Chapter 16

Simulation

You will recall from your previous statistics courses that quantifying uncertainty in statistical inference requires us to get at the **sampling distributions** of things like estimators. When the very strong simplifying assumptions of basic statistics courses do not apply[1], or when estimators are themselves complex objects, like kernel regression curves or histograms, there is little hope of being able to write down sampling distributions in closed form. We get around this by using simulation to approximate the sampling distributions we can't calculate.

16.1 What Do We Mean by "Simulation"?

A stochastic model is a mathematical story about how the data could have been generated. Simulating the model means implementing it, step by step, in order to produce something which should look like the data — what's sometimes called **synthetic data**, or **surrogate data**, or a **realization** of the model. In a stochastic model, some of the steps we need to follow involve a random component, and so multiple simulations starting from exactly the same initial conditions will not give exactly the same outputs or realizations. Rather, will be a distribution over the realizations. Doing large numbers of simulations gives us a good estimate of this distribution.

For a trivial example, consider a model with three random variables, $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$, $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$, with $X_1 \perp\!\!\!\perp X_2$, and $X_3 = X_1 + X_2$. Simulating from this model means drawing a random value from the first normal distribution for X_1, drawing a second random value for X_2, and adding them together to get X_3. The marginal distribution of X_3, and the joint distribution of (X_1, X_2, X_3), are implicit in this specification of the model, and we can find them by running the simulation.

In this particular case, we could also find the distribution of X_3, and the joint distribution, by probability calculations of the kind you learned how to do in your basic probability courses. For instance, X_3 is $\mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$. These analytical

[1] In 36-401, you will have seen results about the sampling distribution of linear regression coefficients when the linear model is true, and the noise is Gaussian with constant variance. As an exercise, try to get parallel results when the noise has a t distribution with 10 degrees of freedom.

probability calculations can usually be thought of as just short-cuts for exhaustive simulations.

16.2 How Do We Simulate Stochastic Models?

16.2.1 Chaining Together Random Variables

Stochastic models are usually specified by sets of conditional distributions for one random variable, given some other variable or variables. For instance, a simple linear regression model might have the specification

$$X \sim \mathcal{N}(\mu_x, \sigma_1^2) \tag{16.1}$$

$$Y|X \sim \mathcal{N}(\beta_0 + \beta_1 X, \sigma_2^2) \tag{16.2}$$

If we knew how to generate a random variable from the distributions given on the right-hand sides, we could simulate the whole model by chaining together draws from those conditional distributions. This is in fact the general strategy for simulating any sort of stochastic model, by chaining together random variables.[2]

What this means is that we can reduce the problem of simulating to that of generating random variables.

16.2.2 Random Variable Generation

Transformations

If we can generate a random variable Z with some distribution, and $V = g(Z)$, then we can generate V. So one thing which gets a lot of attention is writing random variables as transformations of one another — ideally as transformations of easy-to-generate variables.

Example. Suppose we can generate random numbers from the standard Gaussian distribution $Z \sim \mathcal{N}(0,1)$. Then we can generate from $\mathcal{N}(\mu, \sigma^2)$ as $\sigma Z + \mu$. We can generate χ^2 random variables with 1 degree of freedom as Z^2. We can generate χ^2 random variables with d degrees of freedom by summing d independent copies of Z^2.

In particular, if we can generate random numbers uniformly distributed between 0 and 1, we can use this to generate anything which is a transformation of a uniform distribution. How far does that extend?

Quantile Method

Suppose that we know the **quantile function** Q_Z for the random variable X we want, so that $Q_Z(0.5)$ is the median of X, $Q_Z(0.9)$ is the 90th percentile, and general

[2] In this case, we could in principle first generate Y, and then draw from $Y|X$, but have fun finding those distributions. Especially have fun if, say, X has a t distribution with 5 degrees of freedom — a very small change to the specification.

$Q_Z(p)$ is bigger than or equal to X with probability p. Q_Z comes as a pair with the cumulative distribution function F_Z, since

$$Q_Z(F_Z(a)) = a,\ F_Z(Q_Z(p)) = p \tag{16.3}$$

In the **quantile method** (or **inverse distribution transform method**), we generate a uniform random number U and feed it as the argument to Q_Z. Now $Q_Z(U)$ has the distribution function F_Z:

$$\begin{aligned} \Pr(Q_Z(U) \leq a) &= \Pr(F_Z(Q_Z(U)) \leq F_Z(a)) && (16.4)\\ &= \Pr(U \leq F_Z(a)) && (16.5)\\ &= F_Z(a) && (16.6) \end{aligned}$$

where the last line uses the fact that U is uniform on $[0,1]$, and the first line uses the fact that F_Z is a non-decreasing function, so $b \leq a$ is true if and only if $F_Z(b) \leq F_Z(a)$.

Example. The CDF of the exponential distribution with rate λ is $1 - e^{-\lambda x}$. The quantile function $Q(p)$ is thus $-\frac{\log(1-p)}{\lambda}$. (Notice that this is positive, because $1-p < 1$ and so $\log(1-p) < 0$, and that it has units of $1/\lambda$, which are the units of x, as it should.) Therefore, if U Unif$(0,1)$, then $-\frac{\log(1-U)}{\lambda} \sim \text{Exp}(\lambda)$. This is the method used by `rexp()`.

Example. The **Pareto distribution** or **power law** is a two-parameter family, $f(x;\alpha,x_0) = \frac{\alpha-1}{x_0}\left(\frac{x}{x_0}\right)^{-\alpha}$ if $x \geq x_0$, with density 0 otherwise. Integration shows that the cumulative distribution function is $F(x;\alpha,x_0) = 1 - \left(\frac{x}{x_0}\right)^{-\alpha+1}$. The quantile function therefore is $Q(p;\alpha,x_0) = x_0(1-p)^{-\frac{1}{\alpha-1}}$. (Notice that this has the same units as x, as it should.)

Example. The standard Gaussian $\mathcal{N}(0,1)$ does not have a closed form for its quantile function, but there are fast and accurate ways of calculating it numerically (they're what stand behind `qnorm`), so the quantile method can be used. In practice, there are other transformation methods which are even faster, but rely on special tricks.

Since $Q_Z(U)$ has the same distribution function as X, we can use the quantile method, as long as we can calculate Q_Z. Since Q_Z always exists, in principle this solves the problem. In practice, we need to calculate Q_Z before we can use it, and this may not have a closed form, and numerical approximations may be in tractable.[3]

Rejection Method

Another general approach, which avoids needing the quantile function, is the **rejection method**. Suppose that we want to generate Z, with probability density function f_Z, and we have a method to generate R, with p.d.f. ρ, called the **proposal distribution**. Also suppose that $f_Z(x) \leq \rho(x)M$, for some constant $M > 1$. For instance, if f_Z has a limited range $[a,b]$, we could take ρ to be the uniform distribution on $[a,b]$, and M the maximum density of f_Z.

The rejection method algorithm then goes as follows.

[3] In essence, we have to solve the nonlinear equation $F_Z(x) = p$ for x over and over — and that assumes we can easily calculate F_Z.

```
rrejection.1 <- function(dtarget,dproposal,rproposal,M) {
  rejected <- TRUE
  while(rejected) {
    R <- rproposal(1)
    U <- runif(1)
    rejected <- (M*U*dproposal(R) < dtarget(R))
  }
  return(R)
}

rrejection <- function(n,dtarget,dproposal,rproposal,M) {
  replicate(n,rrejection.1(dtarget,dproposal,rproposal,M))
}
```

Code Example 22: An example of how the rejection method would be used. The arguments `dtarget`, `dproposal` and `rproposal` would all be functions. This is not quite industrial-strength code, because it does not let us pass arguments to those functions flexibly. See online code for comments.

1. Generate a proposal R from ρ.
2. Generate a uniform U, independently of R.
3. Is $MU\rho(R) < f_Z(R)$?
    - If yes, "accept the proposal" by returning R and stopping.
    - If no, "reject the proposal", discard R and U, and go back to (1)

If ρ is uniform, this just amounts to checking whether $MU < f_Z(R)$, with M the maximum density of Z.

Computationally, the idea looks like Example 22.

One way to understand the rejection method is as follows. Imagine drawing the curve of $f_Z(x)$. The total area under this curve is 1, because $\int dx f_Z(x) = 1$. The area between any two points a and b on the horizontal axis is $\int_a^b dx f_Z(x) = F_Z(b) - F_Z(a)$. It follows that if we could uniformly sample points from the area between the curve and the horizontal axis, their x coordinates would have exactly the distribution function we are looking for. If ρ is a uniform distribution, then we are drawing a rectangle which just encloses the curve of f_Z, sampling points uniformly from the rectangle (with x coordinates R and y coordinates MU), and only keeping the ones which fall under the curve. When ρ is not uniform, but we can sample from it nonetheless, then we are uniformly sampling from the area under $M\rho$, and keeping only the points which are also below f_Z.

Example. The beta distribution, $f(x;a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$, is defined on the unit interval[4]. While its quantile function can be calculated and so we could use

[4]Here $\Gamma(a) = \int_0^\infty dx e^{-x} x^{a-1}$. It is not obvious, but for integer a, $\Gamma(a) = (a-1)!$. The distribution gets

the quantile method, we could also use the reject method, taking the uniform distribution for the proposals. Figure 16.1 illustrates how it would go for the Beta(5,10) distribution

The rejection method's main drawback is speed. The probability of accepting on any given pass through the algorithm is $1/M$. (EXERCISE: Why?) Thus produce n random variables from it takes, on average, nM cycles. (EXERCISE: Why?) Clearly, we want M to be as small, which means that we want the proposal distribution ρ to be close to the target distribution f_Z. Of course if we're using the rejection method because it's hard to draw from the target distribution, and the proposal distribution is close to the target distribution, it may be hard to draw from the proposal.

The Metropolis Algorithm and Markov Chain Monte Carlo

One very important, but tricky, way of getting past the limitations of the rejection method is what's called the **Metropolis algorithm**. Once again, we have a density f_Z from which we wish to sample. Once again, we introduce a distribution for "proposals", and accept or reject proposals depending on the density f_Z. The twist now is that instead of making independent proposals each time, the next proposal depends on the last accepted value — the proposal distribution is a conditional pdf $\rho(r|z)$.

Assume for simplicity that $\rho(r|z) = rho(z|r)$. (For instance, we could have a Gaussian proposal distribution centered on z.) Then the Metropolis algorithm goes as follows.

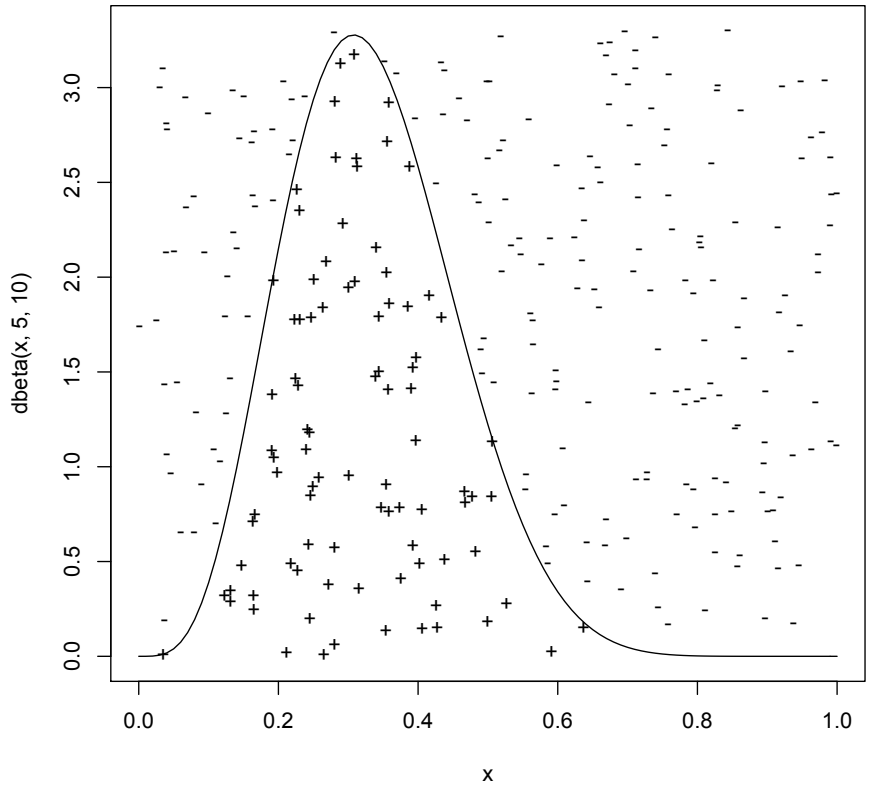
1. Start with value Z_0 (fixed or random).
2. Generate R from the conditional distribution $\rho(\cdot|Z_t)$.
3. Generate a uniform U, independent of R.
4. Is $U \leq f_Z(R)/f_Z(Z_t)$?
    - If yes, set $Z_{t+1} = R$ and go to (2)
    - If not, set $Z_{t+1} = Z_t$ and go to (2)

Mostly simply, the algorithm is run until $t = n$, at which point it returns $Z_1, Z_2, \ldots Z_n$. In practice, better results are obtained if it's run for $n+n_0$ steps, and the first n_0 values of Z are discarded — this is called "burn-in".

Notice that if $f_Z(R) > f_Z(Z_t)$, then R is always accepted. The algorithm always accepts proposals which move it towards places where the density is higher than where it currently is. If $f_Z(R) < f_Z(Z_t)$, then the algorithm accepts the move with some probability, which shrinks as the density at R gets lower. It should not be hard to persuade yourself that the algorithm will spend more time in places where f_Z is high.

It's possible to say a bit more. Successive values of Z_t are dependent on each other, but $Z_{t+1} \perp\!\!\!\perp Z_{t-1}|Z_t$ — this is a Markov process. The target distribution f_Z is

its name because $\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}$ is called the **beta function** of a and b, a kind of continuous generalization of $\binom{a+b}{a}$. The beta distribution arises in connection with problems about minima and maxima, and inference for binomial distributions.




```
M <- 3.3
curve(dbeta(x,5,10),from=0,to=1,ylim=c(0,M))
r <- runif(300,min=0,max=1)
u <- runif(300,min=0,max=1)
below <- which(M*u*dunif(r,min=0,max=1) <= dbeta(r,5,10))
points(r[below],M*u[below],pch="+")
points(r[-below],M*u[-below],pch="-")
```

Figure 16.1: Illustration of the rejection method for generating random numbera from a Beta(5,10) distribution. The proposal distribution is uniform on the range of the beta, which is $[0,1]$. Points are thus sampled uniformly from the rectangle which runs over $[0,1]$ on the horizontal axis and $[0,3.3]$ on the vertical axis, i.e., $M = 3.3$, because the density of the Beta is < 3.3 everywhere. (This is not the lowest possible M but it is close.) Proposed points which fall below the Beta's pdf are marked $+$ and are accepted; those above the pdf curve are marked $-$ and are rejected. In this case, exactly 70% of proposals are rejected.

in fact exactly the stationary distribution of the Markov process. If the proposal distributions have broad enough support that the algorithm can get from any z to any z' in a finite number of steps, then the process will "mix". (In fact we only need to be able to visit points where $f_Z > 0$.) This means that if we start with an arbitrary distribution for Z_0, the distribution of Z_t approaches f_Z and stays there — the point of burn-in is to give this convergence time to happen. The fraction of time Z_t is close to x is in fact proportional to $f_Z(x)$, so we can use the output of the algorithm as, approximately, so many draws from that distribution.[5]

It would seem that the Metropolis algorithm should be superior to the rejection method, since to produce n random values we need only n steps, or $n + n_0$ to handle burn-in, not nM steps. However, this is deceptive, because if the proposal distribution is not well-chosen, the algorithm ends up staying stuck in the same spot for, perhaps, a very long time. Suppose, for instance, that the distribution is bimodal. If Z_0 starts out in between the modes, it's easy for it to move rapidly to one peak or the other, and spend a lot of time there. But to go from one mode to the other, the algorithm has to make a series of moves, all in the same direction, which all reduce f_Z, which happens but is unlikely. It thus takes a very long time to explore the whole distribution. The "best" optimal proposal distribution is make $\rho(r|z) = f_Z(r)$, i.e., to just sample from the target distribution. If we could do that, of course, we wouldn't need the Metropolis algorithm, but trying to make ρ close to f_Z is generally a good idea.

The original **Metropolis algorithm** was invented in the 1950s to facilitate designing the hydrogen bomb. It relies on the assumption that the proposal distribution is symmetric, $\rho(r|z) = \rho(z|r)$. It is sometimes convenient to allow an asymmetric proposal distribution, in which case one accepts R if $U \frac{\rho(R|Z_t)}{\rho(Z_t|R)} \leq \frac{f_Z(R)}{f_Z(Z_t)}$. This is called **Metropolis-Hastings**. Both are examples of the broader class of **Markov Chain Monte Carlo** algorithms, where we give up on getting independent samples from the target distribution, and instead make the target the invariant distribution of a Markov process.

Mixtures and Kernel Density Estimates

Some probability distributions can be written as **mixtures** of other distributions. That is, the pdf can be written in the form

$$f_Z(x) = \sum_{j=1}^{m} w_j f_j(x) \tag{16.7}$$

where the **mixing weights** are non-negative and add up to 1. In this case, the problem of generating a random variable from f_Z can be decomposed into two steps. First, pick a random integer J between 1 and m, with probabilities given by the w_j; then draw from f_J.

[5] And if the dependence between Z_t and Z_{t+1} bothers us, we can always randomly permute them, once we have them.

Notice that kernel density estimates all have the mixture form, but with all weights equal to each other. Thus one simulates from a kernel density estimate by first picking a training point, uniformly at random, and then drawing from the kernel distribution centered on that training point.

Notice also that histogram estimates have this form, with f_j being the uniform distribution on the j^{th} bin, and w_j the total probability of that bin.

Generating Uniform Random Numbers

Everything previously to this rested on being able to generate uniform random numbers, so how do we do that? Well, really that's a problem for computer scientists... But it's good to understand a little bit about the basic ideas.

First of all, the numbers we get will be produced by some deterministic algorithm, and so will be merely **pseudorandom** rather than truly random. But we would like the deterministic algorithm to produce extremely convoluted results, so that its output *looks* random in as many ways that we can test as possible. Dependencies should be complicated, and correlations between easily-calculated functions of successive pseudorandom numbers should be small and decay quickly. (In fact, "truly random" can be defined, more or less, as the limit of the algorithm becoming infinitely complicated.) Typically, pseudorandom number generators are constructed to produce a sequence of uniform values, starting with an initial value, called the **seed**. In normal operation, the seed is set from the computer's clock; when debugging, the seed can be held fixed, to ensure that results can be reproduced exactly.

Probably the simplest example is **incommensurable rotations**. Imagine a watch which fails very slightly, but deterministically, to keep proper time, so that its second hand advances $\phi \neq 1$ seconds in every real second of time. The position of the watch after t seconds is

$$\theta_t = \theta_0 + t\alpha \bmod 60 \tag{16.8}$$

If ϕ is **commensurable** with 60, meaning $\alpha/60 = k/m$ for some integers k, m, then the positions would just repeat every $60k$ seconds. If α is **incommensurable**, because it is an irrational number, then θ_t never repeats. In this case, not only does θ_t never repeat, but it is uniformly distributed between 0 and 60, in the sense that the fraction of time it spends in any sub-interval is just proportional to the length of the interval. (EXERCISE: Why?)

You *could* use this as a pseudo-random number generator, with θ_0 as the seed, but it would not be a very good one, for two reasons. First, exactly representing an irrational number α on a digital computer is impossible, so at best you could use a rational number such that the period $60k$ is large. Second, and more pointedly, the successive θ_t are really too close to each other, and too similar. Even if we only took, say, every 50^{th} value, they'd still be quite correlated with each other.

One way this has been improved is to use *multiple* incommensurable rotations. Say we have a second inaccurate watch, $\phi_t = \phi_0 + \beta t \bmod 60$, where β is incommensurable with both 60 and with α. We record θ_t when ϕ_t is within some small window of 0.[6]

[6]The core idea here actually dates back to a medieval astronomer named Nicholas Oresme in the 1300s,

```
arnold.map <- function(v) {
  theta <- v[1]
  phi <- v[2]
  theta.new <- (theta+phi)%%1
  phi.new <- (theta+2*phi)%%1
  return(c(theta.new,phi.new))
}

rarnold <- function(n,seed) {
  z <- vector(length=n)
  for (i in 1:n) {
    seed <- arnold.map(seed)
    z[i] <- seed[1]
  }
  return(z)
}
```

Code Example 23: A function implementing the Arnold cat map (Eq. 16.10), and a second function which uses it as a pseudo-random number generator. See online version for comments.

Another approach is to use more aggressively complicated deterministic mappings. Take the system

$$\begin{aligned} \theta_{t+1} &= \theta_t + \phi_t \bmod 1 \\ \phi_{t+1} &= \theta_t + 2\phi_t \bmod 1 \end{aligned} \tag{16.9}$$

This is known as "Arnold's cat map", after the great Soviet mathematician V. I. Arnold, and Figure 16.2. We can think of this as the second-hand θ_t advancing not by a fixed amount α every second, but by a varying amount ϕ_t. The variable ϕ_t, meanwhile, advances by the amount $\phi_t + \theta_t$. The effect of this is that if we look at only one of the two coordinates, say θ_t, we get a sequence of numbers which, while deterministic, is uniformly distributed, and very hard to predict (Figure 16.3).

as part of an argument that the universe would not repeat exactly (von Plato, 1994, pp. 279–284).

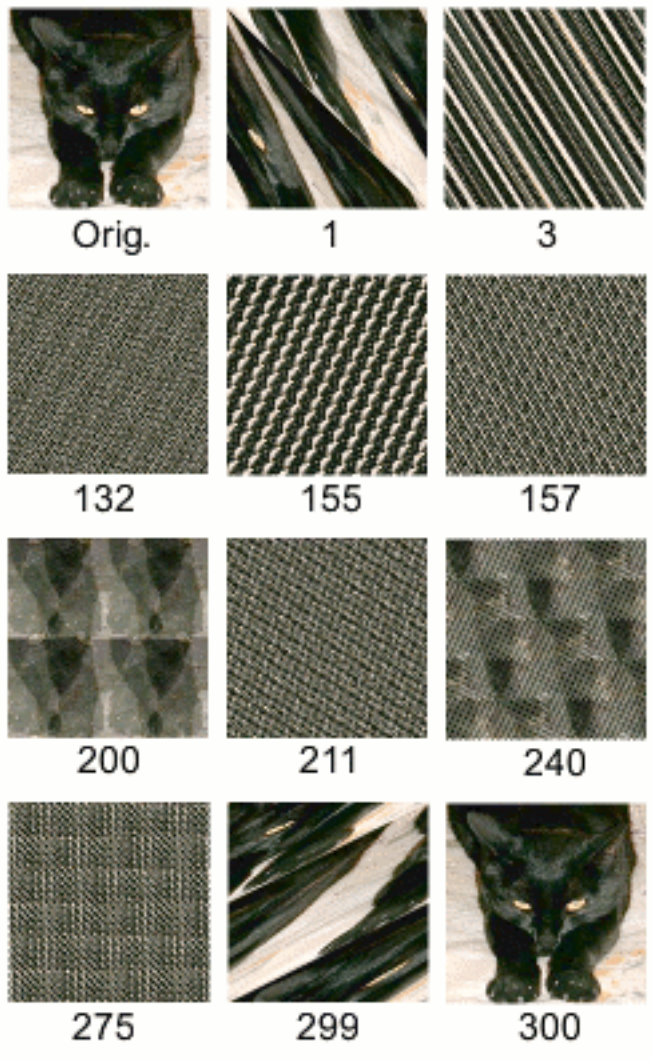
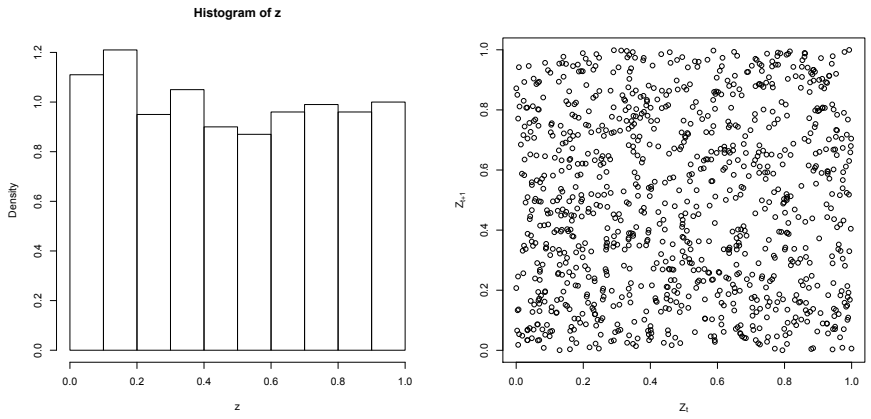


Figure 16.2: Effect of the Arnold cat map. The original image is 300×300, and mapped into the unit square. The cat map is then applied to the coordinates of each pixel separately, giving a new pixel which inherits the old color. (This can most easily seen in the transition from the original to time 1.) The original image re-assembles itself at time 300 because all the original coordinates we multiples of 1/300. If we had sampled every, say, 32 time-steps, it would have taken much longer to see a repetition. In the meanwhile, following the x coordinate of a single pixel from the original image would provide a very creditable sequence of pseudo-random values. (Figure from Wikipedia, s.v. "Arnold's cat map". See also `http://math.gmu.edu/~sander/movies/arnold.html`.)




```
par(mfrow=c(2,1))
z <- rarnold(1000,c(0.11124,0.42111))
hist(z,probability=TRUE)
plot(z[-1000],z[-1],xlab=expression(Z[t]),ylab=expression(Z[t+1]))
```

Figure 16.3: Left: histogram from 1000 samples of the θ coordinate of the Arnold cat map, started from $(0.11124, 0.42111)$. Right: scatter-plot of successive values from the sample, showing that the dependence is very subtle.

16.3 Why Simulate?

There are three major uses for simulation: to understand a model, to check it, and to fit it.

16.3.1 Understanding the Model

We understand a model by seeing what it predicts about the variables we care about, and the relationships between them. Sometimes those predictions are easy to extract from a mathematical representation of the model, but often they aren't. With a model we can simulate, however, we can just run the model and see what happens.

For instance, at the end of the chapter on kernel density estimation, we found the joint density of population growth rates (call that X) and investment rates (call that Y) from the `oecdpanel` data:

```
library(np)
data(oecdpanel)
popinv2 <- npudens(~exp(popgro)+exp(inv),data=oecdpanel)
```

(See Figure 15.4 for a visualization, and the plotting commands.)

Since this is a joint distribution, it implies a certain expected value for Y/X, the ratio of investment rate to population growth rate[7]. Extracting this by direct calculation from `popinv2` would not be easy; we'd need to do the integral

$$\int_{x=0}^{1} dx \int_{y=0}^{1} dy \frac{y}{x} \widehat{f}(x,y) \tag{16.10}$$

To find $\mathbf{E}[Y/X]$ by simulation, however, we just need to generate samples from the joint distribution, say $(\tilde{X}_1, \tilde{Y}_1), (\tilde{X}_2, \tilde{Y}_2), \ldots (\tilde{X}_T, \tilde{Y}_T)$, and average:

$$\frac{1}{T}\sum_{i=1}^{T} \frac{\tilde{Y}_i}{\tilde{X}_i} = \tilde{g}_T \stackrel{T\to\infty}{\longrightarrow} \mathbf{E}\left[\frac{Y}{X}\right] \tag{16.11}$$

where the convergence happens because that's the law of large numbers. If the number of simulation points T is big, then $\tilde{g}_T \approx \mathbf{E}[Y/X]$. How big do we need to make T? Use the central limit theorem:

$$\tilde{g}_T \rightsquigarrow \mathcal{N}(\mathbf{E}[Y/X], \mathrm{Var}[\tilde{g}_1]/\sqrt{T}) \tag{16.12}$$

How do we find the variance $\mathrm{Var}[\tilde{g}_1]$? We approximate it by simulating.

Let's be very concrete about this. Code Example 24 is a function which draws a sample from the fitted kernel density estimate. First let's check that it works, by giving it something easy to do, namely reproducing the means, which we *can* work out:

[7] Economically, we might want to know this because it would tell us about how quickly the capital stock per person grows.

```
rpopinv <- function(n) {
  n.train <- length(popinv2$dens)
  ndim <- popinv2$ndim
  points <- sample(1:n.train,size=n,replace=TRUE)
  z <- matrix(0,nrow=n,ncol=ndim)
  for (i in 1:ndim) {
    coordinates <- popinv2$eval[points,i]
    z[,i] <- rnorm(n,coordinates,popinv2$bw[i])
  }
  colnames(z) <- c("pop.growth.rate","invest.rate")
  return(z)
}
```

Code Example 24: Sampling from the fitted kernel density estimate `popinv2`. Can you see how to modify it to sample from other bivariate density estimates produced by `npudens`? From higher-dimensional distributions?

```
> mean(exp(oecdpanel$popgro))
[1] 0.06930789
> mean(exp(oecdpanel$inv))
[1] 0.1716247
> colMeans(rpopinv(200))
pop.growth.rate     invest.rate
     0.06865678      0.17623612
```

This is pretty satisfactory for only 200 samples, so the simulator seems to be working. Now we just use it:

```
> z <- rpopinv(2000)
> mean(z[,"invest.rate"]/z[,"pop.growth.rate"])
[1] 2.597916
> sd(z[,"invest.rate"]/z[,"pop.growth.rate"])/sqrt(2000)
[1] 0.0348991
```

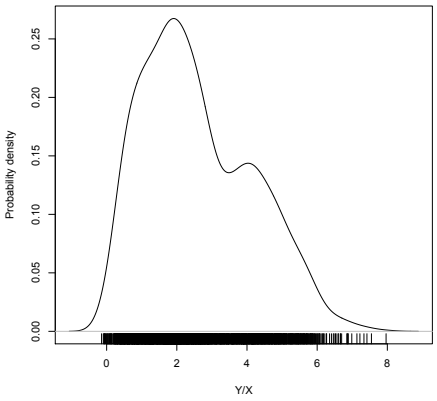
So this tells us that $\mathbf{E}[Y/X] \approx 2.59$, with a standard error of ± 0.035.

Suppose we want not the mean of Y/X but the median?

```
> median(z[,"invest.rate"]/z[,"pop.growth.rate"])
[1] 2.31548
```

Getting the whole distribution of Y/X is not much harder (Figure 16.4). Of course complicated things like distributions converge more slowly than simple things like means or medians, so we want might want to use more than 2000 simulated values for the distribution. Alternately, we could repeat the simulation many times, and look at how much variation there is from one realization to the next (Figure 16.5).

Of course, if we are going to do multiple simulations, we could just average them together. Say that $\tilde{g}_T^{(1)}$, $\tilde{g}_T^{(2)}$, $\ldots \tilde{g}_T^{(s)}$ are estimates of our statistic of interest from s




```
YoverX <- z[,"invest.rate"]/z[,"pop.growth.rate"]
plot(density(YoverX),xlab="Y/X",ylab="Probability density",main="")
rug(YoverX,side=1)
```

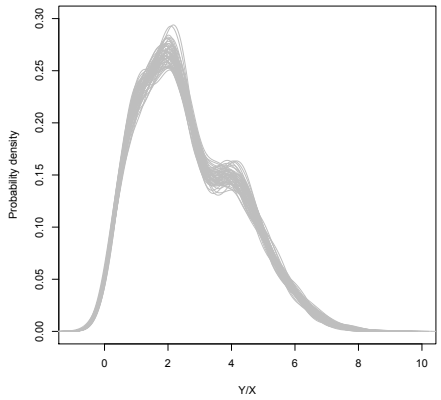
Figure 16.4: Distribution of Y/X implied by the joint density estimate `popinv2`.

independent realizations of the model, each of size T. We can just combine them into one grand average:

$$\tilde{g}_{s,T} = \frac{1}{s}\sum_{i=1}^{s} \tilde{g}_T^{(1)} \tag{16.13}$$

As an average of IID quantities, the variance of $\tilde{g}_{s,T}$ is $1/s$ times the variance of $\tilde{g}_T^{(1)}$.

By this point, we are getting the sampling distribution of the density of a nonlinear transformation of the variables in our model, with no more effort than calculating a mean.




```
plot(0,xlab="Y/X",ylab="Probability density",type="n",xlim=c(-1,10),
     ylim=c(0,0.3))
one.plot <- function() {
  zprime <- rpopinv(2000)
  YoverXprime <- zprime[,"invest.rate"]/zprime[,"pop.growth.rate"]
  density.prime <- density(YoverXprime)
  lines(density.prime,col="grey")
}
invisible(replicate(50,one.plot()))
```

Figure 16.5: Showing the sampling variability in the distribution of Y/X by "overplotting". Each line is a distribution from an estimated sample of size 2000, as in Figure 16.4; here 50 of them are plotted on top of each other. The thickness of the bands indicates how much variation there is from simulation to simulation at any given value of Y/X. (Setting the `type` of the initial `plot` to `n`, for "null", creates the plotting window, axes, legends, etc., but doesn't actually plot anything.)

```
rgeyser <- function() {
  n <- nrow(geyser)
  sigma <- summary(fit.ols)$sigma
  new.waiting <- rnorm(n,mean=fitted(fit.ols),sd=sigma)
  new.geyser <- data.frame(duration=geyser$duration,
                           waiting=new.waiting)
  return(new.geyser)
}
```

Code Example 25: Function for generating surrogate data sets from the linear model fit to `geyser`.

16.3.2 Checking the Model

An important but under-appreciated use for simulation is to *check* models after they have been fit. If the model is right, after all, it represents the mechanism which generates the data. This means that when we simulate, we run that mechanism, and the surrogate data which comes out of the machine should look like the real data. More exactly, the real data should look like a typical realization of the model. If it does not, then the model's account of the data-generating mechanism is systematically wrong in some way. By carefully choosing the simulations we perform, we can learn a lot about how the model breaks down and how it might need to be improved.[8]

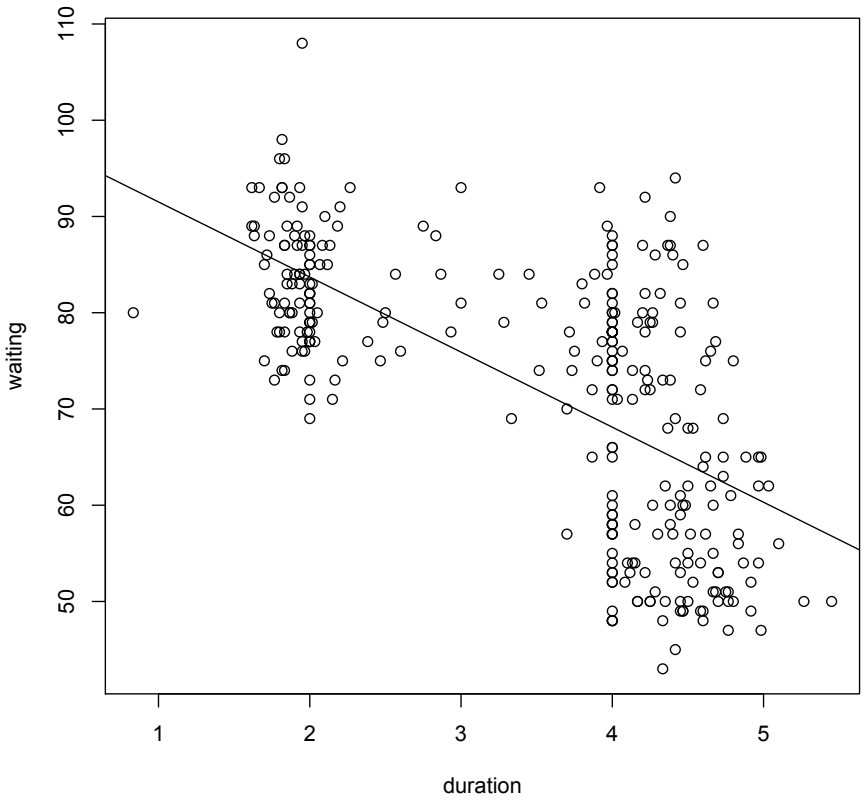
Often the comparison between simulations and data can be done qualitatively and visually. Take, for instance, the data on eruptions of Old Faithful which you worked with in homework 2. In the homework, you fit a regression line to the data by ordinary least squares:

```
library(MASS)
data(geyser)
fit.ols <- lm(waiting~duration,data=geyser)
```

Figure 16.6 shows the data, together with the OLS regression line. It doesn't look that great, but if someone insisted it was a triumph of quantitative vulcanology, how could you show they were wrong?

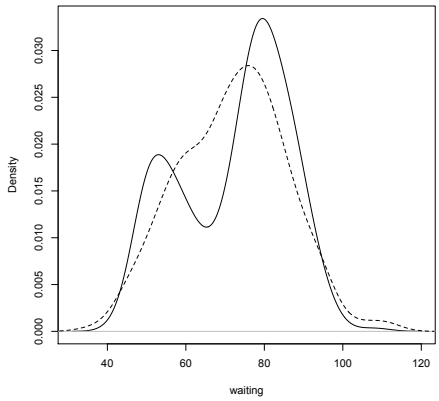
Well, OLS is usually presented as part of a probability model for the response conditional on the input, with Gaussian and homoskedastic noise. In this case, the probability model is $\texttt{waiting} = \beta_0 + \beta_1 \texttt{duration} + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2)$. If we simulate from this probability model, we'll get something we can compare to the actual data, to help us assess whether the scatter around that regression line is really bothersome. Since OLS doesn't require us to assume a distribution for the input variable (here, `duration`), the simulation function in Code Example 25 leaves those values alone, but regenerates values of the response (`waiting`) according the model assumptions.

[8] "Might", because sometimes we're better off with a model that makes systematic mistakes, if they're small and getting it right would be a hassle.




```
plot(geyser$duration,geyser$waiting,xlab="duration",ylab="waiting")
abline(fit.ols)
```

Figure 16.6: Data for the `geyser` data set, plus the OLS regression line.




```
plot(density(geyser$waiting),xlab="waiting",main="",sub="")
lines(density(rgeyser()$waiting),lty=2)
```

Figure 16.7: Actual density of the waiting time between eruptions (solid curve) an that produced by simulating the OLS model (dashed).

A useful principle for model checking is that if we do some exploratory data analyses of the real data, doing the same analyses to realizations of the model should give roughly the same results. This isn't really the case here. Figure 16.7 shows the actual density of `waiting`, plus the density produced by simulating — reality is clearly bimodal, but the model is unimodal. Similarly, Figure 16.8 shows the real data, the OLS line, and a simulation from the OLS model. It's visually clear that the deviations of the real data from the regression line are both bigger and more patterned than those we get from simulating the model, so something is wrong with the latter.

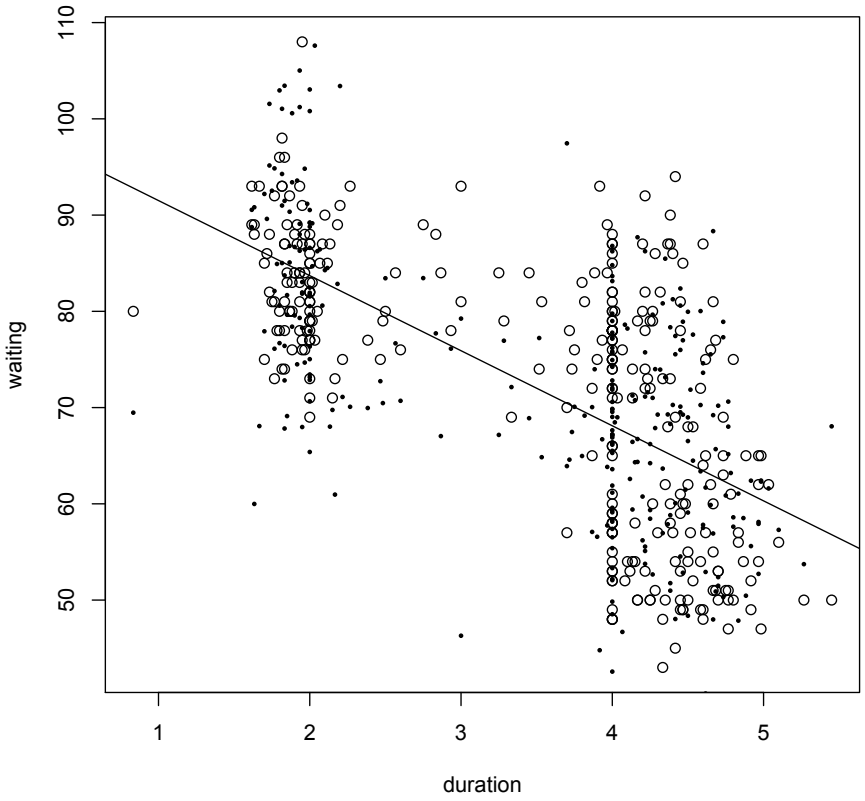
By itself, just seeing that data doesn't look like a realization of the model isn't super informative, since we'd really like to know how the model's broken, and how to fix it. Further simulations, comparing analyses of the data to analyses of the simulation output, are often very helpful here. Looking at Figure 16.8, we might suspect that one problem is heteroskedasticity[9]. In the homework, you estimated a conditional variance function,

```
library(np)
var1 <- npreg(residuals(fit.ols)^2 ~ geyser$duration)
```

which is plotted in Figure 16.9, along with the (constant) variance function of the OLS model.

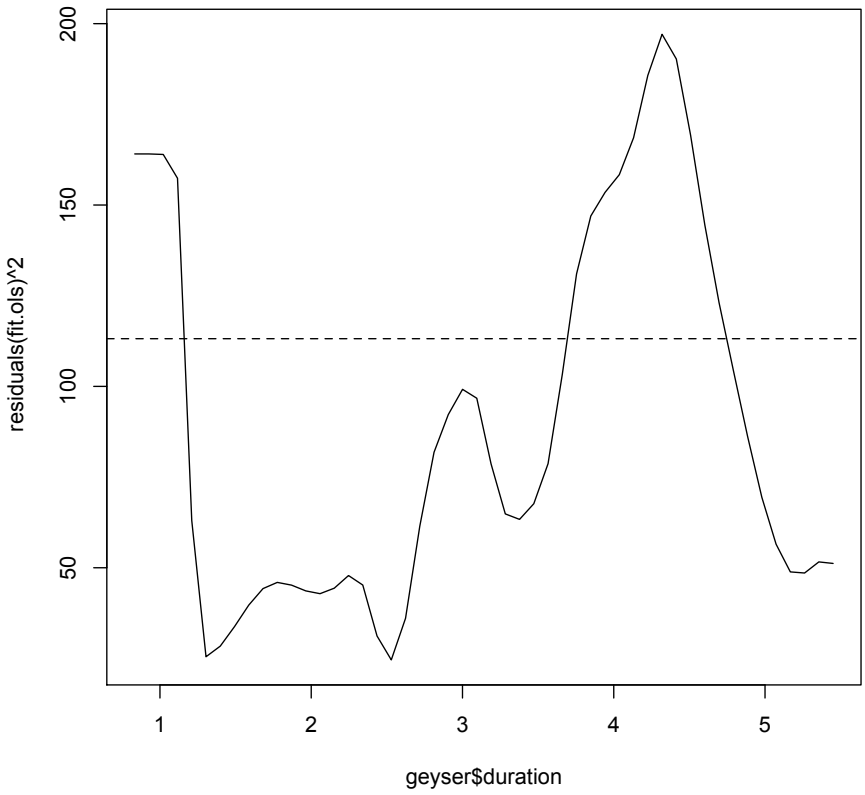
The estimated variance function `var1` does not look particularly flat, but it comes from applying a fairly complicated procedure (kernel smoothing with data-driven bandwidth selection) to a fairly limited amount of data (299 observations).

[9] At the very least, homework 2 should have planted that idea in your mind.




```
plot(geyser$duration,geyser$waiting,xlab="duration",ylab="waiting")
abline(fit.ols)
points(rgeyser(),pch=20,cex=0.5)
```

Figure 16.8: As in Figure 16.6, plus one realization of simulating the OLS model (small black dots).



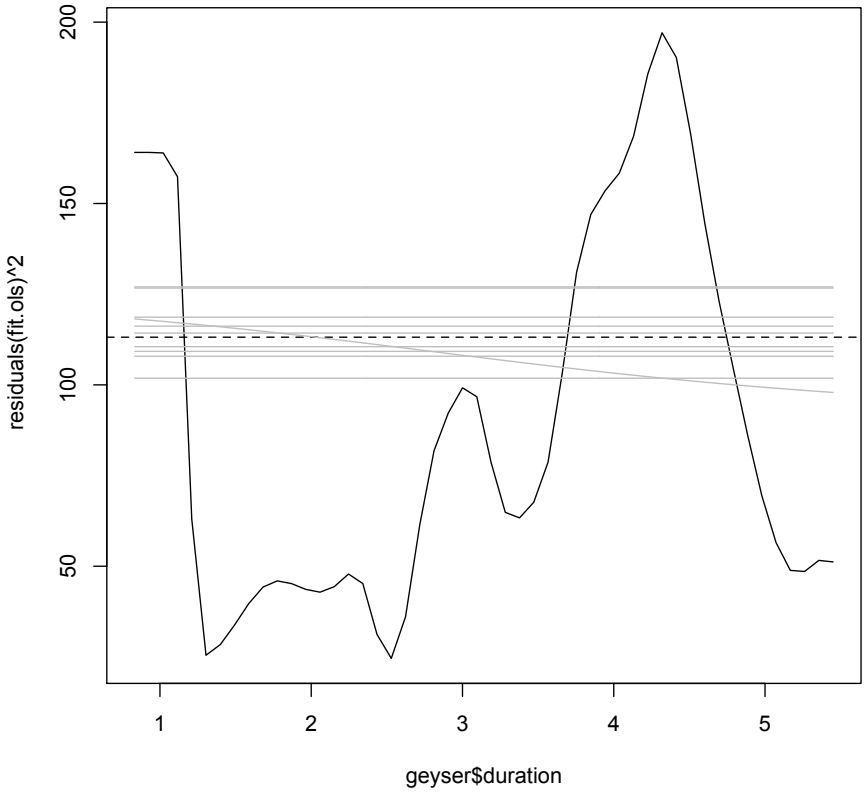

```
plot(var1,xlab="duration",ylab="Conditional variance of waiting time")
abline(h=summary(fit.ols)$sigma^2,lty=2)
```

Figure 16.9: Conditional variance function estimated from the residuals of `fit.ols`, together with the constant variance function assumed by that model (dashed line).

Maybe that's the amount of wiggliness we should *expect* to see due to finite-sample fluctuations? To rule this out, we can make surrogate data from the homoskedastic model, treat it the same way as the real data, and plot the resulting variance functions (Figure 16.10).

These comparisons have been qualitative, but one could certainly be more quantitative about this. For instance, one might measure heteroskedasticity by, say, evaluating the conditional variance at all the data points, and looking at the ratio of the interquartile range to the median. This would be zero for perfect homoskedasticity, and grow as the dispersion of actual variances around the "typical" variance increased. For the data, this is $\frac{111}{129} = 0.86$. Simulations from the OLS model give values around 10^{-15}.

There is nothing particularly special about this measure of heteroskedasticity. (After all, I just made it up.) Whenever we hve some sort of quantittive summary statistic we can calculate on our real data, we can also calculate the same statistic on realizations of the model. The difference will then tell us something about how close the simulations come to the data.




```
plot(var1)
abline(h=summary(fit.ols)$sigma^2,lty=2)
duration.grid <- seq(from=min(geyser$duration),to=max(geyser$duration),
                     length.out=300)
one.var.func <- function() {
  fit <- lm(waiting ~ duration, data=rgeyser())
  var.func <- npreg(residuals(fit)^2 ~ geyser$duration)
  lines(duration.grid,predict(var.func,exdat=duration.grid),col="grey")
}
invisible(replicate(10,one.var.func()))
```

Figure 16.10: As in Figure 16.9, but with variance functions estimated from simulations of the OLS model added in grey.

16.4 The Method of Simulated Moments

Checking whether the model's simulation output looks like the data naturally suggests the idea of *adjusting* the model until it does. This becomes a way of estimating the model — in the jargon, **simulation-based inference**. All forms of this involve adjusting parameters of the model until the simulations *do* look like the data. They differ in what "look like" means, concretely. The most straightforward form of simulation-based inference is the **method of simulated moments**.

16.4.1 The Method of Moments

You will have seen the ordinary **method of moments** in earlier statistics classes. Let's recall the general setting. We have a model with a parameter vector θ, and pick a vector m of moments to calculate. The moments, like the expectation of any variables, are functions of the parameters,

$$m = g(\theta) \tag{16.14}$$

for some function g. If that g is invertible, then we can recover the parameters from the moments,

$$\theta = g^{-1}(m) \tag{16.15}$$

The method of moments *estimator* takes the observed, sample moments $\hat{m}$, and plugs them into Eq. 16.15:

$$\widehat{\theta_{MM}} = g^{-1}(\hat{m}) \tag{16.16}$$

What if g^{-1} is hard to calculate — if it's hard to explicitly solve for parameters from moments? In that case, we can use minimization:

$$\widehat{\theta_{MM}} = \underset{\theta}{\operatorname{argmin}} \|g(\theta) - \hat{m}\|^2 \tag{16.17}$$

For the minimization version, we just have to calculate moments from parameters $g(\theta)$, not vice versa. To see that Eqs. 16.16 and 16.17 do the same thing, notice that (i) the squared[10] distance $\|g(\theta) - \hat{m}\|^2 \geq 0$, (ii) the distance is only zero when the moments are matched exactly, and (iii) there is only θ which will match the moments.

In either version, the method of moments works *statistically* because the sample moments $\hat{m}$ converge on their expectations $g(\theta)$ as we get more and more data. This is, to repeat, a consequence of the law of large numbers.

It's worth noting that nothing in this argument says that m has to be a vector of *moments* in the strict sense. They could be expectations of any functions of the random variables, so long as $g(\theta)$ is invertible, we can calculate the sample expectations of these functions from the data, and the sample expectations converge. When m isn't just a vector of moments, then, we have the **generalized method of moments**.

[10] Why squared? Basically because it makes the function we're minimizing smoother, and the optimization nicer.

It is also worth noting that there's a somewhat more general version of the same method, where we minimize

$$(g(\theta) - \hat{m}) \cdot \mathbf{w}(g(\theta) - \hat{m}) \tag{16.18}$$

with some positive-definite weight matrix $\mathbf{w}$. This can help if some of the moments are much more sensitive to the parameters than others. But this goes beyond what we really need here.

16.4.2 Adding in the Simulation

All of this supposes that we know how to calculate $g(\theta)$ — that we can find the moments exactly. Even if this is too hard, however, we could always *simulate* to approximate these expectations, and try to match the simulated moments to the real ones. Rather than Eq. 16.17, the estimator would be

$$\widehat{\theta_{SMM}} = \underset{\theta}{\operatorname{argmin}} \left\| \tilde{g}_{s,T}(\theta) - \hat{m} \right\|^2 \tag{16.19}$$

with s being the number of simulation paths and T being their size. Now consistency requires that $\tilde{g} \to g$, either as T grows or s or both, but this is generally assured by the law of large numbers, as we talked about earlier. Simulated method of moments estimates like this are generally more uncertain than ones which don't rely on simulation, since it introduces an extra layer of approximation, but this can be reduced by increasing s.[11]

16.4.3 An Example: Moving Average Models and the Stock Market

To give a concrete example, we will try fitting a time series model to the stock market: it's a familiar subject which interests *most* students, and we can check the method of simulated moments here against other estimation techniques.[12]

Our data will consist of about ten year's worth of daily values for the S& P 500 stock index, available on the class website:

```
sp <- read.csv("SPhistory.short.csv")
# We only want closing prices
sp <- sp[,7]
# The data are in reverse chronological order, which is weird for us
sp <- rev(sp)
# And in fact we only want log returns, i.e., difference in logged prices
sp <- diff(log(sp))
```

[11] A common trick is to fix T at the actual sample size n, and then to increase s as much as computationally feasible. By looking at the variance of $\tilde{g}$ across different runs of the model with the same θ, one gets an idea of how much uncertainty there is in $\hat{m}$ itself, and so of how precisely one should expect to be able to match it. If the optimizer has gotten $|\tilde{g}(\theta) - \hat{m}|$ down to 0.02, and the standard deviation of $\tilde{g}$ at constant θ is 0.1, further effort at optimization is probably wasted.

[12] Nothing in what follows, or in the homework, could actually be used to make money, however.

Professionals in finance do not care so much about the sequence of **prices** P_t, as the sequence of **returns**, $\frac{P_t - P_{t-1}}{P_{t-1}}$. This is because making \$1000 is a lot better when you invested \$1000 than when you invested \$1,000,000, but 10% is 10%. In fact, it's often easier to deal with the **log returns**, $X_t = \log \frac{P_t}{P_{t-1}}$, as we do here.

The model we will fit is a **first-order moving average**, or MA(1), model:

$$X_t = Z_t + \theta Z_{t-1} \tag{16.20}$$
$$Z_t \sim \mathcal{N}(0, \sigma^2) \text{ i.i.d.} \tag{16.21}$$

The X_t sequence of variables are the returns we see; the Z_t variables are invisible to us. The interpretation of the model is as follows. Prices in the stock market change in response to news that affects the prospects of the companies listed, as well as news about changes in over-all economic conditions. Z_t represents this flow of news, good and bad. It makes sense that Z_t is uncorrelated, because the relevant part of the news is only what everyone hadn't already worked out from older information[13]. However, it does take some time for the news to be assimilated, and this is why Z_{t-1} contributes to X_t. A negative contribution, $\theta < 0$, would seem to indicate a "correction" to the reaction to the previous day's news.

Mathematically, notice that since Z_t and θZ_{t-1} are independent Gaussians, X_t is a Gaussian with mean 0 and variance $\sigma^2 + \theta^2\sigma^2$. The marginal distribution of X_t is therefore the same for all t. For technical reasons[14], we can really only get sensible behavior from the model when $-1 \leq \theta \leq 1$.

There are two parameters, θ and σ^2, so we need two moments for estimation. Let's try $\text{Var}\left[X_t\right]$ and $\text{Cov}\left[X_t, X_{t-1}\right]$.

$$\begin{aligned}
\text{Var}\left[X_t\right] &= \text{Var}\left[Z_t\right] + \theta^2 \text{Var}\left[Z_{t-1}\right] && (16.22)\\
&= \sigma^2 + \theta^2\sigma^2 && (16.23)\\
&= \sigma^2(1+\theta^2) \equiv v(\theta, \sigma) && (16.24)
\end{aligned}$$

(This agrees with our earlier reasoning about Gaussians, but doesn't need it.)

$$\begin{aligned}
\text{Cov}\left[X_t, X_{t-1}\right] &= \mathbf{E}\left[(Z_t + \theta Z_{t-1})(Z_{t-1} + \theta Z_{t-2})\right] && (16.25)\\
&= \theta \mathbf{E}\left[Z_{t-1}^2\right] && (16.26)\\
&= \theta\sigma^2 \equiv c(\theta, \sigma) && (16.27)
\end{aligned}$$

We can solve the system of equations for the parameters, starting with eliminating

[13] Nobody will ever say "What? It's snowing in Pittsburgh in *February*? I must call my broker!"

[14] Think about trying to recover Z_t, if we knew θ. One might try $X_t - \theta X_{t-1}$, which is almost right, it's $Z_t + \theta Z_{t-1} - \theta Z_{t-1} - \theta^2 Z_{t-2} = Z_t - \theta^2 Z_{t-2}$. Similarly, $X_t - \theta X_{t-1} + \theta^2 X_{t-2} = Z_t + \theta^3 Z_{t-2}$, and so forth. If $|\theta| < 1$, then this sequence of approximations will converge on Z_t; if not, then not. It turns out that models which are not "invertible" in this way are very strange — see Shumway and Stoffer (2000).

σ^2:

$$\frac{c(\theta,\sigma)}{v(\theta,\sigma)} = \frac{\sigma^2\theta}{\sigma^2(1+\theta^2)} \tag{16.28}$$

$$= \frac{\theta}{1+\theta^2} \tag{16.29}$$

$$0 = \theta^2\frac{c}{v} - \theta + \frac{c}{v} \tag{16.30}$$

This is a quadratic in θ,

$$\theta = \frac{1 \pm \sqrt{1-4\frac{c^2}{v^2}}}{2c/v} \tag{16.31}$$

and it's easy to confirm[15] that this has only one solution in the meaningful range, $-1 \leq \theta \leq 1$. Having found θ, we solve for σ^2,

$$\sigma^2 = c/\theta \tag{16.32}$$

The method of moments estimator takes the sample values of these moments, $\hat{v}$ and $\hat{c}$, and plugs them in to Eqs. 16.31 and 16.32. With the S& P returns, the sample covariance is -1.61×10^{-5}, and the sample variance 1.96×10^{-4}. This leads to $\hat{\theta}_{MM} = -8.28 \times 10^{-2}$, and $\widehat{\sigma^2}_{MM} = 1.95 \times 10^{-4}$. In terms of the model, then, each day's news has a follow-on impact on prices which is about 8% as large as its impact the first day, but with the opposite sign.[16]

If we did not know how to solve a quadratic equation, we could use the minimization version of the method of moments estimator:

$$\begin{bmatrix} \widehat{\theta}_{MM} \\ \widehat{\sigma^2_{MM}} \end{bmatrix} = \underset{\theta,\sigma^2}{\operatorname{argmin}} \left\| \begin{array}{c} \sigma^2\theta - \hat{c} \\ \sigma^2(1+\theta^2) - \hat{v} \end{array} \right\|^2 \tag{16.33}$$

Computationally, it would go something like Code Example 26.

The parameters estimated by minimization agree with those from direct algebra to four significant figures, which I hope is good enough to reassure you that this works.

Before we can try out the method of simulated moments, we have to figure out how to simulate our model. X_t is a deterministic function of Z_t and Z_{t-1}, so our general strategy says to first generate the Z_t, and then compute X_t from that. But here the Z_t are just a sequence of independent Gaussians, which is a solved problem for us. The one wrinkle is that to get our first value X_1, we need a previous value Z_0. Code Example 27 shows the solution.

[15] For example, plot c/v as a function of θ, and observe that any horizontal line cuts the graph at only one point.

[16] It would be natural to wonder whether $\widehat{\theta_{MM}}$ is really significantly different from zero. Assuming Gaussian noise, one could, in principle, calculate the probability that even though $\theta = 0$, by chance $\hat{c}/\hat{v}$ was so far from zero as to give us our estimate. As you will see in the homework, however, Gaussian assumptions are very bad for this data. This sort of thing is why we have bootstrapping.

```
ma.mm.est <- function(c,v) {
  theta.0 <- c/v
  sigma2.0 <- v
  fit <- optim(par=c(theta.0,sigma2.0), fn=ma.mm.objective,
               c=c, v=v)
  return(fit)
}

ma.mm.objective <- function(params,c,v) {
  theta <- params[1]
  sigma2 <- params[2]
  c.pred <- theta*sigma2
  v.pred <- sigma2*(1+theta^2)
  return((c-c.pred)^2 + (v-v.pred)^2)
}
```

Code Example 26: Code for implementing method of moments estimation of a first-order moving average model, as in Eq. 16.33. See Appendix 16.6 for "design notes", and the online code for comments.

```
rma <- function(n,theta,sigma2,s=1) {
  z <- replicate(s,rnorm(n=n+1,mean=0,sd=sqrt(sigma2)))
  x <- z[-1,] + theta*z[-(n+1),]
  return(x)
}
```

Code Example 27: Function which simulates `s` independent runs of a first-order moving average model, each of length `n`, with given noise variance `sigma2` and after-effect `theta`. See online for the version with comments on the code details.

```
sim.var <- function(n,theta,sigma2,s=1) {
   vars <- apply(rma(n,theta,sigma2,s),2,var)
   return(mean(vars))
}

sim.cov <- function(n,theta,sigma2,s=1) {
  x <- rma(n,theta,sigma2,s)
  covs <- colMeans(x[-1,]*x[-n,])
  return(mean(covs))
}
```

Code Example 28: Functions for calculating the variance and covariance for specified parameter values from simulations.

What we need to extract from the simulation are the variance and the covariance. It will be more convenient to have functions which calculate these call `rma()` themselves (Code Example 28).

Figure 16.11 plots the covariance, the variance, and their ratio as functions of θ with $\sigma^2 = 1$, showing both the values obtained from simulation and the theoretical ones.[17] The agreement is quite good, though of course not quite perfect.[18]

Conceptually, we could estimate θ by jut taking the observed value $\hat{c}/\hat{v}$, running a horizontal line across Figure 16.11c, and seeing at what θ it hit one of the simulation dots. Of course, there might not be one it hits *exactly*...

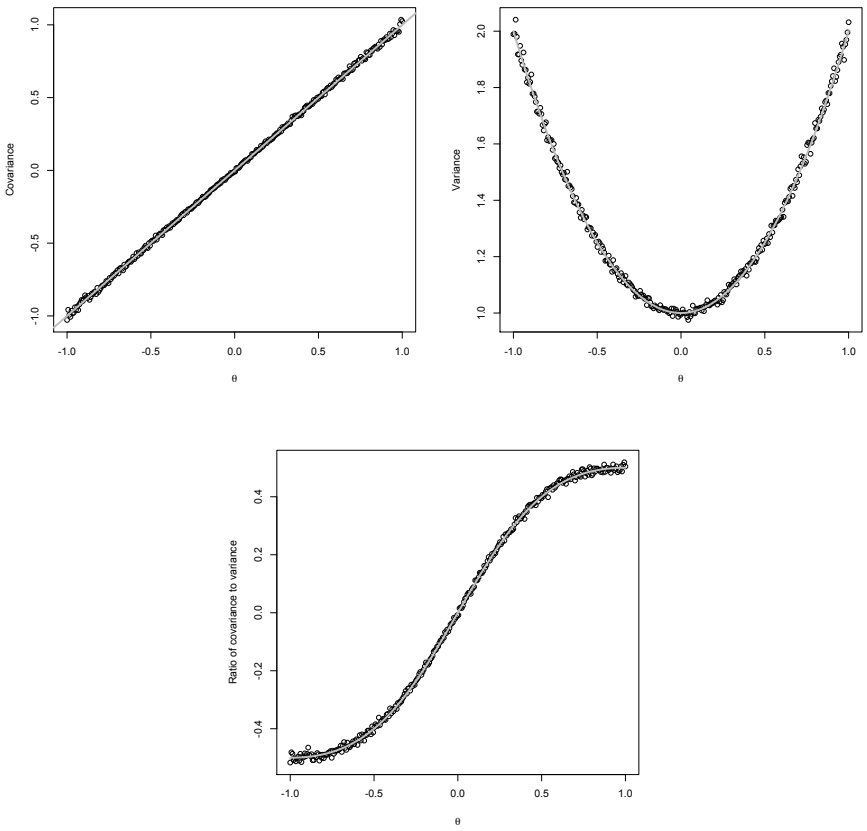
The more practical approach is Code Example 29. The code is practically identical to that in Code Example 26, except that the variance and covariance predicted by given parameter settings now come from simulating those settings, not an exact calculation. Also, we have to say how long a simulation to run, and how many simulations to average over per parameter value.

When I run this, with `s=100`, I get $\widehat{\theta}_{MSM} = -8.36 \times 10^{-2}$ and $\widehat{\sigma}^2_{MSM} = 1.94 \times 10^{-4}$, which is quite close to the non-simulated method of moments estimate. In fact, in this case there is actually a maximum likelihood estimator (`arima()`, after the more general class of models including MA models), which claims $\widehat{\theta}_{ML} = -9.75 \times 10^{-2}$ and $\widehat{\sigma}^2_{ML} = 1.94 \times 10^{-4}$. Since the standard error of the MLE on θ is ± 0.02, this is working essentially as well as the method of moments, or even the method of simulated moments.

In this case, because there *is* a very tractable maximum likelihood estimator, one generally wouldn't use the method of simulated moments. But we can in this case check whether it works (it does), and so we can use the same technique for other models, where an MLE is unavailable.

[17] I could also have varied σ^2 and made 3D plots, but that would have been more work. Also, the variance and covariance are both proportional to σ^2, so the shapes of the figures would all be the same.

[18] If you look at those figures and think "Why not do a nonparametric regression of the simulated moments against the parameters and use the fitted values as $\tilde{g}$, it'll get rid of some of the simulation noise?", congratulations, you've just discovered the smoothed method of simulated moments.




```
theta.grid <- seq(from=-1,to=1,length.out=300)
cov.grid <- sapply(theta.grid,sim.cov,sigma2=1,n=length(sp),s=10)
plot(theta.grid,cov.grid,xlab=expression(theta),ylab="Covariance")
abline(0,1,col="grey",lwd=3)
var.grid <- sapply(theta.grid,sim.var,sigma2=1,n=length(sp),s=10)
plot(theta.grid,var.grid,xlab=expression(theta),ylab="Variance")
curve((1+x^2),col="grey",lwd=3,add=TRUE)
plot(theta.grid,cov.grid/var.grid,xlab=expression(theta),
  ylab="Ratio of covariance to variance")
curve(x/(1+x^2),col="grey",lwd=3,add=TRUE)
```

Figure 16.11: Plots of the covariance, the variance, and their ratio as a function of θ, with $\sigma^2 = 1$. Dots show simulation values (averaging 10 realizations each as long as the data), the grey curves the exact calculations.

```
ma.msm.est <- function(c,v,n,s) {
  theta.0 <- c/v
  sigma2.0 <- v
  fit <- optim(par=c(theta.0,sigma2.0),fn=ma.msm.objective,c=c,v=v,n=n,s=s)
  return(fit)
}

ma.msm.objective <- function(params,c,v,n,s) {
  theta <- params[1]
  sigma2 <- params[2]
  c.pred <- sim.cov(n,theta,sigma2,s)
  v.pred <- sim.var(n,theta,sigma2,s)
  return((c-c.pred)^2 + (v-v.pred)^2)
}
```

Code Example 29: Code for implementing the method of simulated moments estimation of a first-order moving average model.

16.5 Exercises

To think through, not to hand in.

Section 16.4 explained the method of simulated moments, where we try to match expectations of various functions of the data. Expectations of functions are summary statistics, but they're not the *only* kind of summary statistics. We could try to estimate our model by matching any set of summary statistics, so long as (i) there's a unique way of mapping back from summaries to parameters, and (ii) estimates of the summary statistics converge as we get more data.

A powerful but somewhat paradoxical version of this is what's called **indirect inference**, where the summary statistics are the parameters of a *different* model. This second or **auxiliary** model does *not* have to be correctly specified, it just has to be easily fit to the data, and satisfy (i) and (ii) above. Say the parameters of the auxiliary model are β, as opposed to the θ of our real model. We calculate $\widehat{\beta}$ on the real data. Then we simulate from different values of θ, fit the auxiliary to the simulation outputs, and try to match the auxiliary estimates. Specifically, the indirect inference estimator is

$$\widehat{\theta}_{II} = \operatorname*{argmin}_{\theta} ||\tilde{\beta}(\theta) - \widehat{\beta}||^2 \tag{16.34}$$

where $\tilde{\beta}(\theta)$ is the value of β we estimate from a simulation of θ, of the same size as the original data. (We might average together a couple of simulation runs for each θ.) If we have a consistent estimator of β, then

$$\widehat{\beta} \rightarrow \beta \tag{16.35}$$

$$\tilde{\beta}(\theta) \rightarrow b(\theta) \tag{16.36}$$

If in addition $b(\theta)$ is invertible, then

$$\widehat{\theta}_{II} \rightarrow \theta \tag{16.37}$$

For this to work, the auxiliary model needs to have at least as many parameters as the real model, but we can often arrange this by, say, making the auxiliary model a linear regression with a lot of coefficients.

A specific case, often useful for time series, is to make the auxiliary model an **autoregressive model**, where each observation is linearly regressed on the previous ones. A first-order autoregressive model (or "AR(1)") is

$$X_t = \beta_0 + \beta_1 X_{t-1} + \epsilon_t \tag{16.38}$$

where $\epsilon_t \sim \mathcal{N}(0, \beta_3)$. (So an AR(1) has three parameters.)

1. Convince yourself that if X_t comes from an MA(1) process, it can't also be written as an AR(1) model.
2. Write a function, `ar1.fit`, to fit an AR(1) model to a time series, using `lm`, and to return the three parameters (intercept, slope, noise variance).

3. Apply `ar1.fit` to the S&P 500 data; what are the auxiliary parameter estimates?

4. Combine `ar1.fit` with the simulator `rma`, and plot the three auxiliary parameters as functions of θ, holding σ^2 fixed at 1. (This is analogous to Figure 16.11.)

5. Write functions, analogous to `ma.msm.est` and `ma.msm.objective`, for estimating an MA(1) model, using an AR(1) model as the auxiliary function. Does this recover the right parameter values when given data simulated from an MA(1) model?

6. What values does your estimator give for θ and σ^2 on the S& P 500 data? How do they compare to the other estimates?

16.6 Appendix: Some Design Notes on the Method of Moments Code

Go back to Section 16.4.3 and look at the code for the method of moments. There've been a fair amount of questions about writing code, and this is a useful example.

The first function, `ma.mm.est`, estimates the parameters taking as inputs two numbers, representing the covariance and the variance. The real work is done by the built-in `optim` function, which itself takes two major arguments. One, `fn`, is the function to optimize. Another, `par`, is an initial guess about the parameters at which to begin the search for the optimum.[19]

The `fn` argument to `optim` must be a function, here `ma.mm.objective`. The first argument to that function has to be a vector, containing all the parameters to be optimized over. (Otherwise, `optim` will quit and complain.) There can be other arguments, not being optimized over, to that function, which `optim` will pass along, as you see here. `optim` will also accept a lot of optional arguments to control the search for the optimum — see `help(optim)`.

All `ma.mm.objective` has to do is calculate the objective function. The first two lines peel out θ and σ^2 from the parameter vector, just to make it more readable. The next two lines calculate what the moments should be. The last line calculates the distance between the model predicted moments and the actual ones, and returns it. The whole thing could be turned into a one-line, like

```
return(t(params-c(c,v)) %*% (params-c(c,v)))
```

or perhaps even more obscure, but that is usually a bad idea.

Notice that I could write these two functions independently of one another, at least to some degree. When writing `ma.mm.est`, I knew I would need the objective function, but all I needed to know about it was its name, and the promise that it would take a parameter vector and give back a real number. When writing `ma.mm.objective`, all I had to remember about the other function was the promise this one needed to fulfill. In my experience, it is usually easiest to do any substantial coding in this "top-down" fashion[20]. Start with the high-level goal you are trying to achieve, break it down into a few steps, write something which will put those steps together, presuming other functions or programs can do them. Now go and write the functions to do each of those steps.

The code for the method of simulated moments is entirely parallel to these. Writing it as two separate pairs of functions is therefore somewhat wasteful. If I find a mistake in one pair, or thing of a way to improve it, I need to remember to make corresponding changes in the other pair (and not introduce a new mistake). In the long run, when you find yourself writing parallel pieces of code over and over, it is better to try to pull together the common parts and write them *once*. Here, that would mean something like one pair of functions, with the inner one having an argument

[19] Here `par` is a very rough guess based on `c` and `v` — it'll actually be right when `c=0`, but otherwise it's not much good. Fortunately, it doesn't have to be! Anyway, let's return to designing the code

[20] What qualifies as "substantial coding" depends on how much experience you have

which controlled whether to calculate the predicted moments by simulation or by a formula. You may try your hand at writing this.