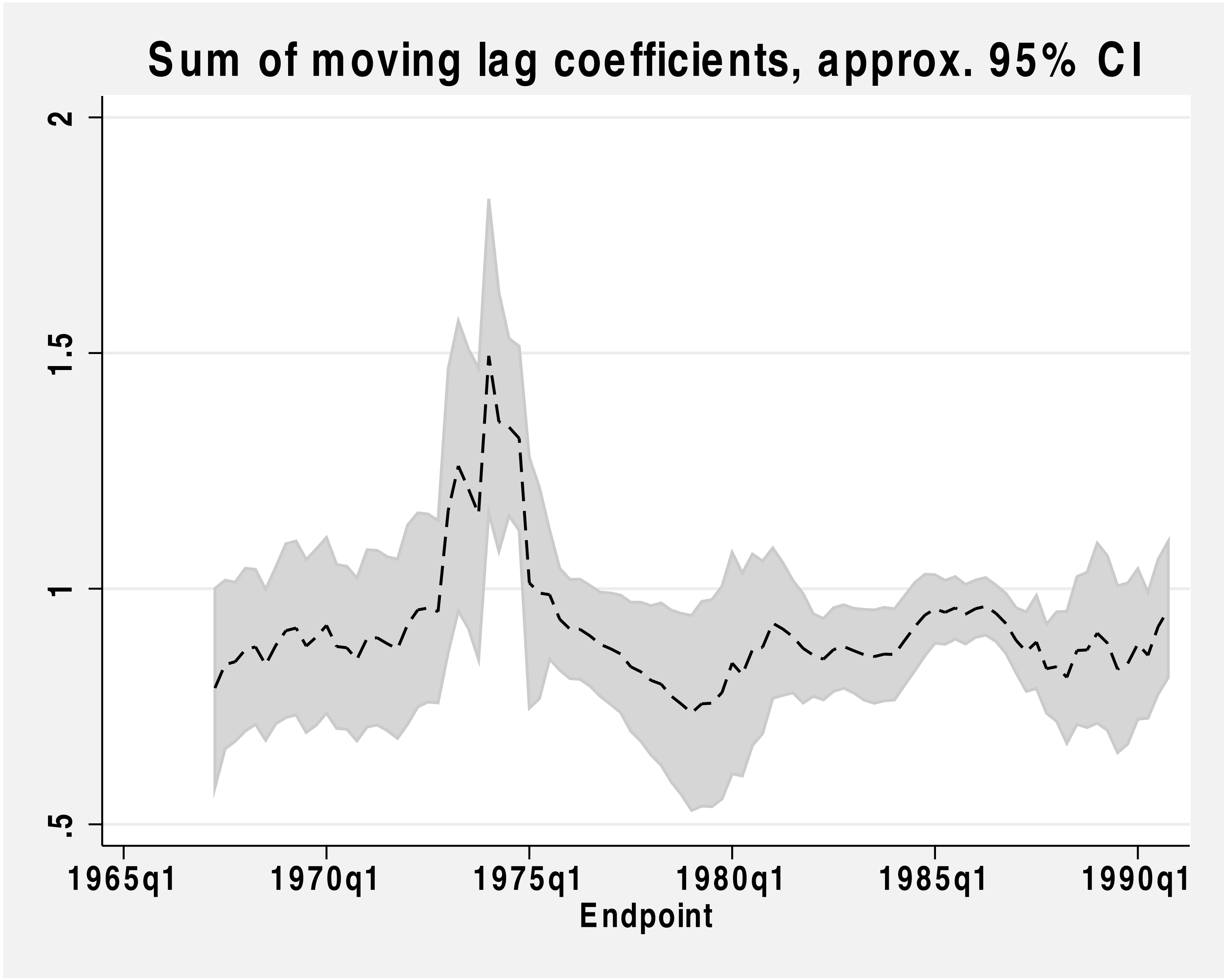
```
 . use wpi1, clear
. qui myregress wpi L(1/4).wpi t, lagvar(wpi) nlags(4)

. return list
scalars:
                 r(se) =  .0082232176260432
                r(sum) =  .9809968042273991
. lincom   L.wpi+L2.wpi+L3.wpi+L4.wpi
 ( 1)  L.wpi + L2.wpi + L3.wpi + L4.wpi = 0
------------------------------------------------------------------------------
         wpi |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         (1) |   .9809968   .0082232   119.30   0.000     .9647067    .9972869
------------------------------------------------------------------------------
```

Having validated the wrapper program by comparing its results with those from `lincom`, we can now invoke it with `rolling`:

```
 . rolling sum=r(sum) se=r(se) ,window(30) : ///
> myregress wpi L(1/4).wpi t, lagvar(wpi) nlags(4)
(running myregress on estimation sample)
Rolling replications (95)
────┼─── 1 ───┼─── 2 ───┼─── 3 ───┼─── 4 ───┼─── 5
..................................................    50
.............................................
```

We can graph the resulting series and its approximate 95% standard error bands with `twoway rarea` and `tsline`:

```
 . tsset end, quarterly
        time variable:  end, 1967q2 to 1990q4
                delta:  1 quarter
. label var end Endpoint
. g lo = sum - 1.96 * se
. g hi = sum + 1.96 * se
. twoway rarea lo hi end, color(gs12) title("Sum of moving lag coefficients, ap
> prox. 95% CI") ///
> ||  tsline sum, legend(off) scheme(s2mono)
```


Sum of moving lag coefficients, approx. 95% CI
2
1.5
1
.5
1965q1
1970q1
1975q1
1980q1
1985q1
1990q1
Endpoint

We now present a second example of ado-file programming, motivated by a question from Stan Hurn. He wanted to compute Granger causality tests (`vargranger`) following VAR estimation in a rolling-window context. To implement this, I wrote program `rgranger`:

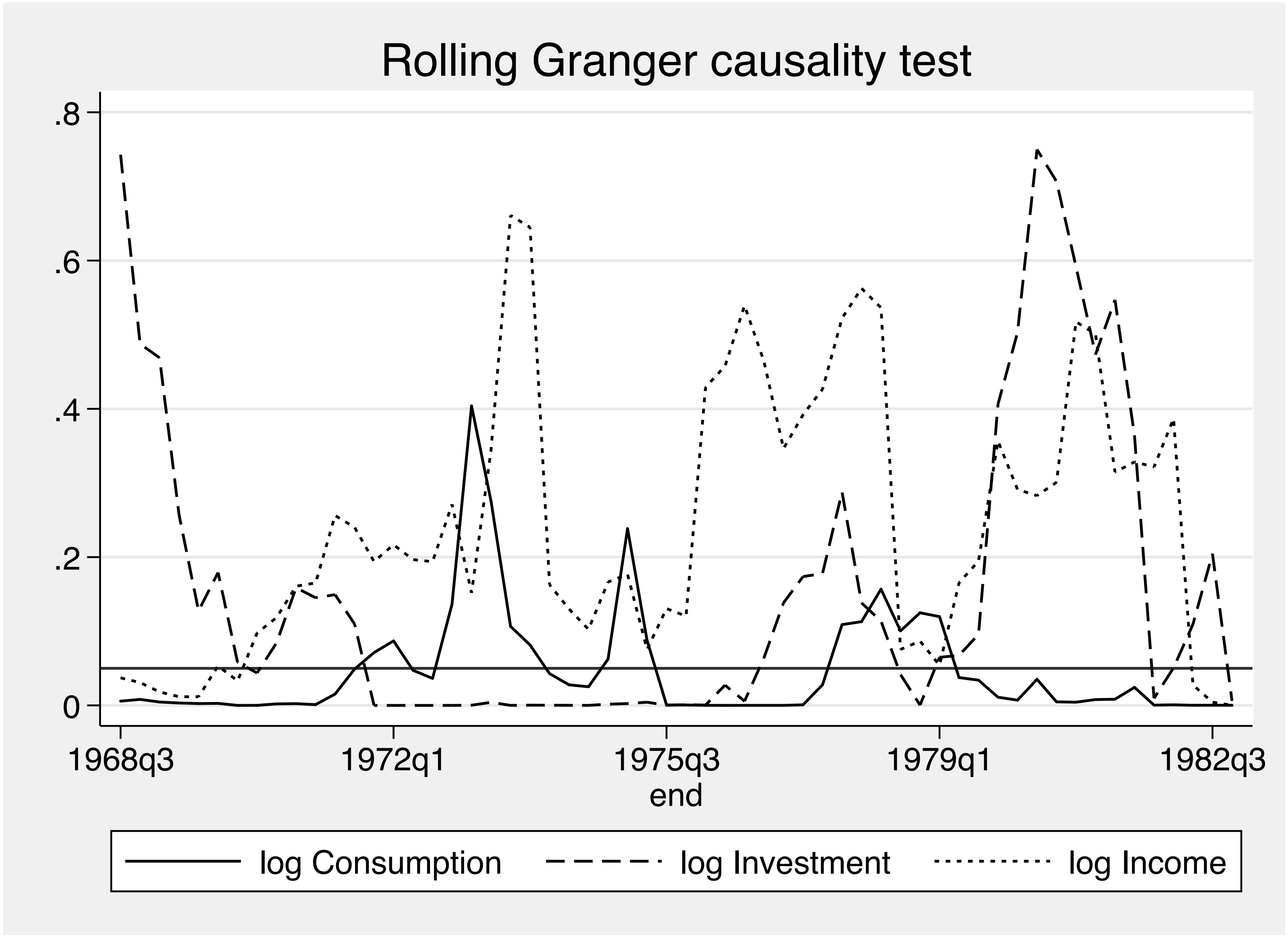
```
 . // program to do rolling granger causality test
. capt prog drop rgranger
. prog rgranger, rclass
  1. syntax varlist(min=2 numeric ts) [if] [in] [, Lags(integer 2)]
  2. var `varlist´ `if´ `in´, lags(1/`lags´)
  3. vargranger
  4. matrix stats = r(gstats)
  5. return scalar s2c   = stats[3,3]
  6. return scalar s2i   = stats[6,3]
  7. return scalar s2y   = stats[9,3]
  8. end
```

We test the program to ensure that it returns the proper results: the *p*-values of the tests for each variable in the VAR, with the null hypothesis that they can appropriately be modeled as univariate autoregression. Inspection of the returned values from `vargranger` showed that they are available in the `r(gstats)` matrix. We can then invoke `rolling` to execute `rgranger`.

```
 . // rolling window regressions
. rolling pc = r(s2c) pi = r(s2i) py = r(s2y), window(35) saving(wald,replace) : ///
> rgranger lconsumption linvestment lincome, lags(4)
(running rgranger on estimation sample)
Rolling replications (58)
────┼──── 1 ───┼──── 2 ───┼──── 3 ───┼──── 4 ───┼──── 5
..................................................    50
........
file wald.dta saved
.
. use wald
(rolling: rgranger)
. tsset end
        time variable:  end, 1968q3 to 1982q4
                delta:  1 quarter
```

With an invocation of `tsline`, we can view the results of this estimation.

```
. su

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
       start |         58        28.5    16.88688          0         57
         end |         58        62.5    16.88688         34         91
          pc |         58    .0475223    .0760982   7.23e-08   .4039747
          pi |         58     .157688    .2109603   1.02e-09   .7498347
          py |         58    .2423784     .180288   .0002369   .6609668
. lab var pc "log Consumption"

. lab var pi "log Investment"

. lab var py "log Income"

. tsline pc pi py, yline(0.05) legend(rows(1)) ylab(,angle(0)) ///
>  ti("Rolling Granger causality test") scheme(s2mono)

. gr export rgranger.pdf, replace
(file /Users/cfbaum/Dropbox/baum/Timberlake2013-2014/Slides/rgranger.pdf written in
```


Rolling Granger causality test
.8
.6
.4
.2
0
1968q3
1972q1
1975q3
1979q1
1982q3
end
log Consumption
log Investment
log Income

egen function programs

The `egen` (Extended Generate) command is open-ended, in that any Stata user can define an additional `egen` function by writing a specialized ado-file program.The name of the program (and of the file in which it resides) must start with `_g`: that is, `_gcrunch.ado` will define the `crunch()` function for `egen`.

To illustrate `egen` functions, let us create a function to generate the 90–10 percentile range of a variable. The syntax for `egen` is:

`egen` [*type*] *newvar* `=` *fcn*(`arguments`) [*if*] [*in*] [*, options*]

The `egen` command, like `generate`, may specify a data type. The `syntax` command indicates that a *newvarname* must be provided, followed by an equals sign and an *fcn*, or function, with *arguments*. `egen` functions may also handle `if` *exp* and `in` *range* qualifiers and options.

We calculate the percentile range using `summarize` with the `detail` option. On the last line of the function, we `generate` the new variable, of the appropriate type if specified, under the control of the `‘touse’` temporary indicator variable, limiting the sample as specified.

```
 . type _gpct9010.ado

*! _gpct9010 v1.0.0  CFBaum 20aug2013
        program _gpct9010
        version 13
        syntax newvarname =/exp [if] [in]
        tempvar touse
        mark `touse´ `if´ `in´
        quietly summarize `exp´ if `touse´, detail
        quietly generate `typlist´ `varlist´ = r(p90) - r(p10) if `touse´
end
```

This function works perfectly well, but it creates a new variable containing a single scalar value. That is a very profligate use of Stata's memory (especially for large `_N`) and often can be avoided by retrieving the single scalar which is conveniently stored by our `pctrange` command.

To be useful, we would like the `egen` function to be *byable*, so that it could compute the appropriate percentile range statistics for a number of groups defined in the data under the control of a `by:` prefix.

The changes to the code are relatively minor. We add an options clause to the `syntax` statement, as `egen` will pass the `by` prefix variables as a `by` *option* to our program. Rather than using `summarize`, we use `egen`'s own `pctile()` function, which is documented as allowing the `by` *prefix*, and pass the options to this function. The revised function reads:

```
. type _gpct9010.ado
*! _gpct9010 v1.0.1  CFBaum 20aug2013
        program _gpct9010
        version 13
        syntax newvarname =/exp [if] [in] [, *]
        tempvar touse p90 p10
        mark `touse´ `if´ `in´
        quietly {
                egen double `p90´ = pctile(`exp´) if `touse´, `options´ p(90)
                egen double `p10´ = pctile(`exp´) if `touse´, `options´ p(10)
                generate `typlist´ `varlist´ = `p90´ - `p10´ if `touse´
        }
end
```

These changes permit the function to produce a separate percentile range for each group of observations defined by the `by`-list.

To illustrate, we use `auto.dta`:

```
 . sysuse auto, clear
(1978 Automobile Data)
. bysort rep78 foreign: egen pctrange = pct9010(price)
```

Now, if we want to compute a summary statistic (such as the percentile range) for each observation classified in a particular subset of the sample, we may use the `pct9010()` function to do so.

Nonlinear least squares estimation

Besides the capabilities for maximum likelihood estimation of one or several equations via the `ml` suite of commands, Stata provides facilities for single-equation nonlinear least squares estimation with `nl` and the estimation of nonlinear systems of equations with `nlsur`.

Although Stata supports interactive use of these commands, I emphasize a more reproducible approach. Rather than hard-coding the syntax in a do-file, I describe the way in which they may be used with a function evaluator program, which is quite similar to the likelihood function evaluators for maximum likelihood estimation, as we shall see.

If you want to use `nl` extensively for a particular problem, it makes sense to develop a *function evaluator program*. That program is quite similar to any Stata `ado`-file or `ml` program. It must be named `nl`*func*`.ado`, where *func* is a name of your choice: e.g., `nlces.ado` for a CES function evaluator.

The stylized function evaluator program contains:

```
program nlfunc
   version 14
   syntax varlist(min=n max=n) if, at(name)
 // extract vars from varlist
 // extract params as scalars from at matrix
 // fill in dependent variable with replace
end
```

As an example, this function evaluator implements estimation of a constant elasticity of substitution (CES) production function:

```
. type nlces.ado
*! nlces v1.0.0  CFBaum 20aug2013
program nlces
   version 13
   syntax varlist(numeric min=3 max=3) if, at(name)
   args logoutput K L
   tempname b0 rho delta
   tempvar kterm lterm
   scalar `b0´ = `at´[1, 1]
   scalar `rho´ = `at´[1, 2]
   scalar `delta´ = `at´[1, 3]
   gen double `kterm´ = `delta´ * `K´^( -(`rho´ )) `if´
   gen double `lterm´ = (1 - `delta´) *`L´^( -(`rho´ )) `if´
   replace `logoutput´ = `b0´ - 1 / `rho´ * ln( `kterm´ + `lterm´ ) `if´
end
```

To use the program `nlces`, call it with the `nl` command, but only include the unique part of its name, followed by `@`:

```
. use production, clear
. nl ces @ lnoutput capital labor, parameters(b0 rho delta) ///
>  initial(b0 0 rho 1 delta 0.5)
(obs = 100)
Iteration 0:  residual SS =  29.38631
...
Iteration 7:  residual SS =  29.36581
      Source |       SS       df       MS
-------------+------------------------------         Number of obs =        100
       Model |  91.1449924     2  45.5724962         R-squared     =     0.7563
    Residual |  29.3658055    97  .302740263         Adj R-squared =     0.7513
-------------+------------------------------         Root MSE      =   .5502184
       Total |  120.510798    99  1.21728079         Res. dev.     =   161.2538

------------------------------------------------------------------------------
    lnoutput |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         /b0 |   3.792158    .099682    38.04   0.000     3.594316    3.989999
        /rho |   1.386993    .472584     2.93   0.004     .4490443    2.324941
      /delta |   .4823616   .0519791     9.28   0.000     .3791975    .5855258
------------------------------------------------------------------------------
  Parameter b0 taken as constant term in model & ANOVA table
```

You could restrict analysis to a subsample with the *if exp* qualifier:

```
nl ces @ lnQ cap lab if industry==33, ...
```

You can also perform post-estimation commands, such as `nlcom`, to derive point and interval estimates of nonlinear combinations of the estimated parameters. In this case, we want to compute the elasticity of substitution, σ:

```
. nlcom (sigma: 1 / ( 1 + [rho]_b[_cons] ))
       sigma:  1 / ( 1 + [rho]_b[_cons] )

------------------------------------------------------------------------------
    lnoutput |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       sigma |   .4189372   .0829424     5.05   0.000     .2543194     .583555
------------------------------------------------------------------------------
```

The `nlsur` command estimates systems of *seemingly unrelated* nonlinear equations, just as `sureg` estimates systems of seemingly unrelated linear equations. In that context, `nlsur` cannot be used to estimate a system of simultaneous nonlinear equations. The `gmm` command, as we will discuss, could be used for that purpose, as could Stata's maximum likelihood commands (`ml`).

Maximum likelihood estimation

For many limited dependent models, Stata contains commands with “canned” likelihood functions which are as easy to use as `regress`. However, you may have to write your own likelihood evaluation routine if you are trying to solve a non-standard maximum likelihood estimation problem.

A key resource is the book *Maximum Likelihood Estimation in Stata*, Gould, Pitblado and Poi, Stata Press: 4th ed., 2010. A good deal of this presentation is adapted from that excellent treatment of the subject, which I recommend that you buy if you are going to work with MLE in Stata.

To perform maximum likelihood estimation (MLE) in Stata, you can write a short Stata program defining the likelihood function for your problem. In most cases, that program can be quite general and may be applied to a number of different model specifications without the need for modifying the program.

Let’s consider the simplest use of MLE: a model that estimates a binomial probit equation, as implemented in Stata by the `probit` command. We code our probit ML program as:

```
program myprobit_lf
  version 14
  args lnf xb
  quietly replace `lnf´ = ln(normal( `xb´ )) ///
      if $ML_y1 == 1
  quietly replace `lnf´ = ln(normal( -`xb´ )) ///
      if $ML_y1 == 0
end
```

This program is suitable for ML estimation in the *linear form* or `lf` context. The local macro `lnf` contains the contribution to log-likelihood of each observation in the defined sample. As is generally the case with Stata's `generate` and `replace`, it is not necessary nor desirable to loop over the observations. In the linear form context, the program need not sum up the log-likelihood.

Several programming constructs show up in this example. The `args` statement defines the program's *arguments*: `lnf`, the variable that will contain the value of log-likelihood for each observation, and `xb`, the linear form: a single variable that is the product of the "X matrix" and the current vector *b*. The arguments are local macros within the program.

The program replaces the values of `lnf` with the appropriate log-likelihood values, conditional on the value of `$ML_y1`, which is the first dependent variable or "y"-variable. Thus, the program may be applied to any 0–1 variable as a function of any set of X variables without modification.

Given the program—stored in the file `myprobit_lf.ado` on the `ADOPATH`—how do we execute it?

```
sysuse auto, clear
gen gpm = 1/mpg
ml model lf myprobit_lf ///
    (foreign = price gpm displacement)
ml maximize
```

The `ml model` statement defines the context to be the linear form (`lf`), the likelihood evaluator to be `myprobit_lf`, and then specifies the model. The binary variable `foreign` is to be explained by the factors `price, gpm, displacement`, by default including a constant term in the relationship. The `ml model` command only defines the model: it does not estimate it. That is performed with the `ml maximize` command.

You can verify that this routine duplicates the results of applying `probit` to the same model. Note that our ML program produces estimation results in the same format as an official Stata command. It also stores its results in the `ereturn list`, so that postestimation commands such as `test` and `lincom` are available.

Of course, we need not write our own binomial probit. To understand how we might apply Stata's ML commands to a likelihood function of our own, we must establish some notation, and explain what the linear form context implies.

The log-likelihood function can be written as a function of variables and parameters:

$$\begin{aligned} \ell &= \ln L\{(\theta_{1j},\theta_{2j},\ldots,\theta_{Ej};y_{1j},y_{2j},\ldots,y_{Dj}),\; j=1,N\} \\ \theta_{ij} &= \mathbf{x}_{ij}\beta_i = \beta_{i0}+x_{ij1}\beta_{i1}+\cdots+x_{ijk}\beta_{ik} \end{aligned}$$

or in terms of the whole sample:

$$\begin{aligned} \ell &= \ln L(\theta_1,\theta_2,\ldots,\theta_E;\mathbf{y}_1,\mathbf{y}_2,\ldots,\mathbf{y}_D) \\ \theta_i &= \mathbf{X}_i\beta_i \end{aligned}$$

where we have D dependent variables, E equations (indexed by i) and the data matrix X_i for the i^{th} equation, containing N observations indexed by j.

In the special case where the log-likelihood contribution can be calculated separately for each observation and the sum of those contributions is the overall log-likelihood, the model is said to meet the *linear form restrictions*:

$$\begin{aligned} \ln \ell_j &= \ln \ell(\theta_{1j}, \theta_{2j}, \ldots, \theta_{Ej}; y_{1j}, y_{2j}, \ldots, y_{Dj}) \\ \ell &= \sum_{j=1}^{N} \ln \ell_j \end{aligned}$$

which greatly simplify the task of specifying the model. Nevertheless, when the linear form restrictions are not met, Stata provides three other contexts in which the full likelihood function (and possibly its derivatives) can be specified.

One of the more difficult concepts in Stata's MLE approach is the notion of ML *equations*. In the example above, we only specified a single equation:

```
    (foreign = price gpm displacement)
```

which served to identify the dependent variable `$ML_y1` to Stata) and the *X* variables in our binomial probit model.

Let's consider how we can implement estimation of a linear regression model via ML. In regression we seek to estimate not only the coefficient vector b but the error variance σ^2. The log-likelihood function for the linear regression model with normally distributed errors is:

$$\ln L = \sum_{j=1}^{N} [\ln \phi\{(y_j - x_j \beta)/\sigma\} - \ln \sigma]$$

with parameters β, σ to be estimated.

Writing the conditional mean of y for the j^{th} observation as μ_j,

$$\mu_j = E(y_j) = x_j \beta$$

we can rewrite the log-likelihood function as

$$\begin{aligned}
\theta_{1j} &= \mu_j = x_{1j}\beta_1 \\
\theta_{2j} &= \sigma_j = x_{2j}\beta_2 \\
\ln L &= \sum_{j=1}^{N} [\ln \phi\{(y_j - \theta_{1j})/\theta_{2j}\} - \ln \theta_{2j}]
\end{aligned}$$

This may seem like a lot of unneeded notation, but it makes clear the flexibility of the approach. By defining the linear regression problem as a two-equation ML problem, we can readily specify equations for both β and σ. In OLS regression with homoskedastic errors, we do not need to specify an equation for σ, a constant parameter, but this approach allows us to readily relax that assumption and consider an equation in which σ itself is modeled as varying over the data.

Given a program `mynormal_lf` to evaluate the likelihood of each observation—the individual terms within the summation—we can specify the model to be estimated with

```
ml model lf mynormal_lf  ///
    (y = equation for y) (equation for sigma)
```

In the homoskedastic linear regression case, this might look like

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) ()
```

where the trailing set of `()` merely indicate that nothing but a constant appears in the “equation” for σ. This `ml model` specification indicates that a regression of `mpg` on `weight` and `displacement` is to be fit, by default with a constant term.

We could also use the notation

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) /sigma
```

where there is a constant parameter to be estimated.

But what does the program `mynormal_lf` contain?

```
program mynormal_lf
  version 14
  args lnf mu sigma
  quietly replace `lnf´ = ///
         ln(normalden( $ML_y1,  `mu´, `sigma´ ))
end
```

We can use Stata's `normalden(x,m,s)` function in this context. The three-parameter form of this Stata function returns the Normal[`m`, `s`] density associated with `x` divided by `s`. Parameter `m`, μ_j in the earlier notation, is the conditional mean, computed as $\mathbf{X}\beta$, while `s`, or σ, is the standard deviation. By specifying an "equation" for σ of `()`, we indicate that a single, constant parameter is to be estimated in that equation.

What if we wanted to estimate a heteroskedastic regression model, in which σ_j is considered a linear function of some variable(s)? We can use the same likelihood evaluator, but specify a non-trivial equation for σ:

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) (price)
```

This would model $\sigma_j = \beta_4\, \mathit{price} + \beta_5$.

If we wanted σ to be proportional to `price`, we could use

```
ml model lf mynormal_lf ///
     (mu: mpg = weight displacement) ///
     (sigma: price, nocons)
```

which also labels the equations as `mu, sigma` rather than the default `eq1, eq2`.

A better approach to this likelihood evaluator program involves modeling σ in log space, allowing it to take on all values on the real line. The likelihood evaluator program becomes

```
program mynormal_lf2
  version 14
  args lnf mu lnsigma
  quietly replace `lnf´ = ///
   ln(normalden( $ML_y1,  `mu´, exp(`lnsigma´ )))
end
```

It may be invoked by

```
    ml model lf mynormal_lf2 ///
         (mpg = weight displacement) /lnsigma, ///
         diparm(lnsigma, exp label("sigma"))
```

Where the `diparm( )` option presents the estimate of σ.

We have illustrated the simplest likelihood evaluator method: the linear form (`lf`) context. It should be used whenever possible, as it is not only easier to code (and less likely to code incorrectly) but more accurate. There are also variants of this method, `lf0, lf1, lf2`. The latter two require you to code the first or first and second derivatives of the log-likelihood function, respectively.

When method `lf` cannot be used—when the linear form restrictions are not met—you may use methods `d0, d1`, or `d2`.

Method `d0`, like `lf`, requires only that you code the log-likelihood function, but in its entirety rather than for a single observation. It is the least accurate and slowest ML method, but the easiest to use when method `lf` is not available.

Method `d1` requires that you code both the log-likelihood function and the vector of first derivatives, or gradients. It is more difficult than `d0`, as those derivatives must be derived analytically and coded, but is more accurate and faster than `d0` (but less accurate and slower than `lf`.

Method `d2` requires that you code the log-likelihood function, the vector of first derivatives and the matrix of second partial derivatives. It is the most difficult method to use, as those derivatives must be derived analytically and coded, but it is the most accurate and fastest method available. Unless you plan to use a ML program very extensively, you probably do not want to go to the trouble of writing a method `d2` likelihood evaluator.

Many of Stata's standard estimation features are readily available when writing ML programs.

You can estimate over a subsample with the standard `if` *exp* or `in` *range* qualifiers on the `ml model` statement.

The default variance-covariance matrix (`vce(oim)`) estimator is based on the inverse of the estimated Hessian, or information matrix, at convergence. That matrix is available when using the default Newton–Raphson optimization method, which relies upon estimated second derivatives of the log-likelihood function.

If any of the quasi-Newton methods are used, you can select the Outer Product of Gradients (`vce(opg)`) estimator of the variance-covariance matrix, which does not rely on a calculated Hessian. This may be especially helpful if you have a lengthy parameter vector. You can specify that the covariance matrix is based on the information matrix (`vce(oim)`) even with the quasi-Newton methods.

The standard heteroskedasticity-robust `vce` estimate is available by selecting the `vce(robust)` option (unless using method `d0`). Likewise, the cluster-robust covariance matrix may be selected, as in standard estimation, with `cluster(`*varname*`)`.

You can estimate a model subject to linear constraints using the standard `constraint` command and the `constraints( )` option on the `ml model` command.

You can specify weights on the `ml model` command, using the weights syntax applicable to any estimation command. If you specify `pweights` (probability weights) robust estimates of the variance-covariance matrix are implied.

You can use the `svy` option to indicate that the data have been `svyset`: that is, derived from a complex survey design.

A method `lf` likelihood evaluator program looks like:

```
program myprog
   version 14
   args lnf theta1 theta2 ...
   tempvar tmp1 tmp2 ...
   qui gen double `tmp1´ = ...
   qui replace `lnf´ = ...
end
```

`ml` places the name of each dependent variable specified in `ml model` in a global macro: `$ML_y1, $ML_y2`, and so on.

`ml` supplies a variable for each equation specified in `ml model` as `theta1`, `theta2`, etc. Those variables contain linear combinations of the explanatory variables and current coefficients of their respective equations. These variables must not be modified within the program.

If you need to compute any intermediate results within the program, use `tempvar`s, and be sure to declare them as `double`. If scalars are needed, define them with a `tempname` statement. Computation of components of the LLF is often convenient when it is a complicated expression.

Final results are saved in `` `lnf' ``: a double-precision variable that will contain the contributions to likelihood of each observation.

The linear form restrictions require that the individual observations in the dataset correspond to independent pieces of the log-likelihood function. They will be met for many ML problems, but are violated for problems involving panel data, fixed-effect logit models, and the Cox proportional hazards model.

Just as linear regression may be applied to many nonlinear models (e.g., the Cobb–Douglas production function), Stata's *linear form restrictions* do not hinder our estimation of a nonlinear model. We merely add equations to define components of the model. If we want to estimate

$$y_j = \beta_1 x_{1j} + \beta_2 x_{2j} + \beta_3 x_{3j}^{\beta_4} + \beta_5 + \epsilon_j$$

with $\epsilon \sim N(0, \sigma^2)$, we can express the log-likelihood as

$$\begin{aligned}
\ln \ell_j &= \ln \phi\{(y_j - \theta_{1j} - \theta_{2j} x_{3j}^{\theta_{3j}})/\theta_{4j}\} - \ln \theta_{4j} \\
\theta_{1j} &= \beta_1 x_{1j} + \beta_2 x_{2j} + \beta_5 \\
\theta_{2j} &= \beta_3 \\
\theta_{3j} &= \beta_4 \\
\theta_{4j} &= \sigma
\end{aligned}$$

The likelihood evaluator for this problem then becomes

```
program mynonlin_lf
  version 11
  args lnf theta1 theta2 theta3 sigma
  quietly replace `lnf´ = ln(normalden( $ML_y1, ///
       `theta1´+`theta2´*$X3^`theta3´, `sigma´ ))
end
```

This program evaluates the LLF using a *global macro*, `X3`, which must be defined to identify the Stata variable that is to play the role of x_3.

By making this reference a global macro, we avoid hard-coding the variable name, so that the same model can be fit on different data without altering the program.

We could invoke the program with

```
global X3 bp0
ml model lf mynonlin_lf (bp = age sex) ///
   /beta3 /beta4 /sigma
ml maximize
```

Thus, we can readily handle a nonlinear regression model in this context of the linear form restrictions, redefining the ML problem as one of four equations.

If we need to set starting values, we can do so with the `ml init` command. The `ml check` command is very useful in testing the likelihood evaluator program and ensuring that it is working properly. The `ml search` command is recommended to find appropriate starting values. The `ml query` command can be used to evaluate the progress of ML if there are difficulties achieving convergence.

Maximum likelihood estimation of distributions' parameters

A natural application for maximum likelihood estimation arises in evaluating the parameters of various statistical distributions. The numerical examples are adapted with thanks from Martin, Hurn, Harris' Cambridge University Press book.

For instance, the probability density function (PDF) of the exponential distribution, which describes the time between events in a Poisson process, is characterized by a single parameter λ:

$$f(x; \lambda) = \lambda e^{-\lambda x}, \ x > 0.$$

The linear-form evaluator for this distribution expresses the log-likelihood contribution of a single observation on the dependent variable,

$$\log L_t(\theta) = \frac{1}{T}(\log \theta - \theta y_t)$$

which is then programmed as

```
 . type myexp_lf.ado

*! myexp_lf v1.0.0  CFBaum 04mar2014
program myexp_lf
  version 13.1
  args lnfj theta
  quietly replace `lnfj´ = 1/$ML_N * ( log(`theta´) - `theta´ * $ML_y1 )
end
```

We illustrate the use of `myexp_lf` on a few observations:

```
 . prog drop _all
. clear
. input y
                 y
  1. 2.1
  2. 2.2
  3. 3.1
  4. 1.6
  5. 2.5
  6. 0.5
  7. end
. ml model lf myexp_lf (y = )
. ml maximize, nolog
initial:       log likelihood =     -<inf>  (could not be evaluated)
feasible:      log likelihood = -1.6931472
rescale:       log likelihood = -1.6931472
                                                  Number of obs   =          6
                                                  Wald chi2(0)    =          .
Log likelihood = -1.6931472                       Prob > chi2     =          .

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |         .5          .5     1.00   0.317     -.479982    1.479982
------------------------------------------------------------------------------

.
```

The Cauchy distribution is the distribution of a random variable that is the ratio of two independent standard normal variables. The PDF of the Cauchy distribution with scale=1 involves a single parameter, θ, which identifies the peak of the distribution, which is both the median and modal value:

$$f(x; \theta, 1) = \frac{1}{\pi} \left[\frac{1}{1 + (x - \theta)^2} \right]$$

Interestingly, both the mean and variance (and indeed all moments) of the Cauchy distribution are undefined.

The linear-form evaluator for this distribution expresses the log-likelihood contribution of a single observation on the dependent variable:

```
 . type mycauchy_lf.ado
*! mycauchy_lf v1.0.0  CFBaum 04mar2014
program mycauchy_lf
  version 13.1
  args lnfj theta
  quietly replace `lnfj´ = -log(_pi)/$ML_N - 1/$ML_N * log(1 + ($ML_y1 - `theta
> ´)^2 )
end
```

We illustrate the use of `mycauchy_lf` on a few observations:

```
 . clear
. input y
                y
  1. 2
  2. 5
  3. -2
  4. 3
  5. 3
  6. end
. ml model lf mycauchy_lf (y= )
. ml maximize, nolog
                                                  Number of obs   =          5
                                                  Wald chi2(0)    =          .
Log likelihood = -2.2476326                       Prob > chi2     =          .

             y        Coef.    Std. Err.       z     P>|z|      [95% Conf. Interval]

         _cons     2.841449    1.177358      2.41    0.016      .5338697     5.149029
```

In financial econometrics, a popular model of interest rates' evolution is that proposed by Vasicek (1977). In this discrete-time framework, changes in the short-term interest rate r_t are modeled as

$$\Delta r_t = \alpha + \beta r_{t-1} + u_t$$

with parameters α, β, σ^2, with σ^2 being the variance of the normally distributed errors u_t.

The PDF of the transitional distribution can be written as:

$$f(r_t | r_{t-1}; \alpha, \delta, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(r_t - \alpha - \delta r_{t-1})^2}{2\sigma^2} \right]$$

where $\delta = 1 + \beta$.

The log-likelihood contribution of a single observation may be expressed as

$$\log L_t(\alpha, \delta, \sigma^2) = -\frac{1}{2T} \log 2\pi - \frac{1}{2T} \log \sigma^2 - \frac{1}{2\sigma^2(T-1)}(r_t - \alpha - \delta r_{t-1})^2, \; t = 2 \ldots T$$

Which may be directly programmed in the linear-form evaluator:

```
 . type myvasicek2_lf.ado
*! myvasicek2_lf v1.0.0  CFBaum 04mar2014
program myvasicek2_lf
  version 13.1
  args lnfj rho sigma2
  quietly replace `lnfj´ = -0.5 * log(2 * _pi) / $ML_N - 0.5 / ($ML_N-1) * log(
> `sigma2´)  ///
  -1 / (2* `sigma2´ * ($ML_N-1)) * ($ML_y1  - `rho´)^2
end
```

When we estimate the model using daily Eurodollar interest rate data, the `alpha` equation contains a slope and intercept which correspond to the parameters δ and α, respectively.

```
 . use eurodata, clear
. generate r = 100 * col4
. generate t = _n
. tsset t
        time variable:  t, 1 to 5505
                delta:  1 unit
. ml model lf myvasicek2_lf (alpha:r = L.r) /sigma2
. ml maximize, nolog
initial:       log likelihood =     -<inf>  (could not be evaluated)
feasible:      log likelihood = -75.295336
rescale:       log likelihood = -2.7731787
rescale eq:    log likelihood = -2.7126501
                                                  Number of obs   =       5504
                                                  Wald chi2(1)    =      77.40
Log likelihood = -.51650389                       Prob > chi2     =     0.0000

------------------------------------------------------------------------------
           r |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
alpha        |
           r |
         L1. |    .993636   .1129437     8.80   0.000     .7722705    1.215002
             |
       _cons |   .0528733    1.02789     0.05   0.959    -1.961754    2.067501
-------------+----------------------------------------------------------------
sigma2       |
       _cons |     .16452   .2326453     0.71   0.479    -.2914564    .6204964
------------------------------------------------------------------------------
```

As Martin et al. show, the estimates of the transitional distribution's parameters may be used to estimate the mean and variance of the stationary distribution:

$$\mu_s = -\frac{\hat{\alpha}}{\hat{\beta}}, \quad \sigma_s^2 = -\frac{\hat{\sigma}^2}{\hat{\beta}(2+\hat{\beta})}$$

We use `nlcom` to compute point and interval estimates of the stationary distribution's parameters.

```
 . di as err _n "Beta"
Beta
. lincom [alpha]L.r - 1
 ( 1)  [alpha]L.r = 1

             r |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
         (1) |   -.006364   .1129437    -0.06   0.955    -.2277295    .2150016

. di as err _n "Mu_s"
Mu_s
. nlcom -1 * [alpha]_cons / ([alpha]L.r - 1)
       _nl_1:  -1 * [alpha]_cons / ([alpha]L.r - 1)

             r |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
       _nl_1 |   8.308242   63.73711     0.13   0.896    -116.6142    133.2307

. di as err _n "Sigma^2_s"
Sigma^2_s
. nlcom -1 * [sigma2]_cons / (([alpha]L.r - 1) * (2 + ([alpha]L.r - 1)))
       _nl_1:  -1 * [sigma2]_cons / (([alpha]L.r - 1) * (2 + ([alpha]L.r - 1)))

             r |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
       _nl_1 |   12.96719    230.131     0.06   0.955    -438.0812    464.0156
```

As a final example, we consider estimation of the Cox–Ingersoll–Ross (CIR, 1985) model of the term structure of interest rates. In their continuous time framework, the interest rate *r* follows the process

$$dr = \alpha(\mu - r)dt + \sigma\sqrt{r}\, dB,\ \ dB \sim N(0, dt)$$

with model parameters α, μ, σ. The interest rate is hypothesized to be mean-reverting to μ at rate α.

CIR show that the stationary distribution of the interest rate is a gamma distribution:

$$f(r_t; \nu, \omega) = \frac{\omega^\nu}{\Gamma(\nu)} r_t^{\nu-1} \exp(-\omega r_t)$$

with unknown parameters ν, ω. $\Gamma(\cdot)$ is the Gamma (generalized factorial) function. The log-likelihood contribution of a single observation may be expressed as

$$\log L(\nu, \omega) = (\nu - 1) \log(r_t) + \nu \log(\omega) - \log \Gamma(\nu) - \omega r_t$$

This may be expressed in the linear-form evaluator as

```
 . type mygamma_lf.ado
*! mygamma_lf v1.0.0  CFBaum 05mar2014
program mygamma_lf
  version 13.1
  args lnfj nu omega
  quietly replace `lnfj´ = (`nu´- 1) * log($ML_y1) + ///
  `nu´ * log(`omega´)  - lngamma(`nu´) - `omega´ * $ML_y1
end
```

We estimate the model from the daily Eurodollar interest rate data, expressed in decimal form. The parameter estimates can then be transformed into an estimate of μ using `nlcom`.

```
 . use eurodata, clear
. rename col4 r
. generate t = _n
. tsset t
        time variable:  t, 1 to 5505
                delta:  1 unit
. ml model lf mygamma_lf (nu:r= ) /omega
. ml maximize, nolog
initial:       log likelihood =     -<inf>  (could not be evaluated)
feasible:      log likelihood =  1791.7893
rescale:       log likelihood =  1791.7893
rescale eq:    log likelihood =  5970.8375
                                                  Number of obs   =       5505
                                                  Wald chi2(0)    =          .
Log likelihood =  10957.375                       Prob > chi2     =          .

------------------------------------------------------------------------------
           r |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
nu           |
       _cons |   5.655586   .1047737    53.98   0.000     5.450233    5.860938
-------------+----------------------------------------------------------------
omega        |
       _cons |   67.63355   1.310279    51.62   0.000     65.06545    70.20165
------------------------------------------------------------------------------
```

```
. di as err _n "Estimated Mu"
Estimated Mu
. nlcom [nu]_cons / ([omega]_cons)
       _nl_1:  [nu]_cons / ([omega]_cons)
--------------------------------------------------------------------------------
             r |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
---------------+----------------------------------------------------------------
         _nl_1 |    .083621   .0004739   176.45   0.000     .0826922    .0845499
--------------------------------------------------------------------------------
```

The estimated μ corresponds to an interest rate anchor of 8.36%.

An ado-file for MLE

We have presented use of Stata's ML facilities in the context of a do-file, in which commands `ml model`, `ml search` and `ml maximize` are used to perform estimation. If you are going to use a particular maximum likelihood evaluator extensively, it may be useful to write an ado-file version of the routine, using ML's noninteractive mode.

An ado-file version of a ML routine can implement all the features of any Stata estimation command, and itself becomes a Stata command with a *varlist* and options, as well as the ability to replay results.

To produce a new estimation command, you must write two ado-files: `newcmd.ado` and `newcmd_ll.ado`, the likelihood function evaluator. The latter ado-file is unchanged from our previous discussion.

The `newcmd.ado` file will characteristically contain references to two programs, internal to the routine: `Replay` and `Estimate`. Thus, the skeleton of the command ado-file looks like:

```
program newcmd
  version 14
  if replay()
    if ("`e(cmd)´" != "newcmd") error 301
    Replay `0´

 else Estimate `0´
end
```

If `newcmd` is invoked by only its name, it will execute the `Replay` subprogram if the last estimation command was `newcmd`. Otherwise, it wlll execute the `Estimate` subprogram, passing the remainder of the command line in local macro `0` to `Estimate`.

The `Replay` subprogram is quite simple:

```
program Replay
   syntax [, Level(cilevel), other-display-options ]
   ml display, level(`level´) other-display-options
end
```

The `Estimate` subprogram contains:

```
program Estimate, eclass sortpreserve
   syntax varlist [if] [in]  [,  vce(passthru) ///
   Level(cilevel) other-estimation-options ///
   other-display-options ]

   marksample touse

   ml model method newcmd_ll  if `touse´, ///
   `vce´ other-estimation-options  maximize

   ereturn local cmd "newcmd"
   Replay, `level´ other-display-options
end
```

The `Estimate` subprogram is declared as `eclass` so that it can return estimation results. The `syntax` statement will handle the use of `if` or `in` clauses, and pass through any other options provided on the command line. The `marksample touse` ensures that only observations satisfying the `if` or `in` clauses are used.

On the `ml model` statement, `method` would be hard-coded as `lf, d0, d1, d2`. The `if ‘touse’` clause specifies the proper estimation sample. The `maximize` option is crucial: it is this option that signals to Stata that ML is being used in its noninteractive mode. With `maximize`, the `ml model` statement not only defines the model but triggers its estimation.

The non-interactive estimation does not display its results, so `ereturn` and a call to the `Replay` subprogram are used to produce output.

Additional estimation options can readily be added to this skeleton. For instance, cluster-robust estimates can be included by adding a `CLuster(varname)` option to the syntax statement, with appropriate modifications to `ml model`. In the linear regression example, we could provide a set of variables to model heteroskedasticity in a `HEtero(varlist)` option, which would then define the second equation to be estimated by `ml model`.

The `mlopts` utility command can be used to parse options provided on the command line and pass any related to the ML estimation process to `ml model`. For instance, a `HEtero(varlist)` option is to be handled by the program, while an `iterate(#)` option should be passed to the optimizer.

A worked example

The likelihood function evaluator for a linear regression model, with the σ parameter constrained to be positive:

```
 . type mynormal_lf.ado

*! mynormal_lf v1.0.1  CFBaum 16feb2014
program mynormal_lf
  version 13.1
  args lnf mu lnsigma
  quietly replace `lnf´ = ln( normalden( $ML_y1,  `mu´, exp(`lnsigma´) ) )
end
```

Because we still want to take advantage of the three-argument form of the `normalden()` function, we pass the parameter `exp('lnsigma')` to the function, which expects to receive σ itself as its third argument. Because the optimization takes place with respect to parameter `lnsigma`, difficulties with the zero boundary are avoided.

The ado-file calling this evaluator is:

```
 . type mynormal.ado

*! mynormal v1.0.1  CFBaum 16feb2014
program mynormal
        version 13.1
        if replay()  {
                if ("`e(cmd)´" != "mynormal") error 301
                Replay `0´
        }
        else Estimate `0´
end

program Replay
        syntax [, Level(cilevel) ]
        ml display, level(`level´)
end

program Estimate, eclass sortpreserve
        syntax varlist [if] [in]  [,  vce(passthru) Level(cilevel) * ]
        mlopts mlopts, `options´
        gettoken lhs rhs: varlist
        marksample touse
        local diparm diparm(lnsigma, exp label("sigma"))
        ml model lf  mynormal_lf (mu: `lhs´ = `rhs´) /lnsigma  ///
        if `touse´, `vce´ `mlopts´ maximize `diparm´
        ereturn local cmd "mynormal"
        ereturn scalar k_aux = 1
        Replay, level(`level´)
end
```

When the `Replay` routine is invoked, the parameter displayed will be `lnsigma` rather than `sigma`. We can deal with this issue by using the `diparm()` option of `ml model`. Now, when the model is estimated, the ancillary parameter `lnsigma` is displayed and transformed into the original parameter space as `sigma`.

```
. sysuse auto, clear
(1978 Automobile Data)

. mynormal price mpg weight turn
initial:       log likelihood =     -<inf>  (could not be evaluated)
feasible:      log likelihood = -811.54531
rescale:       log likelihood = -811.54531
rescale eq:    log likelihood = -808.73926
                                                  Number of obs     =         74
                                                  Wald chi2(3)      =      46.26
Log likelihood = -677.74638                       Prob > chi2       =     0.0000

------------------------------------------------------------------------------
       price |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         mpg |  -72.86501   79.06769    -0.92   0.357    -227.8348    82.10481
      weight |   3.524339   .7947479     4.43   0.000     1.966661    5.082016
        turn |  -395.1902   119.2837    -3.31   0.001    -628.9819   -161.3985
       _cons |   12744.24   4629.664     2.75   0.006      3670.27    21818.22
-------------+----------------------------------------------------------------
    /lnsigma |   7.739796   .0821995    94.16   0.000     7.578688    7.900904
-------------+----------------------------------------------------------------
       sigma |   2298.004   188.8948                      1956.062    2699.723
------------------------------------------------------------------------------
```

We illustrated how the σ parameter in linear regression could be constrained to be positive. What if we want to estimate a model subject to an inequality constraint on one of the parameters? Imagine that we believe that the first slope parameter in a particular regression model must be negative.

To apply this constraint, we now break out the first slope coefficient from the linear combination and give it its own "equation". To refer to the variable `mpg` within the likelihood function evaluator, we must use a global macro. To impose the constraint of negativity on the `mpg` coefficient, we only need specify that the mean equation (for `mu`) is adjusted by subtracting the exponential of the parameter `a`.

```
 . type mynormal_lf_c1.ado

*! mynormal_lf_c1 v1.0.0  CFBaum 16feb2014
program mynormal_lf_c1
        version 13.1
        args lnfj a xb lnsigma
        tempvar mu
        quietly generate double `mu´ = `xb´ - exp(`a´)* $x1
        quietly replace `lnfj´ = ln(normalden($ML_y1, `mu´, exp(`lnsigma´)))
end
```

We can now estimate the model subject to the inequality constraint imposed by the `exp()` function. Because maximizing likelihood functions with inequality constraints can be numerically difficult, we supply starting values to `ml model` from the unconstrained regression.

```
 . global x1 mpg
. qui regress price mpg weight turn
. matrix b0 = e(b), ln(e(rmse))
. matrix b0[1,1] = ln(-1*b0[1,1])
. ml model lf mynormal_lf_c1 (a:) (mu: price = weight turn) /lnsigma, ///
> maximize nolog diparm(lnsigma, exp label("sigma")) from(b0)
initial:       log likelihood =     -<inf>  (could not be evaluated)
feasible:      log likelihood =  -2107.728
rescale:       log likelihood = -1087.8821
rescale eq:    log likelihood =  -897.2948
. ml display
                                                  Number of obs   =         74
                                                  Wald chi2(0)    =          .
Log likelihood = -677.74638                       Prob > chi2     =          .

------------------------------------------------------------------------------
       price |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
a            |
       _cons |   4.288543   1.085203     3.95   0.000     2.161584    6.415501
-------------+----------------------------------------------------------------
mu           |
      weight |   3.524365   .7947492     4.43   0.000     1.966685    5.082044
        turn |  -395.1895   119.2837    -3.31   0.001    -628.9813   -161.3978
       _cons |   12744.04   4629.679     2.75   0.006     3670.034    21818.04
-------------+----------------------------------------------------------------
lnsigma      |
       _cons |   7.739796   .0821995    94.16   0.000     7.578688    7.900904
-------------+----------------------------------------------------------------
```

We use the `nlcom` command to transform the estimated coefficient back to its original space:

```
. nlcom -exp([a]_cons)
       _nl_1:  -exp([a]_cons)

------------------------------------------------------------------------------
       price |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _nl_1 |  -72.86023   79.06812    -0.92   0.357    -227.8309    82.11045
------------------------------------------------------------------------------
```

We have estimated ln(β_{mpg}); `nlcom` back-transforms the estimated coefficient to β_{mpg} in point and interval form.

Variations on this technique are used throughout Stata's maximum likelihood estimation commands to ensure that coefficients take on appropriate values. For instance, the bivariate probit command estimates a coefficient ρ, the correlation between two error processes. It must lie within $(-1, +1)$.

Stata's `biprobit` routine estimates the hyperbolic arctangent (`atanh()`) of ρ, which constrains the parameter itself to lie within the appropriate interval when back-transformed. A similar transformation can be used to constrain a slope parameter to lie within a certain interval on the real line.

GMM estimation

There are various Stata commands, official and user-written, that implement Generalized Method of Moments (GMM) estimation. Stata has a general-purpose GMM command, `gmm`, that can be used to solve GMM estimation problems of any type.

Like the `nl` (nonlinear-least squares) command, `gmm` can be used interactively, but it is often likely to be used in its *function evaluator program* form. In that form, just as with `ml` or the programmed version of `nl`, you write a program specifying the estimation problem.

The GMM function evaluator program, or moment-evaluator program, is passed a *varlist* containing the moments to be evaluated for each observation. Your program replaces the elements of the *varlist* with the 'error part' of the moment conditions. For instance, if we were to solve an OLS regression problem with GMM we might write a moment-evaluator program as:

```
. program gmm_reg
  1.          version 13
  2.          syntax varlist if, at(name)
  3.          qui {
  4.                  tempvar xb
  5.                  gen double `xb´ = x1*`at´[1,1] + x2*`at´[1,2] + ///
>                  x3*`at´[1,3] + `at´[1,4] `if´
  6.                  replace `varlist´ = y - `xb´ `if´
  7.          }
  8. end
```

where we have specified that the regression has three explanatory variables and a constant term, with variable `y` as the dependent variable. The row vector `at()` contains the current values of the estimated parameters. The contents of *varlist* are replaced with the discrepancies, $y - X\beta$, defined by those parameters. A *varlist* is used as `gmm` can handle multiple-equation problems.

To perform the estimation using the standard `auto` dataset, we specify the parameters and instruments to be used in the `gmm` command:

```
. sysuse auto
. gen y = price
. gen x1 = weight
. gen x2 = length
. gen x3 = turn
. gmm gmm_reg, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(weight length turn) onestep nolog
Final GMM criterion Q(b) =  2.43e-16
GMM estimation
Number of parameters =   4
Number of moments    =   4
Initial weight matrix: Unadjusted                    Number of obs   =      74
------------------------------------------------------------------------------
             |               Robust
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         /b1 |   5.382135   1.719276     3.13   0.002     2.012415    8.751854
         /b2 |  -66.17856   57.56738    -1.15   0.250    -179.0086    46.65143
         /b3 |  -318.2055   171.6618    -1.85   0.064    -654.6564    18.24543
         /b0 |   14967.64    6012.23     2.49   0.013     3183.881    26751.39
------------------------------------------------------------------------------
Instruments for equation 1: weight length turn _cons
```

This may seem unusual syntax, but we are just stating that we want to use the regressors as instruments for themselves in solving the GMM problem, as under the hypothesis of $E[u|X] = 0$, the appropriate moment conditions can be written as $EX'u = 0$.

Inspection of the parameters and their standard errors shows that these estimates match those from `regress, robust` for the same model. It is quite unnecessary to use GMM in this context, of course, but it illustrates the way in which you may set up a GMM problem.

To perform linear instrumental variables, we can use the same moment-evaluator program and merely alter the instrument list:

```
. webuse hsng2, clear
(1980 Census housing data)

. gen y = rent

. gen x1 = hsngval

. gen x2 = pcturban

. gen x3 = popden

. gmm gmm_reg, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(pcturban popden faminc reg2-reg4) onestep nolog

Final GMM criterion Q(b) =  150.8821

GMM estimation

Number of parameters =   4
Number of moments    =   7
Initial weight matrix: Unadjusted                 Number of obs   =      50

------------------------------------------------------------------------------
             |               Robust
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         /b1 |   .0022538   .0006785     3.32   0.001      .000924    .0035836
         /b2 |   .0281637   .5017214     0.06   0.955    -.9551922     1.01152
         /b3 |   .0006083   .0012742     0.48   0.633    -.0018891    .0031057
         /b0 |   122.6632   17.26189     7.11   0.000     88.83052    156.4959
------------------------------------------------------------------------------
Instruments for equation 1: pcturban popden faminc reg2 reg3 reg4 _cons
```

These estimates match those produced by `ivregress 2sls, robust`.

Consider solving a nonlinear estimation problem: a binomial probit model, using GMM rather than the usual ML estimator. The moment-evaluator program:

```
. program gmm_probit
  1.          version 13
  2.          syntax varlist if, at(name)
  3.          qui {
  4.                  tempvar xb
  5.                  gen double `xb´ = x1*`at´[1,1] + x2*`at´[1,2] + ///
>                             x3*`at´[1,3] + `at´[1,4] `if´
  6.                  replace `varlist´ = y - normal(`xb´)
  7.          }
  8. end
```

To perform the estimation, we specify the parameters and instruments to be used in the `gmm` command:

```
. webuse hsng2, clear
(1980 Census housing data)

. gen y = (region >= 3)

. gen x1 = hsngval

. gen x2 = pcturban

. gen x3 = popden

. gmm gmm_probit, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(pcturban hsngval popden) onestep nolog

Final GMM criterion Q(b) =  3.18e-21

GMM estimation

Number of parameters =   4
Number of moments    =   4
Initial weight matrix: Unadjusted                    Number of obs   =      50

------------------------------------------------------------------------------
             |               Robust
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         /b1 |   .0000198   .0000146     1.35   0.177    -8.92e-06    .0000484
         /b2 |   .0139055   .0177526     0.78   0.433     -.020889    .0487001
         /b3 |  -.0003561   .0001142    -3.12   0.002    -.0005799   -.0001323
         /b0 |  -1.136154   .9463889    -1.20   0.230    -2.991042    .7187345
------------------------------------------------------------------------------
Instruments for equation 1: pcturban hsngval popden _cons
```

Inspection of the parameters shows that these estimates are quite similar to those from `probit` for the same model. However, whereas `probit` requires the assumption of *i.i.d.* errors, GMM does not. The standard errors produced by `gmm` are robust to arbitrary heteroskedasticity.

As in the case of our linear regression estimation example, we can use the same moment-evaluator program to estimate an instrumental-variables probit model, similar to that estimated by `ivprobit`. Unlike that ML command, though, we need not make any distributional assumptions about the error process in order to use GMM.

```
. gmm gmm_probit, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(pcturban popden rent hsnggrow) onestep nolog
Final GMM criterion Q(b) =  .0470836
GMM estimation
Number of parameters =   4
Number of moments    =   5
Initial weight matrix: Unadjusted                     Number of obs   =      50

------------------------------------------------------------------------------
             |               Robust
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         /b1 |  -6.25e-06   .0000203    -0.31   0.758     -.000046    .0000335
         /b2 |   .0370542   .0333466     1.11   0.266     -.028304    .1024124
         /b3 |  -.0014897   .0013724    -1.09   0.278    -.0041795    .0012001
         /b0 |   -.538059   1.278787    -0.42   0.674    -3.044435    1.968317
------------------------------------------------------------------------------
Instruments for equation 1: pcturban popden rent hsnggrow _cons
```

Although the examples of `gmm` moment-evaluator programs we have shown here largely duplicate the functionality of existing Stata commands, they should illustrate that the general-purpose `gmm` command may be used to solve estimation problems not amenable to any existing commands, or indeed to a maximum-likelihood approach. In that sense, familiarity with `gmm` capabilities is likely to be quite helpful if you face challenging estimation problems in your research.

Programming for prefix commands: simulate, bootstrap

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, we might want to evaluate their empirical rate of convergence when some of the underlying assumptions are not 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.

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.

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 might need draws 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 can 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
```

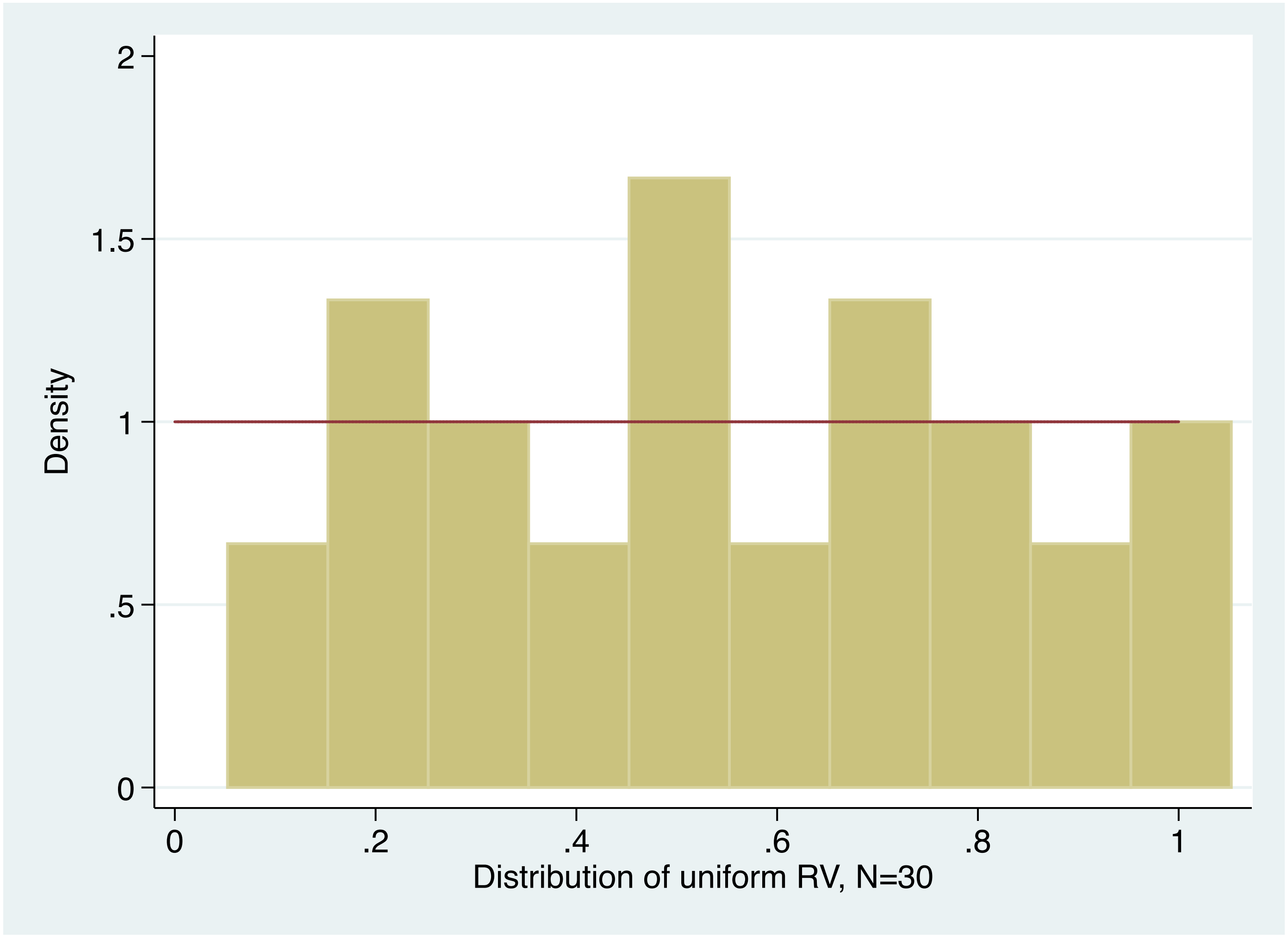
The means of these two variables are close to their theoretical values, as is the variance of the Poisson-distributed variable.

A first illustration of simulation

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 $N \to \infty$. 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.

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 to 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 the `adopath` command 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.

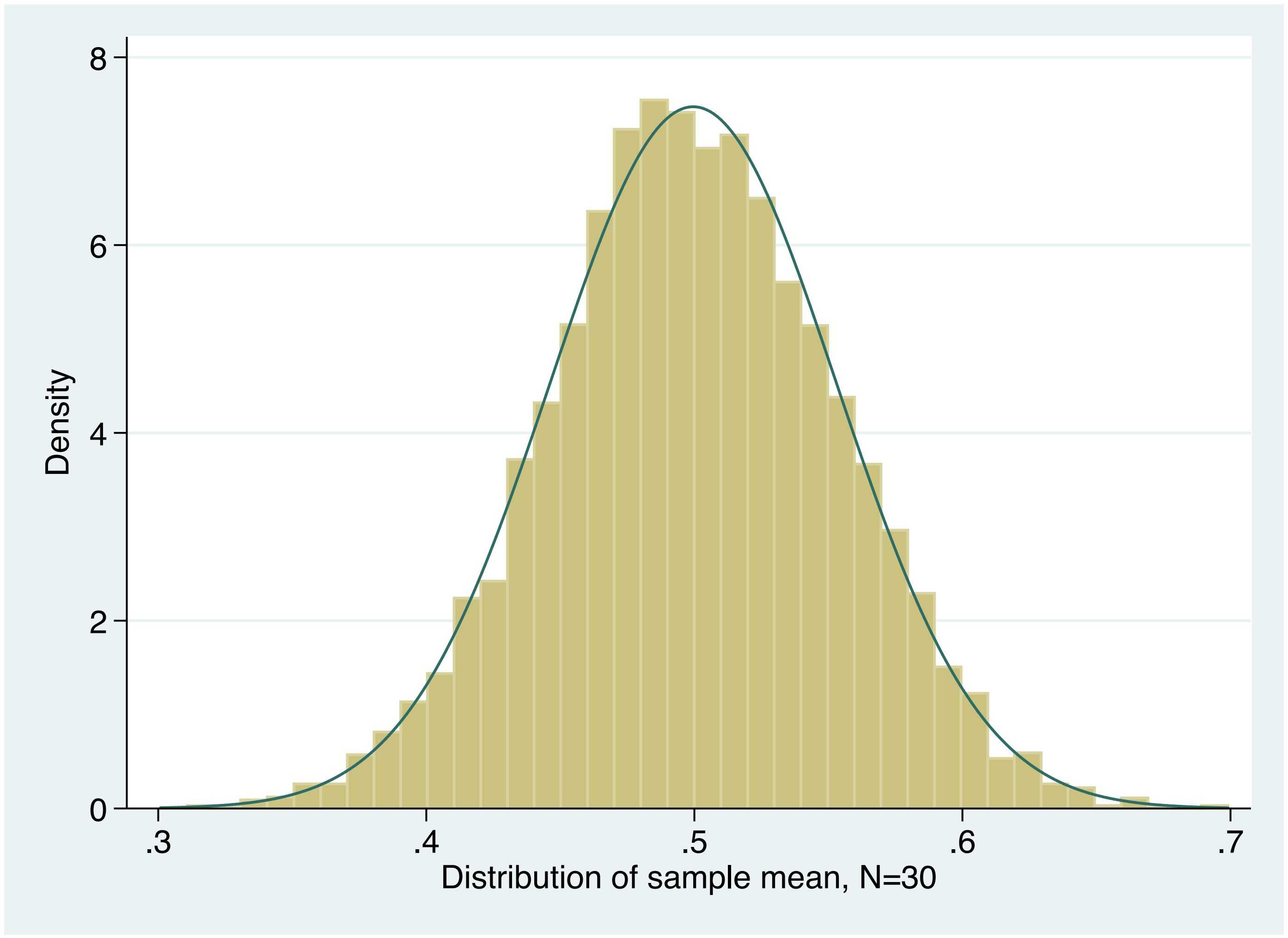
Executing the simulation

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 $\sqrt{(1/12)/30} = 0.0527$.

```
 . 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.

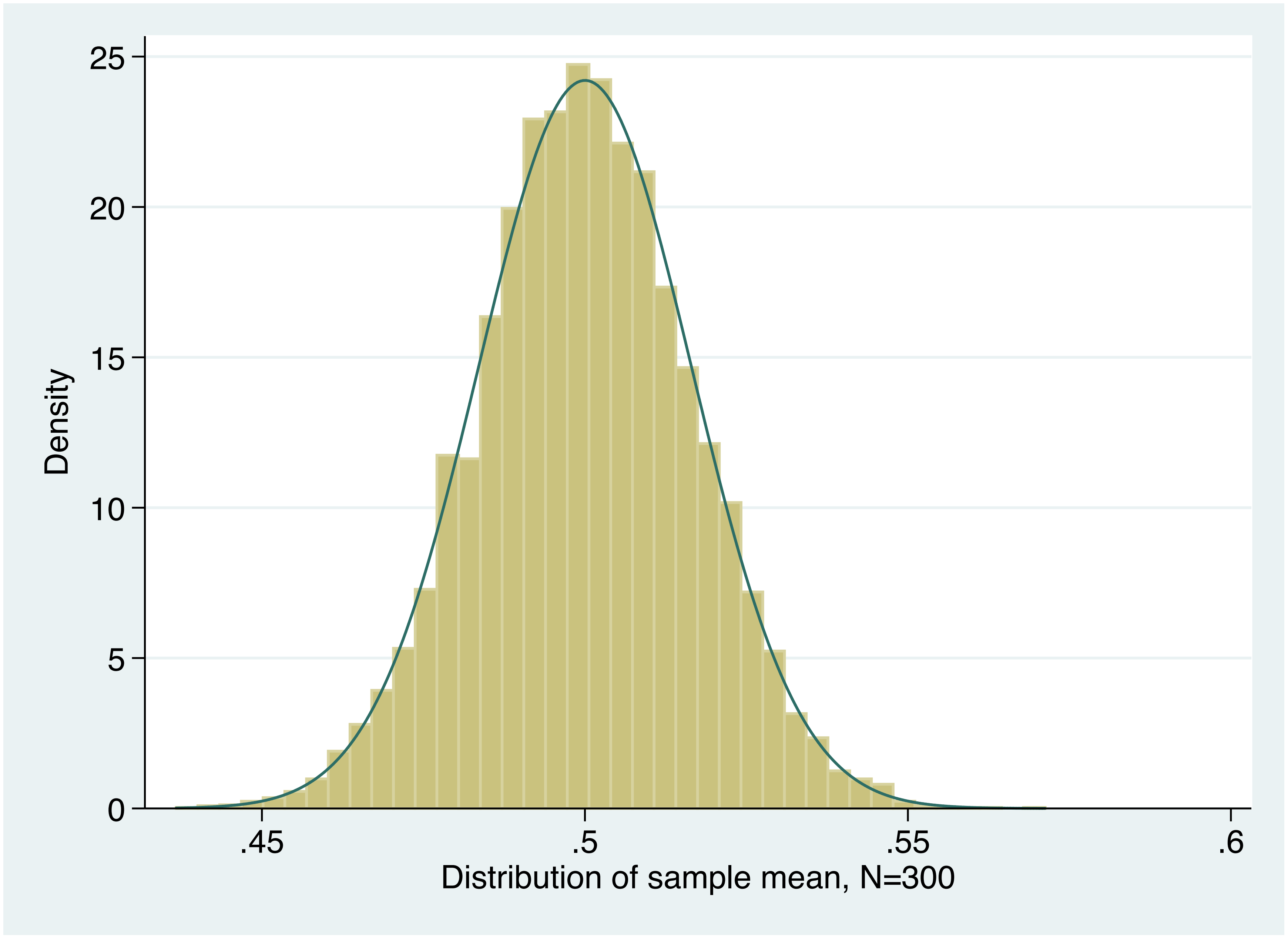
Again, we should check to see that the program works properly with this new feature, and produces the same result as we could manually:

```
 . 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.



Density
0
5
10
15
20
25
.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`

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.

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/λ ~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.

```
 . 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):

```
 . 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.

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 so that we can 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
```

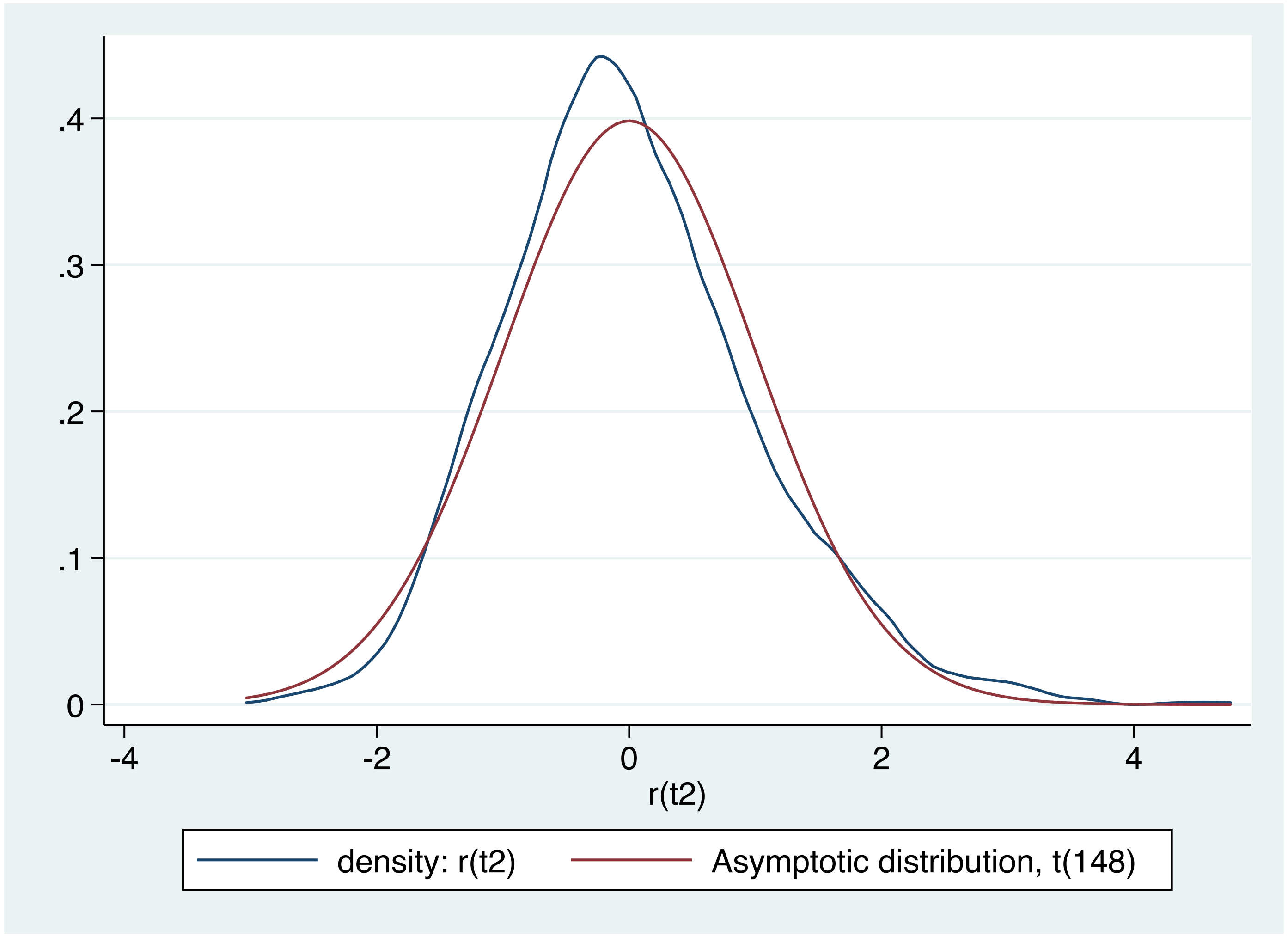
The mean of simulated `b2f` is very close to 2.0, implying the absence of bias. The standard deviation of simulated `b2f` is close to the mean of `se2f`, suggesting that the standard errors are unbiased as well. The mean rejection rate of 0.046 is close to the size of the test, 0.05.

In order to formally evaluate the simulation results, we use the `mean` command 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)))
```

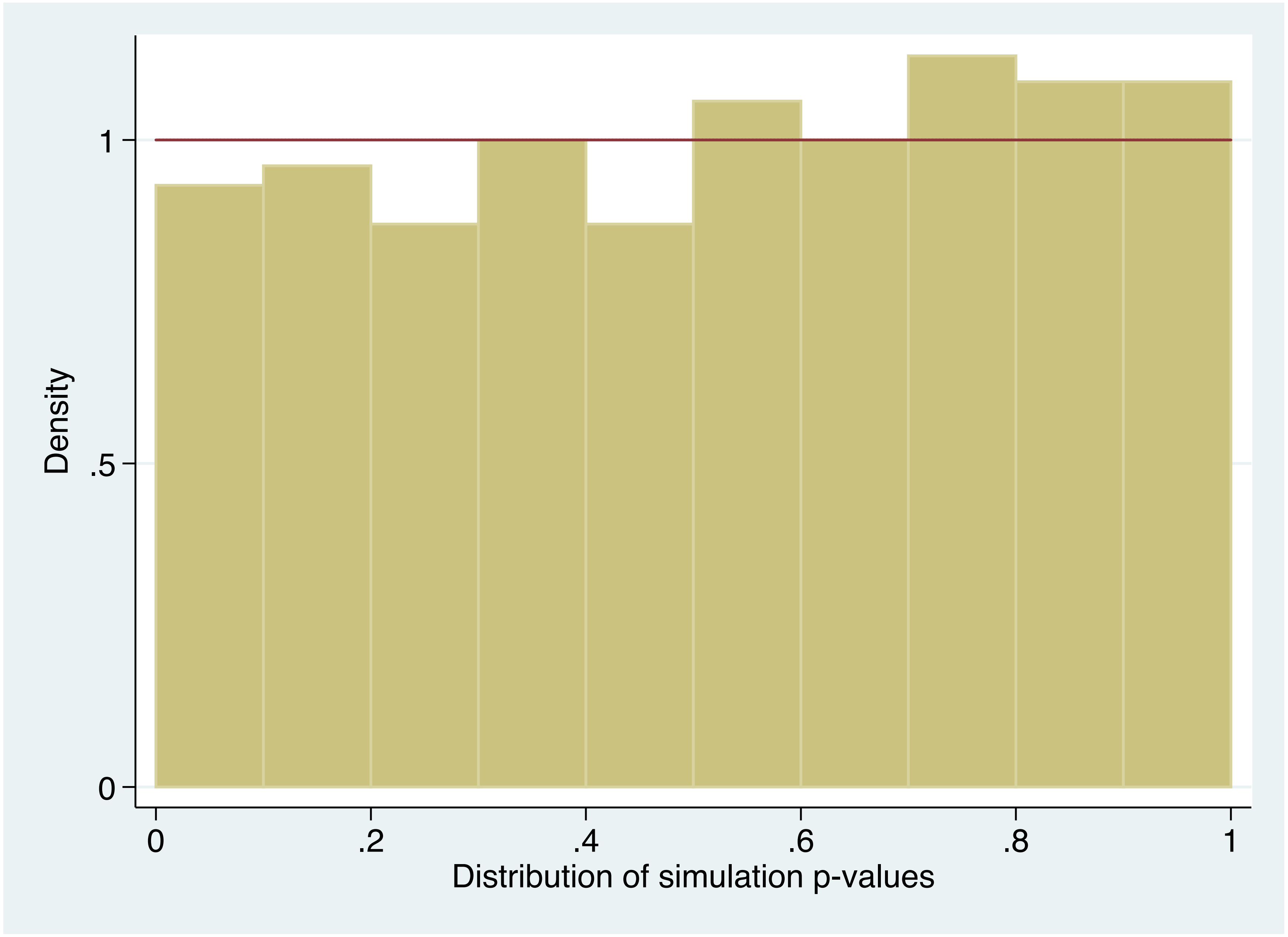


Size of the test

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).


Density
1
.5
0
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.

```
 . 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
-------------+------------------------------           F(  1,   498) = 3368.07
       Model |  5025.95627     1  5025.95627           Prob > F      =  0.0000
    Residual |   743.13261   498  1.49223416           R-squared     =  0.8712
-------------+------------------------------           Adj R-squared =  0.8709
       Total |  5769.08888   499  11.5613004           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
─────────────┴────────────────────────────────────────────────
```

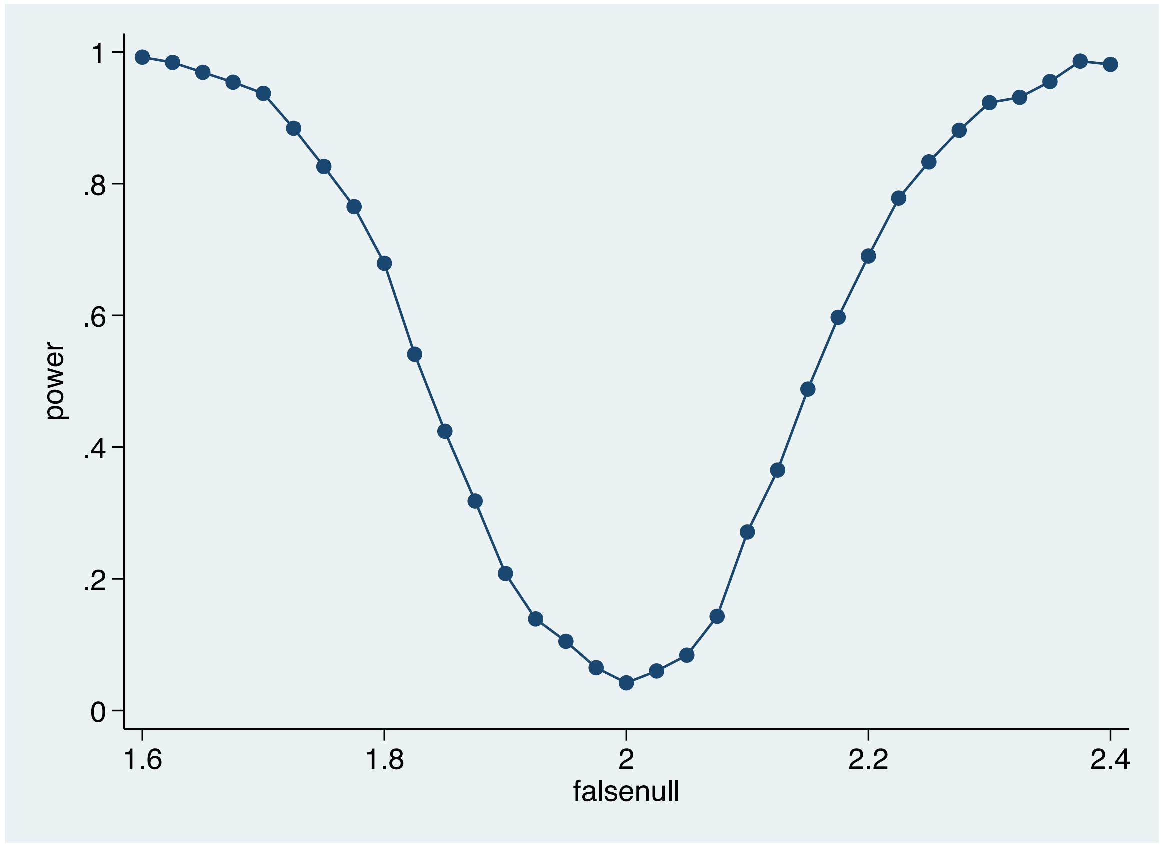
The presence of skewed errors has weakened the ability of the estimates to reject the false null at smaller sample sizes.

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.

```
 . 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 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."

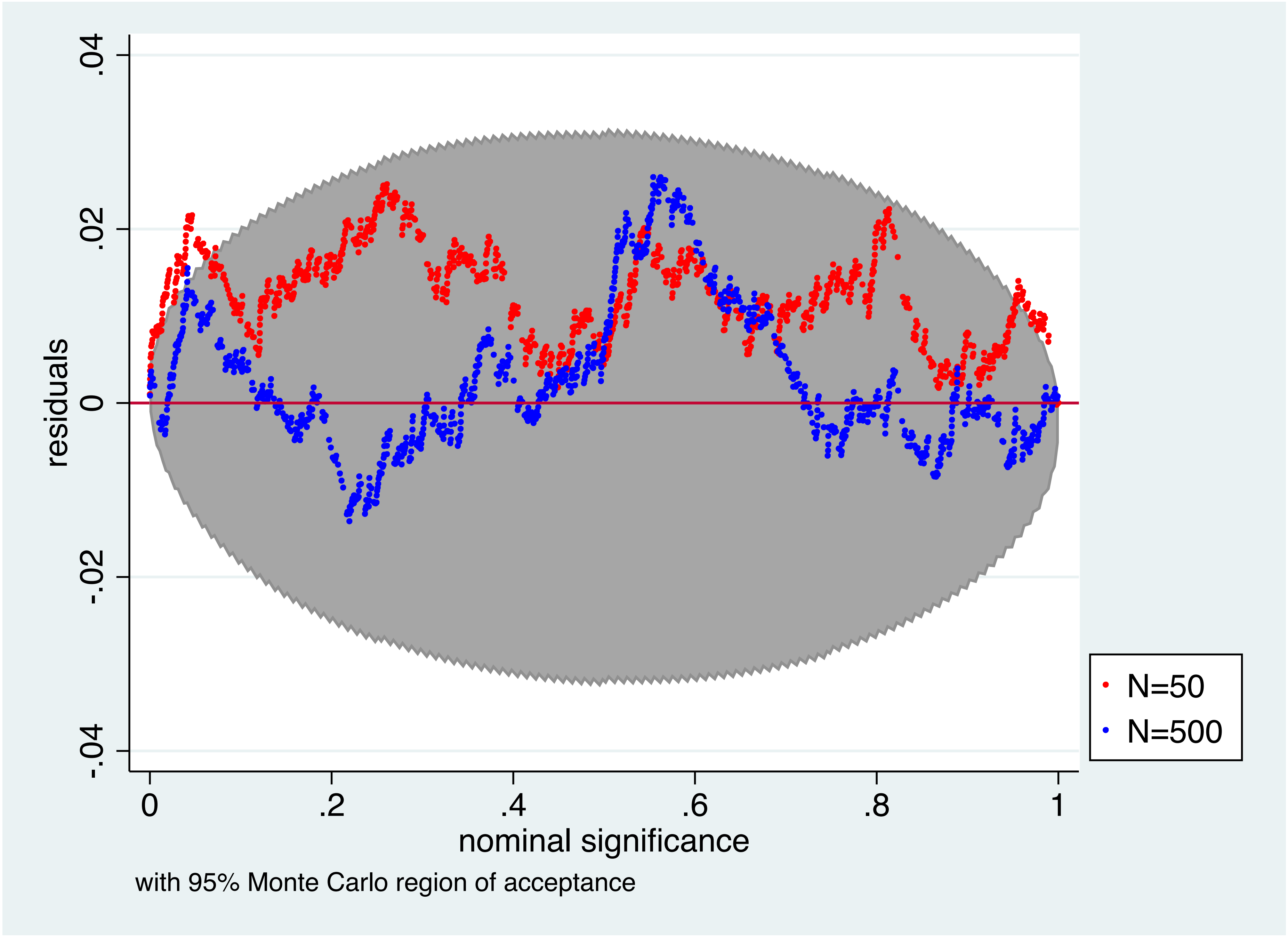
In this example, adapted from the help file, we consider the performance of a *t*-test when the data are not Gaussian, but rather generated by a $\chi^2(2)$, with a mean of 2.0. A *t*-test of the null that $\mu = 2$ is a test of the true null hypothesis. We want to evaluate how well the *t*-test performs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

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.


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

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.

```
 . 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              Number of obs =   10000
-------------+------------------------------           F(  1,  9998) =70216.80
       Model |  320512.118     1  320512.118           Prob > F      =  0.0000
    Residual |  45636.9454  9998  4.56460746           R-squared     =  0.8754
-------------+------------------------------           Adj R-squared =  0.8753
       Total |  366149.064  9999  36.6185682           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 $\sigma^2_v/(\sigma^2_v + \sigma^2_{x^*}) = 0.1$, 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              Number of obs =   10000
-------------+------------------------------           F(  1,  9998) =20632.81
       Model |  246636.774     1  246636.774           Prob > F      =  0.0000
    Residual |   119512.29  9998  11.9536197           R-squared     =  0.6736
-------------+------------------------------           Adj R-squared =  0.6736
       Total |  366149.064  9999  36.6185682           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.

The bootstrap command and prefix

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 $\left(\hat{\theta}_1, \hat{\theta}_2, ..., \hat{\theta}_B\right)$, 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):

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

where the formula has been written to allow $\hat{\theta}$ to be a vector of estimated parameters. The square roots of this variance-covariance matrix are known as the *bootstrap standard errors* of $\hat{\theta}$.

The bootstrap standard errors 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., $\hat{\theta}$, which may be compared with the point estimate of the statistic computed from the original sample, θ_{obs} (for observed). The difference $(\hat{\theta} - \theta_{obs})$ is an estimate of the bias of the statistic; in the presence of a biased point estimate, this bias may be nontrivial. However we cannot use that difference to construct an unbiased estimate, as 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.

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.

The confidence interval is based on the assumption of approximate normality of the sampling (and bootstrap) distribution, and will be reasonable if that assumption is so. You can 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.

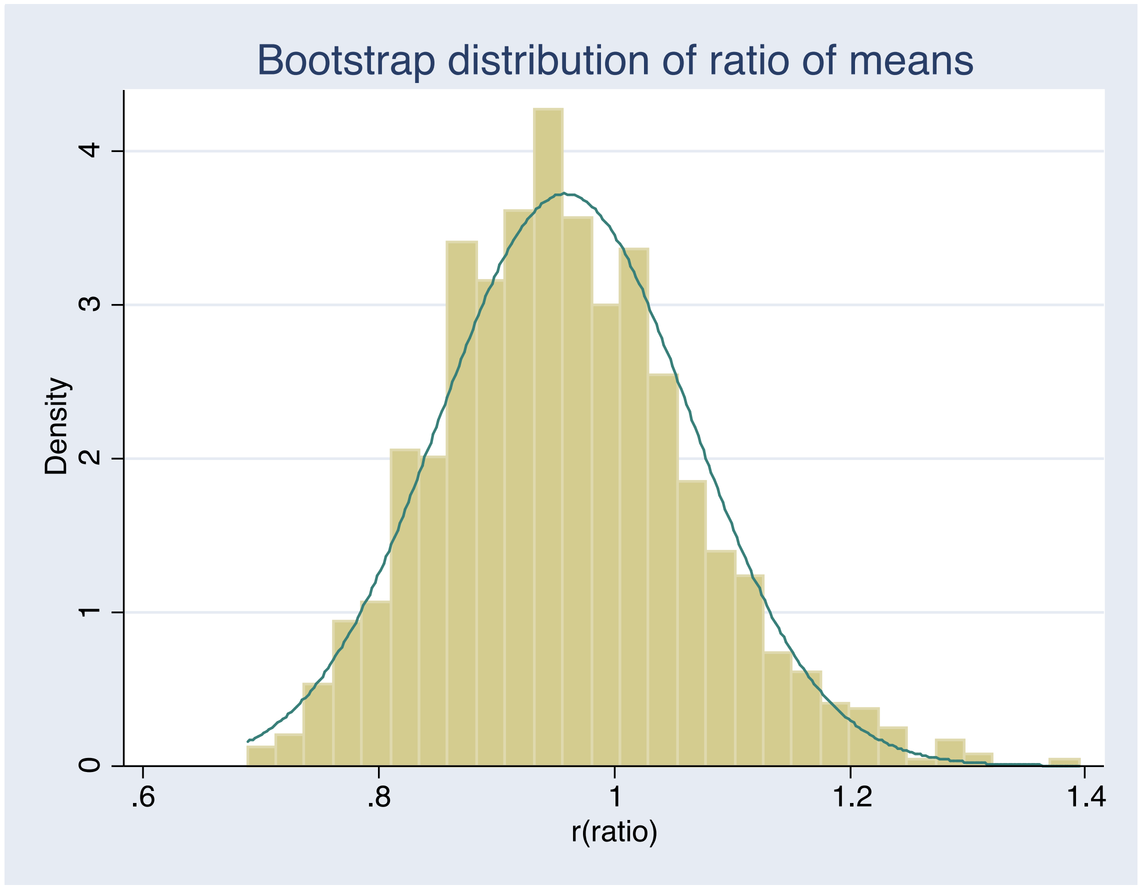
Rather, we could do so by writing a program that returned that ratio:

```
capture program drop muratio
program define muratio, rclass
version 14
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.

```
.
. capture program drop muratio

. program define muratio, rclass
  1.              version 11
  2.              syntax varlist(min=2 max=2)
  3.              tempname ymu
  4.              summarize `1´, meanonly
  5.              scalar `ymu´ = r(mean)
  6.              summarize `2´, meanonly
  7.              return scalar ratio = `ymu´/r(mean)
  8. end

.
. set seed 10101

. local reps 1000

. webuse auto, clear
(1978 Automobile Data)

. tabstat price, by(foreign) stat(n mean semean)

Summary for variables: price
     by categories of: foreign (Car type)

 foreign |         N      mean  se(mean)
---------+------------------------------
Domestic |        52  6072.423  429.4911
 Foreign |        22  6384.682  558.9942
---------+------------------------------
   Total |        74  6165.257  342.8719
----------------------------------------
```

```
(22 missing values generated)
```



Note in the histogram that the empirical distribution is quite visibly skewed.