Monte Carlo Analysis

Andrew Q. Philips*

February 19, 2016

*Ph.D Candidate, Department of Political Science, Texas A&M University, 2010 Allen Building, 4348 TAMU, College Station, TX 77843-4348. aphilips@pols.tamu.edu. http://people.tamu.edu/∼aphilips/.

Contents

1 Motivation

This short paper discusses Monte Carlo simulations, with examples in both R and Stata. Why would we ever want to use a Monte Carlo experiment? Let's start with a motivating example:

```
x <- rnorm(25)
y <- 2*x + rnorm(25)
```

Above we created a data-generating process (or DGP) of 25 observations. Y is a function of a normally distributed X, as well as some random noise. Note that we specified a coefficient on X of 2. Here is what the regression output looks like:

```
summary(lm(y~x))

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.84637 -0.37758 -0.05448  0.58752  1.18899
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.03074    0.14831   0.207    0.838
## x            2.39579    0.18158  13.194 3.25e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7413 on 23 degrees of freedom
## Multiple R-squared:  0.8833,Adjusted R-squared:  0.8782
## F-statistic: 174.1 on 1 and 23 DF,  p-value: 3.255e-12
```

Although the coefficient on X should be 2 (since we specified it ourselves), it is not...the coefficent is almost 2.4. Looks like our OLS example results in a

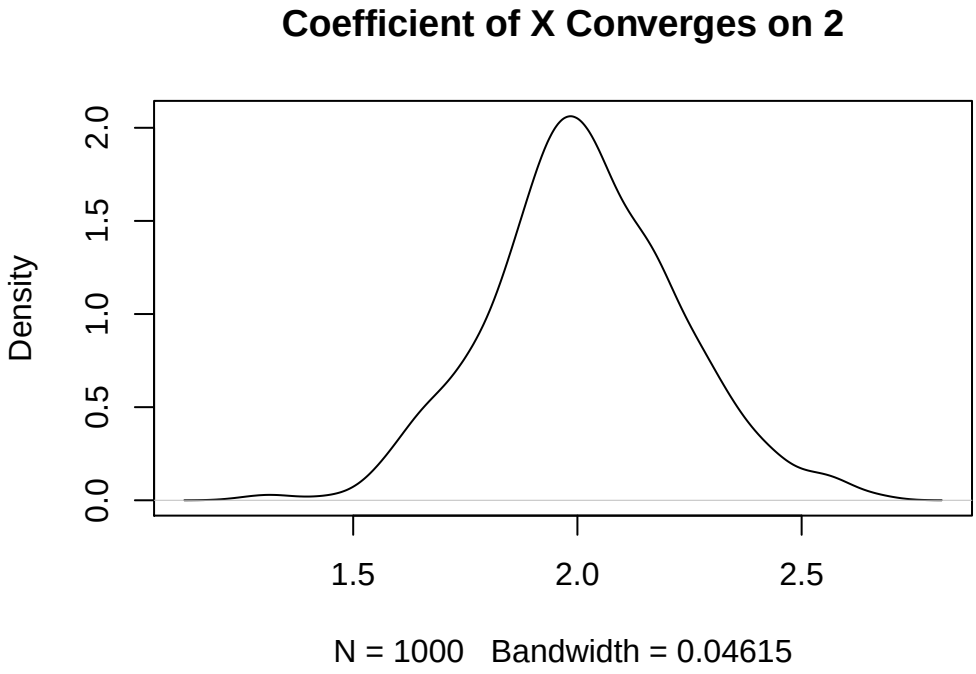
coeffecient that is about 20% higher than what we specified when we created the data. What explains this? We only took one realization (or one draw) of the DGP. Let's re-draw the sample and run the same regression:

```
x <- rnorm(25)
y <- 2*x + rnorm(25)
summary(lm(y~x))


##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.33003 -0.76252  0.05049  0.82558  2.34376
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.1623     0.2335   0.695    0.494
## x             1.9816     0.2285   8.674 1.04e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.15 on 23 degrees of freedom
## Multiple R-squared:  0.7659,Adjusted R-squared:  0.7557
## F-statistic: 75.24 on 1 and 23 DF,  p-value: 1.042e-08
```

The coefficient on X is much closer to 2 this time. If we were to keep doing this, say 1000 times, saved the coefficents from the resulting OLS regressions, and plotted their values, we would get something that looked like this:

```
plot(density(results[,1]), main='Coefficient of X Converges on 2')
```



As the plot shows, it looks like if we were to repeatedly take draws based on our DGP that we created, and plotted the coefficents, we would get something awfully close to the true value that we specified.

In short, the example shown above is the motivation behind Monte Carlo simulations. This approach produces distributions of possible outcome values. Starting with a given sample, we can use Monte Carlos to make stronger inferences about the population; in our example above, this was the data-generating process we specified. Based on a probability distribution we provide, the simulation 'pulls' or draws values at random from this distribution and calculates the probabilistic outcome. In effect, by allowing our parameters (of a function such as an OLS regression) to vary within a set of constraints (the probability distribution), we will end up with a much better sense of what the population values are, and what our degree of confidence in that value is.

Why should we care? For one, measures of uncertainty are important, especially in the social sciences where deterministic outcomes are impossible. Moreover, although many tests and models perform well given extremely large samples, often we have much smaller samples in practice. Or, the samples that we

draw may not be independent. All of these can be tested using Monte Carlo methods.

Monte Carlo methods developed out of the Los Alamos Laboratory under Stanislaw Ulam in the mid-late 1940s. Later, John von Neumann picked up the method, and gave it the name "Monte Carlo." The casinos of Monte Carlo (in Monaco) was where Ulam's uncle would gamble (see Figure 1).[1] These simulations were further developed under such groups as the RAND Corporation.

Figure 1: Monte Carlo

Monte Carlo simulations are useful for a number of reasons. First, we may want to find out the asymptotic properties of our results. One example are the critical values for unit-root tests, many of which do not follow the standard t, Z, or F distributions. By specifying the Data-Generation Process (DGP), we can examine the properties of our test statistic or model at hand or calculate these critical values ourselves.

We can also examine finite-sample properties of a procedure; how many observations are good enough to ensure we correctly reject the null hypothesis at an acceptable number of times, for example. Finite samples are very important to examine in the context of estimators since we rarely have large enough samples where the asymptotic properties hold. Often, Monte Carlo experiments in applied papers will pit one procedure against another and compare their finite-

[1]Figure from: I, Katonams, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2480853.

sample properties.

They can also serve as robustness checks for our models, or as an aid to help diagnose something we may suspect in our sample data. For instance, we may have a model we have specified but want to run simulations on it under certain conditions; we can violate assumptions of our methods (such as the 'BLUE' of OLS or certain distributional assumptions) and see how our models respond. Or we may want to create a simulation that tests our distributional assumptions; although we typically assume Gaussian (i.e. independent and identically distributed) errors, would our model still yield correct inferences with a different distribution, such as a Beta, for instance? Or, could it handle a violation such as small amounts of autocorrelation and still perform well? All of these counterfactuals, or "what-if?" scenarios, can easily be tested through a Monte Carlo approach.

2 What's so Different About it?

What does our regression output usually give us? We have a random vector X, and estimate as data-generation process parameter θ as $E[f(X)] = \theta$.[2] In typical regression analysis we have a single sample estimate for $\hat{\theta}_n$ (i.e. only x). In our example above, we specified the population parameter θ as 2, and when we took our samples, we drew two samples: $\hat{\theta}_1 = 2.396$ and $\hat{\theta}_2 = 1.982$

With Monte Carlo experiments, we draw new samples of X from the underlying population in order to learn about the distribution of $\hat{\theta}_n$. Let's turn back to our earlier example. To make the density plot of θ, I took 1000 simulations, which gives us $n = 1000$ estimates $\hat{\theta}_n$:

```
reps = 1000 # no. of sims
n = 25 # no. of draws
results = matrix(NA,nrow=reps,ncol=1)    # where output is going
i = 1
for(i in 1:reps) {
                x <-rnorm(n)
                y <- 2*x + rnorm(n)
                results[i,1] <- lm(y~x)$coefficients[2] # grab only theta
}
```

[2] I draw (no pun intended) largely from Haugh (2004).

```
# plot the results:
plot(density(results[,1]), main='Coefficient of X Converges on 2')
```

So in this example, going from $i = 1$ to 1000 we generate x_i, set $f_i = f(x_i)$. After we have simulated all 1000 f_i's, we can then calculate $\hat{\theta}_n$:

$$\hat{\theta}_n = \frac{f_1 + f_2 + ... + f_n}{n} \tag{1}$$

Which in our case of 1000 simulations appears as

$$\hat{\theta}_n = \frac{f_1 + f_2 + ... + f_n}{n} = \frac{\sum_{i=1}^{1000} f_n}{1000} \tag{2}$$

```
summary(results[,1])

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.262   1.884   2.008   2.017   2.158   2.674
```

So it looks like with 1000 draws we found a $\hat{\theta}$ very close to $\theta = 2$. According to the Central Limit Theorem, given enough draws we should now have a normal distribution of $\hat{\theta}_n$ centered around true θ (assuming independence of draws). This means that our estimate of $\hat{\theta}_n$ is consistent:

$$\lim_{n \to \infty} \hat{\theta}_n = \theta \tag{3}$$

Our estimate of $\hat{\theta}_n$ will also be unbiased:

$$E[\hat{\theta}_n] = \frac{E[\sum_{i=1}^{1000} f_n]}{1000} = \theta \tag{4}$$

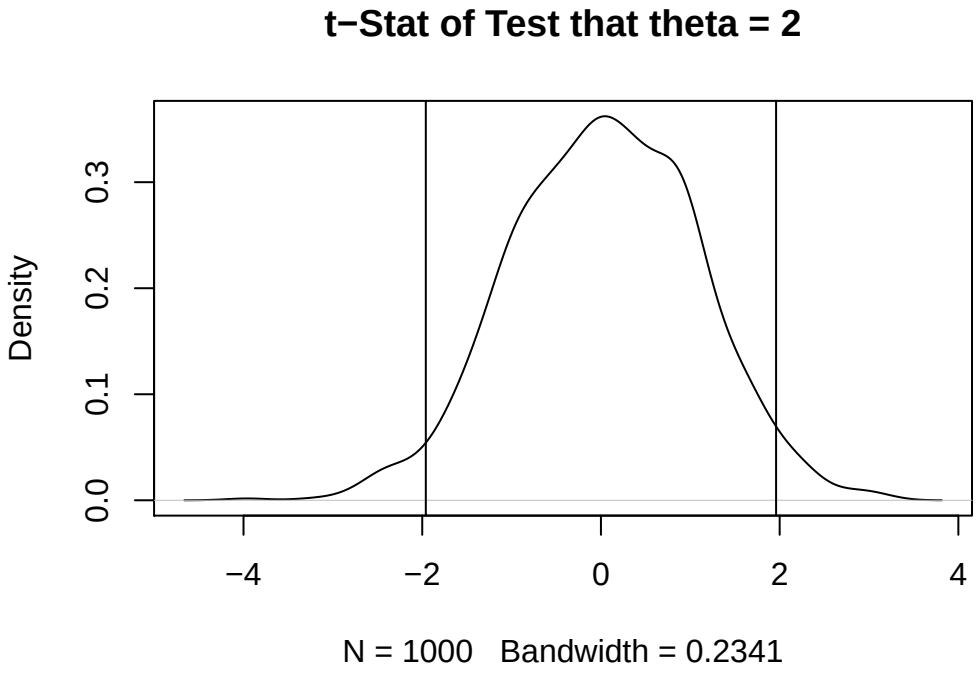
In addition to estimating a convergence around a certain value, we can estimate the probabilities that X is very close to some A subspace (often this is a value of a parameter we specify in the DGP). For example, we can use tests to see if the parameter values we simulate are statistically significant from the true DGP:

$$\theta = E[I_A(X)] \to I_A(X) \tag{5}$$

where $I_A(X) = 1$ if $X \in A$ and 0 otherwise. For instance, if we have a variable x with a parameter $\beta = 2$, for a given confidence interval (say 95%), we can test if the population parameter (2) falls within the confidence interval (calculated by our standard error) 95% of the time. This is relatively straightforward with Monte Carlo simulations; for instance, with 1000 simulations, if ≈950 of the sample parameter values lie within the 95% confidence interval of the population parameter, 2, we can say that our estimates of the standard errors are accurate...i.e. we only reject the null hypothesis when it is true (Type I error) about 5% of the time. We could even calculate Type II error (failure to reject a false null hypothesis) by picking the bounds of our confidence interval (surrounding the population parameter) and calculating the number of times that the sample confidence interval includes the false null areas (any space outside of ≈1.96 standard deviations away from 2, if we are using 95% CIs).

Here is a small example. We simulate our linear regression as before, but now focus on the number of times that we estimate a $\hat{\theta}$ that is significantly (as judged by the p-values coming from a t-test with the null hypothesis that $H_0 : \hat{\theta} = 2$) different from 2.

```
set.seed(023509)
reps = 1000 # no. of sims
n = 25 # no. of draws
results = matrix(NA,nrow=reps,ncol=1)    # where output is going
i = 1
for(i in 1:reps) {
                x <-rnorm(n)
                y <- 2*x + rnorm(n)
                reg <- lm(y ~ x)
                df.ols <- summary.lm(reg)$df[2] # degrees of freedom
                beta <- summary.lm(reg)$coefficient[2,1] # beta of X
                se <- summary.lm(reg)$coefficient[2,2] # std. error of X
                results[i,1] <- (beta - 2)/se # t-test that theta = 2
}
# plot the results:
plot(density(results[,1]), main='t-Stat of Test that theta = 2')
abline(v = 1.96)
abline(v = -1.96)
```




```
quantile(results, c(.025, .975))
```

```
##      2.5%     97.5%
## -2.006965  2.000609
```

As expected, the t-statistic of our test should be centered around 0, and we should only be able to reject the null (have a t-stat greater than approximately 1.96 or below -1.96) about 5 percent of the time. Sure enough, this is the case.

Monte Carlo experiments have also been used to solve intractable equations, such as converging on the critical values of unit root tests.

3 Where are the Random Draws Coming from?

Most commonly, we draw from a normal, uniform or a t distribution. But if we have a suspicion our random draws exhibit some other distributional form, we

should account for this by changing our simulation. We could draw binomial, Poisson, chi-squared, and so on. Programs like R and Stata have many of these distributions as canned procedures, and it is generally good practice to use these rather than program your own distribution.[3] In typical analyses however, such as examining an OLS model, we only use the normal and uniform distributions.

There are procedures related to Monte Carlos, such as the jackknife or the bootstrap. Yet Monte Carlos differ from these two since they are resampling techniques (such as bootstrapping) since resampling takes draws (with replacement) from our sample, not our population. The fact that many draws are taken in resampling makes it somewhat similar to Monte Carlo techniques. To sum up these differences:

- Resampling: Repeated draws from *sample*
- Monte Carlo: Repeated samples from the underlying *population*

However, both methods involve using data we either already have (resampling) or can create (Monte Carlo) to make broader inferences of the population. For more information on resampling methods see Efron and Tibshirani (1986, 1994).

A common number of simulations in an experiment is 1,000. However, for publication, or if the need arises, 10,000 simulations or more may be performed. For work calculating critical values, I have seen 40,000 simulations or more. Usually however, moving from 1,000 to 10,000 simulations makes little difference.[4]

4 What Do I Even Want to Look For?

It is interesting to examine three properties of estimators when conducting Monte Carlo experiments.[5] These are not the only properties to look for, but they tend to be the most common. First, *bias* is a measure of how far off your estimator is from the true estimate: $|\theta - \hat{\theta} = Bias|$ is the absolute bias of the estimator. Second, *efficiency* is how large the variance is around your estimate. Usually this involves examining the measure of uncertainty around the estimate using standard errors.

[3] For more on this, search **help density functions** in Stata.

[4] We could even run a Monte Carlo to see this relationship between precision and computing cost!

[5] Much of this section comes from Carsey (2011).

There is almost always an inverse relationship between these two; an estimator that reduces bias, such as fixed effects, tends to be less efficient than its counterpart, random effects (and random effects will suffer larger bias). Third, simulations can examine *consistency*, or how well the estimator improves (less bias and more efficiency) as the sample size moves towards infinity. Although typically most users of statistical models assume that their model does not suffer from bias and is efficient (and consistent), Monte Carlo methods allow us to investigate these assumptions.

While the above estimator properties are of interest, so are more global properties of the model. One such output worth examining is the Mean Squared Error (MSE—or sometimes Root MSE). With it we can look at the overall performance of our model. In other words, this approach can examine both bias and efficiency of a single parameter estimate, $\hat{\theta}$:

$$\hat{\theta} = Bias(\hat{\theta}, \theta)^2 + Var(\hat{\theta}) \tag{6}$$

MSE has a few weaknesses. Outliers are weighted more due to the squared term. The Root-MSE may be helpful to reduce the influence of outliers. However, bias and inefficiency are still weighted the same—perhaps one matters more than the other? In any case, some other ways to evaluate the performance of a model may be information criterion (AIC/BIC), out of model forecasting, or the level of variance explained.

5 Monte Carlo in Stata

In the examples above we saw how we can create Monte Carlo simulations in R. Stata also makes performing Monte Carlo simulations rather easy through the use of the **simulate** command, which comes as a canned procedure. Another possible command is **postfile**, although that is less of a Monte Carlo command and more of a simple way of combining results we loop over into a single dataset for viewing (just like we did in R). Overall, I find that **simulate** works best if the Monte Carlo is relatively short, while if there are lots of parameters that we have to loop over **postfile** makes it easy to combine and see the output.

Stata and R technically are not drawing random numbers—only pseudo-random numbers can be generated in the program. However, this is probably good enough for us. In reality it is just a random starting point for a set sequence of numbers.

But, those who still need more randomness can go to Random.org. Their randomness generator, "comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs."

Since our output would be different if we re-ran a simulation, it is always a good idea to set the 'seed' at the beginning of a .do file. We did this above using the `set.seed()` command in R. This ensures that the sequence of random numbers Stata generates starts with the same starting values; we will get the same output every time if we set the seed, which is good, especially for replication. If the seed is not set, Stata uses the computer's clock to generate starting values, and you will get a (slightly) different result every time.

In addition, the number of simulations is important. As shown above, a common number is 1000 simulations when conducting a Monte Carlo experiment.

There are a number of typical steps for conducting a Monte Carlo experiment in Stata:

1. Decide on the test statistic that we want to compute (sample mean, sample regression parameters, etc...).

2. Decide on the True statistics from above: either population mean, or the true model parameters. Since we create the Data Generating Process, we know what the true values of the parameters are, along with any 'complexities' we may add (i.e. autocorrelation, heteroskedasticity, omitted variables, etc.). This is the step where we actually write the parametric model that generates the data, i.e. $y_i = 1 + 2 * x_{1i} + 4 * x_{2i} + \varepsilon$. Note that we are including both a systematic component (the constant, and the independent variables x_1 and x_2) as well as a stochastic component ε.[6]

3. Create a program to set up what one simulation trial looks like.

4. Run this first trial. This step is analogous to what we do in most of our applied work where we only have one sample. This is always a good idea to see if our program needs any tweaking.

5. Use the **simulate** command to simulate the number of trials (i.e. sample redraws) we want. If we create a flexible enough program in step 1 we can

[6] If we didn't add the stochastic component, there would be no inherent randomness, and we would precisely estimate θ every time.

also change parts of our experiment to run different types of **simulate** without having to go back and create a whole new program. For instance, we can write the program so Stata asks us to provide the number of observations in the sample for each Monte Carlo experiment. However, it is important to vary only one thing at a time. For instance, if we vary α from 0 to 10 and γ as 0 and 1, we need to simulate all values of α at $\gamma = 0$, and then another set of simulations across α with $\gamma = 1$. If we did not do this, we wouldn't know which movement was producing the changes we saw.

6. Finally we analyze the output. Typically this involves just looking at the mean of the simulation, and standard deviation. Using **sum** to examine summary statistics and **kdensity** for kernel-density plots are particularly helpful here. It may also involve examining precision, power, bias, and more complex hypotheses tests. **simsum** is a user-written program for looking at some of these more complex post-experiment commands.

6 Small Tips to Remember

- Only vary one thing at a time! Try changing the value of a parameter while holding everything else constant, or create nested loops where you vary everything each of the loops.
- If varying a number of parameters/other things, remember that the number of Monte Carlo iterations performed increases very fast. For instance, running 1000 simulations on $t = 10, 20, 50$ means 3000 simulations are performed, since 1000 simulations are run on each value of t. Adding a error correlation, for instance, from $\varepsilon = 0, 0.2, 0.6, 0.8$ means that we now run 1000 simulations where $t = 10$ and $\varepsilon = 0$, 1000 where $t = 10$ and $\varepsilon = 0.2$, 1000 where $t = 10$ and $\varepsilon = 0.6$, and so on...this would result in 12,000 simulations total (1000*3*4=12,000).
- "Thinking is expensive, computers are cheap"
    1. Never program your own distribution/method when a canned procedure will do (also makes your code easier to read). The "randomness creator" is the key part of a Monte Carlo simulation and slight mistakes/differences from the correct distribution will produce wildly different results.

2. Computers have increased their computing power greatly over the past years...this makes simulating much easier. You can simulate even relatively complex Monte Carlos quickly on just a small laptop.
3. Parallel computing is growing in popularity, especially as more and more computers have faster, multiple processors. Basically given a repetitive task (simulations), these programs split the work up evenly between the processors, and at the end combine the data back into one long list. R has the "snow" library and Stata has the user-written "parallel" package. Both can quickly speed up the time it takes to run Monte Carlos.

- Think about how you would like to present your results, and construct your simulation around it. For instance, if you would like to show how your estimator behaves over different levels of observations, construct your Monte Carlo to vary the levels of n. Also be sure what you would like to present: a density function, the mean value of the simulations, the proportion of times a simulation falls above or below some value, and so on.

- Presentation of results are equally important. Often, I find that tables of results mask the substantive findings of the experiment, unless there are only a few entries (typically less than 10). If you are varying lots of things, it may make sense to graphically present the result. For instance, Figure 2 shows the results of a Monte Carlo experiment of a working paper I have. The figure is comparing the rates of Type I error across various cointegration tests. Although there is a lot going on in Figure 2, a table that presented the same amount of information would have had 528 entries (3 levels of observation, 4 combinations of independent variables, and 11 values of autocorrelation, ϕ^x, across 4 different unit root tests)! Often with Monte Carlo simulations you can paint a much clearer story using figures instead of tables.

- Make sure to run one simulation just to make sure your program is performing as you want. Monte Carlos can quickly become complex programming tasks and running a single simulation to check that the program is working is much better than starting the simulation and finding out you have a mistake after it is complete!

- Loops make everything happen, especially when manipulating multiple factors in your simulation.

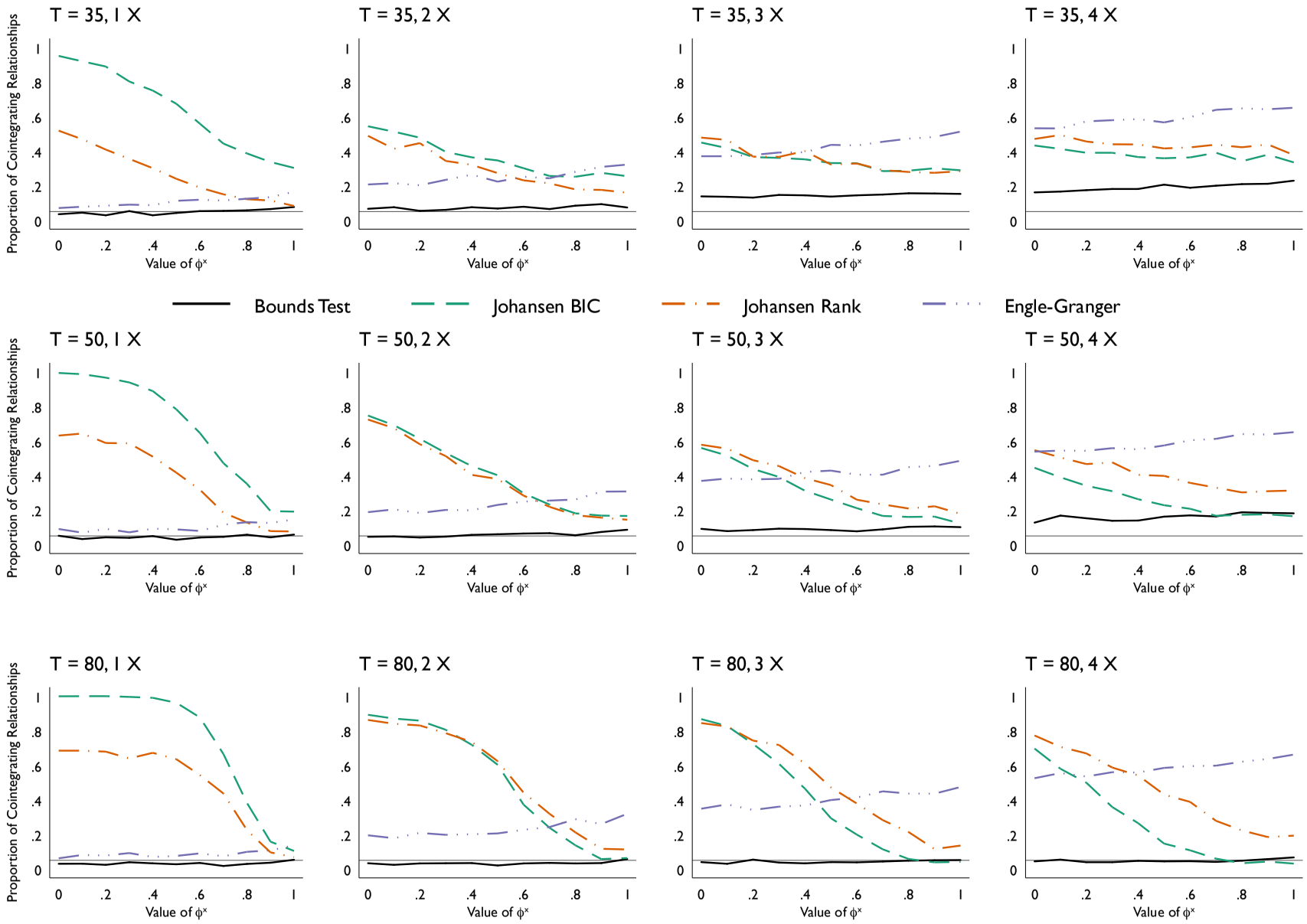


Figure 2: Proportion of Monte Carlo Simulations (falsely) Concluding Cointegration Across Various Methods

Note: From Philips (2016). Each plot shows the proportion of simulations finding (at $p < 0.05$) evidence of one cointegrating relationship with up to k regressors and different numbers of observations across varying levels of autoregression in x_{1t}, using each of the four cointegration testing procedures.

- There is lots of code and help online. See references for more information. Here are some free online help sources: For **R**, see Robert and Casella's presentation to accompany their 2009 book (which is in the references). Also see Carsey (2011). In Stata, look at Baum's 2007 presentation. Another Stata .pdf with pretty good code can be found at the Learn Econometrics website

- Monte Carlo methods are used in other areas too (i.e. Bayesian Markov Chain Monte Carlo methods, machine learning...basically anywhere where closed-form solutions are computationally intractable).[7]

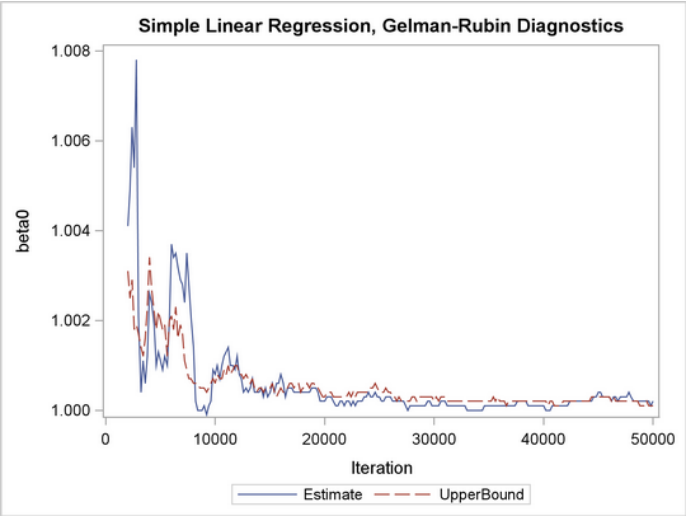


Figure 3: A Markov Chain Monte Carlo Converging on a Stable Estimate

7 Examples

On the download page of my website (found here) there are a number of examples of Monte Carlos in both R and Stata. The first example, Monte Carlo 1, is an example of the Central Limit Theorem from Cameron and Trivedi. In the same files, there is also a simulation that looks at the performance of OLS, Prais-Winsten GLS, and OLS with a lagged dependent variable under varying degrees of autocorrelation.

[7] figure source: https://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug

The next example, Monte Carlo 2, is a Stata file that 'breaks' an instrumental variable regression through two ways. First, we examine what having a weak instrument does to our estimates. Second, we make the instrument invalid by correlating it (over varying degrees) with the error. We explore a variety of ways of graphing these results too.

The third example, Monte Carlo 3, is a Stata file that examines the performance of the Zivot-Andrews test for a structural break under a unit root. The size of the intercept break is varied, and the test statistics are gathered and analyzed.

In another example, Monte Carlo 4, we examine the properties of panel unit root tests. The program allows us to create any number of panels that can be as long as we'd like. We can set the number of unit root series along with the level of autocorrelation in the stationary series...since this likely affects the performance of the tests too. A variety of unit root tests are examined across varying N panels and T time units.

NOTE: Some of these simulations take quite some time to run.

Last, for those interested in resampling methods, I have short Stata and R files covering the bootstrap and jack-knife procedures.

8 Conclusion

The purpose of this paper was to provide a gentle introduction to Monte Carlo methods using both Stata and R. Monte Carlo simulations are a powerful tool that anyone familiar with a bit of statistics can use.

References

Efron, Bradley and Robert J Tibshirani. 1994. *An introduction to the bootstrap*. CRC press.

Efron, Bradley and Robert Tibshirani. 1986. "Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy." *Statistical science* pp. 54–75.

Philips, Andrew Q. 2016. "Have your cake and eat it too: Cointegration and dynamic inference from autoregressive distributed lag models." Working Paper.

Robert, Christian P. and George Casella. 2010. *Introducing Monte Carlo Methods with R*. Springer.

