Useful Stata Commands for Simulation

1. Loops – loops allow you to repeat a section of code in your do file, typically changing one or more elements of that code with each iteration. The part of the code that changes from one iteration to the next is a "local macro variable." You can loop over a sequence of values, or a list of strings or variable names. The appropriate syntax is below where *j* is an arbitrary name for the local macro (you can choose this). It stands in for the value on each iteration:

`foreach j in …`	when looping over a list of strings/values
`foreach j of varlist …`	when looping over a variable list
`forvalues j=#/#`	when looping over a sequence of values

Loops require an open bracket at the end of the first line and a closing bracket on a line by itself at the conclusion of the loop. For example:

```
forvalues j=1/10 {
   display `j'
   }
foreach color in red yellow green {
   display "`color'"
   }
foreach var of varlist income-age {
   sum `var'
   }
```

See Cox (2020) for a helpful introduction to loops and local macros in Stata.

2. Local macro variables – in the examples above, *j*, *color*, and *var* are all local macros. After a local macro is defined, every reference to it must be surrounded with single quotes. Moreover, the opening quote must be "sloped down and to the right" (grave accent); the closing quote must be "sloped down and to the left" (acute accent). In the second example above, the local macro *color* is resolved using single quotes; since the resulting value of *color* is a string (the words red, yellow, and green) and we want to `display` it, the value must be surrounded by double quotes, as with any string. In the first example, *j* is numeric, so no double quotes are needed when it is used in the `display` command.

   Local macro variables can also be defined outside of loops. The key difference between local and global macros (see next item) is that local macro variables are only "active" while your do-file is running. When the do-file stops, they are no longer in memory. The example below creates two local macro variables in a do-file and then uses them in a `display` command.

```
local j=10
display `j'+5
local name Vanderbilt
display "`name' is a top university"
```

Last updated: September 11, 2023

3. Global macro variables – global macros work just like local macros, the main difference being that they remain in memory outside of your do-file. (They are retained until you erase them, replace them, or exit Stata). Unlike local macros, after a global macro is defined, every reference to it must be preceded by a dollar sign `$`. Some basic examples follow:

```
global myname "Sean Corcoran"
display "My name is $myname"
global workdir "C:\My Documents\Stats Class"
cd "$workdir"
```

   Again, if your global macro resolves as a string value and the string value use would normally require the use of double quotes, then be sure to include double quotes as above.

4. Macro utilities – there are lots of commands that allow you to manage macros, but several useful ones are:

`macro dir`	list all macros currently in memory
`macro drop myname`	drop the macro called *myname*
`macro drop _all`	drop all user-defined macros

   Stata also has system-created macros that contain things like the current data's filename, or variables used in previous commands. You will see these when your data is in memory and you type `macro dir`.

   You can even use "extended macros" which can grab other features of your data. For example, the command below creates a local macro called *`inclabel`* that will contain the variable label for the variable called *`income`*. Type `help extended_fcn` for a list of available extended macros.

```
local inclabel : variable label income
```

5. Preserve and restore – Stata allows you to `preserve` your data in its current state, do things with it, and then `restore` the data back to the point at which you `preserve`d it. For example, the following commands keep only females in the data before executing `sum`:

```
preserve
keep if female==1
sum readingscore
restore
```

6. Temporary variables and temporary files – Stata can create variables and data files that exist only while your do-file is running. These are useful if you have no need to retain that variable or dataset beyond the execution of your program. You must first create them using the `tempvar` or `tempfile` command; you can then use the temporary variable or filename later by surrounding it in single quotes (like local macros). Some simple examples are below:

```
tempvar agetimes2
```

```
gen `agetimes2' = age*2
sum `agetimes2'

tempfile boys
tempfile girls
preserve
keep if female==1
save `girls'
restore
preserve
keep if female==0
save `boys'
restore
```

7. Scalars – a scalar is a name assigned to a number or a string (usually a number). When referencing a scalar, no quotes are required. For example:

```
scalar cpi = 1/1.3256
display cpi
gen realgdp = gdp*cpi
```

8. Saved results – Stata typically holds results in memory as scalars or matrices after certain commands are run. For example, after the `summarize` command, the mean is saved as `r(mean)`. Type `return list` after a command to see which results have been saved as scalars. The saved result is lost when you do something new; if you want to retain that number for later, create a scalar that contains it.

```
summarize income
return list
display r(mean)
display r(sd)
scalar meaninc=r(mean)      Saving the mean as a scalar for use later
```

   If the command is an *estimation* command (e.g., `regress`), type `ereturn list` to see the saved results. Some saved results will be scalars, while others will be matrices.

9. Matrices – matrices are objects with `r` rows and `c` columns. The notation `a[r, c]` describes a matrix called `a` with `r` rows and `c` columns. E.g. `a[16, 16]`. Stata has a full matrix language called Mata. You can also easily create and manipulate matrices yourself (beyond the scope of this handout). Type `help matrix` for a list of commands.

10. Draw random samples from your data – if you would like to draw a random sample of size n from your existing dataset, use the commands `sample` or `bsample`. `sample` is a random sample *without replacement*, while `bsample` (bootstrap sample) is a random sample *with replacement*. Note that Stata keeps the sampled observations and drops all others, so you may want to use these together with `preserve` and `restore`.

`sample 25, count`	draws a random sample of n=25
`sample 25`	draws a random sample of 25% of your cases
`bsample 10`	draws a random sample of n=10 *with replacement* (bootstrap)

These commands have options for drawing more complex sampling methods (i.e., not a simple random sample) including stratified and cluster sampling.

11. Stata uses a pseudo-random number generator when selecting random samples and drawing random numbers from a distribution (see item 13 below). What this means is that the results are not truly random but generated by a mathematical algorithm. If you start with the same "seed" value, you will perfectly replicate the same random draws time and time again. This is valuable for replicating work that involves random samples and numbers. To select a seed value, include a `set seed` command at the beginning of your do-file. The seed value can be any number you choose:

```
set seed 12345
```

12. Add empty observations to a dataset – the command `set obs #` will change the number of observations in the current data. If the dataset is empty, it will create # blank observations that you can then fill in (say, if you are generating simulated data). If the dataset already has _N observations, then # must be at least as large as _N.

```
set obs 1000
```

13. Drawing random values from a distribution – Stata can draw random values from a long list of discrete and continuous probability distributions. Some common ones are below. (In each case the variable that will contain the random values is called *x*).

```
gen x = rnormal()          standard normal (0, 1)
gen x = rnormal(m, s)      normal (m, s)
gen x = runiform()         uniform distribution on (0, 1)
gen x = runiform(a, b)     uniform distribution on (a, b)
gen x = rbinomial(n, p)    binomial distribution (# of
                            successes in n trials with p
                            probability of success)
gen x = rt(df)             t-distribution with df degrees of
                            freedom
gen x = runiform()<p       Bernoulli (0-1) variable where p
                            is the probability x=1
```

In Stata, type `help random` for a complete list of random number functions.

The command `drawnorm` draws a sample from a multivariate normal distribution with desired means and covariance/correlation matrix. (You specify the means and covariance matrix; otherwise, it assumes means zero, standard deviation one, and zero correlations

between variables). You can specify the number of observations to be drawn with an empty dataset, otherwise it will use the number of observations in your existing dataset.

```
drawnorm newvarlist, n(#) corr(matrix)
```

Example: draw 100 observations of (x_1, x_2) from a bivariate normal distribution with means (2, 3) and correlation of 0.5:

```
matrix m=(2,3)
matrix C=(1, 0.5 \ 0.5, 1)
drawnorm x1 x2, n(100) corr(C) means(m)
```

14. Monte-Carlo type simulations: the command `simulate` is an easy way of repeatedly executing a Stata command or user-written program while capturing results along the way. For example, suppose you have a user-written program called `ols` that draws random variables from a known distribution and then estimates a simple regression. The command below tells Stata to repeat this command 1000 times while saving the slope coefficient beta each time:

```
simulate beta=_b[x], reps(1000): ols
```

Note _b[x] is Stata's given name for the estimated slope coefficient on a variable called "x" each time the regress command is run. "beta" is a name we are assigning to those values.

For more on user-written programs, see the Stata help menu for `program`.

15. Bootstrapping – nonparametric bootstrap estimation of statistics from a Stata command or user-written program. Note bootstrapping is sampling *with replacement* from your data. The basic syntax is:

```
bootstrap explist, options: command
```

For example, 99 bootstrap replications of the sample mean of x1:

```
bootstrap _b, reps(99) saving(output): mean x1
```

This command executes the `mean` command 99 times, each time with a different bootstrap sample of size _N (the number of observations in your dataset). Note `_b` is Stata's given name for the mean from the `mean` command. See `return list` after `mean` for the available saved scalars. The option "saving" tells Stata to save the accumulated results in a dataset called *output*.

16. Postfile – for collecting results across multiple iterations or replications, as in a Monte Carlo simulation or bootstrap. The command `postfile` sets the stage by specifying variable names and the name of a new Stata file where the results will be saved. The command `post` adds a set of results to said file. `postclose` closes the results file when done.

The following example simulates the drawing of a sample of 25 from a known population (normal, with a mean of 10 and standard deviation 5) and calculation and storage of the sample mean and sample standard deviation:

// `tempname` specifies a temporary holding place where the results will be held
// `tempfile` specifies a temporary file where the results will be saved

```
tempname results
tempfile meantable
```

// `postfile` tells Stata: (1) that results will be iteratively saved to the temporary location *results*; (2) the two items that will be saved—in order—are to be named *x1mean* and *x1sd* (names you choose); (3) *meantable* is the file to which the final results will be saved.

```
postfile `results' x1mean x1sd using `meantable'

forvalues j=1/100 {
   clear
   drawnorm x1, n(25) mean(10) sds(5)
   sum

   // post tells Stata to save the results r(mean) and r(sd) from the sum command to
   the temporary location results
   post `results' (r(mean)) (r(sd))
   }
```

// `postclose` finalizes things by taking results stored in *results* and saving them to *meantable* as specified in the initial `postfile` command.

```
postclose `results'
```

// now view the resulting data stored in temporary file *meantable*

```
use `meantable', clear
histogram x1mean
```