Simulation

November 5, 2013

References:

- Gentle: Computational Statistics
- Monahan: Numerical Methods of Statistics

Many (most?) statistical papers include a simulation (i.e., Monte Carlo) study. The basic idea is that closed form analysis of the properties of a statistical method/model is often hard to do. Even if possible, it usually involves approximations or simplifications. A canonical situation is that we have an asymptotic result and we want to know what happens in finite samples, but often we do not even have the asymptotic result. Instead, we can estimate mathematical expressions using random numbers. So we design a simulation study to evaluate the method/model or compare multiple methods. The result is that the statistician carries out an experiment, generally varying different factors to see what has an effect on the outcome of interest.

The basic strategy generally involves simulating data and then using the method(s) on the simulated data, summarizing the results to assess/compare the method(s).

Most simulation studies aim to approximate an integral, generally an expected value (mean, bias, variance, MSE, probability, etc.). In low dimensions, methods such as Gaussian quadrature are best for estimating an integral but these methods don't scale well [we'll discuss this in the next unit on integration/differentiation], so in higher dimensions we often use Monte Carlo techniques.

1 Monte Carlo considerations

1.1 Monte Carlo basics

The basic idea is that we often want to estimate $\mu \equiv E_f(h(X))$ for $X \sim f$. Note that if h is an indicator function, this includes estimation of probabilities, e.g., $p = P(X \leq x) = F(x) =$

$\int_{-\infty}^{x} f(t)dt = \int I(t \leq x)f(t)dt = E_f(I(X \leq x))$. We would estimate variances or MSEs by having h involve squared terms.

We get an MC estimate of μ based on an iid sample of a large number of values of X from f:

$$\hat{\mu} = \frac{1}{m}\sum_{i=1}^{m} h(X_i),$$

which is justified by the Law of Large Numbers. The simulation variance of $\hat{\mu}$ is $\text{Var}(\hat{\mu}) = \sigma^2/m$, with $\sigma^2 = \text{Var}(h(X))$. An estimator of $\sigma^2 = E_f((h(X) - \mu)^2)$ is $\hat{\sigma}^2 = \frac{1}{m-1}\sum(h(X_i) - \hat{\mu})^2$. So our MC simulation error is based on

$$\widehat{\text{Var}}(\hat{\mu}) = \frac{1}{m(m-1)}\sum_{i=1}^{m}(h(X_i) - \hat{\mu})^2.$$

Sometimes the X_i are generated in a dependent fashion (e.g., sequential MC or MCMC), in which case this variance estimator does not hold because the samples are not IID, but the estimator $\hat{\mu}$ is still correct. [As a sidenote, a common misperception with MCMC is that you should thin your chains because of dependence of the samples. This is not correct - the only reason to thin a chain is if you want to save on computer storage or processing.]

Note that in this case the randomness in the system is very well-defined (as it is in survey sampling, but unlike in most other applications of statistics), because it comes from the RNG that we perform as part of our attempt to estimate μ.

Happily, we are in control of m, so in principle we can reduce the simulation error to as little as we desire. Unhappily, as usual, the standard error goes down with the square root of m.

1.2 Simple example

Suppose we've come up with a fabulous new estimator for the mean of a distribution. The estimator is to take the middle value of the sorted observations as our estimate of the mean of the entire distribution. We work out some theory to show that this estimator is robust to outlying observations and we come up with a snazzy new name for our estimator. We call it the 'median'. Let's denote it as $\tilde{X}$.

Unfortunately, we have no good way of estimating $\text{Var}(\tilde{X}) = E((\tilde{X} - E(\tilde{X}))^2)$ analytically. We decide to use a Monte Carlo estimate

$$\frac{1}{m}\sum_{i=1}^{m}(\tilde{X}_i - \bar{\tilde{X}})^2$$

where $\bar{\tilde{X}} = \frac{1}{m}\sum \tilde{X}_i$. Each $\tilde{X}_i$ in this case is generated by generating a dataset and calculating the

median. In evaluating the variance of the median and comparing it to our standard estimator, the sample mean, what decisions do we have to make in our Monte Carlo procedure?

1.3 Variance reduction

There are some tools for variance reduction in MC settings. One is importance sampling (see Section 3). Others are the use of control variates and antithetic sampling. I haven't personally run across these latter in practice, so I'm not sure how widely used they are and won't go into them here.

In some cases we can set up natural strata, for which we know the probability of being in each stratum. Then we would estimate μ for each stratum and combine the estimates based on the probabilities. The intuition is that we remove the variability in sampling amongst the strata from our simulation.

Another strategy that comes up in MCMC contexts is *Rao-Blackwellization*. Suppose we want to know $E(h(X))$ where $X = \{X_1, X_2\}$. Iterated expectation tells us that $E(h(X)) = E(E(h(X)|X_2)$. If we can compute $E(h(X)|X_2) = \int h(x_1, x_2) f(x_1|x_2) dx_1$ then we should avoid introducing stochasticity related to the X_1 draw (since we can analytically integrate over that) and only average over stochasticity from the X_2 draw by estimating $E_{X_2}(E(h(X)|X_2)$. The estimator is

$$\hat{\mu}_{RB} = \frac{1}{m} \sum_{i=1}^{m} E(h(X)|X_{2,i})$$

where we either draw from the marginal distribution of X_2, or equivalently, draw X, but only use X_2. Our MC estimator averages over the simulated values of X_2. This is called Rao-Blackwellization because it relates to the idea of conditioning on a sufficient statistic. It has lower variance because the variance of each term in the sum of the Rao-Blackwellized estimator is Var$(E(h(X)|X_2)$, which is less than the variance in the usual MC estimator, Var$(h(X))$, based on the usual iterated variance formula: $V(X) = E(V(X|Y)) + V(E(X|Y)) \Rightarrow V(E(X|Y)) < V(X)$.

2 Random number generation (RNG)

At the core of simulations is the ability to generate random numbers, and based on that, random variables. On a computer, our goal is to generate sequences of pseudo-random numbers that behave like random numbers but are replicable. The reason that replicability is important is so that we can reproduce the simulation.

2.1 Generating random uniforms on a computer

Generating a sequence of random standard uniforms is the basis for all generation of random variables, since random uniforms (either a single one or more than one) can be used to generate values from other distributions. Most random numbers on a computer are pseudo-random. The numbers are chosen from a deterministic stream of numbers that behave like random numbers but are actually a finite sequence (recall that both integers and real numbers on a computer are actually discrete and there are finitely many distinct values), so it's actually possible to get repeats. The seed of a RNG is the place within that sequence where you start to use the pseudo-random numbers.

Many RNG methods are sequential congruential methods. The basic idea is that the next value is

$$u_k = f(u_{k-1}, \ldots, u_{k-j}) \text{mod } m$$

for some function, f, and some positive integer m . Often $j = 1$. *mod* just means to take the remainder after dividing by m. One then generates the random standard uniform value as u_k/m, which by construction is in $[0, 1]$.

Given the construction, such sequences are periodic if the subsequence ever reappears, which is of course guaranteed because there is a finite number of possible values given that all the values are remainders of divisions by a fixed number . One key to a good random number generator (RNG) is to have a very long period.

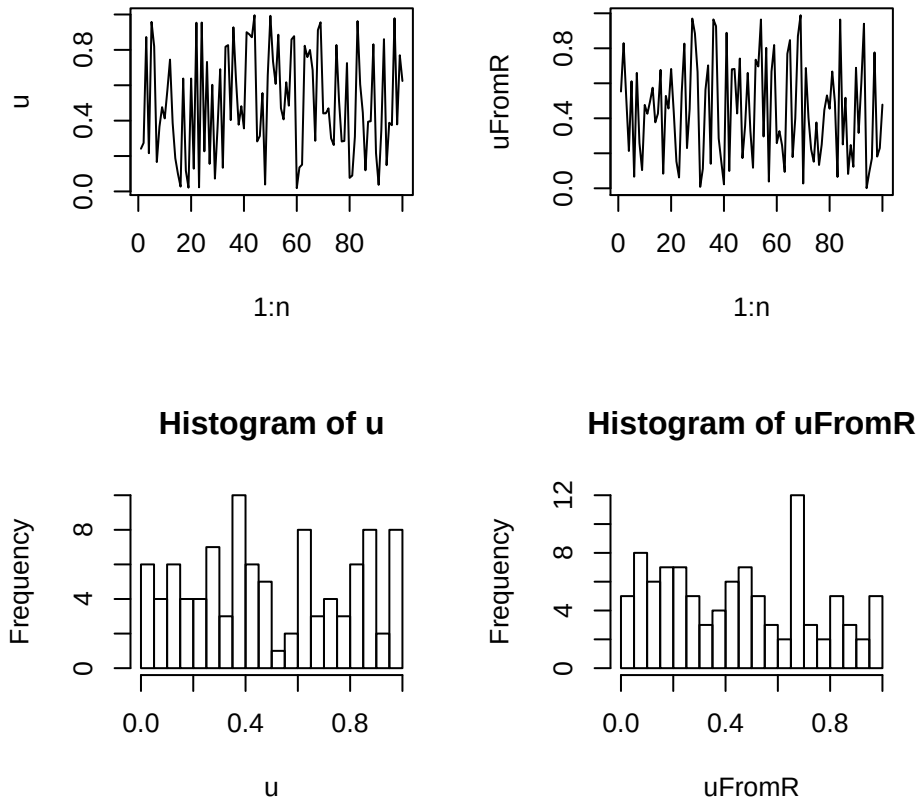
An example of a sequential congruential method is a basic linear congruential generator:

$$u_k = (au_{k-1}) \text{mod } m$$

with integer a, m, and u_i values. Here the periodicity can't exceed $m - 1$ (the method is set up so that we never get $u_k = 0$ as this causes the algorithm to break), so we only have $m - 1$ possible values. The seed is the initial state, u_0 - i.e., the point in the sequence at which we start. By setting the seed you guarantee reproducibility since given a starting value, the sequence is deterministic. In general a and m are chosen to be large, but of course they can't be too large if they are to be represented as computer integers. The standard values of m are Mersenne primes, which have the form $2^p - 1$ (but these are not prime for all p), with $m = 2^{31} - 1$ common. Here's an example of a linear congruential sampler:

```
n <- 100
a <- 171
m <- 30269
u <- rep(NA, n)
u[1] <- 7306
```

```
for (i in 2:n) u[i] <- (a * u[i - 1])%%m
u <- u/m
uFromR <- runif(n)
par(mfrow = c(2, 2))
plot(1:n, u, type = "l")
plot(1:n, uFromR, type = "l")
hist(u, nclass = 25)
hist(uFromR, nclass = 25)
```



A wide variety of different RNG have been proposed. Many have turned out to have substantial defects based on tests designed to assess if the behavior of the RNG mimics true randomness. Some of the behavior we want to ensure is uniformity of each individual random deviate, independence of sequences of deviates, and multivariate uniformity of subsequences. One test of a RNG that many RNGs don't perform well on is to assess the properties of k-tuples - subsequences of length k, which should be independently distributed in the k-dimensional unit hypercube. Unfortunately, linear congruential methods produce values that lie on a simple lattice in k-space, i.e., the points

are not selected from q^k uniformly spaced points, where q is the the number of unique values. Instead, points often lie on parallel lines in the hypercube.

Combining generators can yield better generators. The Wichmann-Hill is an option in R and is a combination of three linear congruential generators with $a = \{171, 172, 170\}$, $m = \{30269, 30307, 30323\}$, and $u_i = (x_i/30269 + y_i/30307 + z_i/30323) \bmod 1$ where x, y, and z are generated from the three individual generators. Let's mimic the Wichmann-Hill manually:

```
RNGkind("Wichmann-Hill")
set.seed(0)
saveSeed <- .Random.seed
uFromR <- runif(10)
a <- c(171, 172, 170)
m <- c(30269, 30307, 30323)
xyz <- matrix(NA, nr = 10, nc = 3)
xyz[1, ] <- (a * saveSeed[2:4])%%m
for (i in 2:10) xyz[i, ] <- (a * xyz[i - 1, ])%%m
for (i in 1:10) print(c(uFromR[i], sum(xyz[i, ]/m)%%1))
```

```
## [1] 0.4626 0.4626
## [1] 0.2658 0.2658
## [1] 0.5772 0.5772
## [1] 0.5108 0.5108
## [1] 0.3376 0.3376
## [1] 0.3576 0.3576
## [1] 0.413 0.413
## [1] 0.1329 0.1329
## [1] 0.255 0.255
## [1] 0.9202 0.9202
```

```
# we should be able to recover the current value of the seed
xyz[10, ]
```

```
## [1] 20696  2593  4576
```

```
.Random.seed[2:4]
```

```
## [1] 20696  2593  4576
```

By default R uses something called the Mersenne twister, which is in the class of generalized feedback shift registers (GFSR). The basic idea of a GFSR is to come up with a deterministic generator of bits (i.e., a way to generate sequences of 0s and 1s), B_i, $i = 1, 2, 3, \ldots$. The pseudo-random numbers are then determined as sequential subsequences of length L from $\{B_i\}$, considered as a base-2 number and dividing by 2^L to get a number in $(0, 1)$. In general the sequence of bits is generated by taking B_i to be the *exclusive or* [i.e., 0+0 = 0; 0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 0] summation of two previous bits further back in the sequence where the lengths of the lags are carefully chosen.

Additional notes Generators should give you the same sequence of random numbers, starting at a given seed, whether you ask for a bunch of numbers at once, or sequentially ask for individual numbers.

When one invokes a RNG without a seed, they generally have a method for choosing a seed, often based on the system clock.

There have been some attempts to generate truly random numbers based on physical randomness. One that is based on quantum physics is http://www.idquantique.com/true-random-number-generator/quantis-usb-pcie-pci.html. Another approach is based on lava lamps!

2.2 RNG in R

We can change the RNG in R using *RNGkind()*. We can set the seed with *set.seed()*. The seed is stored in *.Random.seed*. The first element indicates the type of RNG (and the type of normal RV generator). The remaining values are specific to the RNG. In the demo code, we've seen that for Wichmann-Hill, the remaining three numbers are the current values of $\{x, y, z\}$.

In R the default RNG is the Mersenne twister (`?RNGkind`), which is considered to be state-of-the-art – it has some theoretical support, has performed reasonably on standard tests of pseudorandom numbers and has been used without evidence of serious failure. Plus it's fast (because bitwise operations are fast). In fact this points out one of the nice features of R, which is that for something as important as this, the default is generally carefully chosen by R's developers. The particular Mersenne twister used has a periodicity of $2^{19937} - 1 \approx 10^{6000}$. Practically speaking this means that if we generated one random uniform per nanosecond for 10 billion years, then we would generate 10^{25} numbers, well short of the period. So we don't need to worry about the periodicity! The seed for the Mersenne twister is a set of 624 32-bit integers plus a position in the set, where the position is `.Random.seed[2]`.

We can set the seed by passing an integer to *set.seed()*, which then sets as many actual seeds as required for a given generator. Here I'll refer to the integer passed to *set.seed()* as *the* seed. Ideally, nearby seeds generally should not correspond to getting sequences from the stream that are

closer to each other than far away seeds. According to Gentle (CS, p. 327) the input to *set.seed()* should be an integer, $i \in \{0, \ldots, 1023\}$, and each of these 1024 values produces positions in the RNG sequence that are "far away" from each other. I don't see any mention of this in the R documentation for *set.seed()* and furthermore, you can pass integers larger than 1023 to *set.seed()*, so I'm not sure how much to trust Gentle's claim. More on generating parallel streams of random numbers below.

So we get replicability by setting the seed to a specific value at the beginning of our simulation. We can then set the seed to that same value when we want to replicate the simulation.

```
set.seed(0)
rnorm(10)

##  [1] -0.09400  0.19476 -0.41913 -0.21971 -0.65887 -0.55566  0.08172
##  [8]  0.20599  0.97703 -0.07111

set.seed(0)
rnorm(10)

##  [1] -0.09400  0.19476 -0.41913 -0.21971 -0.65887 -0.55566  0.08172
##  [8]  0.20599  0.97703 -0.07111
```

We can also save the state of the RNG and pick up where we left off. So this code will pick where you had left off, ignoring what happened in between saving to *savedSeed* and resetting.

```
set.seed(0)
rnorm(5)

## [1] -0.0940  0.1948 -0.4191 -0.2197 -0.6589

savedSeed <- .Random.seed
tmp <- sample(1:50, 2000, replace = TRUE)
.Random.seed <- savedSeed
rnorm(5)

## [1] -0.55566  0.08172  0.20599  0.97703 -0.07111
```

In some cases you might want to reset the seed upon exit from a function so that a user's random number stream is unaffected:

```
f = function(args) {
    oldseed <- .Random.seed
    # other code
    .Random.seed <<- oldseed
}
```

Note the need to reassign to the global variable *.Random.seed*.

RNGversion() allows you to revert to RNG from previous versions of R, which is very helpful for reproducibility.

The RNGs in R generally return 32-bit (4-byte) integers converted to doubles, so there are at most 2^{32} distinct values. This means you could get duplicated values in long runs, but this does not violate the comment about the periodicity because the two values after the two duplicated numbers will not be duplicates of each other – note there is a distinction between the values as presented to the user and the values as generated by the RNG algorithm.

One way to proceed if you're using both R and C is to have C use the R RNG, using the *unif_rand* (corresponding to *runif()*) and *norm_rand* (corresponding to *rnorm()*) functions in the *Rmath* library. This way you have a consistent source of random numbers and don't need to worry about issues with RNG in C. If you call C from R, this should approach should also work; you could also generate all the random numbers you need in R and pass them to the C function.

Note that whenever a random number is generated, the software needs to retain information about what has been generated, so this is an example where a function must have a side effect not observed by the user. R frowns upon this sort of thing, but it's necessary in this case.

2.3 Random slippage

If the exact floating point representations of a random number sequence differ, even in the 14th, 15th, 16th decimal places, if you run a simulation long enough, such a difference can be enough to change the result of some conditional calculation. Suppose your code involves:

```
> if(x>0) { } else{ }
```

As soon as a small difference changes the result of testing *x>0*, the remainder of the simulation can change entirely. This happened to me in my thesis as a result of the difference of an AMD and Intel processor, and took a while to figure out.

2.4 RNG in parallel

We can generally rely on the RNG in R to give a reasonable set of values. One time when we want to think harder is when doing work with RNG in parallel on multiple processors. The worst

thing that could happen is that one sets things up in such a way that every process is using the same sequence of random numbers. This could happen if you mistakenly set the same seed in each process, e.g., using *set.seed(mySeed)* in R on every process.

2.4.1 The naive approach

The naive approach is to use a different seed for each process. E.g., if your processes are numbered id=1,...,p, with *id* unique to a process, using `set.seed(id)` on each process. This is likely not to cause problems, but raises the danger that two (or more sequences) might overlap. For an algorithm with dependence on the full sequence, such as an MCMC, this probably won't cause big problems (though you likely wouldn't know if it did), but for something like simple simulation studies, some of your 'independent' samples could be exact replicates of a sample on another process.

2.4.2 The clunky but effective approach

One approach to avoid the problem is to do all your RNG on one process and distribute the random deviates to the other processes, but this can be infeasible with many random numbers.

2.4.3 The sophisticated approach

More generally to avoid this problem, the key is to use an algorithm that ensures sequences that do not overlap. In R, there are two packages that deal with this, *rlecuyer* and *rsprng*. We'll go over *rlecuyer*, as I've heard that rsprng is deprecated (though there is no evidence of this on CRAN) and *rsprng* is (at the moment) not available for the Mac.

The L'Ecuyer algorithm has a period of 2^{191}, which it divides into subsequences of length 2^{127}.

Here's how you initialize independent sequences on different processes when using the *parallel* package's parallel apply functionality (illustrated here with *parSapply()*.

```
require(parallel)

## Loading required package:  parallel

require(rlecuyer)

## Loading required package:  rlecuyer

nSims <- 250
testFun <- function(i) {
    val <- runif(1)
```

```
    return(val)
}

nCores <- 4
RNGkind()

## [1] "Wichmann-Hill" "Inversion"

cl <- makeCluster(nCores)
iseed <- 0
clusterSetRNGStream(cl = cl, iseed = iseed)
RNGkind()  # clusterSetRNGStream sets RNGkind as L'Ecuyer-CMRG

## [1] "Wichmann-Hill" "Inversion"

# but it doesn't show up here on the master
res <- parSapply(cl, 1:nSims, testFun)
clusterSetRNGStream(cl = cl, iseed = iseed)
res2 <- parSapply(cl, 1:nSims, testFun)
identical(res, res2)

## [1] TRUE

stopCluster(cl)
```

If you want to explicitly move from stream to stream, you can use *nextRNGStream()*. For example:

```
RNGkind("L'Ecuyer-CMRG")
seed <- 0
set.seed(seed)  ## now start M workers
s <- .Random.seed
for (i in 1:M) {
    s <- nextRNGStream(s)
    # now send s to worker i as .Random.seed
}
```

When using *mclapply()*, you can use the *mc.set.seed* argument as follows (note that *mc.set.seed* is TRUE by default, so you should get different seeds for the different processes by default), but one

needs to invoke `RNGkind("L'Ecuyer-CMRG")` to get independent streams via the L'Ecuyer algorithm.

```
require(parallel)
require(rlecuyer)
RNGkind("L'Ecuyer-CMRG")
res <- mclapply(seq_len(nSims), testFun,
mc.cores = nSlots, mc.set.seed = TRUE)
# this also seems to auotmatically reset the seed when it is run
res2 <- mclapply(seq_len(nSims), testFun,
mc.cores = nSlots, mc.set.seed = TRUE)
identical(res,res2)
```

The documentation for *mcparallel()* gives more information about reproducibility based on *mc.set.seed*.

With foreach

Getting independent streams One question is whether *foreach* deals with RNG correctly. This is not documented, but the developers (Revolution Analytics) are well aware of RNG issues. Digging into the underlying code reveals that the *doMC* and *doParallel* backends both invoke *mclapply()* and set *mc.set.seed* to TRUE by default. This suggests that the discussion above r.e. *mclapply()* holds for *foreach* as well, so you should do `RNGkind("L'Ecuyer-CMRG")` before your foreach call. For *doMPI*, as of version 0.2, you can do something like this, which uses L'Ecuyer behind the scenes:

```
cl <- makeCluster(nSlots)
setRngDoMPI(cl, seed=0)
```

Ensuring reproducibility While using *foreach* as just described should ensure that the streams on each worker are are distinct, it does not ensure reproducibility because task chunks may be assigned to workers differently in different runs and the substreams are specific to workers, not to tasks.

For *doMPI*, you can specify a different RNG substream for each task chunk in a way that ensures reproducibility. Basically you provide a list called *.options.mpi* as an argument to *foreach*, with *seed* as an element of the list:

```
nslaves <- 4
library(doMPI, quietly = TRUE)

##
## Attaching package:  'Rmpi'
## The following object is masked from 'package:rlecuyer':
##
##    .onUnload

cl <- startMPIcluster(nslaves)

##  4 slaves are spawned successfully. 0 failed.

registerDoMPI(cl)
result <- foreach(i = 1:20, .options.mpi = list(seed = 0)) %dopar% {
    out <- mean(rnorm(1000))
}
result2 <- foreach(i = 1:20, .options.mpi = list(seed = 0)) %dopar% {
    out <- mean(rnorm(1000))
}
identical(result, result2)

## [1] TRUE
```

That single seed then initializes the RNG for the first task, and subsequent tasks get separate substreams, using the L'Ecuyer algorithm, based on *nextRNGStream()*. Note that the *doMPI* developers also suggest using the *chunkSize* option (also specified as an element of *.options.mpi*) when using *seed*. See `?"doMPI-package"` for more details.

For other backends, such as *doParallel*, there is a package called *doRNG* that ensures that *foreach* loops are reproducible. Here's how you do it:

```
rm(result, result2)
nCores <- 4
library(doRNG, quietly = TRUE)

## Loading required package:  registry

library(doParallel)
registerDoParallel(nCores)
```

```
result <- foreach(i = 1:20, .options.RNG = 0) %dorng% {
    out <- mean(rnorm(1000))
}
result2 <- foreach(i = 1:20, .options.RNG = 0) %dorng% {
    out <- mean(rnorm(1000))
}

## Warning:  error 'Interrupted system call' in select
## Warning:  error 'Interrupted system call' in select

identical(result, result2)

## [1] TRUE
```

Alternatively, you can do:

```
rm(result, result2)
library(doRNG, quietly = TRUE)
library(doParallel)
registerDoParallel(nCores)
registerDoRNG(seed = 0)
result <- foreach(i = 1:20) %dopar% {
    out <- mean(rnorm(1000))
}
registerDoRNG(seed = 0)
result2 <- foreach(i = 1:20) %dopar% {
    out <- mean(rnorm(1000))
}
identical(result, result2)

## [1] TRUE
```

3 Generating random variables

There are a variety of methods for generating from common distributions (normal, gamma, beta, Poisson, t, etc.). Since these tend to be built into R and presumably use good algorithms, we won't go into them. A variety of statistical computing and Monte Carlo books describe the various

methods. Many are built on the relationships between different distributions - e.g., a beta random variable (RV) can be generated from two gamma RVs.

Also note that you can call the C functions that implement the R distribution functions as a library (*Rmath*), so if you're coding in C or another language, you should be able to make use of the standard functions: *{r,p,q,d}{norm,t,gamma,binom,pois,etc.}* (as well as a variety of other R math functions, which can be seen in *Rmath.h*). Phil Spector has a writeup on this (http://www.stat.berkeley.edu/classes/s243/rmath.html) and material can also be found in the *Writing R Extensions* manual on CRAN (section 6.16).

3.1 Multivariate distributions

The *mvtnorm* package supplies code for working with the density and CDF of multivariate normal and t distributions.

To generate a multivariate normal, we've seen the standard method based on the Cholesky decomposition:

```
U <- chol(covMat)
crossprod(U, nrow(covMat))
```

For a singular covariance matrix we can use the Cholesky with pivoting, setting as many rows to zero as the rank deficiency. Then when we generate the multivariate normals, they respect the constraints implicit in the rank deficiency.

3.2 Inverse CDF

Most of you know the inverse CDF method. To generate $X \sim F$ where F is a CDF and is an invertible function, first generate $Z \sim \mathcal{U}(0,1)$, then $x = F^{-1}(z)$. For discrete CDFs, one can work with a discretized version. For multivariate distributions, one can work with a univariate marginal and then a sequence of univariate conditionals: $f(x_1)f(x_2|x_1)\cdots f(x_k|x_{k-1},\ldots,x_1)$, when the distribution allows this analytic decomposition.

3.3 Rejection sampling

The basic idea of rejection sampling (RS) relies on the introduction of an auxiliary variable, u. Suppose $X \sim F$. Then we can write $f(x) = \int_0^{f(x)} du$. Thus X is the marginal density of X in the joint density, $(X,U) \sim \mathcal{U}\{(x,u) : 0 < u < f(x)\}$. Now we'd like to use this in a way that relies only on evaluating $f(x)$ without having to draw from f.

To implement this we draw from a larger set and then only keep draws for which $u < f(x)$. We choose a density, g, that is easy to draw from and that can *majorize* f, which means there exists a constant c s.t. , $cg(x) \geq f(x)\ \forall x$. In other words we have that $cg(x)$ is an upper envelope for $f(x)$. The algorithm is

1. generate $x \sim g$
2. generate $u \sim \mathcal{U}(0, 1)$
3. if $u \leq f(x)/cg(x)$ then use x; otherwise go back to step 1

The intuition here is graphical: we generate from under a curve that is always above $f(x)$ and accept only when u puts us under $f(x)$ relative to the majorizing density. A key here is that the majorizing density have fatter tails than the density of interest, so that the constant c can exist. So we could use a t to generate from a normal but not the reverse. We'd like c to be small to reduce the number of rejections because it turns out that $\frac{1}{c} = \frac{\int f(x)dx}{\int cg(x)dx}$ is the acceptance probability. This approach works in principle for multivariate densities but as the dimension increases, the proportion of rejections grows, because more of the volume under $cg(x)$ is above $f(x)$.

If f is costly to evaluate, we can sometimes reduce calculation using a lower bound on f. In this case we accept if $u \leq f_{\text{low}}(y)/cg_Y(y)$. If it is not, then we need to evaluate the ratio in the usual rejection sampling algorithm. This is called squeezing.

One example of RS is to sample from a truncated normal. Of course we can just sample from the normal and then reject, but this can be inefficient, particularly if the truncation is far in the tail (a case in which inverse CDF suffers from numerical difficulties). Suppose the truncation point is greater than zero. Working with the standardized version of the normal, you can use an translated exponential with lower end point equal to the truncation point as the majorizing density (Robert 1995; Statistics and Computing, and see calculations in the demo code). For truncation less than zero, just make the values negative.

3.4 Adaptive rejection sampling

The difficulty of RS is finding a good enveloping function. Adaptive rejection sampling refines the envelope as the draws occur, in the case of a continuous, differentiable, log-concave density. The basic idea considers the log of the density and involves using tangents or secants to define an upper envelope and secants to define a lower envelope for a set of points in the support of the distribution. The result is that we have piecewise exponentials (since we are exponentiating from straight lines on the log scale) as the bounds. We can sample from the upper envelope based on sampling from a discrete distribution and then the appropriate exponential. The lower envelope is

used for squeezing. We add points to the set that defines the envelopes whenever we accept a point that requires us to evaluate $f(x)$ (the points that are accepted based on squeezing are not added to the set). We'll talk this through some in class.

3.5 Importance sampling

Importance sampling (IS) allows us to estimate expected values, with some commonalities with rejection sampling.

$$E_f(h(X)) = \int h(x)\frac{f(x)}{g(x)}g(x)dx$$

so $\hat{\mu} = \frac{1}{m}\sum_i h(x_i)\frac{f(x_i)}{g(x_i)}$ for x_i drawn from $g(x)$, where $w_i^* = f(x_i)/g(x_i)$ act as weights. Often in Bayesian contexts, we know $f(x)$ only up to a normalizing constant. In this case we need to use $w_i = w_i^*/\sum_i w_i^*$.

Here we don't require the majorizing property, just that the densities have common support, but things can be badly behaved if we sample from a density with lighter tails than the density of interest. So in general we want g to have heavier tails. More specifically for a low variance estimator of μ, we would want that $f(x_i)/g(x_i)$ is large only when $h(x_i)$ is very small, to avoid having overly influential points.

This suggests we can reduce variance in an IS context by oversampling x for which $h(x)$ is large and undersampling when it is small, since $\text{Var}(\hat{\mu}) = \frac{1}{m}\text{Var}(h(X)\frac{f(X)}{g(X)})$. An example is that if h is an indicator function that is 1 only for rare events, we should oversample rare events and then the IS estimator corrects for the oversampling.

What if we actually want a sample from f as opposed to estimating the expected value above? We can draw x from the unweighted sample, $\{x_i\}$, with weights $\{w_i\}$. This is called sampling importance resampling (SIR).

3.6 Ratio of uniforms

If U and V are uniform in $C = \{(u,v) : 0 \leq u \leq \sqrt{f(v/u)}$ then $X = V/U$ has density proportion to f. The basic algorithm is to choose a rectangle that encloses C and sample until we find $u \leq f(v/u)$. Then we use $x = v/u$ as our RV. The larger region enclosing C is the majorizing region and a simple approach (if $f(x)$and $x^2 f(x)$ are bounded in C) is to choose the rectangle, $0 \leq u \leq \sup_x \sqrt{f(x)}$, $\inf_x x\sqrt{f(x)} \leq v \leq \sup_x x\sqrt{f(x)}$.

One can also consider truncating the rectangular region, depending on the features of f.

Monahan recommends the ratio of uniforms, particularly a version for discrete distributions (p. 323 of the 2nd edition).

4 Design of simulation studies

Let's pose a concrete example. This is based on a JASA paper that you'll look at more carefully in a problem set. Suppose one is modeling data as being from a mixture of normal distributions:

$$f(y;\theta) = \sum_{h=1}^{m} w_k f(y;\mu_h,\sigma_h)$$

where $f(y;\mu_h,\sigma_h)$ is a normal density with mean μ_h and s.d. σ_h. A statistician has developed methodology for carrying out a hypothesis test for $H_0 : m = m_0$ vs. $H_a : m > m_0$.

First, what are the key issues that need to be assessed to evaluate their methodology? What do we want to know to assess a hypothesis test?

Second, what do we need to consider in carrying out a simulation study to address those issues?

4.1 Basic steps of a simulation study

1. Specify what makes up an individual experiment: sample size, distributions, parameters, statistic of interest, etc.
2. Determine what inputs, if any, to vary; e.g., sample sizes, parameters, data generating mechanisms
3. Write code to carry out the individual experiment and return the quantity of interest
4. For each combination of inputs, repeat the experiment m times. Note this is an embarrassingly parallel calculation (in both the data generating dimension and the inputs dimension(s).
5. Summarize the results for each combination of interest, quantifying simulation uncertainty
6. Report the results in graphical or tabular form

4.2 Overview

Since a simulation study is an experiment, we should use the same principles of design and analysis we would recommend when advising a practicioner on setting up an experiment.

These include efficiency, reporting of uncertainty, reproducibility and documentation.

In generating the data for a simulation study, we want to think about what structure real data would have that we want to mimic in the simulation study: distributional assumptions, parameter values, dependence structure, outliers, random effects, sample size (n), etc.

All of these may become input variables in a simulation study. Often we compare two or more statistical methods conditioning on the data context and then assess whether the differences between methods vary with the data context choices. E.g., if we compare an MLE to a robust estimator, which is better under a given set of choices about the data generating mechanism and how sensitive is the comparison to changing the features of the data generating mechanism? So the "treatment variable" is the choice of statistical method. We're then interested in sensitivity to the conditions.

Often we can have a large number of replicates (m) because the simulation is fast on a computer, so we can sometimes reduce the simulation error to essentially zero and thereby avoid reporting uncertainty. To do this, we need to calculate the simulation standard error, generally, $s/\sqrt{m}$ and see how it compares to the effect sizes. This is particularly important when reporting on the bias of a statistical method.

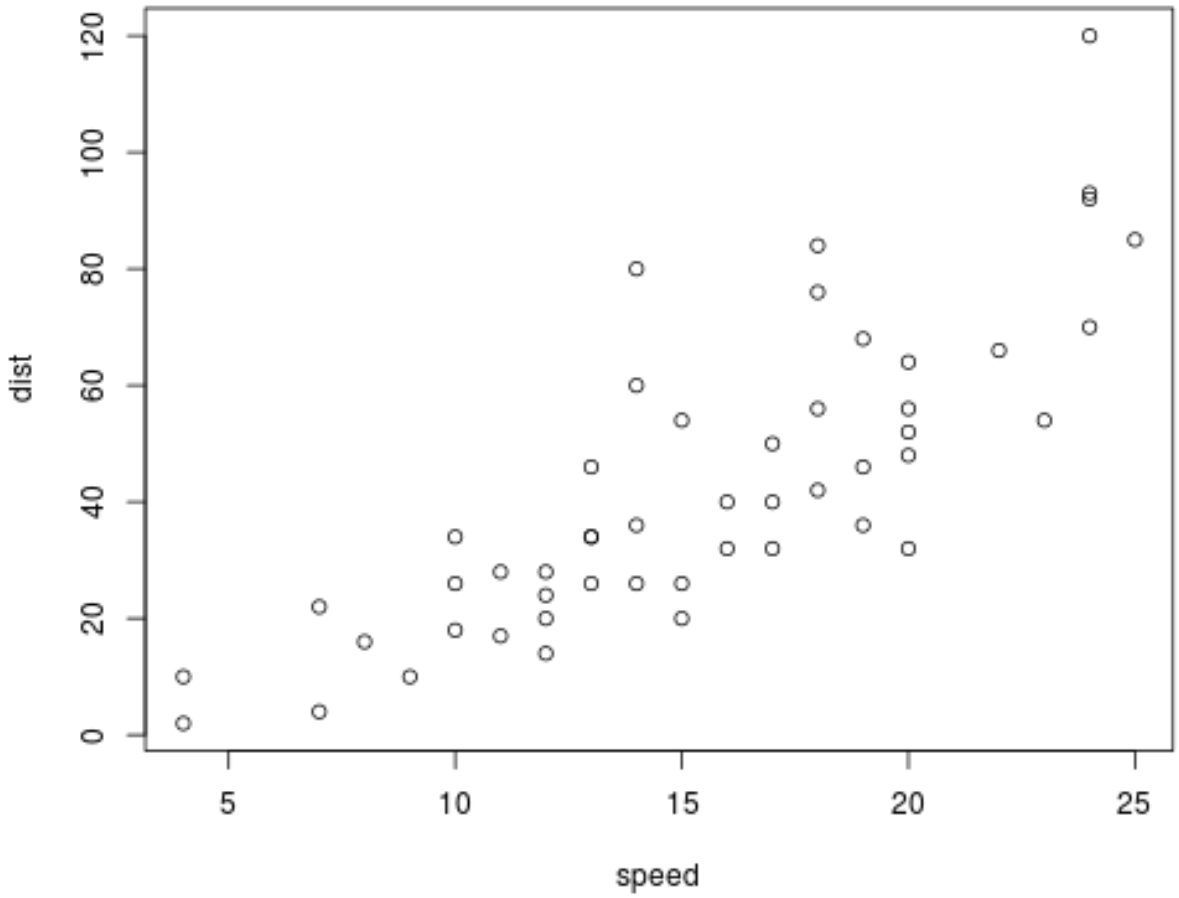
We might denote the data, which could be the statistical estimator under each of two methods as Y_{ijklq}, where i indexes treatment, j, k, l index different additional input variables, and $q \in \{1, \ldots, m\}$ indexes the replicate. E.g., j might index whether the data are from a t or normal, k the value of a parameter, and l the dataset sample size (i.e., different levels of n).

One can think about choosing m based on a basic power calculation, though since we can always generate more replicates, one might just proceed sequentially and stop when the precision of the results is sufficient.

When comparing methods, it's best to use the same simulated datasets for each level of the treatment variable and to do an analysis that controls for the dataset (i.e., for the random numbers used), thereby removing some variability from the error term. A simple example is to do a paired analysis, where we look at differences between the outcome for two statistical methods, pairing based on the simulated dataset.

One can even use the "same" random number generation for the replicates under different conditions. E.g., in assessing sensitivity to a t vs. normal data generating mechanism, we might generate the normal RVs and then for the t use the same random numbers, in the sense of using the same quantiles of the t as were generated for the normal - this is pretty easy, as seen below. This helps to control for random differences between the datasets.

```
devs <- rnorm(100)
tdevs <- qt(pnorm(devs), df = 1)
plot(devs, tdevs)
abline(0, 1)
```



4.3 Experimental Design

A typical context is that one wants to know the effect of multiple input variables on some outcome. Often, scientists, and even statisticians doing simulation studies will vary one input variable at a time. As we know from standard experimental design, this is inefficient.

The standard strategy is to discretize the inputs, each into a small number of levels. If we have a small enough number of inputs and of levels, we can do a full factorial design (potentially with replication). For example if we have three inputs and three levels each, we have 3^3 different treatment combinations. Choosing the levels in a reasonable way is obviously important.

As the number of inputs and/or levels increases to the point that we can't carry out the full factorial, a fractional factorial is an option. This carefully chooses which treatment combinations to omit. The goal is to achieve balance across the levels in a way that allows us to estimate lower level effects (in particular main effects) but not all high-order interactions. What happens

is that high-order interactions are aliased to (confounded with) lower-order effects. For example you might choose a fractional factorial design so that you can estimate main effects and two-way interactions but not higher-order interactions.

In interpreting the results, I suggest focusing on the decomposition of sums of squares and not on statistical significance. In most cases, we expect the inputs to have at least some effect on the outcome, so the null hypothesis is a straw man. Better to assess the magnitude of the impacts of the different inputs.

When one has a very large number of inputs, one can use the Latin hypercube approach to sample in the input space in a uniform way, spreading the points out so that each input is sampled uniformly. Assume that each input is $\mathcal{U}(0, 1)$ (one can easily transform to whatever marginal distributions you want). Suppose that you can run m samples. Then for each input variable, we divide the unit interval into m bins and randomly choose the order of bins and the position within each bin. This is done independently for each variable and then combined to give m samples from the input space. We would then analyze main effects and perhaps two-way interactions to assess which inputs seem to be most important.

5 Implementation of simulation studies

5.1 Computational efficiency

Parallel processing is often helpful for simulation studies. The reason is that simulation studies are embarrassingly parallel - we can send each replicate to a different computer processor and then collect the results back, and the speedup should scale directly with the number of processors we used. Since we often need to some sort of looping, writing code in C/C++ and compiling and linking to the code from R may also be a good strategy, albeit one not covered in this course.

Handy functions in R include *expand.grid()* to get all combinations of a set of vectors and the *replicate()* function in R, which will carry out the same R expression (often a function call) repeated times. This can replace the use of a *for* loop with some gains in cleanliness of your code. Storing results in an array is a natural approach.

```
require(fields)
thetaLevels <- c("low", "med", "hi")
n <- c(10, 100, 1000)
tVsNorm <- c("t", "norm")
levels <- expand.grid(thetaLevels, tVsNorm, n)
# example of replicate() -- generate m sets correlated normals
```

```
set.seed(0)
genFun <- function(n, theta = 1) {
    u <- rnorm(n)
    x <- runif(n)
    Cov <- exp(-rdist(x)/theta)
    U <- chol(Cov)
    return(cbind(x, crossprod(U, u)))
}
m <- 20
simData <- replicate(m, genFun(100, 1))
dim(simData)  # 100 observations by {x, y} values by 20 replicates

## [1] 100   2  20
```

5.2 Analysis and reporting

Often results are reported simply in tables, but it can be helpful to think through whether a graphical representation is more informative (sometimes it's not or it's worse, but in some cases it may be much better).

You should set the seed when you start the experiment, so that it's possible to replicate it. It's also a good idea to save the current value of the seed whenever you save interim results, so that you can restart simulations (this is particularly helpful for MCMC) at the exact point you left off, including the random number sequence.

To enhance reproducibility, it's good practice to post your simulation code (and potentially data) on your website or as supplementary material with the journal. One should report sample sizes and information about the random number generator.

Here are JASA's requirements on documenting computations:

"Results Based on Computation - Papers reporting results based on computation should provide enough information so that readers can evaluate the quality of the results. Such information includes estimated accuracy of results, as well as descriptions of pseudorandom-number generators, numerical algorithms, computers, programming languages, and major software components that were used."