Lab Manual for Biol 5221

©Alan Rogers and Jon Seger

April 22, 2024

Contents

Project 4

Simulating Genetic Drift and Mutation

Allele frequencies change for several reasons, one of which is sheer chance. These random changes, which we call *genetic drift*, occur for many reasons. When a heterozygote produces a gamete, that gamete may inherit either of the two parental alleles—the choice between them is random. Similarly, there are random forces that affect how long we live, whether we succeed in mating, and how many children we produce if we do. All of these effects contribute to genetic drift, but population geneticists seldom try to model them in detail.

Instead, we think of genetic drift more abstractly. To produce each new generation, we imagine that the genes of the parental generation are sampled to produce those of the offspring. The size of this sample equals the number of gene copies in the offspring generation. On average, the allele frequency of the offspring generation is the same as that of the parental generation, but in any particular case it will be slightly higher or lower. The smaller the sample, the larger these random changes are likely to be—the larger, in other words, is the effect of genetic drift.

The conventional way to model genetic drift involves the "urn" metaphor. We think of the parental gene pool as an "urn" from which gametes (gene copies) are drawn at random, and we think of the offspring gene pool as an initially *empty* urn to be populated by the gametes drawn from the parents. Probability theory tells us about the distributions of outcomes to expect, given the population size (N) and the allele frequency (p) among the parents. But what if we want to observe a particular instance of this process, or more to the point, a sequence of such instances, representing the long-term evolution of the population? Before computers, this was very difficult. Now it's easy and enlightening.

4.1 The easy way to sample from an urn

Each time you draw a ball, it is red with a probability (p) that equals the relative frequency of red balls within the urn. This is just like tossing a coin, whose probability of "heads" is equal to p. You already *know* how to do that because we did it in a previous project. Thus, you have all the tools you need to write a computer simulation of genetic drift. In this project, however, we will show you an easier way.

In that previous project, you tossed coins using Python's `random` function. We could use that here again, but there's an easier way. On the class website there is a file called `pgen.py` which contains several functions that we have written for you. One of these is called `bnldev`. As explained on on page 51, `bnldev` generates random values (deviates) from the binomial distribution. Before using it, you must `import` the `bnldev` function into your program. After that, the command `bnldev(N,p)` will generate a binomial random deviate with parameters `N` (the number of trials) and `p` (the probability of "success" on each trial).

Here is an example:

```
from pgen import bnldev

for i in range(5):
    print(bnldev(10000, 0.5))
```

These three lines of code do five repetitions of Kerrich's coin experiment. In each repetition, `bnldev(10000, 0.5)` simulates the entire process of tossing a fair coin 10000 times. Try this on your computer. You will be amazed at how fast it runs.

4.2 Iterating over a list

Before we get started in earnest, you need to learn a Python trick. Below, you will need to run a simulation repeatedly, each time with a different value of the mutation rate, u. We want to show you two ways of doing that. Instead of doing an entire simulation, each pass through the loop in my examples will simply print the current value of u.

Here is the first approach:

```
u = [0.0001, 0.001, 0.01]

for i in range(len(u)):
    print(u[i])
```

Consider how this works. The `range` command generates the following list: `[0,1,2]`.[1] Then the `for` command iterates through this list. It is not necessary, however, to go through this extra step. The `for` command will iterate through *any* list, whether it was produced by `range` or not. Here is a re-write that avoids the `range` and `len` commands:

```
for u in [0.0001, 0.001, 0.01]:
    print(u)
```

The second approach is simpler, and not only because it leaves out `range` and `len`. It also allows one to refer to the current mutation rate simply as `u` rather than using the more cumbersome `u[i]`. Our bias in favor of the second method is pretty obvious, but you can use whichever method you please.

[1]This is not quite true in Python 3.x, but we don't need to worry about the difference.

4.3 Simulating drift and mutation

On his own page 56, Gillespie presents a program that simulates genetic drift. Here is an improved version, which uses `bnldev`:

```
 1   from __future__ import division, print_function # not needed for Python 3.x
     from pgen import bnldev

     twoN = 100      # population size (number of gene copies)
 5   maxgen = 100000 # maximum generation count

     p = 0.5       # initial frequency of allele A
     H = 2*p*(1-p) # initial heterozygosity
     g = 0         # count generations
10
     # Loop continues until heterozygosity is exhausted or we reach
     # the maximum number of generations.
     print("%4s %8s %8s" % ("g", "p", "H"))
     while H > 0 and g < maxgen:
15       # Draw balls from urn. x is a binomial random
         # variate representing the number of copies of A
         # in the next generation.
         x = bnldev(twoN, p)

20       p = x/twoN
         H = 2*p*(1-p)
         g += 1
         print("%4d %8.3f %8.3f" % (g, p, H))
```

You will find this code on the lab page of the course web site in a file called `drift.py`. Download it and run it a few times.

Once you get the program going, it should run to completion in 100 generations or so. The middle column of output is `p`, which should wobble around and end up either at 0 or 1. Meanwhile, `H` declines fairly steadily from 0.5 to 0.

Make sure you understand the code. Line 1 is not necessary if you're running a recent version of Python. It's there so that the code will run correctly under older versions of Python. In line 18, `x` is the number of copies of some allele (say A_1) in the new generation. Using `x`, the program calculates new values of `p` and `H`. Finally, it increments the count of generations and prints a line of output.

This program simulates drift in its purest form—no other evolutionary force is involved. In this exercise, you will add mutation into this simulation. In the end, you will need to loop over several values of u, the mutation rate. For the moment, however, start small by defining a single value. Put the line `u = 0.01` just after line 6 in your program. (Refer to the listing above for the line numbering.) You have now defined a mutation rate. We will assume that the rate of mutation from

A_1 to A_2 is the same as the rate from A_2 to A_1. How should these assumptions be incorporated into the simulation?

The mathematical details depend on the order of events during the life cycle. Let us suppose that the life cycle has the following steps:

1. Each individual produces many many gametes. Because there are so many gametes, the frequency of A_1 among them is essentially identical to that among parents. In other words, there is no genetic drift at this stage. A fraction u of the copies of A_1 become A_2, and a fraction u of the A_2 become A_1. As a result, the frequency of A_1 becomes

   $$p' = p(1-u) + (1-p)u.$$

   Among the gametes, a fraction p were allele A_1. Of these, a fraction $1-u$ failed to mutate and are therefore still A_1. Thus, $p(1-u)$ is the fraction of the gametes that carried A_1 and did not change, whereas $(1-p)u$ is the fraction that were A_2 before mutation but are A_1 after.

2. From these gametes, a smaller number $(2N)$ of gene copies is chosen to become adults in the next generation. This is where drift happens. In the new generation, the number of copies of A_1 among the adults is a binomial random variable with parameters $2N$ and p'.

Exercise

We are finally in a position to do something interesting. When the mutation rate is very low, heterozygosity behaves as though there were no mutation at all—it declines to 0. This is *not* true, however, for larger values of u. The question is, *under what circumstances is the effect of mutation large enough to matter?* In other words, what is the smallest value of u for which H is appreciably greater than 0? The answer will certainly depend on u, because u measures the force of mutation. It may also depend on N, if the answer turns out to depend on the force of mutation relative to that of drift. To answer these questions, the first step is to put mutation into your drift program:

1. Modify your drift program so that mutation occurs in each generation. This will involve adding a line of code in between lines 14 and 15 in the listing above. Run your modified program a few times. It should still run to fixation, but it should take more generations.

2. Modify your program so that it runs the simulation repeatedly, once for each of the several mutation rates. Use at least four rates, which span the range from 0.0001 to 0.1, using the method discussed above. Think carefully about how you want to divide this interval up. Do you want to space the rates evenly on an arithmetic scale? On a logarithmic scale?[2] Does it matter? You will need to add a `for` statement just after line 6. This new loop will enclose lines 7–23 of the original program, and those original lines will need to be indented. Remove the print statement (line 23). The new program should have a single print statement, which

[2]On an arithmetic scale, the difference between each pair of adjacent mutation rates is the same. For example: 1,2,3, and 4 are equally spaced on an arithmetic scale. On a logarithmic scale, the *ratios* are the same. For example: 1, 10, 100, and 1000.

should execute at the end of each pass through the outer loop. It should print the current values of the following quantities:

$2N$	twice population size
u	mutation rate per gene copy per generation
$4Nu$	twice the number of mutations per generation in the population as a whole
H	heterozygosity
g	count of generations

Format it so that the numbers line up in columns. In the output, pay close attention to the printed value of `gen`. It tells you how many generations elapsed before either fixation of one allele or the other. If neither allele was fixed, then `H` will be greater than 0, and `gen` will equal `maxgen`.

3. Use this program in an effort to answer the question posed above. Search for the smallest value of u (and $4Nu$) for which H is appreciably greater than 0. Let's call these the *critical values* of u and $4Nu$. To make your answers more precise, play around with the range of u values in your simulation. But do not get carried away. We are not expecting high precision. Just try for an answer that is correct within roughly a factor of 5.

4. Having found the critical values under a specific assumption about $2N$, do it again with a larger value of $2N$—one that is 10 or 100 times as large. As population size increases, do the two critical values go up, go down, or stay about the same? Is the pattern simpler in terms of u or $2N$ or $4Nu$?

5. Write a sentence or two summarizing the pattern in your results. Attempt to answer the question posed above.

What to submit Upload your code to Canvas as a plain-text file with extension `.py`. This is exactly the format produced by Idle; you shouldn't have to change anything. In future labs, you may need to write more than one program. If so, upload multiple files. In addition, upload a "lab report"—a document in `.pdf` or `.docx` format, which contains relevant output and a paragraph or two explaining what you did and what you learned.

What not to include in the lab report The lab report should not include computer code. Neither should it include extraneous output. While you're debugging a program, it's often useful to print the values of variables during intermediate steps, or that print a line of output on each pass through a loop. This output does not belong in your lab report. Remove these `print` statements before generating the output that you will submit.

Project 5

Simulating Gene Genealogies

This project introduces a procedure—a sort of recipe—for simulating genetic drift. In computing, recipes are called "algorithms." The one below is called the "coalescent algorithm." It is based on the theory about gene genealogies that you have been learning in lecture and in the text. We will give you a simple version of the algorithm and ask you to modify it.

You used a different method to simulate genetic drift in Project 4. There, you kept track of each copy of every allele in the entire population. In the new approach described here, we keep track only of the copies within our sample. This is one of several reasons why the coalescent approach is so powerful. This approach also differs from the traditional one in going backwards through time.

As we travel forward in time, genetic drift reduces variation within populations and increases that between them. Thus, we focus on these variables in forward-time simulations. In backwards time, drift implies that individuals share ancestors. The smaller the population, the more recent these ancestors are likely to be. Thus, backward-time simulations focus on shared ancestors. Yet in spite of this difference in focus, both kinds of simulation describe the same biology.

5.1 Exponential and Poisson random variates

In this project you will generate random variates from two probability distributions, the exponential and the Poisson. (To refresh your memory about these, see *Just Enough Probability.*) The `random` module of the Python Standard Library contains a function called `expovariate`, which generates exponential random variates. Python has no built-in generator for Poisson random variates, however, so we provide one called `poidev` in our module `pgen.py`. You will need to download `pgen.py` from the class web site. To make these functions available to Python, start your program with

```
from random import expovariate
from pgen import poidev
```

The `expovariate` function (described on page 51) returns a value drawn at random from the exponential distribution. The single parameter of this distribution is called the hazard or the rate. In this project, we will be interested the hazard of a coalescent event. In class we discussed this hazard in the context of the standard model for neutral genes and constant population size. For example, in a sample of two gene copies, the hazard of a coalescent event is $h = 1/2N$ per generation, where $2N$ is the number of gene copies in the population. We can use `expovariate` to simulate the time since the last common ancestor (LCA) of two random gene copies. For example,

```
>>> from random import expovariate
>>> twoN = 1000.0
>>> expovariate(1.0/twoN)
380.65658779723157
```

Here, we handed `expovariate` a hazard of $1/2N$, and it returned a value of 380.66 generations. Think of this as the number of generations since the LCA of a pair of gene copies drawn at random from a population in which $2N = 1000$. Do this a few times yourself in Python's interactive window to get a feel for how such intervals vary in length.

To sample random values from the Poisson distribution, we use the `poidev` function, which is described on page 52. We will use this distribution to model the number of mutations that occur within a given gene genealogy. In this context, the mean is uL, the product of the mutation rate u and the total branch length L. (The total branch length is the sum of the lengths of all branches within the genealogy.) The command `poidev(u*L)` generates a random number that represents the number of mutations within such a genealogy. Here are a couple of examples:

```
>>> from pgen import poidev
>>> u = 1/1000
>>> L = 4000
>>> poidev(u*L)
7
>>> poidev(u*L)
5
```

The two returned values—7 and 5—represent the numbers of mutations within two random genealogies within which $L = 4000$ and $u = 1/1000$. Do this a few times yourself.

Combining the exponential and Poisson distributions We usually do not know the lengths of branches in a gene genealogy. What we do know are the values of genetic statistics such as S, the number of segregating sites. In class we discussed the model of infinite sites, which implies that S equals the number of mutations that occurred along the genealogy. How can we simulate that?

Let us go back to the simplest case: the genealogy of a pair of neutral genes in a population consisting of $2N$ gene copies and with mutation rate u. We proceed in two steps. In the first step, we use `expovariate(1.0/twoN)` to get a random value of t, the depth of the genealogy. Since there are two lines of descent, the total branch length is $L = 2t$, and the expected number of mutations is $2ut$. Next, we call `poidev(2*u*t)` to get the number of mutations along a random genealogy. Here's an example:

```
>>> from pgen import poidev
>>> from random import expovariate
>>> u = 1/1000
>>> twoN = 4000.0
>>> poidev(2*u*expovariate(1.0/twoN))
15
```

The result—15—represents the number of mutations between a random pair of genes drawn from a population in which $2N = 4000$ and $u = 1/1000$. Do this yourself a few times yourself.

Usually, we work with much larger samples, and we cannot get t directly from `expovariate`. In this project, you will use the coalescent algorithm to find the total branch length (L) of a random gene genealogy. Then you will use `poidev` to add mutations, just as in the preceding paragraphs.

5.2 The Coalescent Algorithm

The coalescent algorithm begins in the current generation with a sample of K gene copies. As we trace the ancestry of this sample backwards in time, it will occasionally happen that two of the ancestral genes are copies of a single gene in the previous generation. This is called a coalescent event. With each coalescent event, the number of ancestral gene copies is reduced by one until eventually only a single gene copy remains. This is the LCA of all the genes in the sample.

The time interval between two adjacent coalescent events is called a *coalescent interval.* As you learned in class, the length of each interval is an exponential random variable. In the computer program, we can use this fact to step from interval to interval. We need not concern ourselves with individual generations.

As you learned in class, the hazard of a coalescent event is

$$h = x(x - 1)/4N$$

per generation, where $2N$ is the number of gene copies in the population, and x is the number (during some previous generation) of gene copies with descendants in the modern sample. Below, we use this fact together with `expovariate` to obtain simulated lengths of coalescent intervals.

The coalescent algorithm is very simple to code. Here's a program called `coal_depth.py`, which uses it:

```
from random import expovariate
K = 30       # sample size (number of gene copies)
twoN = 1000  # population size (number of gene copies)

tree_depth = 0.0 # age of last common ancestor in generations

# Each pass through loop deals with one coalescent interval.
while K > 1:
    h = K*(K-1)/(2.0*twoN) # hazard of a coalescent event
    t = expovariate(h)       # time until next coalescent event
    tree_depth += t
    print("%2d %7.2f %7.2f     REMOVE ME" % (K, t, tree_depth))
    K -= 1                   # must be the last line in the loop
```

```
15  print("Tree depth:", tree_depth, "generations")
```

Download this program from the lab page of the class web site, and run it a few times. At the end of each run, it reports the depth of a random gene tree.

In this code, `K` represents the number of gene copies. We set it initially to 30, the number of gene copies in the modern sample. This value is reduced by one with each coalescent interval. The heart of the algorithm is the `while` loop in lines 8–13. Each pass through that loop deals with one coalescent interval. Within the loop, the program calculates the coalescent hazard and hands it to `expovariate`, which provides the length (t) of the current coalescent interval. This version of the program does nothing but add these values together to obtain the time since the LCA of the sample. At the very end of the loop, we subtract 1 from `K`, because after a coalescent event has occurred, there is 1 fewer lineage. As you modify this code, make sure this statement remains at the very end of the `while` loop.

Exercise

1. The code above generates simulated values of the tree's depth—the number of generations from the present back to the LCA. Modify the code so that instead of depth, it simulates the total branch length (L). To see how to do this, consider an interval during which there were 18 distinct gene copies in the tree. If that interval lasted 45 generations, then it would contribute 45 to the depth of the tree, as shown in line 11 of the code above. However, this interval would contribute 45×18 to L. Modify the code accordingly.

2. Once your program is producing random values of L, add a line near the top that sets $u = 0.001$. Then, at the very end, use `poidev` as explained above to get the number of mutations in the genealogy. (This number is a Poisson-distributed random variable with mean uL.) At this stage, the program should produce a single line of output showing the number of mutations on a random tree, given the population size and mutation rate.

3. A few years ago, Jorde et al published data in which the number of segregating sites was $S = 82$ for a mitochondrial sample from 77 Asians. This is a good estimate of the number of mutations in the mitochondrial genealogy of these subjects. In these data, the mutation rate is thought to be about $u = 0.001$ per sequence. Let us entertain the hypothesis that the Asian female population was historically very small: say $2N = 5000$. Use these values to set `u`, `twoN`, and `K` within your program, and then run it 20 times, keeping track of the results.

4. Use these results together with the observed data ($S = 82$) to evaluate the idea that the simulation model describes reality. How many of the simulated values are smaller than 82? How many are larger? Does the observed value represent an unusual outcome, or is it pretty typical of the simulated values?

In the last step above, we are not asking for any detailed quantitative analysis. Just compare the observed to the simulated values and make a subjective judgement. We'll show you in the next project how to put such conclusions on a firmer basis. Your project report should include (1) your program, (2) the 20 simulated values of S, and (3) a few sentences summarizing the conclusion that you reached in the last step.

Project 6

Using Simulation to Test a Statistical Hypothesis

In Project 5 you used a coalescent simulation to evaluate a hypothesis about $2N$ and u, two parameters that play central roles within population genetics. Your evaluation was subjective because we did not ask for any formal statistical test. In this project, you'll do it right. But first, you need to learn some basic principles. We begin with two definitions.

6.1 A statistic and its sampling distribution

All data have noise. They are noisy because of measurement error, because of sampling error, and because of unpredictable factors involved in the phenomena we study. This is why we need statistical methods in the first place. Things that we calculate from data are called *statistics*. Familiar examples include the mean and variance of a sample. Less familiar examples include estimates of the site frequency spectrum and the mismatch distribution. When a gene genealogy is estimated from data, even that is a statistic. Because they are based on noisy data, statistics are noisy too. In this context, "noise" refers to random variation. Scientists deal with such randomness using probability theory, and you should be familiar with that before reading further. (See *Just Enough Probability.*)

Scientific hypotheses are often tested by comparing a statistic to its sampling distribution. To get this idea across, we re-visit a simple problem: that of tossing a coin. As you learned in *Just Enough Probability*, John Kerrich tossed a coin 10,000 times and observed 5070 heads. This number is a statistic. How could we use it to test a hypothesis?

Suppose we suspected that Kerrich's coin was slightly unfair, that the probability (p) of heads was 0.45 on each toss. To test this idea, we would need to know whether the value that Kerrich observed is unusual in similar experiments for which p really does equal 0.45. Suppose that we somehow managed to manufacture such a coin and then used it in 100,000 repetitions of Kerrich's experiment. Suppose in addition that in none of these was the number of heads as large as Kerrich's value, 5070. Then Kerrich's result would clearly rank as unusual, and this might lead us to doubt the original hypothesis that $p = 0.45$. This is the reasoning that usually underlies tests of statistical hypotheses. The difficulty is that usually (as here) these repetitions are impossible to carry out.

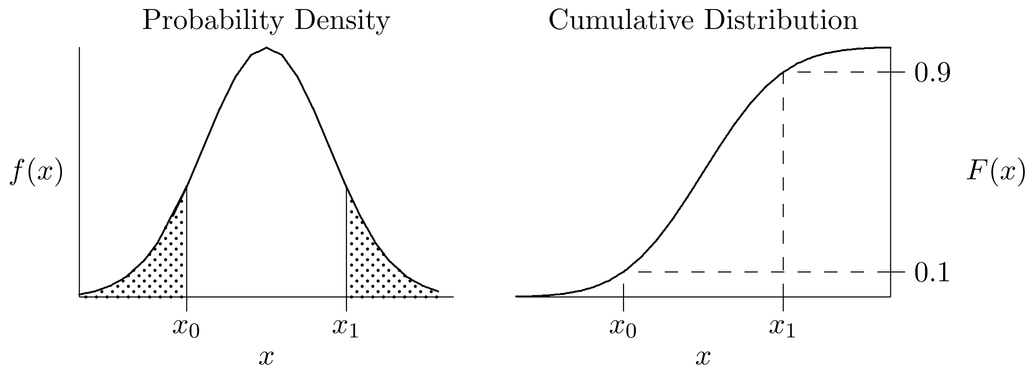

Figure 6.1: Two ways to graph a sampling distribution. Areas under the density function correspond to probabilities. For example, the left shaded region comprises a fraction 0.1 of the total area, because 0.1 is the probability that $X < x_0$. On the right, the vertical axis shows $F(x)$, the probability that $X < x$. For example, $F(x_0) = 0.1$ and $F(x_1) = 0.9$.

There is no way to manufacture a coin for which p is exactly 0.45. Even if that were possible, who would volunteer to toss that coin $100,000 \times 10,000 = 1$ billion times?

Nonetheless, let's pretend that we had really done 100,000 repetitions of Kerrich's experiment under conditions that guarantee that $p = 0.45$. We could calculate the number of heads from each repetition and then summarize these values using a *frequency distribution.* Since the number of repetitions was so large, this frequency distribution would approximate a special sort of probability distribution called a *sampling distribution.*

The sampling distribution of a statistic is its probability distribution, but only in a restricted sense: it is the probability distribution implied by a particular hypothesis. Different hypotheses imply different sampling distributions, even for the same statistic. In our example, we assumed that $p = 0.45$. A different hypothesis would imply a different sampling distribution.

In short, *a sampling distribution is the probability distribution of a statistic, as implied by a particular hypothesis.*

6.2 Using sampling distributions to test hypotheses

Let us postpone the question of how one gets a sampling distribution. For the moment, our focus is on using one to test a hypothesis. It is sometimes useful to distinguish between a random variable and the particular values it may take. In this section, we use upper case for the former and lower case for the latter. Thus, X represents a random variable and has a probability distribution, but x is just a number.

Figure 6.1 shows two ways to graph the sampling distribution of a hypothetical statistic, X. In the left panel, the vertical axis shows the *probability density function*, f. Areas under the curve correspond to probabilities in the sampling distribution of X. The shape of this particular density function may look familiar, but that has no relevance here. Sampling distributions can have any shape.

In the density function in the left panel, the two "tails" of the distribution are shaded. These tails comprise only a small fraction of the total area under the curve. For this reason, the observed value of our statistic is unlikely to fall within either tail. If it does, then either (a) we have observed an unusual chance event, or (b) our hypothesis and its implied sampling distribution are incorrect. When this happens, we say that the hypothesis has been rejected.

This does not mean it is wrong, for correct hypotheses may be rejected too. The probability of such an error is called α and is equal to the shaded area of the graph. To minimize these errors, we usually choose small values of α, such as 0.05 or 0.01. In figure 6.1, we have set α to a very large value (0.2) so that the shading is easy to see.

The right panel of Figure 6.1 shows another way to graph the same sampling distribution. The vertical axis shows $F(x)$, the probability of observing a value less than or equal to x. This is called the *cumulative distribution function.* For example, in the figure's left panel, a fraction 0.1 of the area under the curve lies to the left of x_0 and a fraction 0.9 lies to the left of x_1. Consequently, the right panel shows that $F(x_0) = 0.1$ and that $F(x_1) = 0.9$.

We explained above how f can be used to test a statistical hypothesis: if the observed value falls within the shaded tails, then we reject. This idea can be re-expressed in terms of F because F and f contain the same information. In terms of F, we reject if

$$F(x_{\text{obs}}) \leq \alpha/2, \qquad \text{or} \qquad F(x_{\text{obs}}) \geq 1 - \alpha/2, \tag{6.1}$$

where x_{obs} is the observed value of the statistic, and α is the level of significance.[1] This is just another way of saying that the observed value fell into one of the tails.

In testing a statistical hypothesis, one can work either with the density function (f) or with the cumulative distribution function (F), but the latter approach is often easier. We don't need to know the whole sampling distribution—just a single value, $F(x_{\text{obs}})$. As we shall see, this value is easy to estimate by computer simulation.

6.3 Sampling distributions from computer simulations

Long ago there was only one way to figure out the sampling distribution implied by a hypothesis—it required a sharp pencil and a mathematical turn of mind. That approach doesn't always work however, even for gifted mathematicians. Fortunately there is now another way: we can estimate sampling distributions from computer simulations.

In an informal way, you have done this already. In Project 5, you generated several simulated values of a genetic statistic (S) under a particular hypothesis about u and $2N$. You then compared these simulated values to the one ($S_{\text{obs}} = 82$) that Jorde obtained from real data. Near the end of that assignment, we asked: "How many of the simulated values are smaller than 82? How many are larger?" These are questions about $F(S_{\text{obs}})$. We then asked: "Does the observed value represent an

[1]Consider first the left shaded tail. This tail contains half the shaded area and thus has probability $\alpha/2$. This means that $F(x_0) = \alpha/2$. If x_{obs} falls within this tail, then it must be true that $F(x_{\text{obs}}) \leq \alpha/2$. We can reject any hypothesis for which this condition is true. Applying the same reasoning to the right shaded tail, we reject if $F(x_{\text{obs}}) \geq 1 - \alpha/2$.

Simulation	x	Simulation	x
1	4439	6	4508
2	4474	7	4521
3	4484	8	4527
4	4489	9	4558
5	4495	10	4566

Table 6.1: The number (x) of heads in ten draws from the binomial distribution with $N = 10,000$ and $p = 0.45$.

unusual outcome, or is it pretty typical of the simulated values?" In answering that question, you used the same logic that we explained above. The approach is more formal this time, but the idea is the same.

To see how this works in detail, let us return to our hypothesis about Kerrich's coin. Our statistic (x_{obs}) is the number of heads in 10,000 tosses, which equalled 5070 in Kerrich's data. We are interested in $F(x_{\text{obs}})$ under the hypothesis that $p = 0.45$. The conditions of Kerrich's experiment imply that the number of heads is drawn from a binomial distribution. This distribution has two parameters: the number (N) of trials and the probability (p) of "heads" on each trial. Our hypothesis assumes that $p = 0.45$, and we can take $N = 10,000$, because that is how many times Kerrich tossed the coin. We need to calculate $F(5070)$ from a binomial distribution with these parameter values.

We could do this calculation by summing across the binomial distribution function, but we'll do it here by computer simulation. The procedure is simple: over and over, we simulate Kerrich's experiment under the conditions of our hypothesis. Each simulation generates a value of x, the number of heads. Table 6.1 shows the result of 10 such simulations. In every one, the simulated value of x is smaller than $x_{\text{obs}} = 5070$, the value that Kerrich observed. Already we begin to suspect that our hypothesis is inconsistent with the data. Let us relate this to the probability, $F(x_{\text{obs}})$, discussed above.

As you know, probabilities are estimated by relative frequencies. We can estimate $F(x_{\text{obs}})$ as the relative frequency of observations such that $x \leq x_{\text{obs}}$. Every one of the 10 observations in Table 6.1 satisfies this condition. Thus, we estimate $F(x_{\text{obs}})$ as $10/10 = 1$. Our sample is small, so this may not be a very good estimate. But let us take it at face value for the moment. To test a hypothesis, we must first specify α, so let us set $\alpha = 0.05$. With this significance level, equation 6.1 says that we can reject if $F(x_{\text{obs}}) \leq 0.025$ or ≥ 0.975. Our estimate is $F(x_{\text{obs}}) = 1$, which is clearly greater than 0.975. Thus, we reject the hypothesis—or we would do so if we trusted our estimate of $F(x_{\text{obs}})$.

A convincing answer will require many more simulations, a potentially tedious project. Here is a program called `cointest.py`, which removes the tedium:

```
from pgen import bnldev

xobs = 5070  # Observed value of statistic.
nreps = 100  # Number of repetitions to do.

```

```
    p = 0.45      # As assumed by hypothesis.
    F = 0.0       # Will hold Pr[X <= xobs]

    for i in range(nreps):
10      x = bnldev(10000, p) # Number of heads in one experiment
        if x <= xobs:        # Count number of x's that are <= xobs
            F += 1

    F /= nreps               # Turn count into a fraction.
15
    print("F[%d] = %6.3f for hypothesis: p=%5.3f" % (xobs, F, p))
```

In this code, each execution of line 10 repeats Kerrich's experiment, using the function `bnldev`. (This function is described in Project 4 and documented more fully on p. 51 of appendix B.3.1.) Lines 11–12 count the number of such repetitions for which the outcome is less than or equal to the observed value 5070, and line 14 converts the resulting count into a fraction. Download this code from the class web site and run it. You should get something that looks like this:

```
F[5070] =  1.000 for hypothesis: p=0.45
```

The critical piece here is the value 1.000. This is the fraction of simulations for which the simulated value was less than or equal to Kerrich's value, 5070. It estimates $F(x_{\text{obs}})$. Even in this larger sample, every simulation resulted in fewer heads than Kerrich saw. If our hypothesis about p is correct, then Kerrich saw something remarkable. This is probably not what happened. It seems more likely that our hypothesis about p is way off the mark.

This estimate of $F(x_{\text{obs}})$ is a lot more reliable than the previous one, because it is based on 100 simulations rather than 10. It would be more accurate still if we had done 10,000. We did not start with 10,000, because it's wise to keep the number small until you get the code running. Try increasing `nreps` to 10,000. Does it makes a difference?

Exercise

1. Revise your coalescent code from Project 5 so that it tests the same hypothesis ($u = 0.001$ and $2N = 5000$) that we considered there, using the same data ($S_{\text{obs}} = 82$, $K = 77$). To do this, use `cointest.py` as your model. Simply replace line 10 with the simulation that you wrote in Project 5. Like `cointest.py`, your program should produce a single line of output. Show us the program and its output, and write a sentence or two saying whether and why the hypothesis can be rejected.

The tricky parts of this exercise are getting the indentation right and making sure that each variable is initialized where it needs to be.

Before proceeding, make sure you understand what this program does. In principle, it is exactly what you did in Project 5. The program does the same simulation many times and allows you to tell whether the observed value, $S_{\text{obs}} = 82$, is unusually large, unusually small, or typical, under a particular evolutionary hypothesis.

6.4 Confidence intervals

In the preceding section, we tested a single hypothesis about the value of p. It would be more interesting to examine a range of p values in order to see which can be rejected and which cannot. Here is a modified program called `coin_ci.py`, which does this:

```
 1  #!/usr/bin/python
    from pgen import bnldev

    xobs = 5070  # Observed value of statistic.
 5  nreps = 1000 # Number of repetitions to do.

    for p in [0.49, 0.495, 0.5, 0.505, 0.51, 0.515, 0.52, 0.525]:
        F = 0.0

10      for i in range(nreps):
            x = bnldev(10000, p)
            if x <= xobs:
                F += 1

15      F /= nreps

        print("F[%d] = %6.3f for hypothesis: p=%5.3f" % (xobs, F, p))
```

Notice how little this differs from `cointest.py`. Line 6 has been re-written, and lines 7–16 have been shifted to the right. We also bumped `nreps` up to 1000. Apart from these minor changes, the two programs are identical. These changes create an additional loop, which does a separate test for each of the values of p listed on line 6. Make sure you understand why the values of `xobs` and `nreps` are set outside this loop, and why that of `F` is set inside it.

Download this program and get it running. Your output should look similar to this:

```
F[5070] =  1.000 for hypothesis: p=0.490
F[5070] =  0.995 for hypothesis: p=0.495
F[5070] =  0.921 for hypothesis: p=0.500
F[5070] =  0.675 for hypothesis: p=0.505
F[5070] =  0.259 for hypothesis: p=0.510
F[5070] =  0.067 for hypothesis: p=0.515
F[5070] =  0.004 for hypothesis: p=0.520
F[5070] =  0.001 for hypothesis: p=0.525
```

Each line of this output tests a different hypothesis. We can reject the first two (at $\alpha = 0.05$) because $F(x_{\text{obs}}) \geq 0.975$. We can reject the last two because $F(x_{\text{obs}}) \leq 0.025$. The hypotheses we *cannot* reject include all values within the range $0.5 \leq p \leq 0.515$. These values—the ones we cannot reject—are called the *confidence interval* of p.

There is a little ambiguity here. We can reject $p = 0.495$ but not $p = 0.5$, so the lower boundary of the confidence interval must lie somewhere between these values. Exactly where, we cannot say. There is a similar ambiguity involving the upper boundary. It is safe however to say that the confidence interval is enclosed by the slightly larger interval $0.495 < p < 0.52$. The limits (0.495 and 0.52) of this larger interval are clearly *outside* the confidence interval, for we were able to reject them. This is why we use the strict inequality symbol ($<$). To be conservative, it is good practice to use this larger interval when reporting a confidence interval.

Exercise

2. Revise your coalescent code using `coin_ci.py` as a model, and estimate the confidence interval of $2N$. Show us your code, the output, and a brief explanation. Your explanation does not have to be long. Just explain what you have learned about the parameter $2N$. Which hypotheses about $2N$ can we reject, and which are still on the table?

At this point, we hope you can see (a) that testing a statistical hypothesis just amounts to asking whether your observation is unusual, and (b) that a confidence interval is just a summary of hypothesis tests. The methods of statistics are complex, but the underlying ideas are simple.

6.5 p-hacking

This lab has introduced the classical approach to testing a statistical hypothesis. Although such tests play an important role in science, they can also mislead. To see how, we encourage you to spend some time with this website:

`https://fivethirtyeight.com/features/science-isnt-broken`

The interactive tool there will enable you to prove either that Democrats are good for the economy or that Republicans are, depending on your own preconceptions.

That bit is discouraging, but keep reading. The second part of the article describes a study in which 29 teams of scientists each used identical data in an effort to answer the same question: do referees give more red cards to dark-skinned soccer players than to light-skinned ones? The teams of scientists used different methods and got different answers. Yet if you focus on the results that were statistically significant, you'll find that they tell a consistent story.

Project 7

Simulating Selection and Drift

In Project 4, you incorporated the effect of mutation into a model of genetic drift. Here, you will modify that code once again to incorporate the effect of selection and then use the resulting code in an experiment.

In his section 3.1, Gillespie lays out the algebra of natural selection. At the top of page 62, he presents a classical formula for the frequency (p') of allele A_1 among offspring, given the genotypic fitnesses and the allele frequency of the parents:

$$p' = \frac{p^2 w_{11} + p(1-p)w_{12}}{\bar{w}} \tag{7.1}$$

Here, w_{ij} is the fitness of genotype A_iA_j, and

$$\bar{w} = p^2 w_{11} + 2p(1-p)w_{12} + (1-p)^2 w_{22}$$

is mean fitness.

This formula gives the allele frequency expected among offspring in the absence of genetic drift. In a finite population, however, there will also be variation around this expected value. We can model this process using a modified version of the urn model of genetic drift. In the standard urn model, we are equally likely to choose any ball in the urn. If p is the fraction of balls that are black, then p is also the probability that the ball we choose will be black. Now, however, there is a bias: one allele has higher fitness than the other. In the urn metaphor, balls of one color (say black) are more likely to be drawn. A ball drawn from this biased urn is black with probability p', as given by equation 7.1. You will use this idea below in order to simulate the combined effects of selection and drift. As you will see, the procedure is almost identical to the one you used in Project 4, where you modeled the combined effects of mutation and drift.

Exercise

1. Begin either with your code from Project 4, or else with a fresh copy of file `drift.py` from the class web site. Next, modify this code so that it performs the simulation many times. You did this with the coalescent simulation in Project 6, so you already know how. You need something along the following lines:

```
1   twoN = 40              # population size
2   nreps = 100            # number of replicates
3   for rep in range(nreps):
4         p = 1.0/twoN
5         <then the "while loop" from the drift simulation>
```

Note that line 5 is not real code. You should replace it with code from your own program.

It will be useful to simplify the drift simulation a bit. In the previous lab, the inner loop began like this:

```
while H > 0 and g < maxgen:
```

This required that we define `H`, `maxgen`, and `g` ahead of time, and then keep the variables up to date. None of this is needed for the current lab, so get rid of the code defining and/or updating these variables, and replace the `while` statement above with

```
while 0.0 < p < 1.0:
```

This loop will continue until the allele we're tracking is either fixed or lost. It's simpler, because it doesn't require any additional variables.

With this set up, each run of the program will perform 100 replicate simulations. So far, the program does not simulate selection. It just does 100 runs of the drift simulation. Later on, you will want to make `nreps` much larger. Notice that $p = 1/2N$ at the beginning of each replicate. With this setup, we can interpret each replicate as a model of a newly arisen mutant allele.

Make sure you understand why some variables (`twoN` and `nreps`) are set before the `for` loop, and the others are set within the loop but before any other code. Beginners often get this sort of thing wrong, so make sure you understand what is happening here. What would happen if `twoN` were set inside the `for` loop?[1] What would happen if `nreps` were set inside the `for` loop?[2] What would happen if `gen` and `p` were set before the `for` loop rather than within it?[3]

Once you understand the code, take out all the old print statements. At the very end—after all replicates have run—print the fraction of replicates in which A_1 was fixed ($p = 1$) rather than lost ($p = 0$).

2. Now modify the program to implement the biased urn model that we discussed above. At the top of your program, define the following constants: $2N = 40$, $s = 0.1$, $h = 0.5$, $w_{11} = 1 + s$, $w_{12} = 1 + hs$, and $w_{22} = 1$. This says that A_1 is favored and that there is no dominance. hs is

[1]The code would still run correctly, but not quite as fast. This is because `twoN` is a constant whose value never needs to change.

[2]You must set `nreps` before you first use it, in the `range` statement. Otherwise the code will break. Once `range` executes, it doesn't matter whether you set `nreps` again, although doing so is a waste of time.

[3]On the second pass through the loop, things would break because `gen` and `p` would begin the loop with the values they had at the end of the first pass.

the fitness advantage of heterozygotes over A_2A_2 homozygotes. Note the difference between this fitness scheme and the one in the text.

Next, modify the value of p just before the call to `bnldev`, and then use this modified p in the call to `bnldev`. You did this in Project 4 in order to model mutation. Now you will do it again (this time using Eqn 7.1) in order to model selection. (Incidentally, you may be tempted to define a new variable called `p'`. This will not work, because the "`'`" character is illegal within Python names. Use some other name instead, or simply redefine the variable `p`.)

3. So far, you've got a program that runs the same simulation `nreps` times. Now you want to repeat that entire process for each of the following values of $2N$: 500, 1000, 2000 and 4000. This will involve enclosing the existing code in a `for` loop, as we illustrated in `coin_ci.py` (see p. 18 of Project 6).

4. Once you have the program working, it is time to collect some data. Set `nreps` to 10000, and run the simulation with each of the following values of s: 0, 0.001, and 0.02. You will discover that this takes awhile for the case in which $s = 0$. Use the results to make a graph with $\log_{10} 2N$ on the horizontal axis and "fraction fixed" on the vertical. Draw three lines: one connecting the points with $s = 0$, one for $s = 0.001$, and one for $s = 0.02$. Summarize the graph in words. How does the probability of fixation respond to population size? Write a paragraph interpreting your results using the theory discussed in Gillespie and in class.

Hint: To save yourself some work, you may want to ask Python to calculate the $\log_{10}$ values rather than using your calculator. To do this, put

```
from math import log10
```

at the top of your program. Then `log10(100)` will give $\log_{10} 100$ (which equals 2).

Project 8

Using HapMap

Several large databases of human genetic variation have become available during the past decade, and they have revolutionized the field. This project will introduce you to one of them, the HapMap. The HapMap project has genotyped millions of single-nucleotide polymorphisms (SNPs) throughout the human genome within samples from several human populations. You will learn how to download and manipulate these data, and will then use them to explore the relationship between observed and expected heterozygosity in HapMap SNPs.

8.1 Choosing a population and chromosome at random

The original HapMap provided data on the following populations:

Label	Sample
`YRI`	90 Yorubans from Nigeria
`CEU`	90 Utahns of western and northern European ancestry
`CHB`	45 Han Chinese from Beijing
`JPT`	44 Japanese from Tokyo
`JPT+CHB`	combined Chinese and Japanese samples

Although recent releases of HapMap have added additional populations, we will focus on these. The data from these populations are made available as plain text files, one for each chromosome within each population. You will download one of these files onto your computer, but first you must figure out which one.

For this project, you will choose a chromosome and population at random. To do so, cut and paste the following program into the Python interpreter, and run it once.

```
from random import choice, randrange

print("Your chromosome is number", randrange(1,23))
print("Your population is", choice(["YRI", "JPT+CHB", "CEU"]))
```

It will tell you the number of your chromosome, and the name of your population. This program uses two new functions—`choice` and `randrange`—from Python's `random` module. We'll be using these at several points in this project, so you should experiment with them until you understand how they work. They are described in section B.2, page 50.

8.2 Downloading data

Now that you know your chromosome and population, you are ready to visit the HapMap web site. This is a task that you'll need to do for several projects, so we've put the instructions into a self-contained appendix, which you'll find on page 48. Please download two data files: the one for your own chromosome and population, and also the one for chromosome 22, population `JPT`.

From here on, we'll assume that you have already downloaded the appropriate data file for this project. We recommend that you put all relevant files into the `Documents` folder. Before proceeding, make sure that the data file and `pgen.py` are both in the same folder (or directory), and that Python can execute the command `from pgen import *` without generating an error.

8.3 Using the `pgen` module to work with HapMap data

The `pgen` module provides several tools for use with HapMap data files. These are described in section B.3.2 (page 52). Here, we merely illustrate their use. At the top of your program, you will need the line:

```
from pgen import *
```

The portion of the program that uses HapMap data should begin like this:

```
pop = "JPT"
chromosome = 22
hds = hapmap_dataset(hapmap_fname(chromosome, pop))
```

Here, the call to `hapmap_fname` tries to find the name of an appropriate HapMap data set—one for chromosome 22 and population `JPT`. If it succeeds, it hands that name to `hapmap_dataset`, which reads the file and stores it in a useful form (more on this in a moment). In the last line, the assignment statement creates a variable called `hds`, which points to this data structure. We use the name `hds` to help us remember that it refers to an object of type "`hapmap_dataset`," but you can use any name you please.

But what exactly is an object of type "`hapmap_dataset`?" Well, it is a lot like a Python list. Like a list, it has a length:

```
>>> len(hds)
11235
```

And like a list, it also has a sequence of data values that we can access like this:

```
>>> snp = hds[3]
```

Now `snp` refers to the data value at position 3 within the data set. This data value is an object of another specially-created type, called "`hapmap_snp`."

An object of type `hapmap_snp` has a variety of types of data. Having defined `snp`, we can type

```
>>> snp.chromosome
'22'
>>> snp.position
14603201
>>> snp.id
'rs1041770'
```

These commands report (1) the chromosome on which this SNP lies, (2) the position of the SNP in base pairs, measured from the end of the chromosome, and (3) the "rs-number," a name that uniquely identifies this SNP. You might think that the rs-number was redundant, once we know the SNP's position. However, the position values change a little with each new build of the data base, whereas the rs-numbers are invariant from build to build.[1] Thus, the rs-number is essential if you want to find a SNP that is discussed in the literature.

`hapmap_snp` also contains all the genetic data for the SNP. For example,

```
>>> snp.alleles
['G', 'T']
```

returns a list containing the two alleles that are present at this locus. In the original data file, the genotype of each individual would have been represented as GG, GT, or TT. In `hapmap_snp`, these genotypes are recoded as numbers:

```
>>> snp.gtype
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

Each integer counts the number of copies of `alleles[0]` in one individual. In these data, allele 0 is G, so the integers 0, 1, and 2 correspond to genotypes TT, TG, and GG. Since the genotypes are now integers, they have a mean and a variance, which you can access like this:

```
>>> snp.mean
0.066666666666666666
>>> snp.variance
0.062222222222222222
```

Objects of type `hapmap_snp` behave like lists. The command `snp[3]` returns the genotypic value at position 3, `len(snp)` and `snp.sampleSize` both return the number of genotypes in the data, and `sum(snp)` returns the sum of the genotypic values.

In these examples, `snp` is the name of a variable that points to an object of type `hapmap_snp`. But we could have used any other name just as well. For example,

[1]Occasionally, rs-numbers do change. This happens when it is discovered that two numbers actually refer to the same SNP. Then the higher number is retired, and lower number becomes the official designation.

```
>>> another_snp = hds[23]
>>> another_snp.alleles
['A', 'C']
```

8.3.1 Some examples

This section shows how `pgen`'s HapMap facilities can be used to solve problems. Read them carefully, and try them out in Python's interactive interpreter. They will all be useful later on.

Working with `gtype` In a miniature sample of 8 genotypes, `gtype` might look like this:

```
>>> snp.gtype
[1, 0, 0, 1, 1, 2, 0, 1]
```

Each number counts the copies of allele 0 in an individual genotype. The sum of these values—in this case 6—is the number of copies of this allele in the data set. Since each genotype has two genes, the total number of genes is $2 \times 8 = 16$. The frequency of allele 0 is therefore $p = 6/16$. On the other hand, the average of the data values is $\bar{x} = 6/8$. The allele frequency is exactly half the mean and is thus amazingly easy to calculate:

```
>>> p = 0.5*snp.mean
```

Just remember that this is the frequency of allele 0—the one that is listed first in `snp.alleles`.

Counting genes and genotypes As explained above, the number of copies of allele 0 is just the sum of genotypic values:

```
>>> n0 = sum(snp)
```

What about the number of heterozygotes? The variable `snp.gtype` is a Python list. As such, it has a built-in method called `count` that will count the number of copies of any given value within the list. In our data, every heterozygote is represented by the integer "1." We can count the number of heterozygotes with a single command:

```
>>> snp.gtype.count(1)
```

Here, `count` is a built-in method that would work on any Python list. To calculate the frequency of heterozygotes:

```
>>> snp.gtype.count(1)/snp.sampleSize
```

Looping over SNPs There are several ways to loop over the SNPs in a `hapmap_dataset`. If you want to include *all* the SNPs, the code is very simple:

```
>>> for snp in hds:
...     print(snp.mean)
```

This works because objects of type `hapmap_snp` behave like lists. Today's project involves making a graph. If you tried to graph all the SNPs, the graph would be incomprehensible. Consequently, we will look only at a limited sample. The easy way to do this is as follows:

```
>>> for i in range(50):
...     snp = choice(hds)
...     print(snp.mean)
```

This loop includes 50 SNPs chosen at random from among all those in the data. The code makes use of the `choice` function, which we introduced on p. 23 and discuss more fully on p. 51.

Exercise

Create a `hapmap_dataset` for chromosome 22 of population `JPT`. From this dataset, select the SNP whose index is 83, i.e. `hds[83]`. Use this SNP to answer the following questions: (1) What is the nucleotide state (A, T, G, or C) of allele 0? (2) How many copies of this allele are present? (3) What is its relative frequency? (4) What is the expected frequency of heterozygotes at Hardy-Weinberg equilibrium? (5) What is the observed frequency of hetetozygotes? (6) What is the position (in base pairs) of this SNP on the chromosome?

Project 9

Heterozygosity of HapMap SNPs

Heterozygosity is perhaps the most fundamental measure of genetic variation. In this project, we'll study the heterozygosity of SNPs within the HapMap data base. For each SNP, we'll calculate not only the observed heterozygosity, but also the value expected at Hardy-Weinberg equilibrium. We'll be interested to see how well the two numbers agree.

The exercise will involve modifying the incomplete code below. You can download it from the lab website, where it is called `haphetinc.py`.

```
 1   from pgen import *
     from random import choice

     chromosome = 22
 5   pop = 'JPT'
     hds = hapmap_dataset(hapmap_fname(chromosome,pop))

     for i in range(20):
         snp = choice(hds)
10
         ohet = 0.0  # REPLACE ME
         ehet = 0.0  # REPLACE ME

         print("%5.3f %5.3f" % (ohet, ehet))
```

The guts of this little program is the loop in lines 8–14. Each time through the loop, we choose a SNP at random from the data set. Then we set the observed (`ohet`) and expected (`ehet`) heterozygosities, and print these values. Only, as you can see, the present code just sets these values to 0.

Exercise

1. Change lines 11-12 so that they calculate the observed and expected heterozygosities of each SNP.

So far, your program merely lists the expected and observed heterozygosity values. It is hard to get a good sense of these values, just by staring at the numbers. You'll appreciate the pattern better after making a scatter plot. Before you can do this, you must first get the data values into lists.

9.1 Storing values in a list

As you calculate values of `ohet` and `ehet`, you will need to store them in a list. Before you can do so, the list must first exist. Thus, you will need to create an empty list before the main loop in your program. For example, you might use a command like `obs_vals = 3*[None]` to create a list with room for three observed values. Here, the keyword "`None`" is Python's way of representing an unknown value. In subsequent commands, you replace these with real values. Here is how it works:

```
>>> obs_val = 2*[None]
>>> obs_val
[None, None]
>>> obs_val[0] = 3.0
>>> obs_val[1] = 111
>>> obs_val
[3.0, 111]
```

Below, you will create two lists (one for `ohet` and one for `ehet`) to store the values you calculate within the loop in the program above. Your lists should be defined *before* the loop and should be long enough to hold all the values you will calculate within the loop.

It is also possible to start with an empty list:

```
>>> obs_val = []
>>> obs_val
[]
```

Then you can add values using the `append` method, which works with any list:

```
>>> obs_val.append(3.0)
>>> obs_val.append(111)
>>> obs_val
[3.0, 111]
```

Either way, you end up with the same answer. The first method is faster with large jobs, but the second is easier. Feel free to use either approach.

9.2 Making a scatter plot

There are several Python packages for making high-quality graphics. If you are working on your own machine at home, you may want to download either `matplotlib` or `gnuplot.py`. Unfortunately, none of these packages is available in the Mac Lab at Marriott Library. As a partial solution to this problem, we have implemented a crude graphics function within `pgen.py`. It constructs a scatter plot out of ordinary text characters—a style of graphics that has not been common since the 1970s.

The function `charplot` is described on page 53. Here is a listing that illustrates its use:

```
from pgen import charplot

x = [0, 1, 2, 3,  4,  5,  6,  7,  8,  9]
y = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

charplot(x, y)
```

When this program runs, it makes a scatter plot of the (x_i, y_i) values, with x on the horizontal axis and y on the vertical. Try it yourself. The function accepts several optional arguments, which you can use if you wish to control the number of tick marks or the dimensions of the plot. See page 53 for details.

Exercise

2. Instead of printing the heterozygosity values, put them into two lists: one for the expected values and the other for the observed values. At the end of the loop, make a scatter plot, with expected heterozygosity on the horizontal axis and the observed value on the vertical axis.

3. Use your program to produce a graph. Experiment to find the best number of SNPs. With a ruler and pencil, draw a line on the graph representing the function $y(x) = x$. (To help you draw this line, you may want to append a dummy pair of data values such as $(0.7, 0.7)$ before calling `charplot`.) If observed values always equaled expected ones, all points would fall on this line. Write a short paragraph describing the pattern you see. Is there good agreement between observed and expected values, or do you see a discrepancy? Does variation about the expected value increase with the expected value, or does it decrease or stay the same? Why might this be?

What to hand in Your answers to step 1, the final version of your program, the graph, and a paragraph discussing the results.

Project 10

Simulating Selection and Drift at Two Loci

This project is motivated by the current widespread interest in using *linkage disequilibrium* (LD) to detect selective sweeps. The goal is to find out how linkage disequilibrium (as measured by r^2) behaves when one locus is undergoing a selective sweep and the other is neutral. You will build a simulation model involving selection, recombination, and drift at two loci.

In the end, you will need to tabulate values of a continuous variable, r^2. We've written a Python class called `Tabulator`, which simplifies this task. See sec. C.3, p. 58, for details.

We assume that selection acts at one locus, where allele A is favored over allele a. At a linked locus, alleles B and b are neutral. The simulation starts with $p_A = 1/2N$ and stops when $p_A \geq 1/2$. At that point, it prints out p_A, p_B, and r. By doing a number of replicate simulations, you will get a sense of how r behaves when p_A is near 1/2. On the class website, you will find a program called `twolocinc.py`, which does most of this job. Your task is to fill in the pieces that we have left out, and then to run the program a few times to collect data. But first, we need to tell you about some nuts and bolts.

10.1 Sampling from the multinomial distribution

Thus far, our simulations have involved a single locus with two alleles. To model drift in such a system, we used the metaphor of an urn containing balls of two colors. Now we need to keep track of four types of chromosome (or gamete): AB, Ab, aB, and ab. The urn metaphor still works, but now the urn contains balls of four types. In a slight modification of Gillespie's notation, we write the relative frequencies of the four gamete types as:

Gamete	AB	Ab	aB	ab
Rel. freq.	x_0	x_1	x_2	x_3

Our initial subscript is 0 (rather than 1) for consistency with subscripting in Python.

Suppose the urn contains four types of balls in these frequencies. You draw $2N$ balls at random from the urn, replacing each one and then shaking the urn before selecting the next ball. Among the balls that you selected, we represent the number of each type as follows:

Gamete	AB	Ab	aB	ab
Count	n_0	n_1	n_2	n_3

In other words, n_0 is the number of balls of type AB that were drawn from the urn, and so on. Each time you repeat this experiment, the values in the array $[n_0, n_1, n_2, n_3]$ will vary at random. The probability distribution that describes this variation is called the *multinomial*. In the special case when there are only two types of balls, there is no difference between the multinomial and binomial distributions. The multinomial distribution is more general because it applies regardless of the number of types of balls.

We are not going to tell you much about the multinomial. All your program needs to do is draw samples from it, and we have provided a function called `mnldev` that simplifies this task. This function is described on page 51. To use it, you need to specify the number of balls to draw, and the relative frequency of each type within the urn. Here's an illustration:

```
>>> from pgen import mnldev
>>> mnldev(5, [0.2, 0.3, 0.5])
[2, 0, 3]
>>> mnldev(5, [0.2, 0.3, 0.5])
[1, 1, 3]
```

In these calls to `mnldev`, the first argument is the integer 5. This says that we want to draw 5 balls. The second argument is a list containing 3 relative frequencies. For each call, `mnldev` returns a list of 3 integers that sums to 5. These returned values are samples from the multinomial distribution. They represent the numbers of balls drawn, with one number for each of the 3 types.

In your application, you will want to draw `twoN` balls from an urn whose relative frequencies are stored in a Python list called `x`. The returned counts should be stored in a list called `n`. Thus, the call to `mnldev` will look like `n = mnldev(twoN, x)`.

Ordinarily, a list of relative frequencies must be normalized so that it sums to 1.0. Making sure that this is true often involves an extra step in computer code. You can omit this step, because `mnldev` does the normalizing automatically. The call `mnldev(5, [2, 3, 5])` is exactly the same as `mnldev(5, [0.2, 0.3, 0.5])`. We'll explain below how to use this trick to simplify your code.

10.2 An incomplete program

On the course website you'll find a program called `twolocinc.py`, which looks like this:

```
# twolocinc.py
from pgen import mnldev, Tabulator
from math import sqrt

twoN = 500
s = 10/twoN                      # selective advantage of allele A
```

```
c = 0.001                     # recombination rate
w = [1.0+s, 1.0+s, 1.0, 1.0]  # fitnesses of AB, Ab, aB, & ab

# Tabulate values in [0,1] into 10 bins
tab = Tabulator(low=0, high=1, nBins=10)

print("2N=%d s=%f c=%f" % (twoN, s, c), end=' ')
print("Fitnesses:", w)
print("%6s %6s %6s" % ("pA", "pB", "rsq"))
trials = 0
ngot = 0
nwant = 50
while ngot < nwant:
    trials += 1
    x = [1.0/twoN, 0, 0.5-1.0/twoN, 0.5] # freqs of AB, Ab, aB, & ab
    pA = x[0]+x[1]
    pB = x[0]+x[2]
    while True:
        # Insert code here to adjust x for recombination
        # Insert code here to adjust x for gametic selection
        n = mnldev(twoN, x)    # sample from multinomial
        x = [z/twoN for z in n]
        pA = x[0]+x[1]
        pB = x[0]+x[2]
        if pA==0 or pA>=0.5 or pB==0 or pB==1:
            break

    if pA >= 0.5 and (0 < pB < 1):
        # Insert code here to calculate rsq.
        rsq = 0.0
        print("%6.3f %6.3f %6.3f" % (pA, pB, rsq))
        tab += rsq
        ngot += 1

print(f"Trials: {trials}")
# Print the tabulation in a readable format.
print(tab)
```

This program will run, but it is incomplete. It does neither recombination nor selection and does not calculate the value of r^2. These are the parts you need to add. But first, get it running.

In its overall structure, this program is a lot like the one you wrote in Project 7. The top section defines parameters and prints them. Two of these deserve comment. We will simulate the history of a large number of newly-arisen mutations. Most of these will be quickly lost, and these we will ignore. We are interested in mutations that manage to increase to a frequency of 1/2. The parameter `nwant` specifies the number of these successful mutations that we want to study. The variable `ngot` holds the number of successful mutations we have encountered so far. The outer loop continues until `ngot == nwant`.

The main part of the program consists of two nested loops. Each pass through the outer loop simulates the history of one mutation, which is initially present in only a single copy. The inner loop models the history of that mutant, and each pass through that loop corresponds to a single generation. The inner loop stops when either of the loci becomes monomorphic, or when the A allele reaches a frequency of 1/2. This entire process is repeated many times—once for each mutant allele—under the control of the outer loop. This new program differs from that of Project 7 in that there are four gamete types rather than just two alleles.

This becomes apparent on line 21, which initializes a list containing the relative frequencies of the four gamete types. In the one-locus model, you would have written `p = 1.0/twoN` at this point. The new code needs to make some assumption about the state of the population just before the initial mutation. The one made by line 21 is completely arbitrary: just before the mutation, allele A didn't exist, and allele B had frequency 0.5, so the four gamete frequencies were $x = [0, 0, 1/2, 1/2]$. Then one of the aB gametes mutated to become an AB. The resulting list of gamete frequencies is $x = [1/2N, 0, 1/2 - 1/2N, 1/2]$, as indicated on line 21.

After setting the value of `x` on line 21, we set `pA` and `pB` on lines 22–23. These depend on `x` and must therefore be reset each time `x` changes, as seen on lines 28–29.

Each pass through the inner loop (lines 24–32) corresponds to a single generation. Based on current gamete frequencies, we sample from the multinomial distribution. This yields a list of counts, which represent the number of copies of each gamete type in the next generation. After converting these back into relative frequencies, we check to see if it is time to stop. That is all there is to it. If this seems familiar to you, that is probably because the program you wrote in Project 7 worked the same way.

Note that lines 34–39 execute only after we drop out of the inner loop. The `if` statement there makes sure that nothing prints unless both loci are polymorphic. We are not interested in monomorphic cases. Line 38 adds the current value of r^2 to the Tabulator, and line 43 prints the tabulated values. (Tabulator is described in section C.3, p. 58.)

Make sure this program runs before you begin changing it. With the current parameter values, you'll find that only a few of the replicates generate output. The program prints fitnesses, a recombination rate, and r^2, but these are misleading. As it stands, there is no recombination and no selection, and r^2 is simply set to zero. Your job is to rectify that.

Exercise

1. **Recombination.** We're going to model recombination using a trick that you have already used twice before. In Project 4, you incorporated mutation into a simulation of drift by adjusting allele frequencies before calling `bnldev`. Then in Project 7 you used the same trick in order to model selection. Now you will use that trick once again to model recombination. There is only one real difference. In those earlier projects, there were only two alleles so you only had to adjust the value of one allele frequency. Now you must adjust the values of all

four gamete frequencies and then call `mnldev` instead of `bnldev`.

In his Eqn. 4.1 (p. 102), Gillespie shows how recombination changes the frequency of one gamete type. There are analogous formulas for all four types:

$$\begin{aligned} x_0' &= x_0(1-c) + cp_Ap_B \\ x_1' &= x_1(1-c) + cp_A(1-p_B) \\ x_2' &= x_2(1-c) + c(1-p_A)p_B \\ x_3' &= x_3(1-c) + c(1-p_A)(1-p_B) \end{aligned}$$

where c is the recombination rate, p_A is the frequency of A, and p_B is the frequency of B. By the time the program reaches line 20 in the code above, it knows the value of c and all the values in the list x. From these, it is easy to calculate $p_A = x_0 + x_1$ and $p_B = x_0 + x_2$. This is all you need to evaluate the equations above, which yield the frequencies of gamete types *after* recombination. Add code that does this just after line 23 above. However you decide to do this, make sure your results end up in the list named `x`.

2. **Selection.** Before calling `mnldev`, we need to adjust the gamete frequencies once again—this time for the effect of selection. The idea is the same as before. We have a theoretical formula that converts pre-selection frequencies into post-selection frequencies. As input to this process, we use the values in the Python list `x`, which have already been adjusted for recombination.

   For simplicity, assume that selection acts at the gamete stage rather than the diploid stage. (This makes our formulas a little simpler than Gillespie's.) After selection, the new gamete frequencies are
$$x_i' = x_i w_i / \bar{w}$$
   where $\bar{w} = \sum x_i w_i$ is the mean fitness. You have all the x_i and w_i values, so it is easy to calculate $\bar{w}$. Given $\bar{w}$, it is easy to calculate x_i' for all four gamete types.

   But before you start, take another look at the formula. In it, $\bar{w}$ plays the role of a normalizing constant. It is there so that the new gamete frequencies will sum to 1. As discussed above, `mnldev` will do this normalizing for us, so we don't have to. It is sufficient (and a little faster) to set $x_i' = x_i w_i$, without bothering to calculate $\bar{w}$. With this simplification, the list that we hand to `mnldev` contains values that are proportional to gamete frequencies, and that is good enough. Add code that does this just after line 24. As before, make sure that your results end up in the list named `x`.

3. **Linkage disequilibrium.** You have one more change to make. One conventional measure of linkage disequilibrium is the correlation between loci,
$$r = \frac{x_0x_3 - x_1x_2}{\sqrt{p_A(1-p_A)p_B(1-p_B)}}$$
   Note that this r is *not* the recombination rate. That is why we used a different symbol (c) for that. The numerator here equals D, a measure of LD that was discussed in lecture and in the text. Modify line 34 so that it calculates r^2.

4. **Collect data.** Set $2N = 5000$ and $c = 0.001$. Run the program once with $s = 0/2N$, once with $s = 10/2N$, and once with $s = 100/2N$. (These correspond to $2Ns = 0$, $2Ns = 10$, and $2Ns = 100$.) The program will take awhile to run when $s = 0$, because only a few of the simulated mutations will reach a frequency of $1/2$. For each value of s, Tabulator will print the frequency distribution of the values of r^2. How easy would it be to distinguish strong from weak selection, based on the value of r^2?

Project 11

Linkage Disequilibrium in the Human Genome

In Project 10 we modeled selection and drift at two linked loci. The focus there was on *linkage disequilibrium* (LD), which was measured using the statistic r, a form of correlation coefficient. It is easy to estimate r, provided that we can tell which nucleotides occur together on individual chromosomes. Unfortunately, we are often ignorant about this. We are ignorant, in other words, about "gametic phase." This makes it hard to measure LD with any data set that—like the HapMap—consists of unphased genotypes.

There are several solutions. The simplest is to estimate LD using a second correlation coefficient (discussed below), which is easy to estimate from diploid genotypes at two loci. This is useful, because it turns out that this second correlation is a good estimate of the first[1]. To distinguish between the two correlation coefficients, let us refer to the first as r_H (since it correlates Haploid gametes), and to the second as r_D (since it correlates Diploid genotypes). In this project, we will estimate r_D, and interpret its value as an estimate of r_H.

In this project we use these estimates to study how LD varies across large regions of chromosome. Each of you will study a different chromosome. You'll compare the pattern of LD in three populations. This project will require working with correlations, so that is where we begin.

11.1 Correlating diploid genotypes

In raw HapMap data, each genotype is a character string. The `pgen` module, however, recodes these as integers, as explained on page 25. For example, genotypes TT, CT, and CC might be recoded as 0, 1, and 2. The table below shows genotypes at two neighboring SNP loci on chromosome 22 in the Utah HapMap population (CEU). It was made by the program sketched below.

[1]Rogers, A.R. & C. Huff. 2009. *Genetics* 182(3):839–844.

```
locus A [rows] (C/T) at 22181701
locus B [cols] (A/G) + 793
      0  1 2
 0 [ 90 41 2]
 1 [  0 21 5]
 2 [  0  0 3]
rAB = 0.612, r^2 = 0.375, n = 162
```

Locus A is at position 22181701, and locus B is 793 base pairs to the right of it. Locus A has two nucleotides, C and T, whereas locus B has two others, A and G. The 3×3 table shows the distribution of genotypes at the two loci, with labels 0, 1 and 2 to indicate the genotypic values at the two loci. If you study the table, you'll see that the two loci are not independent. Individuals in row 0 tend to fall in column 0; those in row 1 tend to fall in column 1; and all those in row 2 fall in column 2. There is thus a positive relationship between the genotypic values of the two loci. This reflects a similar association in the underlying genotypes: individuals with genotype TT at locus A tend to have genotype GG at locus B, and so on. The last line of the report above summarizes the strength of this association. The diploid correlation coefficient is $r_D = 0.612$, and its square is $r_D^2 = 0.375$. Now this is *not* the same as the statistic r_H that we used in Project 10 to measure LD. Yet as discussed above, it is a good estimate of r_H. This justifies our use of it in what follows as a measure of LD.

The correlation between two variables is based on their covariance, and the concept of covariance is related to that of variance. So let us begin there. As explained in section 2.2 of JEPr, the variance V_X of X is its average squared deviation from the mean:

$$V_X = E[(X - \bar{X})^2].$$

Similarly, the covariance (JEPr section 2.3) is the average product of the deviations of X and Y from their respective means:

$$\text{Cov}(X, Y) = E[(X - \bar{X})(Y - \bar{Y})].$$

$\text{Cov}(X, Y)$ is positive when large values of X often pair with large Y, negative when large X often pairs with small Y, and zero when neither variable predicts the other. The maximum possible (absolute) value of $\text{Cov}(X, Y)$ is $\sqrt{V_X V_Y}$, so the natural way to define a dimensionless correlation coefficient is as

$$r_D = \text{Cov}(X, Y)/\sqrt{V_X V_Y},$$

which can range from -1 to $+1$.

11.2 Calculating variances and covariances

The convenient ways to compute variances and covariances are also closely related. You may recall from JEPr that $V_X = E(X^2) - E(X)^2$ and that $\text{Cov}(X, Y) = E(XY) - E(X)E(Y)$. Thus, we can calculate the variance from the averages of X and of X^2. The covariance is only slightly more complicated: we need the averages of X, of Y, and of the "cross-product" XY. In JEPr, these averages were across probability distributions, because we were discussing the variances and covariances of random variables. We are now concerned with data, so we average across the values in our data. Apart from that, the procedure is the same.

You will not need to calculate variances in today's main assignment, for they are calculated automatically when you create an object of type `hapmap_dataset`. You will however need to calculate covariances. The first step in today's assignment is to figure out how. It will help to think first about variances, because that calculation is so similar. Consider the listing below.

```
# Return the variance of the values in xvec
def var(xvec):
    m = msq = 0.0
    for x in xvec:
        m += x
        msq += x*x
    n = float(len(xvec))
    m /= n              # mean
    msq /= n            # mean square
    return msq - m*m    # variance

x = [5, 25, 36, 37, 41, 50, 60, 73, 75, 99]
y = [15, 16, 21, 44, 49, 62, 71, 73, 78, 94]

print("Var(x):", var(x))
print("Var(y):", var(y))
```

This code is available on the class web site, where it is called `var.py`. Download it, run it, and see what it does. You should get two lines of output, which report the variances of the two data sets.

How would this code need to change if we wanted to calculate a covariance rather than a variance? For one thing, we would want to step through the two data lists simultaneously in order to accumulate the sums of X, Y, and XY. There are several ways to do this, the simplest of which is involves a facility called `zip`, which you have not yet seen. Let us pause for a moment to discuss this new facility.

`zip(xv, yv)` is a Python facility that steps through the elements of several sequences (lists, tuples, strings, or whatever). For example,

```
>>> xv = ['x1', 'x2']
>>> yv = ['y1', 'y2']
>>> for x, y in zip(xv, yv):
...     print(x, y)
...
x1 y1
x2 y2
```

Each time through the loop, `x` and `y` are automatically set equal to corresponding elements from the lists `xv` and `yv`. Here we have "zipped" a pair of lists, but one can also zip three or more.

To use this facility in calculating a covariance, you would begin with a framework like this:

```
def cov(xvec, yvec):
    mx = my = mxy = 0.0
    for x, y in zip(xvec, yvec):
```

Exercise

1. Write a new function called `get_cov`, which calculates the covariance between two lists, and use it to print the covariance as the last line in the program.

11.3 Smoothing data

In this project, you will be comparing LD between tens of thousands of pairs of loci. Without some way of simplifying the output, you would drown in data. In the end, you will plot the results rather than just staring at numbers. This will help, but it is not enough—the noise in the LD estimates would still obscure the pattern. We can get rid of much of this noise by *smoothing* the data. This is the purpose of the `scatsmooth` function, which is available within `pgen.py` and is described on page 53.

There are many ways to smooth data, and `scatsmooth` implements perhaps the simplest. It divides the X axis into bins of equal width, and calculates the mean of Y within each bin. For example, suppose that we have data in two Python lists called `x` and `y`. To smooth them, we could type

```
>>> bin_x, bin_y, bin_n = scatsmooth(x, y, 5, 0, 40)
```

As you can see, `scatsmooth` takes five arguments: (1) `x`, a list of horizontal coordinate values; (2) `y`, a list of vertical coordinate values; (3) the number of bins (5 in this case); (4) the low end of the first bin; and (5) the high end of the last bin. In this example, we have asked `scatsmooth` to divide the interval from 0 to 40 into 5 bins. If you leave off the last two arguments, `scatsmooth` will calculate them from the data. `scatsmooth` returns three values, each of which is a list. The first (`bin_x` in this example) contains the midpoints of the X-axis values of the bins. The second (`bin_y`) contains the mean Y-axis values. The third returned list (`bin_n`) contains the numbers of observations within the bins. In your own code, you can name the returned values whatever you like; you don't need to call them `bin_x`, `bin_y`, and `bin_n`.

We can now manipulate the three returned lists any way we please. For example, here is a listing that prints their values.

```
>>> for xx, yy, nn in zip(bin_x, bin_y, bin_n):
...     print("%8.3f %8.3f %4d" % (xx, yy, nn))
...
   4.000    4.000    5
```

```
    12.000   19.000   10
    20.000   39.000   10
    28.000   59.000   10
    36.000   74.000    5
```

Experiment with `scatsmooth` until you understand how it works.

11.4 An incomplete program

On the website, you will find a program called `rscaninc.py`, which is an incomplete version of the program you will need for this project. Here is the listing:

```
 1   from pgen import *
     from random import randrange

     nreps = 500
 5   chromosome = 22
     pop = 'JPT'
     window = 200

     # DEFINE GET_COV AND GET_RSQ FUNCTIONS HERE
10
     hds = hapmap_dataset(hapmap_fname(chromosome,pop))

     distvec = []
     rsqvec = []
15   for rep in range(nreps):
         i = randrange(len(hds) - window) # index of random snp
         # scan right
         for j in range(i+1, len(hds)):
             kilobases = abs(hds[j].position - hds[i].position)*0.001
20           if kilobases > window:
                 break
             distvec.append(kilobases)
             rsqvec.append( get_rsq(hds[i], hds[j]) )

25   print("Chromosome %d pop %s; %d focal SNPs, %d values of rsq" % \
         (chromosome, pop, nreps, len(rsqvec)))

     # YOUR CODE GOES HERE
```

You will need to insert code of your own just after lines 9 and 28. You do not need to modify anything else. Nonetheless, let us step through the existing code to see what it does.

Lines 1–7 import modules and define variables for later use. The main loop (lines 15–23) looks at a large number (`nreps`) of randomly-selected SNPs, which we will call "focal SNPs." The inner loop compares this focal SNP with each neighboring SNP within an adjoining region whose size (in kb) is given by the variable `window`. You need not worry about how lines 19–21 work. All you need to know is this: by the time we reach line 22, `i` is the index of a random SNP, and `j` is the index of a neighboring SNP. The goal is to find out (a) the distance between these two SNP loci in kb, and (b) their LD value, r_D^2. The first of these data values is calculated for you: it is in the variable `kilobases`. The second is obtained from a call to `get_rsq`, which you can see on line 23. At the bottom of the loop (lines 22–23), both data values (`kilobases` and r_D^2) are appended to their respective lists. In line 25 we have dropped out of the main loop, and both data lists are complete. The program prints a line of output and stops.

Exercise

In this exercise, the goal is to examine the relationship between LD and the distance that separates loci on a chromosome. You will study the LD-distance relationship in three different human populations.

2. We assume that you already know which chromosome you are working with. If not, refer to section 8.1 of Project **??**.

3. This week you will need HapMap data files for three populations, `CEU`, `YRI`, and `JPT`. Please download them as explained in appendix A. For debugging, you may want to use the dummy data set described on page 48.

4. Download `rscaninc.py` from the class web site and save it as `rscan.py`. Modify it so that it specifies the right chromosome.

5. Paste your `get_cov` function definition into the program just after line 9. Immediately below, define a function called `get_rsq`, which returns r_D^2.

6. At the end of the program, add code that uses `scatsmooth` to smooth the data over the interval from 0 to `window`, using 20 bins. Treat `distvec` as your X-axis variable and `rsqvec` as your Y-axis variable. Then print the smoothed data in a table. The table should contain a row for each bin, one column for `dist`, one for r_D^2, and one for n (the numbers of observations within bins).

Make sure your program runs before proceeding.

7. Set `nreps` to 500 and `window` to 200. Run the program once with `pop` set equal to each of the following values: `CEU`, `YRI`, and `JPT`. These values refer to the European, African, and Japanese populations. If you have been using the dummy data set, you will also need to reset `chromosome`.

8. Use the data to make a graph with `dist` on the horizontal axis and r_D^2 on the vertical. You should end up with three curves—one for each population—on a single graph. You may do

this by hand, with *Excel*, or however you please. Label it so that we can tell which curve refers to which population.

9. Write a paragraph or two describing the pattern in the data and (if possible) suggesting an explanation. Pay particular attention to the differences (if any) between populations.

What to turn in (1) your code, (2) the output from the three runs, (3) the graph, and (4) your prose describing and interpreting the results.

Project 12

Linkage Disequilibrium Near the Human Lactase Gene

Most mammals are able to digest milk only during infancy. Later in life, their bodies stop producing the enzyme *lactase* and are thus unable to digest milk sugar (*lactose*). Some humans however continue to make lactase throughout life, a condition called *lactase persistance.* (It's opposite is *lactose intolerance.*) This condition is common in European (and some African) populations and seems to result from a single point-mutation in the promoter region of the lactase gene. Todd Bersaglieri and his colleagues (*Am. J. Hum. Genet.*, 74:1111–1120, 2004) argue that the European mutant arose a few thousand years ago and has been strongly favored by natural selection. Their evidence for this comes from the amount of linkage disequilibrium (LD) in this genomic region. In this project, you'll see for yourself. You will estimate LD in the region around the putative advantageous mutation and then compare this estimate to the LD on chromosome 2 as a whole.

12.1 An incomplete program

As usual, we provide code that does much of the work. You will find this code on the class web site in a file called `lactaseinc.py`. Here is what it looks like:

```
from pgen import *
from random import randrange

reach = 5000
chromosome = 99
pop = "CEU"
focal_position = 136325116  # position of SNP rs4988235

# REPLACE THIS WITH YOUR OWN get_rsq FUNCTION
def get_rsq(snp_x, snp_y):
    return 0.0

```

```
# Find average rsq between one SNP (hds[mid]) and all other SNPs that
# are nearby on the chromosome.  Return None if no such SNPs are
# found.  Average will include all SNPs between x-reach and x+reach,
# where x is the position of the SNP at hds[mid]
def window_rsq(hds, mid, reach):
    midsnp = hds[mid]
    lo = hds.find_position(midsnp.position - reach)
    hi = hds.find_position(midsnp.position + reach)
    if lo == hi:             # make sure hi > lo
        return None

    rsqsum = n = 0
    for i in range(lo, hi+1):
        if i == mid:
            continue
        rsq = get_rsq(hds[i], midsnp)
        if rsq != None:
            n += 1
            rsqsum += rsq

    return rsqsum / float(n)

hds = hapmap_dataset(hapmap_fname(chromosome, pop))
focndx = hds.find_position(focal_position)
print("looking for SNP at pos %d; nearest is at %d" \
      % (focal_position, hds[focndx].position))
if focndx == len(hds)-1:
    print("Error: couldn't find focal_position", focal_position)
    exit(1)
rsq_mean_obs = window_rsq(hds, focndx, reach)

print("Mean rsq w/i %d kb of %s: %f" % (round(reach/1000.0),
                                         hds[focndx].id, rsq_mean_obs))

# From here on, the program should estimate the probability that a
# random region in this chromosome would have as much LD as the region
# around focal_position.

# To get you started, here is how to set up a loop that will examine "nreps"
# different regions, randomly distributed across the chromosome.
tail = 0     # Count replicates >= rsq_mean_obs
nreps = 100
for rep in range(nreps):
        ndx = randrange(len(hds)) # index of a random SNP
57
        # Your code goes here. It should calculate rsq_mean_sim for ndx,
        # and add 1 to tail if this value is >= rsq_mean_obs
60
    # At the end, divide tail by nreps and print the answer.
```

Lines 4–7 define several parameters. The parameter `reach` determines determines the size of the window within which you will estimate LD. In this initial version of the program, the window reaches from 5000 bases to the left of the focal SNP to 5000 bases to the right. (That is why it is called "`reach`.") The chromosome number is set to 99, so that we can use the dummy data file during debugging. (See page 48.) In the end, you'll want to set `chromosome = 2` in order to analyze the real data. Line 7 identifies the SNP that is thought to be under selection. This SNP was identified in the Bersaglieri paper mentioned above. It's label is "rs4988235," and in our data it lies at position specified on line 7 of the code above. (This number may change in future versions of HapMap, but the label should remain the same.)

In the HapMap data for this target SNP, one of the genotypes has a missing value. Our pgen software eliminates SNPs with missing values, so this site will not appear in our data. For this reason, the call to `hds.find_position(focal_position)` will not find the target SNP. Instead, it will find the nearest SNP. That's OK, because there is so much LD in this region that SNPs in LD with the target SNP will also be in LD with this nearest SNP.

Lines 9–11 define the stub of a function that is supposed to calculate r^2—except that it doesn't in this incomplete code. Replace this function definition with your own version from Project 11. The function `window_rsq` is new. It takes three arguments, `hds` (the data set), `mid` (an index into that data set), and `reach` (discussed above). The function compares a central SNP (`hds[mid]`) to each SNP in the region around it. If no other SNPs are found in this region, the function returns `None`. Otherwise it returns the average value of r^2 between the SNPs in this region and the central SNP. The calls to `find_position` will seem mysterious, since you have not yet seen this function used. If you are curious, see page 52.

In line 35, we are done defining things and ready to do real work. In rapid succession, we define the dataset (`hds`), find the index (`focndx`) of the "focal SNP" (the one that is supposedly under selection), and calculate the mean value (`rsq_mean_obs`) of r^2 within the region around this focal SNP.

Exercise

1. Download the relevant files for the current project. You will need: (a) `pgen.py`, (b) the incomplete Python program `lactaseinc.py`, (c) the HapMap data file for chromosome 2 in the European population (`CEU`), and (d) the dummy data file for "chromosome 99." For instructions on downloading HapMap data files, see appendix A. For instructions on getting the dummy data file, see page 48.

2. Replace lines 10–11 with your own working copy of the `get_rsq` function. You'll need the `cov` function too. You won't need the `var` function, because variances have already been calculated

for you (see pages 25 and 53).

At this point, your program should calculate the mean value of r^2 in the region surrounding the putatitive selected SNP.

3. Add code to the end of the program that calculates
   (a) the mean r^2 value within 100 randomly selected regions on chromosome 2,
   (b) the fraction of these regions whose mean r^2 is greater than or equal to the "observed value," i.e. the value for the region surrounding the putative selected SNP.

Hint: To calculate r^2 in randomly selected regions, choose random SNPs as we did in Project 11, and then hand each one to function `window_rsq`. To calculate the fraction of these values that is greater than or equal to the observed value, proceed as we did in Project 6.

4. Now it is time to analyze real data, so set `chromosome` equal to 2. Then run the program with `reach` set to each of the following values: 20,000, 200,000, and 1,000,000. These runs will take longer, so be patient.

5. Write a paragraph discussing the results. Is the LD around the lactase gene unusual on chromosome 2? If so, how far does the region of unusual LD seem to reach? The answer to this question should be based on the tail probabilities you calculated for the three values of `reach`.

Appendix A

Downloading HapMap data

A.1 Interacting with the HapMap ftp server

The HapMap data are available via an ftp site at NCBI. There is a link to that site under the "lab" page of the course website. Or to go directly there, point your web browser at

```
https://ftp.ncbi.nlm.nih.gov/hapmap/genotypes/2008-10_phaseII/fwd_strand/non-redundant
```

This will bring you to a list of files with names like `genotypes_chr22_JPT_r24_nr.b36_fwd.txt.gz` In these file names, the string "`chr22_JPT`" means "chromosome 22" in population "`JPT`" (Japan). Scroll down the list to the file you need. Then right-click and select "`Save link as`" in order to download the file onto your own computer.

The file will end with "`.gz`," which indicates that it has been compressed using the `gnuzip` program. You don't need to decompress it, because `pgen.py` can read files that are compressed in this way.

A.2 The dummy data file

As you are debugging your programs, you must execute it after each edit. This can be a slow process unless the program executes very fast. Unfortunately, even the smaller HapMap data files are pretty large, and parsing them repeatedly can slow down the process of programming.

To solve this problem, we have created a dummy HapMap data file called `genotypes_chr99_CEU_r23a_nr.b36_fwd.txt`. In contains only a small number of SNPs, and your program can parse it almost instantly. We encourage you to use this file rather than any real data until you get your code running. To do so, place the file in the same folder with your other programs and specify

```
chromosome = 99
pop = 'CEU'
```

The dummy data file is under `Data` on the lab page of the class web site.

A.3 Put the data files into the folder where you plan to use them

Once the data file is on your machine, you need to figure out where (in which folder) to put it. If you work on your own machine, it's probably best to create a separate folder. On the Marriott Macs, it makes more sense just to work on the desktop. Whatever you decide, make sure that the folder exists and that it contains your HapMap data file along with a copy of `pgen.py`. Then launch Idle, get into the interactive window, and try importing `pgen`. To do so, just type "`from pgen import *`". If this works, nothing will print. If you get an error message, then Idle was unable to find the file `pgen.py`. To fix this problem, see section B.4 on page 53.

A.4 Downloading all the HapMap files

On your home machine, you may want to download *all* of the HapMap files, for your own personal use. You would not want to do this by pointing and clicking on each file, one after the other. It is much easier to use the program `ftp`, and connect to `ftp.hapmap.org`. If you want to do this, we'll be happy to show you how.

Appendix B

More Python

In writing this lab manual, we have introduced various Python constructs that were not covered in JEPy. We collect them here for convenient reference.

B.1 Basic Python

`zip(xv, yv)` is a Python facility that steps through the elements of several sequences (lists, tuples, strings, or whatever). For example,

```
>>> xv = ['x1', 'x2']
>>> yv = ['y1', 'y2']
>>> for x, y in zip(xv, yv):
...     print(x, y)
...
x1 y1
x2 y2
```

Each time through the loop, `x` and `y` are automatically set equal to corresponding elements from the lists `xv` and `yv`. This function is often used with a pair of lists, but it will also work with three or more. (This paragraph is a copy of one on page 39.)

B.2 The `random` module

This module is part of Python's standard library and is documented in full on the Python website: `http://docs.python.org/3.1/library`. We have made use of the functions listed below. Before using them, be sure to import the `random` module with a line of code such as

```
from random import *
```

After that command executes, you will have access to all the facilities of the `random` module. We list only a few of these here—the ones used in the lab projects but not covered by JEPy.

`choice(seq)` returns a random element from the sequence `seq`, which may be a list, a tuple, a string, or any Python object with similar properties. For example, `choice("abc")` returns `"a"`, `"b"`, or `"c"` with equal probability.

`expovariate(h)` returns a random value drawn from the exponential distribution with hazard h, or mean $1/h$.

`randrange(a, b)` returns a random integer from $a, a+1, \ldots, b-1$. The first argument is optional and defaults to 0 if omitted. Thus, `randrange(3)` returns either 0, 1, or 2 with equal probability, but `randrange(1,3)` returns either 1 or 2.

B.3 The pgen module

This module is available on the class website: `http://www.anthro.utah.edu/~rogers/ant5221/lab` as a file called `pgen.py`. Download that file, and place it in the same folder (or directory) as the Python program you are writing. At the top of your own Python program, import the `pgen` module as follows:

```
from pgen import *
```

B.3.1 Random numbers

`bnldev(n,p)` returns a random deviate drawn from the binomial distribution. When the function is called, n is a integer, the number of trials, and p is a float, the probability of "success" on each trial. Consider for example the experiment of tossing an unfair coin for which the probability of "heads" is 0.6 on each toss. To simulate the experiment of tossing this coin 30 times, we could use the command `x = bnldev(30,0.6)`. After the command executes, x will hold an integer between 0 and 30, which represents the number of "heads."

`mnldev(n, p)` returns a random deviate drawn from the multinomial distribution. This distribution generalizes the binomial. It might be used, for example, to describe the process of sampling balls with replacement from an urn containing balls of several colors. The number of balls drawn is called the number of *trials*. The probability that a random ball is (say) red is equal to the relative frequency of such balls within the urn. When `mnldev` is called, n (an integer) is the number of trials, and p is a list or tuple whose ith entry (p_i) is the probability that outcome i is observed on any given trial. The values in p only need to be *proportional* to the corresponding probabilities. They do not need to sum to unity; they only need to be positive numbers. For example, suppose that the urn contains 300 red balls, 100 black ones, and 50 white ones. To simulate the experiment of drawing 10 balls (with replacement), we could use the command `x = mnldev(10, [300, 100, 50])`. After the command executes, x will point to a list that contains the number of red balls drawn, then the number of black ones, and then the number of white.

`poidev(m)` returns a random deviate drawn from the Poisson distribution with mean m. For example, the command `x = poidev(10)` sets x to a value drawn from the Poisson distribution with mean 10.

B.3.2 For HapMap

The `pgen` module contains several facilities for working with HapMap genotype data. For a gentle introduction to these methods, see section 8.3.

`hapmap_fname(chromosome, population)` returns the name of the file (on your own computer) containing data for a given chromosome and population. It can find this file only if it resides in Python's current working directory (see section B.4).

`hapmap_dataset(filename)` creates an object of type `hapmap_dataset`. Such objects include the following data:

`filename` the name of the original HapMap data file

`snps` a list of objects of type `hapmap_snp`, each containing data from a single snp.

Objects of type `hapmap_dataset` are created as follows:

```
chromosome = 22
pop = "CEU"
hds = hapmap_dataset(hapmap_fname(chromosome, pop))
```

Now `hds` is an object of type `hapmap_dataset`. Its instance variables can be accessed like this:

```
>>> hds.filename
'/home/rogers/hapmap/hapmap-r23/genotypes_chr22_JPT_r23a_nr.b36_fwd.txt'
```

The variable `hds` behaves like a list or tuple of SNPs. `len(hds)` returns the number of SNPs in the data set, and `hds[3]` returns a pointer to the SNP at position 3 within the data set. The first SNP is `hds[0]`, and the last is `hds[len(hds)-1]`. Objects of type `hapmap_dataset` provide the following methods:

`find_position(pos)` returns the index of the SNP whose position on chromosome (measured in base pairs) is *closest to* (but not necessarily identical to) `pos`. The argument `pos` should be a positive integer. If `pos >= len(self)`, the function returns `len(self)-1`.

`find_id(id)` returns the index of SNP whose identifying string ("rs number") equals `id`. On entry, `id` should be a string, formatted as in HapMap data files. For example: "rs4284202." If `id` is not present in the data, the function returns `len(self)`.

`hapmap_snp` An object of this type represents all the data for a single SNP. Each such object includes the following data values:

- `id` the "rs number" (a unique identifier) of this SNP
- `alleles` a list of the alleles present at locus
- `chromosome` the label of the chromosome on which this SNP resides
- `position` an integer; the position of the SNP on the chromosome
- `gtype` a list of genotype data. Each item in the list is an integer, the number of copies of `alleles[0]` in an individual genotype.
- `sampleSize` number of values in `gtype`.
- `mean` the mean of `gtype`
- `variance` the variance of `gtype`

Suppose that `snp` is an object of type `hapmap_snp`. Then its data values can be accessed by syntax such as `snp.mean`, `snp.variance`, and so on. In addition, `snp` behaves like a list or tuple of genotypes. For example, `len(snp)` returns the number of genotypes (as does `snp.sampleSize`), and `snp[4]` returns the genotype at position 4.

B.3.3 Plotting

`charplot(x, y, nticks, outputheight, outputwidth)` prints scatterplots on terminals without graphics capabilities. On entry, x is a list of x-axis values and y a list of y-axis values. The other arguments are optional and specify the number of tick marks per axis, and the height and width of the output in character units. The specified number of tick marks is advisory only. The program will do its best to use tick marks that are as close as possible to the number requested without being ugly.

`scatsmooth(x, y, n, minx, maxx)` smooths a set of `(x,y)` data by dividing the range of `x` values into `n` equally-spaced bins, and calculating the average `y`-value within each bin. Function returns `(bin_x, bin_y, bin_n)`. For bin `i`, `bin_x[i]` is the midpoint of the `x` values, `bin_y[i]` is the mean of `y`, and `bin_n[i]` is number of observations. All parameters except `x` and `y` are optional. If `n` is omitted, the default value is 10. If `minx` and `maxx` are omitted, they default to `min(x)` and `max(x)`. See page 40.

B.4 Helping Python find input files

In these projects, Python will need to read several kinds of input files: HapMap data files, `pgen.py`, and the programs that you write. If it fails to find these files, nothing works. Here is an interaction that illustrates the problem:

```
>>> from pgen import *
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named pgen
```

Here, I tried to import the `pgen` module, and Python let me know that it could not find it. To avoid this problem, you must tell Python where your data files are.

Python searches for input files in a list of folders called the "path." To help it find these files, do one of two things: either put your own input files into one of the folders in Python's path, or else modify that path so that it includes the folder where your files reside. To examine the path, use Idle's "Path Browser," which you will find under the "File" menu.

B.4.1 Putting your own files into Python's path

On Macs, Python searches the Documents folder by default. If you put all your files there, Python will have no trouble finding them. This is the easiest approach and is the one to use when you are working in the lab at Marriott.

B.4.2 Putting your files into some other folder

If you work on your own computer, you may not want to clutter up Documents with the files from this class. On my own machine, I keep the material for this class in a separate folder called "pgen." You might want to do something similar.

If you launch Idle from an icon, it will not initially know about the folder that contains your files. But if you open a .py file within this folder and execute it, Idle automatically adds this folder to the path. After that, you can execute commands like `import pgen` without difficulty.

Those of us who work with the command line interface have it even easier. If you launch Idle from the command line, it automatically adds the current working directory to its search path and thus has no difficulty finding files.

B.4.3 Manipulating Python's current working directory

Python looks for input in a folder called the "current working directory," or CWD. This works well if you launched Python from the command line, for then the CWD is just the directory within which you typed the "python" command. But if you launched Python by clicking on an icon, the CWD is unlikely to be set to a useful value. You can, however, manipulate it by typing commands into Python's interpreter.

To do so, you must first import the `os` module:

```
>>> import os
```

You can then check the CWD like this:

```
>>> os.getcwd()
'/home/rogers'
```

This tells me that the CWD is the directory `/home/rogers`. If my input file is in that directory, Python will find it.

But suppose it is in a subdirectory of `/home/rogers` called `pgen`. I can change the CWD by typing

```
>>> os.chdir("pgen")
```

This changes the CWD to `/home/rogers/pgen`, provided that this directory exists on my computer. To check that this worked, use the `os.getcwd` command once again:

```
>>> os.getcwd()
'/home/rogers/pgen'
```

In the preceding example, the `os.chdir` command was very simple, because we were moving into a subdirectory of the directory we started in. If you need to make a more drastic move, specify the entire path name. For example, the command

```
>>> os.chdir("/home/rogers/pgen")
```

would would move us to `/home/rogers/pgen` no matter where we started from.

Appendix C

Tabulating Frequency Distributions

C.1 Introduction

To see the pattern in data, we often need to summarize it in a compact form. Tabulations are one such summary. They tell us how many observations are in each of several categories. This appendix describes two methods for tabulating data in Python, one for discrete data and one for continuous data.

C.2 Tabulating discrete data

Consider the following data set, which represents genotypes at a single nucleotide site:

```
>>> gtype = ['AA', 'AA', 'AT', 'TA', 'TA', 'AA', 'TA']
```

There are 7 observations in this data set, but all of them fall into 3 categories. We can use Python's `set` function to extract these three categories:

```
>>> for i in set(gtype):
...     print(i)
...
AA
AT
TA
```

In these data, we have both `AT` and `TA`. The distinction between these ordinarily doesn't matter in genetics. To convert all these into a common format, we can sort the characters within each genotype as follows:

```
>>> gtype = ["".join(sorted(i)) for i in gtype]
```

This uses a list comprehension (indicated by the square brackets) to create a new list. Within the list comprehension, the builtin function `sorted` turns each genotype into a sorted list of individual characters. Then `"".join` converts each list into a character string. In the end, we have a data set with 7 observations but only two genotypes, `AA` and `AT`:

```
>>> for i in set(gtype):
...     print(i)
...
AA
AT
```

To count the occurrences of each genotype in the data, we make use of the fact that `gtype` is defined as a Python list, and lists have a builtin method called `count`, which counts the occurrences of its argument in the list. For example,

```
>>> gtype.count('AA')
3
```

Here's a snippet of code that counts the occurrences of each genotype in the data:

```
>>> h = {}
>>> for i in set(gtype):
...     h[i] = gtype.count(i)
...
>>> print(h)
{'AA': 3, 'AT': 4}
```

This defines a Python dictionary called `h`, which is initially empty. Then we iterate across the genotypes, using the `count` method to add observations to the dictionary. Finally, we print the dictionary. The output says that the data contain 3 copies of `AA`, and 4 of `AT`.

This is helpful, but the output is hardly pretty. It would be better to print the output as a table, with genotypes in one column and counts in another. Here's how:

```
>>> for key in sorted(h.keys()):
...     print(key, h[key])
...
AA 3
AT 4
```

Here, I've used the `keys` method that is built into Python dictionaries to get a list of genotypes. I then hand this list to `sorted`, which sorts the list of genotypes. We then iterate across these genotypes, printing the genotype and the corresponding count in two columns.

Suppose you had a list of genotypes, but you wanted to tabulate alleles rather than genotypes. The trick is to begin by concatenating all the genotypes into a single character string:

```
>>> dnaseq = ''.join(gtype)
>>> for i in set(dnaseq):
...     print(i, dnaseq.count(i))
...
A 10
T 4
```

C.3 Tabulating continuous data

When working with continuous data, the number of different values may be enormous. There is thus little point in counting the occurrences of each value. We need a different approach. With continuous variables, it makes more sense to group the data into bins. This is tedious, so we have written a Python module that does the heavy lifting. It is in a file called `pgen.py`, which you can download from the class web site. Put it in the same directory (or folder) as your other Python programs.

Here is a program that tabulates continuous data into 5 bins:

```
from pgen import Tabulator  # read module Tabulator

data = [0.39760402926232336, 0.38844722349063998, 0.15828823308128848,
        0.21675307512013373, 0.67759054634129579, 0.63336008432108437,
        0.7791838758913473, 0.11329205659594056, 0.088616376501101851,
        0.27797955173023731]

# Construct a Tabulator named tab.  The range [0,1] is divided into 5
# bins, and the Tabulator counts the number of values within each
# bin. It also counts the values that fall above or below the range [0,1].

tab = Tabulator(low=0.0, high=1.0, nBins=5)

# Each pass through the for loop examines one piece of data
for x in data:
    tab += x     # Add 1 to the count in the relevant bin

print(tab)       # print the tabulated data
```