11

MONTE CARLO

Monte Carlo (Italian and Spanish for Mount Carl) is a city in Monaco famous for its casinos, and has more glamorous associations with its name than Reno or Atlantic City.

Monte Carlo methods are thus about randomization: taking existing data and making random transformations to learn more about it. But although the process involves randomness, its outcome is not just the mere whim of the fates. At the roulette table, a single player may come out ahead, but with millions of suckers testing their luck, casinos find that even a 49–51 bet in their favor is a reliable method of making money. Similarly, a single random transformation of the data will no doubt produce a somehow distorted impression, but reapplying it thousands or millions of times will present an increasingly accurate picture of the underlying data.

This chapter will first look at the basics of random number generation. It will then discuss the general process of describing parameters of a distribution, parameters of a data set, or a distribution as a whole via Monte Carlo techniques. As a special case, bootstrapping is a method for getting a variance out of data that, by all rights, should not be able to give you a variance. Nonparametric methods also make a return in this chapter, because shuffling and redrawing from the data can give you a feel for the odds that some hypothesized event would have occurred; that is, we can write hypothesis tests based on resampling from the data.

```
gsl_rng *apop_rng_alloc(int seed){
  static int first_use = 1;
   if (first_use){
        first_use --;
        gsl_rng_env_setup();
   }
    gsl_rng *setme = gsl_rng_alloc(gsl_rng_taus2);
    gsl_rng_set(setme, seed);
    return setme;
}
```

Listing 11.1 Allocating and initializing a random number generator.

11.1 RANDOM NUMBER GENERATION

We need a stream of random numbers, but to get any programming done, we need a *replicable* stream of random numbers.[1]

There are two places where you will need replication. The first is with debugging, since you don't want the segfault you are trying to track down to appear and disappear every other run. The second is in reporting your results, because when a colleague asks to see how you arrived at your numbers, you should be able to reproduce them exactly.

Of course, using the same stream of numbers every time creates the possibility of getting a lucky draw, where *lucky* can mean any of a number of things. The compromise is to use a collection of deterministic streams of numbers that have no apparent pattern, where each stream of numbers is indexed by its first value, the *seed*.[2] The GSL implements such a process.

Listing 11.1 shows the innards of the `apop_rng_alloc` function from the Apophenia library to initialize a `gsl_rng`. In all cases, the function takes in an integer, and then sets up the random number generation (RNG) environment to produce new numbers via the Tausworth routine. Fans of other RNG methods can check the GSL documentation for setting up a `gsl_rng` with alternative algorithms. On the first call, the function calls the `gsl_rng_env_setup` function to work some internal magic in the GSL. Listings 11.6 and 11.7 below show an example using this function.

[1] Is it valid to call a replicable stream of seemingly random numbers *random*? Because such RNGs are arguably not random, some prefer the term *pseudorandom number generator* (PRNG) to describe them. This question is rooted in a philosophical question into which this book will not delve: what is the difference between perceived randomness given some level of information and true randomness? See, e.g., Good (1972, pp 127–8).

[2] Formally, the RNG produces only one stream, that eventually cycles around to the beginning. The seed simply specifies where in the cycle to begin. But because the cycle is so long, there is little loss in thinking about each seed producing a separate stream.

If you initialize one RNG stream with the value of another, then they are both at the same point in the cycle, and they will follow in lock-step from then on. This is to be avoided; if you need a sequence of streams, you are better off just using a simple list of seeds like 0, 1, 2, ...

RANDOM NUMBER DISTRIBUTIONS Now that you have a random number generator, here are some functions that use it to draw from all of your favorite distributions. Input an RNG as allocated above plus the appropriate parameters, and the GSL will transform the RNG as necessary.

```
double gsl_ran_bernoulli (gsl_rng *r, double p);
double gsl_ran_beta (gsl_rng *r, double a, double b);
double gsl_ran_binomial (gsl_rng *r, double p, int n);
double gsl_ran_chisq (gsl_rng *r, double df);
double gsl_ran_fdist (gsl_rng *r, double df1, double df2);
double gsl_ran_gaussian (gsl_rng *r, double sigma);
double gsl_ran_tdist (gsl_rng *r, double df);
double gsl_ran_flat (gsl_rng *r, double a, double b);
double gsl_rng_uniform (gsl_rng *r);
```

- The *flat* distribution is a Uniform[A,B) distribution. The Uniform[0,1) distribution gets its own no-options function, `gsl_rng_uniform(r)`.
- The Gaussian draw assumes a mean of zero, so if you intend to draw from, e.g., a $\mathcal{N}(7, 2)$, then use `gsl_ran_gaussian(r, 2) + 7`.
- The `apop_model` struct includes a `draw` method that works like the above functions to make random draws, and allows standardization with more exotic models like the histogram below; see the example in Listing 11.5, page 361.

An example: the Beta distribution The Beta distribution is wonderful for all sorts of modeling, because it can describe such a wide range of probability functions for a variable $\in [0, 1]$. For example, you saw it used as a prior on page 259. But its α and β parameters may be difficult to interpret; we are more used to the mean and variance. Thus, Apophenia provides a convenience function, `apop_beta_from_mean_var`, that takes in μ and σ^2 and returns an appropriate Beta distribution, with the corresponding values of α and β.

As you know, the variance of a Uniform$[0, 1]$ is exactly $\frac{1}{12}$, which means that the Beta distribution will never have a variance greater than $\frac{1}{12}$ (and close to $\frac{1}{12}$, perverse things may happen computationally for $\mu \not\approx \frac{1}{2}$).[3] The mean of a function that has positive density iff $x \in [0, 1]$ must be $\in (0, 1)$. If you send `apop_beta_from_mean_var` values of μ and σ^2 that are outside of these bounds, the function will return `GSL_NAN`.

What does a Beta distribution with, say, $\mu = \frac{3}{8}, \sigma^2 = \frac{1}{24} = .041\overline{66}$ look like? Listing 11.2 sets up an RNG, makes a million draws from a Beta distribution, and plots the result.

[3] More trivia: the Uniform$[0, 1]$ is symmetric, so its skew is zero. Its kurtosis is $\frac{1}{80}$.

```
1  #include <apop.h>
2
3  int main(){
4      int  draws = 1e7;
5      int  bins = 100;
6      double mu  = 0.492; //also try 3./8.
7      double sigmasq = 0.093; //also try 1./24.
8      gsl_rng *r  = apop_rng_alloc(0);
9      apop_data *d  = apop_data_alloc(0, draws, 1);
10         apop_model *m = apop_beta_from_mean_var(mu, sigmasq);
11         for (int i =0; i < draws; i++)
12             apop_draw(apop_data_ptr(d, i, 0), r, m);
13         Apop_settings_add_group(&apop_histogram, apop_histogram, d, bins)
14         apop_histogram_normalize(&apop_histogram);
15         apop_histogram_plot(&apop_histogram, NULL);
16 }
```

Listing 11.2 Building a picture of a distribution via random draws. Online source: `drawbeta.c`.

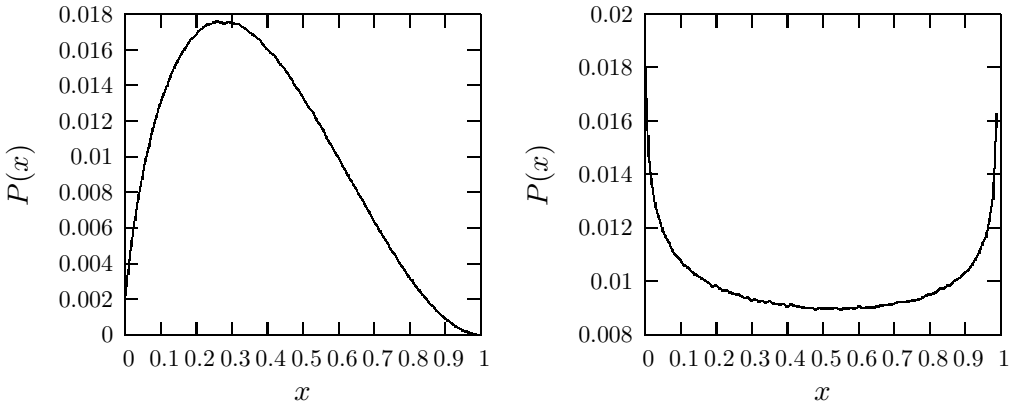

Figure 11.3 The flexible Beta distribution. Run via `drawbeta | gnuplot`.

- Most of the code simply names constants; the first real action occurs on line 11, where `apop_beta_from_mean_var` takes in μ and σ^2 and returns an `apop_model` representing a Beta distribution with the appropriate parameters.
- In line 14, the data set `d` is filled with random draws from the model. The `apop_draw` function takes in a pointer-to-`double` as the first argument, and puts a value in that location based on the RNG and model sent as the second and third arguments. Thus, you will need to use `apop_data_ptr`, `gsl_vector_ptr`, or `gsl_matrix_ptr` with `apop_draw` to get a pointer to the right location. The slightly awkward pointer syntax means that no copying or reallocation is necessary, so there is less to slow down drawing millions of numbers.
- The final few lines of code take the generic `apop_histogram` model, set it to use

the data `d` and the given number of bins, normalize the histogram thus produced to have a total density of one, and plot the result.

The output of this example (using 1e7 draws) is at left in Figure 11.3; by contrast, the case where $\mu = 0.492$, $\sigma^2 = 0.093$ is pictured at right.

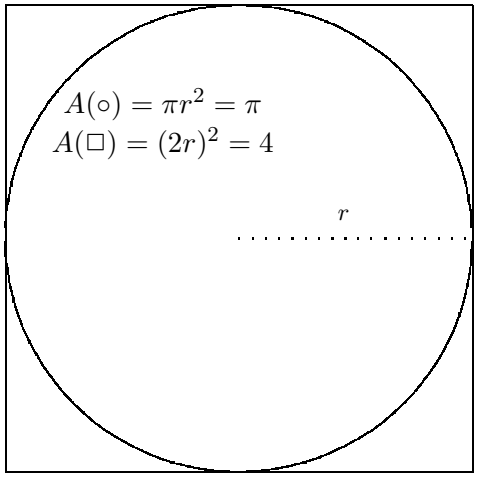

Figure 11.4 A circle inscribed in a square. The ratio of the area of the circle to the area of the square is $\pi/4$. Online source: `squarecircle.gnuplot`.

$\mathbb{Q}_{11.1}$

As per Figure 11.4, when a circle is inscribed inside a square, the ratio of the area of the circle to the square is $\pi/4$. Thus, if we randomly draw 100 points from the square, we expect $100\pi/4$ to fall within the circle.
Estimate π via random draws from a square. For i in zero to about 1e8:

- Draw x_i from a Uniform$[-1, 1]$ distribution; draw y_i from a Uniform$[-1, 1]$ distribution.
- Determine whether (x_i, y_i) falls within the unit circle, meaning that $\sqrt{x_i^2 + y_i^2} \leq 1$ (which is equivalent to $x_i^2 + y_i^2 \leq 1$).
- Every 10,000 draws, display the proportion of draws inside the circle times four. How close to π does the estimate come? (*Hint*: it may be clearer to display `fabs(M_PI - pi_estimate)` instead of the estimate itself.)

DRAWING FROM YOUR OWN DATA Another possibility, beyond drawing from famous distributions that your data theoretically approximates, would be to draw from your actual data.

If your data are in a vector, then just draw a random index and return the value at that index. Let *r* be an appropriately initialized `gsl_rng`, and let *your_data* be a `gsl_vector` from which you would like to make draws. Then the following one-liner would make a single draw:

```
gsl_vector_get(your_data, gsl_rng_uniform_int(r, your_data->size));
```

※ *Drawing from histograms* There are a few reasons for your data to be in a histogram form—a rough probability mass function—like the ones used for plotting data in Chapter 5, for describing data in Chapter 7, for goodness of fit tests in Chapter 9, and as the output from the `apop_update` function. Here, we will draw from them to produce artificial data sets.

```
1  #include <apop.h>
2  gsl_rng *r;
3
4  void plot_draws(apop_model *m, char *outfile){
5      int draws = 2e3;
6      apop_data *d = apop_data_alloc(0, draws,1);
7          for(size_t i=0; i < draws; i++)
8              apop_draw(apop_data_ptr(d, i, 0), r, m);
9          apop_model *h3 = apop_estimate(d, apop_histogram);
10         apop_histogram_print(h3, outfile);
11 }
12
13 int main(){
14         r = apop_rng_alloc(1);
15         apop_db_open("data-wb.db");
16         apop_data *pops = apop_query_to_data("select population+0.0 p from pop where p>500");
17         apop_model *h = apop_estimate(pops, apop_histogram);
18         apop_histogram_print(h, "out.hist");
19         plot_draws(apop_estimate(pops, apop_lognormal), "lognormal_fit");
20         printf("set xrange [0:2e5]; set yrange [0:0.12]; \n \
21                     plot 'out.hist' with boxes, 'lognormal_fit' with lines\n");
22 }
```

Listing 11.5 Draw from the actual histogram of country populations, and from the exponential distribution that most closely fits. Online source: `drawfrompop.c`.

For example, say that we would like to generate communities whose populations are distributed the way countries of the world are distributed. Listing 11.5 does

this two ways. The simplest approach is to simply generate a histogram of world populations and make draws from that histogram.

- Line 17 creates a filled histogram by filling the un-parametrized base `apop_histogram` model with a list of populations.

In some cases, there are simply not enough data for the job. The World Bank data set lists 208 countries; if your simulation produces millions of communities, the repetition of 208 numbers could produce odd effects. The solution presented here is to estimate a Lognormal distribution, and then draw from that ideal distribution. Line 21 does the model fit and then sends the output of the `apop_estimate` function to the `plot_draws` function, which makes a multitude of draws from the ideal distribution, and then plots those. You can see that the result is smoother, without the zero-entry bins that the real-world data has.

- Lines 7–8 fill column zero of `d` with data, then line 9 turns that data into a histogram.
- The easiest way to put two data sets on one plot is to write both of them to separate files (lines 18 and 10), and then call those files from a Gnuplot script (lines 20–21).

※ *Seeding with the time* There are situations where a fixed seed is really not what you want—you want different numbers every time you run your program. The easiest solution is to seed using the `time` function. The standard library function `time(NULL)` will return the number of seconds that have elapsed since the beginning of 1970, which is roughly when UNIX and C were first composed. As I write this, `time` returns 1,147,182,523—not a very good way to tell the time. There are a number of functions that will turn this into hours, minutes, or months; see any reference on C's standard library (such as the GNU C Library documentation) for details. But this illegible form provides the perfect seed for an RNG, and you get a new one every second.

Listing 11.6, `time.c`, shows a sample program that produces ten draws from an RNG seeded with the time. This is not industrial-strength random number generation, because patterns could conceivably slip in. For an example of the extreme case, try compiling `time.c`, and run it continuously from the shell. In a Bourne-family shell (what you are probably using on a POSIX system), try

```
while true; do ./time; done
```

You should see the same numbers a few dozen times until the clock ticks over, and then another stream repeated several times. [You can get your command prompt

```
#include <time.h>
#include <apop.h>

int main(){
    gsl_rng *r = apop_rng_alloc(time(NULL));
    for (int i =0; i< 10; i++)
        printf("%.3g\t", gsl_rng_uniform(r));
    printf("\n");
}
```

Listing 11.6 Seeding an RNG using the time. Online source: `time.c`.

back using <ctrl-c>.] If you have multiple processors and run one simulation on each, then runs that start in the same second will be replicating each other. Finally, if you ever hope to debug this program, then you will need to write down the time started so that you can replicate the runs that break:

```
//I assume the database is already open and has a one-column
//table named runs. The time is a long integer
//in the GNU standard library, so its printf tag is %li.
long int right_now = time(NULL);
apop_query("insert into runs (%li);", right_now);
gsl_rng *r = apop_rng_alloc(right_now);
```

Caveats aside, if you just want to see some variety every time the program runs, then seeding with the time works fine.

※ *The standard C RNG* If the GSL is not available, the standard C library includes a `rand` function to make random draws and an `srand` function to set its seed. E.g.:

```
#include <stdlib.h>
srand(27);
printf("One draw from a U[0,1]: %g", rand()/(RAND_MAX +0.0));
```

The GSL's RNGs are preferable for a few reasons. First, `gsl_ran_max(r)` is typically greater than `RAND_MAX`, giving you greater variation and precision. Second, the C language standard specifies that there must be a `rand` function, but not how it works, meaning two machines may give you different streams of random numbers for the same seed.

Finally, `rand` gives your entire program exactly one stream of numbers, while you can initialize many `gsl_rng`s that will be independent of each other. For example, if you give every agent in a simulation its own RNG, you can re-run the simulation

with one agent added or removed (probably at a breakpoint in GDB) and are guaranteed that the variation is due to the agent, not RNG shifts. Here is some sample code to clarify how such a setup would be initialized:

```
typedef struct agent{
    long int agent_number;
    gsl_rng *r;
    ...
} agent;

void init_agent(agent *initme, int agent_no){
    initme->agent_number = agent_no;
    initme->r = apop_rng_init(agent_no);
    ...
}
```

- Random number generators produce a deterministic sequence of values. This is a good thing, because we could never debug a program or replicate results without it. Change the stream by initializing it with a different seed.
- Given a random number generator, you can use it to draw from any common distribution, from a histogram, or from a data set.

11.2 DESCRIPTION: FINDING STATISTICS FOR A DISTRIBUTION

For many statistic-distribution pairs, there exists a closed-form solution for the statistic: the kurtosis of a $\mathcal{N}(\mu, \sigma)$ is $3\sigma^4$, the variance of a Binomial distribution is $np(1-p)$, et cetera. You can also take recourse in the Slutsky theorem, that says that given an estimate r for some statistic ρ and a continuous function $f(\cdot)$, then $f(r)$ is a valid estimate of $f(\rho)$. Thus, sums or products of means, variances, and so on are easy to calculate as well.

However, we often find situations where we need a global value like a mean or variance, but have no closed-form means of calculating that value. Even when there is a closed-form theorem that begins *in the limit as* $n \to \infty$*, it holds that...*, there is often evidence of the theorem falling flat for the few dozen data points before us.

One way to calculate the expected value of a statistic $f(\cdot)$ given probability distribution $p(\cdot)$ would be a numeric integral over the entire domain of the distribution. For a resolution of 100,000 slices, write a loop to sum

$$E[f(\cdot)|p(\cdot)] = \frac{f(-500.00)\cdot p(-500.00)}{100,000} + \frac{f(-499.99)\cdot p(-499.99)}{100,000} + \cdots + \frac{f(499.99)\cdot p(499.99)}{100,000} + \frac{f(500.00)\cdot p(500.00)}{100,000}.$$

This can be effective, but there are some details to be hammered out: if your distribution has a domain over $(-\infty, \infty)$, should you integrate over $[-3, 3]$ or $[-30, 30]$? You must decide up-front how fine the resolution will be, because (barring some tricks) each resolution is a new calculation rather than a modification of prior calculations. If you would like to take this approach, the GSL includes a set of numerical integration functions.

Another approach is to evaluate $f(\cdot)$ at values randomly drawn from the distribution. Just as Listing 11.2 produced a nice picture of the Beta distribution by taking enough random draws, a decent number of random draws can produce a good estimate of any desired statistic of the overall distribution. Values will, by definition, appear in proportion to their likelihood, so the $p(\cdot)$ part takes care of itself. There is no cutoff such that the tails of the distribution are assumed away. You can incrementally monitor $E[f(\cdot)]$ at 1,000 random draws, at 10,000, and so on, to see how much more you are getting with the extra time.

An example: the kurtosis of a t distribution You probably know that a t distribution is much like a Normal distribution but with fatter tails, but probably not how much fatter those tails are. The kurtosis of a vector is easy to calculate—just call `apop_vector_kurtosis`. By taking a million or so draws from a t distribution, we can produce a vector whose values cover the distribution rather well, and then find the kurtosis of that vector.

Listing 11.7 shows a program to execute this procedure.

- The `main` function just sets up the header of the output table (see Table 11.8) and calls `one_df` for each df.
- The `for` loop on lines 6–7 does the draws, storing them in the vector `v`.
- Once the vector is filled, line eight calculates the partially normalized kurtosis. That is, it calculates raw kurtosis over variance squared; see the box on page 230 on the endless debate over how best to express kurtosis.

The closed-form formula for the partially-normalized kurtosis of a t distribution with $df > 4$ degrees of freedom is $(3df - 6)/(df - 4)$. For $df \leq 4$, the kurtosis is undefined, just as the variance is undefined for a Cauchy distribution (i.e., a t distribution with $df = 1$). At $df = 5$, it is finite, and it monotonically decreases as df continues upwards.

```
1  #include <apop.h>
2
3  void one_df(int df, gsl_rng *r){
4    long int i, runct = 1e6;
5    gsl_vector  *v = gsl_vector_alloc(runct);
6      for (i=0; i< runct; i++)
7            gsl_vector_set(v, i, gsl_ran_tdist(r, df));
8      printf("%i\t %g", df, apop_vector_kurtosis(v)/gsl_pow_2(apop_vector_var(v)));
9      if (df > 4)
10           printf("\t%g", (3.*df - 6.)/(df-4.));
11     printf("\n");
12     gsl_vector_free(v);
13 }
14
15 int main(){
16   int df, df_max = 31;
17   gsl_rng *r = apop_rng_alloc(0);
18     printf("df\t k (est)\t k (actual)\n");
19     for (df=1; df< df_max; df++)
20           one_df(df, r);
21 }
```

Listing 11.7 Monte Carlo calculation of kurtoses for the t distribution family. Online source: `tdistkurtosis.c`.

Table 11.8 shows an excerpt from the simulation output, along with the true kurtosis.

The exact format for the variance of the estimate of kurtosis will not be given here ($\mathbb{Q}$: use the methods here to find it), but it falls with df: with $df < 4$, we may as well take the variance of the kurtosis estimate as infinite, and it shrinks as df grows. Correspondingly, the bootstrap estimates of the kurtosis are unreliable for $df = 5$ or 6, but are more consistent for df over a few dozen. You can check this by re-running the program with different seeds (e.g., replacing the zero seed on line 17 of the code with `time(NULL)`).

$\sum$

- By making random draws from a model, we can make statements about global properties of the model that improve in accuracy as the number of draws $\to \infty$.
- The variance of the Monte Carlo estimation of a parameter tends to mirror the variance of the underlying parameter. The maximum likelihood estimator for a parameter achieves the Cramér–Rao lower bound, so the variance of the Monte Carlo estimate will be larger (perhaps significantly so).

df	k (est)	k (analytic)
1	183640	
2	5426.32	
3	62.8055	
4	19.2416	
5	8.5952	9
6	6.00039	6
7	4.92161	5
8	4.52638	4.5
9	4.17413	4.2
10	4.00678	4
15	3.55957	3.54545
20	3.37705	3.375
25	3.281	3.28571
30	3.23014	3.23077

Table 11.8 The fatness of the tails of t distributions at various df.

11.3 INFERENCE: FINDING STATISTICS FOR A PARAMETER

The t distribution example made random draws from a closed-form distribution, in order to produce an estimate of a function of the distribution parameters. Conversely, say that we want to estimate a function of the data, such as the variance of the mean of a data set, $\widehat{\mathrm{var}}(\hat{\mu}(\mathbf{X}))$. We have only one data set before us, but we can make random draws from $\mathbf{X}$ to produce several values of the statistic $\hat{\mu}(\mathbf{X}_r)$, where $\mathbf{X}_r$ represents a random draw from $\mathbf{X}$, and then estimate the variance of those draws. This is known as the *bootstrap* method of estimating a variance.

BOOTSTRAPPING THE STANDARD ERROR

The core of the bootstrap is a simple algorithm:

Repeat the following m times:
  Let $\tilde{\mathbf{X}}$ be n elements randomly drawn from the data, with replacement.
  Write down the statistic(s) $\beta(\tilde{\mathbf{X}})$.
Find the *standard error* of the m values of $\beta(\tilde{\mathbf{X}})$.

This algorithm bears some resemblance to the steps demonstrated by the CLT demo in Listing 9.1 (page 298): draw m iid samples, find a statistic like the mean of each, and then look at the distribution of the several statistics (rather than the underlying data itself). So if $\beta(\tilde{\mathbf{X}})$ is a mean-like statistic (involving a sum over

n), then the CLT applies directly, and the artificial statistic approaches a Normal distribution. Thus, it makes sense to apply the usual Normal distribution-based test to test hypotheses about the true value of β.

```
1  #include "oneboot.h"
2
3  int main(){
4      int rep_ct = 10000;
5      gsl_rng *r = apop_rng_alloc(0);
6      apop_db_open("data-census.db");
7      gsl_vector *base_data = apop_query_to_vector("select in_per_capita from income where
           sumlevel+0.0 =40");
8      double RI = apop_query_to_float("select in_per_capita from income where sumlevel+0.0
           =40 and geo_id2+0.0=44");
9      gsl_vector *boot_sample = gsl_vector_alloc(base_data->size);
10     gsl_vector *replications = gsl_vector_alloc(rep_ct);
11     for (int i=0; i< rep_ct; i++){
12         one_boot(base_data, r, boot_sample);
13         gsl_vector_set(replications, i, apop_mean(boot_sample));
14     }
15     double stderror = sqrt(apop_var(replications));
16     double mean = apop_mean(replications);
17     printf("mean: %g; standard error: %g; (RI-mean)/stderr: %g; p value: %g\n",
18         mean, stderror, (RI-mean)/stderror, 2*gsl_cdf_gaussian_Q(fabs(RI-mean), stderror));
19 }
```

Listing 11.9 Bootstrapping the standard error of the variance in state incomes per capita. Online source: `databoot.c`.

```
#include "oneboot.h"

void one_boot(gsl_vector *base_data, gsl_rng *r, gsl_vector* boot_sample){
    for (int i =0; i< boot_sample->size; i++)
        gsl_vector_set(boot_sample, i,
                gsl_vector_get(base_data, gsl_rng_uniform_int(r, base_data->size)));
}
```

Listing 11.10 A function to produce a bootstrap draw. Online source: `oneboot.c`.

Listings 11.10 and 11.9 shows how this algorithm is executed in code. It tests the hypothesis that Rhode Island's income per capita is different from the mean.

- Lines 4–8 of Listing 11.9 are introductory material and the queries to pull the requisite data. For Rhode Island, this is just a scalar, used in the test below, but for the rest of the country, this is a vector of 52 numbers (one for each state, commonwealth, district, and territory in the data).

- Lines 11–14 show the main loop repeated m times in the pseudocode above, which makes the draws and then finds the mean of the draws.
- Listing 11.10 is a function to make a single bootstrap draw, which will be used in a few other scripts below. The `one_boot` function draws with replacement, which simply requires repeatedly calling `gsl_rng_uniform_int` to get a random index and then writing down the value at that index in the data vector.
- Lines 15 and 16 of Listing 11.10 find the mean and standard error of the returned data, and then lines 17–18 run the standard hypothesis test comparing the mean of a Normal distribution to a scalar.
- Recall the discussion on page 300 about the standard deviation of a data set, which is the square root of its variance, σ; and the standard deviation of the mean of a data set, which is $\sigma/\sqrt{n}$. In this case, we are interested in the distribution of $\beta(\tilde{\mathbf{X}})$ itself, not the distribution of $E(\beta(\tilde{\mathbf{X}}))$s, so we use σ instead of $\sigma/\sqrt{n}$.

$\mathbb{Q}_{11.2}$ Rewrite `databoot.c` to calculate the standard error of `base_data` directly (without bootstrapping), and test the same hypothesis using that standard error estimate rather than the bootstrapped version. Do you need to test with your calculated value of $\hat{\sigma}$ or with $\hat{\sigma}/\sqrt{n}$?

```
1  #include <apop.h>
2
3  int main(){
4      int draws = 5000, boots = 1000;
5      double mu = 1., sigma = 3.;
6      gsl_rng *r = apop_rng_alloc(2);
7      gsl_vector *d = gsl_vector_alloc(draws);
8      apop_model *m = apop_model_set_parameters(apop_normal, mu, sigma);
9      apop_data *boot_stats = apop_data_alloc(0, boots, 2);
10     apop_name_add(boot_stats->names, "mu", 'c');
11     apop_name_add(boot_stats->names, "sigma", 'c');
12         for (int i =0; i< boots; i++){
13             for (int j =0; j< draws; j++)
14                 apop_draw(gsl_vector_ptr(d, j), r, m);
15             apop_data_set(boot_stats, i, 0, apop_vector_mean(d));
16             apop_data_set(boot_stats, i, 1, sqrt(apop_vector_var(d)));
17         }
18         apop_data_show(apop_data_covariance(boot_stats));
19         printf("Actual:\n var(mu) %g\n", gsl_pow_2(sigma)/draws);
20         printf("var(sigma): %g\n", gsl_pow_2(sigma)/(2*draws));
21 }
```

Listing 11.11 Estimate the covariance matrix for the Normal distribution Online source: `normalboot.c`.

A Normal example Say that we want to know the covariance matrix for the estimates of the Normal distribution parameters, $(\hat{\mu}, \hat{\sigma})$. We could look it up and find that it is

$$\begin{bmatrix} \frac{\sigma^2}{n} & 0 \\ 0 & \frac{\sigma^2}{2n} \end{bmatrix},$$

but that would require effort and looking in books.[4] So instead we can write a program like Listing 11.11 to generate the covariance for us. Running the program will show that the computational method comes reasonably close to the analytic value.

- The introductory material up to line 11 allocates a vector for the bootstrap samples and a data set for holding the mean and standard deviation of each bootstrap sample.
- Instead of drawing permutations from a fixed data set, this version of the program produces each artificial data vector via draws from the Normal distribution itself, on lines 13–14, and then lines 15–16 write down statistics for each data set.
- In this case, we are producing two statistics, μ and σ, so line 18 finds the covariance matrix rather than just a scalar variance.

$\mathbb{Q}_{11.3}$ On page 321, I mentioned that you could test the claim that the kurtosis of a Normal distribution equals $3\sigma^4$ via a bootstrap. Modify `normalboot.c` to test this hypothesis. Where the program currently finds means and variances of the samples, find each sample's kurtosis. Then run a t test on the claim that the mean of the vector of kurtoses is $3\sigma^4$.

$\sum$

- ➤ To test a hypothesis about a model parameter, we need to have an estimate of the parameter's variance.
- ➤ If there is no analytic way to find this variance, we can make multiple draws from the data itself, calculate the parameter, and then find the variance of that artificial set of parameters. The Central Limit Theorem tells us that the artificial parameter set will approach a well-behaved Normal distribution.

[4] E.g., Kmenta (1986, p 182). That text provides the covariance matrix for the parameters $(\hat{\mu}, \hat{\sigma}^2)$. The variance of $\hat{\mu}$ is the same, but the variance of $\hat{\sigma}^2 = \frac{2\sigma^4}{n}$.

11.4 DRAWING A DISTRIBUTION

To this point, we have been searching for global parameters of either a distribution or a data set, but the output has been a single number or a small matrix. What if we want to find the entire distribution?

We have at our disposal standard RNGs to draw from Normal or Uniform distributions, and this section will present are a few techniques to transform those draws into draws from the distribution at hand. For the same reasons for which random draws were preferable to brute-force numeric integration, it can be much more efficient to produce a picture of a distribution via random draws than via *grid search*: the draws focus on the most likely points, the tails are not cut off at an arbitrary limit, and a desired level of precision can be reached with less computation.

And remember, *anything* that produces a nonnegative univariate measure can be read as a subjective likelihood function. For example, say that we have real-world data about the distribution of firm sizes. The model on page 253 also gives us an artificial distribution of firm sizes given input parameters such as the standard error (currently hard-coded into the model's `growth` function). Given a run, we could find the distance $d(\sigma)$ between the actual distribution and the artificial. Notice that it is a function of σ, because every σ produces a new output distribution and therefore a new distance to the actual. The most likely value of σ is that which produces the smallest distance, so we could write down $L(\sigma) = 1/d(\sigma)$, for example.[5] Then we could use the methods in this chapter and the last to find the most likely parameters, draw a picture of the likelihood function given the input parameters, calculate the variance given the subjective likelihood function, or test hypotheses given the subjective likelihood.

IMPORTANCE SAMPLING

There are many means of making draws from one distribution to inform draws from another; Train (2003, Chapter 9) catalogs many of them.

For example, *importance sampling* is a means of producing draws from a new function using a well-known function for reference. Let $f(\cdot)$ be the function of interest, and let $g(\cdot)$ be a well-known distribution like the Normal or Uniform. We want to use the stock Normal or Uniform distribution RNGs to produce an RNG for an arbitrary function.

For $i = 1$ to a few million:
    Draw a new point, x_i, from the reference distribution $g(\cdot)$.
    Give the point weighting $f(x_i)/g(x_i)$.
Bin the data points into a histogram (weighting each point appropriately).

[5] Recall the invariance principle from page 351, that basic transformations of the likelihood function don't change the data therein, so we don't have to fret over the exact form of the subjective likelihood function, and would get the same results using $1/d$ as using $1/d^2$ or $1/(d + 0.1)$, for example.

At the end of this, we have a histogram that is a valid representation of $f(\cdot)$, which can be used for making draws, graphing the function, or calculating global information.

What reference distribution should you use? Theoretically, any will do, but engineering considerations advise that you pick a function that is reasonably close to the one you are trying to draw from—you want the high-likelihood parts of $g(\cdot)$ to match the high-likelihood parts of $f(\cdot)$. So if $f(\cdot)$ is generally a bell curve, use a Normal distribution; if it is focused near zero, use a Lognormal or Exponential; and if you truly have no information, fall back to the Uniform (perhaps for a rough exploratory search that will let you make a better choice).

MARKOV CHAIN MONTE CARLO *Markov Chain Monte Carlo* is an iterative algorithm that starts at one state and has a rule defining the likelihood of transitioning to any other given state from that initial state—a Markov Chain. In the context here, the state is simply a possible value for the parameter(s) being estimated. By jumping to a parameter value in proportion to its likelihood, we can get a view of the probability density of the parameters. The exact probability of jumping to another point is given a specific form to be presented shortly.

This is primarily used for Bayesian updating, so let us review the setup of that method. The analysis begins with a prior distribution expressing current beliefs about the parameters, $P_{\text{prior}}(\boldsymbol{\beta})$, and a likelihood function $P_{\text{L}}(\mathbf{X}|\boldsymbol{\beta})$ expressing the likelihood of an observation given a value of the parameters. The goal is to combine these two to form a posterior; as per page 258, Bayes's Rule tells us that this is

$$P_{\text{post}}(\beta|\mathbf{X}) = \frac{P_{\text{L}}(\mathbf{X}|\beta)P_{\text{prior}}(\beta)}{\int_{\forall B\in\mathbb{B}} P_{\text{L}}(\mathbf{X}|B)P_{\text{prior}}(\beta)dB}$$

The numerator is easy to calculate, but the denominator is a global value, meaning that we can not know it with certainty without evaluating the numerator at an infinite number of points. The Metropolis–Hastings algorithm, a type of MCMC, offers a solution.

The gist is that we start at an arbitrary point, and draw a new candidate point from the prior distribution. If the candidate point is more likely than the current point (according to P_{L}), jump to it, and if the candidate point is less likely, perhaps jump to it anyway. After a burn-in period, record the values. The histogram of recorded values will be the histogram of the posterior distribution. To be more precise, here is a pseudocode description of the algorithm:

Begin by setting β_0 to an arbitrary starting value.
For $i = 1$ to a few million:
    Draw a new proposed point, β_p, from P_{prior}.
    If $P_{\text{L}}(\mathbf{X}|\beta_p) > P_{\text{L}}(\mathbf{X}|\beta_{i-1})$
        $\beta_i \leftarrow \beta_p$
    else
        Draw a value $u \sim \mathcal{U}[0, 1]$
        If $u < P_{\text{L}}(\mathbf{X}|\beta_p)/P_{\text{L}}(\mathbf{X}|\beta_{i-1})$
            $\beta_i \leftarrow \beta_p$
        else
            $\beta_i \leftarrow \beta_{i-1}$
    If $i > 1,000$ or so, record β_i.
Report the histogram of β_is as the posterior distribution of β.

As should be evident from the description of the algorithm, it is primarily used to go from a prior to a posterior distribution. To jump to a new point β_{new}, it must first be chosen by the prior (which happens with probability $P_{\text{prior}}(\beta_{\text{new}})$) and then will be selected with a likelihood roughly proportional to $P_{\text{L}}(\mathbf{x}|\beta_{\text{new}})$, so the recorded draw will be proportional to $P_{\text{prior}}(\beta_{\text{new}})P_{\text{L}}(\mathbf{x}|\beta_{\text{new}})$. This rough intuitive argument can be made rigorous to prove that the points recorded are a valid representation of the posterior; see Gelman *et al.* (1995).

Why not just draw from the prior and multiply by the likelihood directly, rather than going through this business of conditionally jumping? It is a matter of efficiency. The more likely elements of the posterior are more likely to contribute a large amount to the integral in the denominator of the updating equation above, so we can estimate that integral more efficiently by biasing draws toward the most likely posterior values, and the jumping scheme achieves this with fewer draws than the naïve draw-from-the-prior scheme does.

You have already seen one concrete example of MCMC: the simulated annealing algorithm from Chapter 10. It also jumps from point to point using a decision rule like the one above. The proposal distribution is basically an improper uniform distribution—any point is as likely as any other—and the likelihood function is the probability distribution whose optimum the simulated annealing is seeking. The difference is that the simulated annealing algorithm makes smaller and smaller jumps; combined with the fact that MCMC is designed to tend toward more likely points, it will eventually settle on a maximum value, at which point we throw away all the prior values. For the Bayesian updating algorithm here, jumps are from a fixed prior, and tend toward the most likely values of the posterior, but the $1,000\text{th}$ jump is as likely to be a long one as the first, and we use every point found to produce the output distribution.

At the computer, the function to execute this algorithm is `apop_update`. This function takes in a parametrized prior model, an unparametrized likelihood and data, and outputs a parametrized posterior model.

The output model will be one of two types. As with the Beta/Binomial example on page 259, there are many well-known *conjugate distributions*, where the posterior will be of the same form as the prior distribution, but with updated parameters. For example, if the prior is named *Gamma distribution* (which the `apop_gamma` model is named) and the likelihood is named *Exponential distribution* (and `apop_exponential` is so named), then the output of `apop_update` will be a closed-form Gamma distribution with appropriately updated parameters. But if the tables of closed form conjugates offer nothing applicable, the system will fall back on MCMC and output an `apop_histogram` model with the results of the jumps.

$\mathbb{Q}_{11.4}$

Check how close the MCMC algorithm comes to the closed-form model.

- Pick an arbitrary parameter for the Exponential distribution, β_I. Generate ten or twenty thousand random data points.
- Pick a few arbitrary values for the Gamma distribution parameters, $\boldsymbol{\beta}_{\text{prior}}$.
- Estimate the posterior, by sending the two distributions, the randomly-generated data, and $\boldsymbol{\beta}_{\text{prior}}$ to `apop_update`.
- Make a copy of the `apop_exponential` model and change its name to something else (so that `apop_update` won't find it in the conjugate distribution table). Re-send everything to `apop_update`.
- Use a goodness-of-fit test to find how well the two output distributions match.

$\mathbb{Q}_{11.5}$

The output to the updating process is just another distribution, so it can be used as the prior for a new updating step. In theory, distributions repeatedly updated with new data will approach putting all probability mass on the true parameter value. Continuing from the last exercise:

- Regenerate a new set of random data points from the Exponential distribution using the same β_I.
- Send the new data, the closed-form posterior from the prior exercise, and the same Exponential model to the updating routine.
- Plot the output.
- Once you have the generate/update/plot routine working, call it from a `for` loop to generate an animation beginning with the prior and continuing with about a dozen updates.
- Repeat with the MCMC-estimated posteriors.

11.5 NON-PARAMETRIC TESTING

Section 11.3 presented a procedure for testing a claim about a data set that consisted of drawing a bootstrap data set, regenerating a statistic, and then using the many draws of the statistic and the CLT to say something about the data. But we can test some hypotheses by simply making draws from the data and checking their characteristics, without bothering to produce a statistic and use the CLT.

To give a concrete example of testing without parametric assumptions, consider the permutation test. Say that you draw ten red cards and twenty black cards from what may be a crooked deck. The hypothesis is that the mean value of the red cards you drew equals the mean of the black cards (counting face cards however you like). You could put the red cards in one pile to your left, the black cards in a pile to your right, calculate $\overline{\mathbf{x}}_{\mathrm{red}}$, $\hat{\sigma}_{\mathrm{red}}$, $\overline{\mathbf{x}}_{\mathrm{black}}$, and $\hat{\sigma}_{\mathrm{black}}$, assume $\overline{\mathbf{x}}_{\mathrm{black}} - \overline{\mathbf{x}}_{\mathrm{red}} \sim t$ distribution, and run a traditional t test to compare the two means.

But if μ_{red} really equals μ_{black}, then the fact that some of the cards you drew are black and some are red is irrelevant to the value of $\overline{\mathbf{x}}_{\mathrm{left}} - \overline{\mathbf{x}}_{\mathrm{right}}$. That is, if you shuffled together the stacks of black and red cards, and dealt out another ten cards to your left and twenty to your right, then $\overline{\mathbf{x}}_{\mathrm{left}} - \overline{\mathbf{x}}_{\mathrm{right}}$ should not be appreciably different. If you deal out a few thousand such shuffled pairs of piles, then you can draw the distribution of values for $\overline{\mathbf{x}}_{\mathrm{left}} - \overline{\mathbf{x}}_{\mathrm{right}}$. If $\overline{\mathbf{x}}_{\mathrm{red}} - \overline{\mathbf{x}}_{\mathrm{black}}$ looks as if it is very far from the center of that distribution, then we can reject the claim that the color of the cards is irrelevant.

What is the benefit of all this shuffling and redealing when we could have just run a t test to compare the two groups? On the theoretical level, this method relies on the assumption of the bootstrap principle rather than the assumptions underlying the CLT. Instead of assuming a theoretical distribution, you can shuffle and redraw to produce the distribution of outcome values that matches the true sample data (under the assumption that $\mu_{\mathrm{red}} = \mu_{\mathrm{black}}$), and then rely on the bootstrap principle to say that what you learned from the sample is representative of the population from which the sample was drawn.

On a practical level, the closer match to the data provides real benefits. Lehmann & Stein (1949) showed that the permutation test is more powerful—it is less likely to fail to reject the equality hypothesis when the two means are actually different.

In pseudocode, here is the test procedure:[6]

[6]The test is attributed to Chung & Fraser (1958). From p 733: "The computation took one day for programming and two minutes of machine time."

$\mu \leftarrow |\overline{\mathbf{x}}_{\text{red}} - \overline{\mathbf{x}}_{\text{black}}|$
$\mathbf{d} \leftarrow$ the joined test and control vectors.
Allocate a million-element vector $\mathbf{v}$.
For $i = 1$ to a million:
    Draw (without replacement) from $\mathbf{d}$ a vector the same size as $\overline{\mathbf{x}}_{\text{red}}$, $\mathbf{L}$.
    Put the other elements of $\mathbf{d}$ into a second vector, $\mathbf{R}$.
    $\mathbf{v}_i \leftarrow |\overline{\mathbf{L}} - \overline{\mathbf{R}}|$.
$\alpha \leftarrow$ the percentage of $\mathbf{v}_i$ less than μ.
Reject the claim that $\mu = 0$ with confidence α.
Fail to reject the claim that $\mu = 0$ with confidence $1 - \alpha$.

$\mathbb{Q}_{11.6}$ Baum *et al.* (2008) sampled genetic material from a large number of cases with bipolar disorder and controls who were confirmed to not have bipolar disorder. To save taxpayer money, they pooled the samples to form the groups listed in the file `data-genes` in the code supplement. Each line of that file lists the percent of the pool where the given SNP (single nucleotide polymorphism) has a given marker.
For which of the SNPs can we reject the hypothesis of no difference between the cases and controls? Write a program to read the data into a database, then use the above algorithm to test whether we reject the hypothesis of no difference in marker frequency for a given SNP, and finally write a `main` that runs the test for each SNPs in the database. The data here is cut from 550,000 SNPs, so base the Bonferroni correction on that many tests.

Other nonparametric tests tend to follow a similar theme: given a null hypothesis that some bins are all equiprobable, we can develop the odds that the observed data occurred. See Conover (1980) for a book-length list of such tests, including Kolmogorov's method from page 323 of this book.

TESTING FOR BIMODALITY As the finale to the book, Listing 11.12 shows the use of kernel densities to test for multimodality. This involves generating a series of kernel density estimates of the data, first with the original data, and then with a few thousand bootstrap estimates, and then doing a nonparametric test of the hypothesis that the distribution has fewer than n modes, for each value of n.

Recall the kernel density estimate from page 262, which was based on the form

$$\hat{f}(t, \mathbf{x}, h) = \frac{\sum_{i=1}^{n} \mathcal{N}((t - x_i)/h)}{n \cdot h},$$

where $\mathbf{x}$ is the vector of n data points observed, $\mathcal{N}(y)$ is a Normal$(0, 1)$ PDF evaluated at y, and $h \in \mathbb{R}^+$ is the bandwidth. Let k be the number of modes.

Figure 7.15 (p 262) showed that as h rises, the spike around each point spreads out and merges with other spikes, so k falls, until eventually the entire data set is subsumed under one single-peaked curve, so $k = 1$. Thus, there is a monotonic relationship between h and the number of modes in the density function induced by h.

Silverman (1981) offers a bootstrap method of testing a null hypothesis of the form *the distribution has more than k modes*. Let h_0 be the smallest value of h such that $\hat{f}(t, X, h_0)$ has more than k modes; then the null hypothesis is that h_0 is consistent with the data.

We then draw a few hundred bootstrap samples from the data, $\mathbf{x}'_1, \ldots, \mathbf{x}'_{200}$; write down $\hat{f}(t, \mathbf{x}'_1, h_0), \ldots, \hat{f}(t, \mathbf{x}'_{200}, h_0)$; and count the modes of each of these functions. Silverman shows that, thanks to the monotonic relationship between h and the number of modes, the percentage of bootstrap distributions with more than k modes matches the likelihood that $\hat{f}(t, \mathbf{x}, h_0)$ is consistent with the data. If we reject this hypothesis, then we would need a lower value of h, and therefore more modes, to explain the data.

Listing 11.12 shows the code used to produce these figures and test a data set of television program ratings for bimodality. Since it draws a few hundred bootstrap samples, link it with `oneboot.c` from page 368.

- The `countmodes` function makes two scans across the range of the data. The first generates a histogram: at each point on the x-axis, it piles up the value at that point of a Normal distribution centered at every data point. The result will be a histogram like those pictured in Figure 7.15, page 262, or those that are produced in the Gnuplot animation the program will write to the `kernelplot` file. The second pass checks for modes, by simply asking each point whether it is higher than the points to its left and right.
- $h \to k$: For each index `i`, `ktab[i]` is intended to be the smallest value of `h` that produces fewer than `i` modes. The `fill_kmap` function calls `countmodes` to find the number of modes produced by a range of values of `h`. The function runs from the largest `h` to the smallest, so the last value of `h` written to any slot will be the smallest value that produces that mode count.[7]
- $k \to p$: Now that we have the smallest bandwidth h that produces a given k using the actual data, `boot` does the bootstrapping: `one_boot` (p 368) uses a straightforward `for` loop to produce a sample, then the same `countmodes` function used above is applied to the artificial sample. If the mode count is larger than the number of modes in the original, it is a success for the hypothesis. The function then

[7] Also, the `for` loop in this function demonstrates a pleasant trick for scanning a wide range: instead of stepping by a fixed increment, it steps by percentages, so it quickly scans the hundreds but looks at the lower end of the range in more detail.

returns the percent successes, which we can report as a p value for the hypothesis that the distribution has more than k modes.

- In some parts of the output, such as for $k = 12$, the battery of tests finds low confidence that there are more than k modes and high confidence that there are more than $k + 1$ modes. If both tests were run with the same value of h, this would be inconsistent, but we are using the smallest level of smoothing possible in each case, so the tests for k and $k + 1$ parameters have little to do with each other. For small k, the output is monotonic, and shows that we can be very confident that the data set has more than three modes, and reasonably confident that it has more than four.

```
#include "oneboot.h"

double modect_scale, modect_min, modect_max,
        h_min=25, h_max=500, max_k = 20,
        boot_iterations = 1000,
        pauselength = 0.1;
char outfile[] = "kernelplot";

void plot(apop_data *d, FILE *f){
    fprintf(f, "plot '-' with lines\n");
    apop_data_print(d, NULL);
    fprintf(f, "e\npause %g\n", pauselength);
}

int countmodes(gsl_vector *data, double h, FILE *plothere){
  int len =(modect_max-modect_min)/modect_scale;
  apop_data *ddd = apop_data_calloc(0, len, 2);
  double sum, i=modect_min;
    for (size_t j=0; j < ddd->matrix->size1; j ++){
        sum = 0;
        for (size_t k = 0; k< data->size; k++)
            sum += gsl_ran_gaussian_pdf((i-gsl_vector_get(data,k))/h,1)/(data->size*h);
        apop_data_set(ddd, j, 0, i);
        apop_data_set(ddd, j, 1, sum);
        i+=modect_scale;
    }
    int modect =0;
    for (i = 1; i< len-1; i++)
        if(apop_data_get(ddd,i,1)>=apop_data_get(ddd,i-1,1)
            && apop_data_get(ddd,i,1)>apop_data_get(ddd,i+1,1))
                modect++;
    if (plothere) plot(ddd, plothere);
    apop_data_free(ddd);
    return modect;
}

void fill_kmap(gsl_vector *data, FILE *f, double *ktab){
    for (double h = h_max; h> h_min; h*=0.99){
```

```
            int val = countmodes(data, h, f);
            if (val <max_k)
                ktab[val − 1] = h;
        }
}

double boot(gsl_vector *data, double h0, int modect_target, gsl_rng *r){
  double over_ct = 0;
  gsl_vector *boots = gsl_vector_alloc(data->size);
    for (int i=0; i < boot_iterations; i++){
        one_boot(data, r, boots);
        if (countmodes(boots, h0, NULL) > modect_target)
            over_ct++;
    }
    gsl_vector_free(boots);
    return over_ct/boot_iterations;
}

apop_data *produce_p_table(gsl_vector *data, double *ktab, gsl_rng *r){
  apop_data *ptab = apop_data_alloc(0, max_k, 2);
    apop_name_add(ptab->names, "Mode", 'c');
    apop_name_add(ptab->names, "Likelihood of more", 'c');
    for (int i=0; i< max_k; i++){
        apop_data_set(ptab, i, 0, i);
        apop_data_set(ptab, i, 1, boot(data, ktab[i], i, r));
    }
    return ptab;
}

void setvars(gsl_vector *data){ //rescale based on the data.
    double m1 = gsl_vector_max(data);
    double m2 = gsl_vector_min(data);
    modect_scale = (m1−m2)/200;
    modect_min = m2−(m1−m2)/100;
    modect_max = m1+(m1−m2)/100;
}

int main(){
    APOP_COL(apop_text_to_data("data−tv", 0,0), 0, data);
    setvars(data);
    FILE *f = fopen(outfile, "w");
    apop_opts.output_type = 'p';
    apop_opts.output_pipe = f;
    double *ktab = calloc(max_k, sizeof(double));
    fill_kmap(data, f, ktab);
    fclose(f);
    gsl_rng *r = apop_rng_alloc(3);
    apop_data_show(produce_p_table(data, ktab, r));
}
```

Listing 11.12 Silverman's kernel density test for bimodality. Online source: `bimodality.c`

Another way to speed the process in `bimodality.c` is to clump the data before summing the Normal distributions. If there are three points at $-.1$, 0, and .1, then this will require three calculations of the Normal PDF for every point along the real line. If they are clumped to 0, then we can calculate the Normal PDF for $\mu = 0$ times three, which will run three times as fast.

$\mathbb{Q}_{11.7}$ Write a function to group nearby data points into a single point with a given weight (you can use the `weight` element of the `apop_data` structure to record it). Rewrite the `countmodes` function to use the clumped and weighted data set. How much clumping do you need to do before the results degrade significantly?

$\sum$

- Via resampling, you can test certain hypotheses without assuming parametric forms like the t distribution.
- The typical kernel density estimate consists of specifying a distribution for every point in the data set, and then combining them to form a global distribution. The resulting distribution is in many ways much more faithful to the data.
- As the bandwidth of the sub-distributions grows, the overall distribution becomes smoother.
- One can test hypotheses about multimodality using kernel densities, by finding the smallest level of smoothing necessary to achieve n-modality, and bootstrapping to see the odds that that level of smoothing would produce n-modality.