Introduction to Monte Carlo Simulations with Applications in R Using the `SimDesign` Package

Phil Chalmers

November 30, 2015

Overview

1. What are Monte Carlo simulations (MCSs), and why do we do them?
2. Meta-statistics to summarise MCS results
3. Hands-on coding of MCSs, and overview of the `SimDesign` package in R
4. Important considerations and principles to follow
5. Presenting results

What are Monte Carlo Simulations?

Numerical experiments using data randomly sampled from a selection of probability distributions.

- Given some design condition under investigation, generate some data, analyse said data, repeat R times, and summarise the obtained results
- Requires computers to perform these experiments, usually with a good general purpose/statistical programming language (R, SAS, python, C++)
- General setup is *embarrassingly parallel*. Makes these experiments ideal for modern super-computers, user-built Beowulf clusters, or even computers on completely different networks

Why should we care?

Rationale:

- Regarding parameter estimates, sampling properties must be established so that they can be used with confidence in empirical data
  - Furthermore, estimators may perform better than others in different settings
- Small sample properties may not be known (even when large-sample asymptotics are)
- Analytic results may not be possible (e.g., sampling distribution of $\frac{1}{n}\sum_{i=1}^{n}|x_i - \bar{x}|$?)
- Assumptions are required for *all* models . . . but what happens when these assumptions are violated?

Typical issues investigated in MCSs

- Is an estimator **biased**? What are its sampling properties (i.e., **standard error**)?
- How do different estimators compare in terms of **efficiency**?
- Is an estimator **consistent**/**robust** when assumptions are violated?
- Do the confidence intervals demonstrate **nominal coverage rates**?
- Given some nominal detection rate (α), how **powerful** is a test statistic when the null hypothesis is false; does it retain the null the appropriate proportion of times when the null is true (**Type I error**)?

Reasoning of MCSs

For population parameter ψ (given some pre-specified conditions), let $\hat{\psi} = f(D)$ be a sample estimate given some randomly sampled dataset (D).

- Say that we had a number of randomly sampled datasets from this population. For each dataset, we could compute $\hat{\psi}_1 = f(D_1)$, $\hat{\psi}_2 = f(D_2)$, ..., $\hat{\psi}_R = f(D_R)$.
- Under the Central Limit Theorem, as the number of random samples $R \to \infty$ then $\psi \equiv E[f(D)]$.
- Therefore, given our finite datasets we could use the approximation $\psi \approx \frac{1}{R} \sum_{r=1}^{R} f(D_r)$ when R is large enough.

Reasoning of MCSs

To use this property of the Central Limit Theorem in MCSs, if we were to randomly generate datasets with the desired empirical properties (often implied by the model we want to analyse it with) then $\hat{\psi}_r = f(D_r)$ would be a single instance from the true sampling distribution of ψ.

- Repeating the data generation process to obtain D_r many times, and collecting the results in the form $\hat{\psi}_r = f(D_r)$, will give us a random sample of the true sampling distribution of the statistic.
- From here, we can make inferences about the properties of the statistic by analysing different properties of this collected set of estimates (mean, standard deviation, quartiles, etc).

Simple Example

Question

How does trimming the sample mean affect estimation of the population mean (μ) when the data are drawn from a Gaussian distribution? Investigate using different sample sizes.

Simple Example (generate + analyse)

```
R <- 1000
mu <- 1
N <- c(30, 90, 270)
mat <- matrix(0, R, 3)
colnames(mat) <- c('mean', 'trim.1', 'trim.2')
res <- list('30'=mat, '90'=mat, '270'=mat)

for(r in 1L:R){
    for(n in N){
        dat <- rnorm(n, mean = mu, sd = 1)
        M0 <- mean(dat)
        M1 <- mean(dat, trim = .1)
        M2 <- mean(dat, trim = .2)
        res[[as.character(n)]][r, ] <-
            c(M0, M1, M2)
    } #end n
} #end r
```

Sample of estimator distributions

```
head(res[['30']])
```

```
##         mean trim.1 trim.2
## [1,] 1.082  1.163  1.195
## [2,] 1.235  1.207  1.172
## [3,] 0.830  0.890  0.895
## [4,] 1.149  1.107  1.054
## [5,] 0.938  0.983  1.044
## [6,] 1.011  1.007  0.985
```

Sample of estimator distributions

```
head(res[['270']])
```

```
##          mean trim.1 trim.2
## [1,] 0.993  0.998  1.004
## [2,] 0.918  0.921  0.917
## [3,] 0.925  0.933  0.949
## [4,] 1.006  1.001  0.985
## [5,] 0.969  0.977  0.988
## [6,] 0.997  0.992  0.968
```

Sample of estimator distributions

Recall that $\mu = 1$ in this simulation.

```
sapply(res, colMeans)
```

```
##              30 90    270
## mean    0.996  1 0.999
## trim.1 0.996  1 0.999
## trim.2 0.996  1 0.999
```

Standard Error

From Wikipedia: *The standard error (SE) is the standard deviation of the sampling distribution of a statistic, most commonly of the mean.*

- After collecting our estimates we have successfully obtained an unbiased sample of the true sampling distribution.
- Hence, the standard deviation of the collection of sample estimates is an estimate of the *standard error*!

Simple Example (summarise)

```
sapply(res, function(x) apply(x, 2, sd))
```

```
##              30     90     270
## mean    0.182 0.105 0.0611
## trim.1 0.187 0.109 0.0631
## trim.2 0.195 0.115 0.0658
```

Recall that for the mean estimator (no trimming), the sampling error is $SE = \sigma/\sqrt{N}$.

```
sapply(N, function(n, SD = 1) SD / sqrt(n))
```

```
## [1] 0.1826 0.1054 0.0609
```

Flow of MCSs

All MCSs follow the same work-flow:

- **generate** data from some probability sampling distribution (implied by some model) given a fixed set of **design** parameters/conditions,
- **analyse** these data to obtain information about the statistical estimators, detection rates, confidence intervals, etc,
- Repeat the **generate** and **analyse** operations a number of times and collect their results, and **summarise** this information using relevant meta-statistics.

Summarising results

What we really want is some way to compare different estimators using standard methods.

bias : Good estimators should be unbiased. Bias estimates therefore should be close to 0.

$$bias = \frac{1}{R}\sum_{r=1}^{R}(\hat{\psi}_r - \psi)$$

root-mean square error (**RMSE**). Good estimators should demonstrate *minimal sampling error* when recovering the population values; the less sampling error, the better.

$$RMSE = \sqrt{\frac{1}{R}\sum_{r=1}^{R}(\hat{\psi}_r - \psi)^2}$$

$$MSE \equiv RMSE^2$$

Summarising results

Note that $RMSE \geq SD$. Although SD is equivalent to the standard error (SE), it doesn't account for the potential *bias* in the parameter estimates.

- Recall that $SE = SD = \sqrt{\frac{1}{R}\sum_{r=1}^{R}(\hat{\psi}_r - \bar{\psi})^2}$, where $\bar{\psi}$ is used instead of ψ. Hence, $RSME \equiv SE$ iff $bias \equiv 0$.
- Bias contributes to population recovery accuracy with the relationship $RMSE = \sqrt{bias^2 + SE^2}$
- Therefore, $RMSE$ captures information about how variable the sampling variability *as well as* how much the bias negatively contributes to the accuracy of recovering population parameters.

Simple Example (summarise)

```
sapply(res, function(x, pop) colMeans(x - pop),
       pop = mu)
```

```
##                30       90       270
## mean    -0.00409 0.00299 -0.001445
## trim.1  -0.00380 0.00237 -0.000814
## trim.2  -0.00356 0.00275 -0.000553
```

```
(RMSE <- sapply(res, function(x, pop)
    sqrt(colMeans((x - pop)^2)), pop = mu))
```

```
##              30    90    270
## mean    0.182 0.105 0.0611
## trim.1  0.187 0.109 0.0630
## trim.2  0.195 0.115 0.0658
```

Summarising results

For unbiased estimators we can also find the **relative efficiency** of the estimators by simply forming ratios between the MSE values (treating one estimator as the reference).

$$RE_i = (RMSE_i / RMSE_1)^2$$

$$RE_i = MSE_i / MSE_1$$

Naturally, when $i = 1$ then $RE \equiv 1$. Values greater than 1 indicate less efficiency than the reference, while values less than 1 indicate more.

Simple Example

```
RE <- t((t(RMSE) / RMSE[1,])^2)
RE
```

```
##             30   90  270
## mean      1.00 1.00 1.00
## trim.1    1.05 1.08 1.07
## trim.2    1.14 1.20 1.16
```

Coverage, Type I Errors, and Power

Other types of MCSs are possible that don't necessarily focus on parameter recovery per say.

- For instance, we may be interested in whether a proposed *confidence interval* contains the population parameter at some advertised rate $(1 - \alpha)$. This is commonly referred to as **coverage**.

```
# 95% confidence interval coverage
CIs <- matrix(0, 10000, 2)
for(i in 1:10000)
    CIs[i,] <- t.test(rnorm(100, mean = 2.5))$conf.int
1 - mean(CIs[,1] > 2.5 | CIs[,2] < 2.5)
```

```
## [1] 0.949
```

Coverage, Type I Errors, and Power

Analogously, we may be interested in whether *p*-values return the correct detection rates.

- **Type I Errors**: Reject the Null hypothesis when it is *true*. This should happen at a rate of α.

```
ps <- numeric(10000); alpha <- .05
for(i in 1:10000) ps[i] <- t.test(rnorm(100))$p.value
mean(ps < alpha)
```

```
## [1] 0.0476
```

- **Power**: Reject the Null hypothesis when it is *false* (given α). Higher rate $=$ better.

```
for(i in 1:10000) ps[i] <- t.test(rnorm(100, mean=0.2))$p.value
mean(ps < alpha)
```

```
## [1] 0.506
```

Organizing Monte Carlo Simulations

Organizing Monte Carlo Simulations

This area is often taken for granted. If the design of the MCS is not well thought out beforehand then you may find yourself in a world of headaches and remorse....

Thinking A Little More Clearly

Previous slides are enough to help you understand 99% of the simulation work out there, and technically are enough for you to write your own simulations. That's great, but....

- DON'T DO IT THIS WAY!
- There are better, more organized, less error prone ways.
- Let's start with the three main F-ing problems with the previous coding approach:
  - for-loops
  - functions
  - features

I Got 99 Problems and a For-Loop Makes 100, 101, 102, ...

The `for-loop` philosophy is generally how one approaches MCSs in general purpose programming languages (at least at first). This leads to some pretty annoying problems:

- Loops can be rearranged, and may be deeeeeeeeeeply nested
- Objects can be accidentally rewritten in larger programs (e.g., `n <- 10 ... for(n in 1:N)`)
- Objects don't have to be standard (could be list, matrices, vectors, etc). This can create confusion
- Notation is non-standard (*what the heck does n mean?!?!*). Requires extra documentation
- Code is generally harder to read and keep organized
- Hard to make extensible (e.g., add more distributional shapes to the example)

Simple Example Code (Again)

```
R <- 1000
mu <- 1
N <- c(30, 90, 270)
mat <- matrix(0, R, 3)
colnames(mat) <- c('mean', 'trim.1', 'trim.2')
res <- list('30'=mat, '90'=mat, '270'=mat)

for(r in 1L:R){
    for(n in N){
        dat <- rnorm(n, mean = mu, sd = 1)
        M0 <- mean(dat)
        M1 <- mean(dat, trim = .1)
        M2 <- mean(dat, trim = .2)
        res[[as.character(n)]][r, ] <-
            c(M0, M1, M2)
    } #end n
} #end r
```

(Less) Simple Example Code

```
R <- 1000; mu <- 1
dist <- c('norm', 'chi')
N <- c(30, 90, 270); mat <- matrix(0, R, 3)
colnames(mat) <- c('mean', 'trim.1', 'trim.2')
tmp <- list('norm'=mat, 'chi'=mat)
res <- list('30'=tmp, '90'=tmp, '270'=tmp)

for(r in 1L:R){
    for(d in dist){
        for(n in N){
            dat <- if(d == 'norm') rnorm(n, mean=mu)
                else rchisq(n, df = mu)
            M0 <- mean(dat)
            M1 <- mean(dat, trim = .1)
            M2 <- mean(dat, trim = .2)
            res[[as.character(n)]][[d]][r, ] <-
                c(M0, M1, M2)
        } #end n
    } #end d
} #end r
```

(Less) Simple Example Code

```
head(res[['270']][['chi']])
```

```
##         mean trim.1 trim.2
## [1,] 0.909  0.659  0.571
## [2,] 1.030  0.725  0.627
## [3,] 1.036  0.751  0.605
## [4,] 1.098  0.722  0.614
## [5,] 0.892  0.645  0.504
## [6,] 1.125  0.757  0.598
```

Functions to Avoid Hard-to-find Errors

- All of the meta-statistics previously described could be explicitly coded each time they are required. However, this is unintuitive and very error prone.
- Instead, `functions` should be written and utilized, such as `bias()`, `RMSE()`, empirical detection rates/coverage, and so on. These should be recycled and reused once they are well tested.
- In order to do this though, data should be kept in an easily accessible and predictable form (not nested-lists)

Features Which Should be Considered

- Saving/resuming temporary simulation state in case of power-outages/crashes
- Automatically re-drawing data when analysis functions fail (but tracking how functions failed)
- Outputting files to completely save analysis results (and making sure to do this uniquely)
- Making efficient use of RAM (e.g., discarding data objects, avoiding large empty storage objects)
- Parallel computation support, including support on larger clusters or across independent nodes/computers
- Clear and easy debugging (for-loops make it difficult to debug anything due to littered workspace)

... enter the `SimDesign` package.

The SimDesign package

SimDesign

Provides tools to help organize Monte Carlo simulations in R. The tools provided control the structure and back-end of the Monte Carlo simulations by utilizing a generate-analyse-summarise strategy. The functions control common simulation issues such as re-simulating non-convergent results, support parallel back-end computations, save and restore temporary files, aggregate results across independent nodes, and provide native support for debugging.

The `SimDesign` package

Philosophy of the `SimDesign` package is to *explicitly* follow the work-flow:

- Given some **design** conditions

$$(\textit{generate} \rightarrow \textit{analyse})^R \rightarrow \textit{summarise}$$

In the lingo of R, **design** is some pre-defined object to be indexed and acted upon while **three functions** are used and recycled over and over again to complete the MCS steps.

The SimDesign package

```
Design <- data.frame(...)

Generate <- function(...) ...
Analyse <- function(...) ...
Summarise <- function(...) ...

results <- runSimulation(design = Design,
                         replications = 1000,
                         generate = Generate,
                         analyse = Analyse,
                         summerise = Summarise)
```

The `SimDesign` package

`SimDesign` requires

- **design** is a `data.frame` with the relevant simulations conditions to be investigated in each row, and each major MCS condition to be in each column (e.g., sample size, effect size, distribution, etc),
- `generate()`, `analyse()`, and `summarise()` are all user-defined R functions (with a specific set of inputs and required output properties), and
- passing all objects, functions, and optional arguments to `runSimulation()` to control the MCS flow and features

The SimDesign package

The SimDesign package contains a selection of convenience functions as well.

- bias(), RMSE(), and RE(): bias, RMSE, and RE. Accepts statistics in deviation form if that happens to be easier.
- ECR() and EDR(): empirical coverage and detection rates.

The SimDesign package

One important feature in SimDesign is that the internal functions are wrapped within try() calls automatically. Hence, if an routine fails within any of the respective functions (i.e., a stop() call is thrown) then SimDesign will silently deal with the error while also tracking what the error message was.

- Error messages are appended into the returned data.frame object with the number of times they occurred
- Users may throw their own stop() calls if useful (e.g., when a model converges but not before the maximum number of iterations were reached)
- SimDesign has built-in safety termination when more than 50 consecutive errors are thrown. This avoids simulations getting stuck in infinite loops, even when running code across different cores

The SimDesign package

Thankfully, the package allows some initial hand-holding to get you started. Simply run one of the following commands after loading the package:

```
library(SimDesign)
SimFunctions()
SimFunctions('my-awesome-simulation')
```

The SimDesign package

SimDesign

Let's explore how the previously described simulations can be performed with SimDesign instead of using for-loops.

General simulation considerations

- Design your simulation experiment like you would design a real-world experiment. Use thought over bruit force
- Ensure that the description of your simulation is *clear* and *reproducable*. Others should be able to replicate your design
- Throw errors early and often, and run analysis functions that are known to take the most time/are the most likely to fail first. You don't want to do extra computations if you don't need to!

General simulation considerations

- Each combination of conditions you are interested in grows exponentially. So be careful not to choose too many, especially if you can reason that they will be irrelevant a priori
- That being said, don't just choose conditions that you think will be favorable. . . .you may be pleasantly surprised!

Writing Principles

1. Start testing your simulation with a small number of replications/conditions to make sure it runs through correctly. Increase once the majority of the bugs have been worked out
2. Document strange aspects of your simulation/code that you are likely to forget about
3. Save the state of your simulation! Helps to resume later in case something goes wrong, but also gives you something to track for longer simulations (`runSimulation(..., save = TRUE)`)
4. Useful to output your results as well (`runSimulation(..., save_results = TRUE)`), especially for debugging purposes

Presenting results

- Present your results as if you were dealing with a real-world experiment
- Check for interaction effects via ANOVA/regression methods, and look at effect sizes for an indication of what conditions are the most influential in the results
- Use figures (box-plots, scatter plots, power curves), condensed/marginalized tables (with rounded numbers), descriptions, etc, to highlight the important aspects of your findings
  - Table-plots can be useful here as a nice hybrid

Conclusion

MCSs are important to quantitative researchers, so it's important to understand how they work and how to construct them.

- *Practice makes perfect*. Make trial runs of your code to make sure it works before doing anything serious. The more often you write, the better you will get at it
  - Use of version control systems like `git` are useful here
- *Be critical*. Evaluate the simulation as if it were someone else's. What would you criticize about your design/presentation? Is the purpose of the simulation obvious?

Conclusion

- *Have fun and explore*. This is one of the few areas where exploring can be fun and rewarding. Even if you don't have a simulation study in mind, write about topics:
  - You've had trouble grasping,
  - Have always wondered about,
  - Are skeptical of.

Further Information

- Gentle, J. E. (1985). *Monte Carlo methods*. In S. Kotz & N. L. Johnson (Eds.), The encyclopedia of statistical sciences (Vol. 5, pp. 612-617). New York: Wiley.
- Paxton, P., Curran, P., Bollen, K. A., Kirby, J., & Chen, F. (2001). Monte Carlo Experiments: Design and Implementation. *Structural Equation Modeling, 8*, 287-312.
- Mooney, C. Z. (1997). *Monte Carlo simulation*. Thousand Oaks, CA: Sage.

The SimDesign package

- `aggregate_simulations()`: when saving results on independent nodes (using only a fraction of the total `replication` number) this function aggregates the saved files into one weighted simulation result.

```
runSimulation(..., replications = 500,
              filename = 'file1')
runSimulation(..., replications = 500,
              filename = 'file2')
# reads-in .rds files in working directory
Full <- aggregate_simulations()
```

For nodes linked on a LAN network (i.e., a Beowulf cluster) then the setup simply requires passing the argument `MPI = TRUE`. See the example in `help(runSimulation)`.