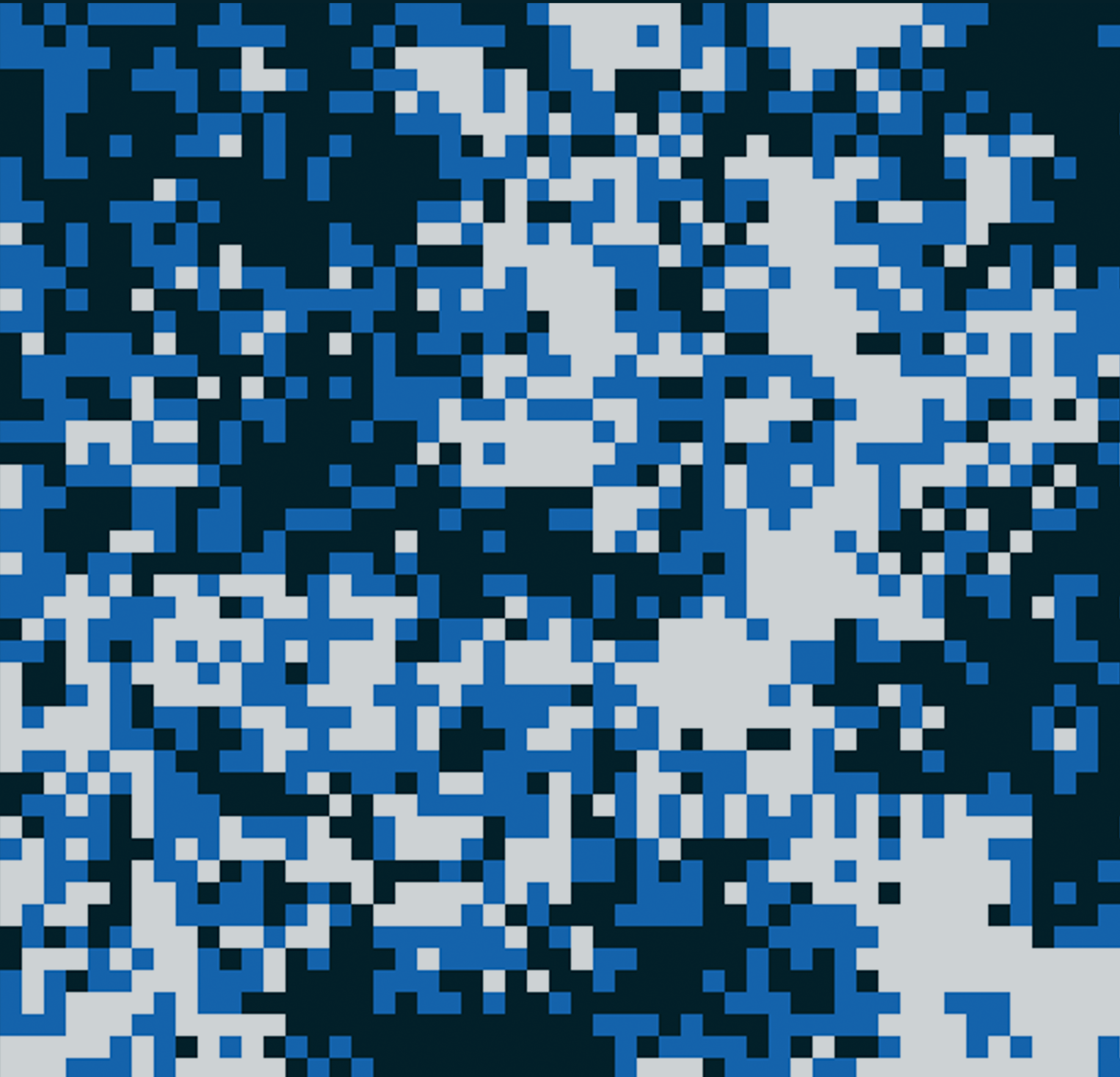
Nature, In Code

Learning programming while discovering the rules that govern life

Marcel Salathé



Nature in Code

Biology in Javascript - Learning programming while discovering the rules that govern life

Marcel Salathé

This book is for sale at http://leanpub.com/natureincode

This version was published on 2016-12-07

To Rahel, Jonas, and Elina

Contents

3. Genetic Drift: The Power of Chance

In the previous chapter, we have established that under Hardy-Weinberg assumptions - infinite population size, random mating, no mutation or selection, etc. - nothing ever changes. Allele frequencies stay the same forever, and if the genotype frequencies are not currently at Hardy-Weinberg frequencies, they will get there in a single generation, and then remain there forever.

It's now time to relax some if these assumptions. The first assumption that we are going to relax is that of the infinite population size. We are now going to assume that populations are finite. It turns out that this has enormous effects for evolution. When populations are finite, chance effects will start to play a role. These chance effects are stronger when populations are smaller. It is intuitively easy to understand this. Suppose you toss a coin - you know that on average, it will come up heads half of the time, and tails the other half of the time. However, if you only toss the coin a few times, you can get very non-even distributions of heads and tails. Say you toss it ten times. It may end up tossing 5 heads and 5 tails (5:5), but you will often find yourself tossing 6:4, or 7:3. Sometimes you'll even get 8:2 or 9:1; and even a 10:0, although very rare, wouldn't be totally unexpected. If you now increase your coin tossing efforts by one order of magnitude, and toss your coin 100 times, you would notice that it's very rare to get beyond a 30:70 ratio, and you would hardly ever hit 20:80 or below. And in fact, getting tails or heads 100 times in a row is so unlikely, it's hard to come up with a comparison. The best I can come up with is that it's about as likely as winning the lottery four consecutive times. Increase your coin tossing efforts again by a order of magnitude (1000 tosses), and anything less equal than 450:550 becomes highly improbable.

You get the idea: The more often you toss the coin, the closer you will get to the perfect equilibrium that you would expect. The implication of this is that *small population sizes will be more prone to random chance effects than large populations*. However, all finite population sizes are prone to chance effects, and over evolutionary timescales, even large population sizes where random chance effects are weak in the short term may eventually show signs of these effects. When allele frequencies change over time (i.e. when evolution occurs) due to these random effects, we call it *genetic drift.*

This is an important chapter. It's important because it introduces the concept of randomness, a key concept in all of biology. One could even argue that the main difference between the dynamics of living systems (biological systems), and the dynamics of non-living systems (chemical and physical systems) is randomness. The science of evolutionary biology has been dominated for a long time by the perspective of natural selection. Nowadays, we understand that natural selection is not the only process affecting evolution. Which force is more important for evolution - natural selection or genetic drift - is one of the great debates in contemporary evolutionary biology. But even those favoring natural selection would agree that randomness, and genetic drift, is a major force. In this chapter, we will shed some light on this force, and on its consequences.

Randomness

Randomness is the idea of events occurring at random, by chance. If they happen by chance, it's impossible to predict them. The more sophisticated term for randomness is stochasticity - random events are said to be *stochastic*.

While random events cannot be predicted, we can still say something about their probability of occurring. This is the main idea behind the theory of probability. For example, a fair coin comes up heads half of the time, and tails the other half of the time. While we can't predict the outcome of a single coin toss, we still know that the chance, or probability, that the coin comes up heads, is exactly 50% (in reality, no coin is exactly fair, and the probability of heads is probably slightly lower or higher than 50%, but in a fair coin, the probability is 50% by definition). Because we know that each coin toss comes up 50% heads and 50% tails, we can calculate the probability that ten coin tosses come up heads 80% of time, and tails 20% of the time, for example. For any ten coin tosses, it is completely impossible to predict exactly how many will come up heads; but because we can calculate the probability of any outcome, we can at least get a good sense of what to expect. For example, we can calculate that the probability of 80% heads 20% tails is 4.39% like so: $\frac{\frac{10*9}{2}}{2^{10}} = 0.0439$ (don't worry about how to calculate this, but if you are interested, here is a good explanation: http://gwydir.demon.co.uk/jo/probability/info.htm[1]). An outcome like this isn't unlikely, but you wouldn't want to bet the farm on it either.

Before we get to the biology of randomness, I want to jump straight to code. The great thing about having fast computers is that we can simulate randomness over and over again, in a very short amount of time. This allows us to run millions of coin tossing experiments in a computer in a split second.

In JavaScript, the quintessential method to produce randomness is `Math.random()`. It takes no arguments, but produces a random number between 0 and 1. Let's try it out. Create a new HTML document with a `<script>` element, and log the output of the method like so:

```
console.log(Math.random());
```

Save and load the file in the browser, and take a look at the console output. Indeed, a random number between 0 and 1 has been generated. Verify this by reloading the page over and over again. You'll notice that the number is different, every time you reload the page, as it should be.

Let's go ahead and verify the claim I made. I claimed that `Math.random()` produces a random number between 0 and 1. I'm going to clarify this further by saying not only will it generate a random number between 0 and 1, but that each number is equally likely to be generated. If that is true, then the average output of `Math.random()` should be 0.5. To verify that, let's use a `for` loop to create many random numbers, and then calculate the average of these numbers:

[1] http://gwydir.demon.co.uk/jo/probability/info.htm

```
var sum = 0;
var repeats = 100;
for (var i=0; i<repeats; i=i+1) {
    sum = sum + Math.random();
}
var average = sum / repeats;
console.log("The average is", average);
```

If you followed the examples in the previous chapters, this should now look familiar. At first, we are declaring two variables, `sum` and `repeats`. Then, we're using a `for` loop to add to `sum` whatever `Math.random()` returns, summing up the random numbers. We do this exactly as many times as defined by the variable `repeats`. Then, we simply calculate and print the average.

When I do this, I'm getting an average of

```
0.533668438885361
```

Upon reload, I'm getting

```
0.46173790065106
```

Another reload gives me

```
0.4819509003055282
```

and so on. This isn't very precise, so let's maybe increase the number of repeats. You should be able to increase repeats to 100,000 without having your browser breaking a sweat. With 100,000 repeats, I'm now getting

```
0.5000435156048741
0.5006688627070328
0.4989869923099107
```

when I reload the document three times. Already much closer!

Try increasing your repeats by a factor of ten, for as long as your browser can handle it (stop when it takes a few seconds). On my laptop, I can go up to 1 billion repeats before I have to wait a few seconds, and then I'm getting

```
0.499998771590113
0.49996582762364633
0.5000288064449279
```

As you can see, the more repeats, the closer we get to 0.5. However, keep in mind that floating point numbers have a limited accuracy, as discussed in the previous chapter. Usually, you won't ever notice this, because the inaccuracy occurs at insignificant digits, but if you keep adding numbers, for example, the inaccuracy can potentially add up too. The point of this exercise was not to lead you astray with floating point accuracy, which you will probably never encounter as a problem. The point was to introduce you to `Math.random()` - and to demonstrate how powerful our computers are nowadays. In just a few seconds, my laptop computer generated a billion numbers and added them up!

A note on Math.random()

Above, I introduced `Math.random()` as a method that produces a random number between 0 and 1. If you now think that this does neither include 0 nor 1, you could be forgiven, and indeed, for the purposes of this book, it wouldn't matter. But I want you to know that while it is correct that 1 is not included, 0 is technically included. In other words, it is in principle possible to get a perfect 0 by calling `Math.random()`, while it is entirely impossible to ever get a 1.

Now let's go back to the previous coin tossing example. I mentioned that if I tossed a fair coin ten times, the probability of 80% heads and 20% tails is 4.39%. How can we verify this in JavaScript? By actually running the coin-tossing experiment in the computer! This is a so-called *simulation*, where you simulate a process from the real world in the virtual world of a computer.

First, let's implement the coin tossing. Create a new HTML file and have it execute the following script:

```
var coins = 10;
var heads = 0;
var tails = 0;
for (var i = 0; i < coins; i = i + 1) {
    if (Math.random() <= 0.5) {
        heads = heads + 1;
    }
    else {
        tails = tails + 1;
    }
}
console.log(heads,"heads",tails,"tails");
```

This should give you an output like `4` "heads" `6` "tails", and on reload, the output should change (although not always - by chance you might get the same outcome two times in a row).

Let's go through this code in detail, because it introduces a new concept we haven't met yet: that of control flow. The code setup is straightforward - you initiate three variables, and then implement a `for` loop throwing the coins using `Math.random()`. The variables `heads` and `tails` should act as counters, so that whenever the coin comes up heads (i.e. when `Math.random()` returns a value that is smaller or equal to `0.5`), we increase the `heads` counter by one, and whenever the coin comes up tails (i.e. when `Math.random()` returns a value that is greater than `0.5`), we increase the `tails` counter by one.

In other words, the execution of some of the code (e.g. increase `heads` counter by one) is *conditional* - the code should only be executed if a given condition is met. As you can imagine, this is a key concept in programming. The way this concept can be implemented in JavaScript is as follows:

```
if (condition) {
    statement1
}
else {
    statement2
}
```

This is easy to read: if the `condition` is true, then execute `statement1`, otherwise execute `statement2`. Sometimes, the otherwise statement is simply to do nothing, in which case you can omit the `else` altogether and simply write:

```
if (condition) {
    statement
}
```

Finally, there is also an `else if`, in case you have multiple conditions:

```
if (condition1) {
    statement1
}
else if (condition2) {
    statement2
}
else {
    statement3
}
```

If `condition1` is true, then only `statement1` is executed. If `condition1` is false, but `condition2` is true, then `statement2` is executed. If both conditions are false, then `statement3` is executed.

Sometimes, you see this type of code:

```
if (condition)
    statement1;
```

This is technically correct - if the statement is only one line of code, you can theoretically omit the curly brackets. *Don't ever do this* - it will almost certainly introduce nasty errors down the line. You may for example want to add a second statement later on, and may end up doing something like this:

```
if (condition)
    statement1;
    statement2;
```

You might think that `statement2` will be executed only if `condition` is true, but that would be wrong. The code above is equivalent to:

```
if (condition) {
    statement1;
}
statement2;
```

This means that `statement2` will always be executed, which is not at all what you wanted. There's a simple rule to avoid this danger: *always use curly brackets when you use if and else statements*.

We need to understand one more thing about control flow - that of the condition itself. A condition must always evaluate to be either true or false. Like all programming languages, JavaScript provides a special type called *boolean*, which can either be true or false (the other two types you have encountered so far were numbers and strings). You can use this type in normal variables, like so:

```
var is_ready = false;
```

A condition, on the other hand, is not a variable that you set yourself - it's an expression that is either true or false. I'm going to show you a few examples of expressions that evaluate to true or false, and while there are many more, these are the ones you need when dealing with numbers.

Smaller than:
`3 < 4` evaluates to true
`4 < 3` evaluates to false

Smaller or equal than:
`4 <= 4` evaluates to true
`5 <= 4` evaluates to false

Greater than:
`4 > 3` evaluates to true
`3 > 4` evaluates to false

Greater or equal than:
`4 >= 4` evaluates to true
`4 >= 5` evaluates to false

Equals:
`4 == 4` evaluates to true
`4 == 5` evaluates to false

This last expression is the source of most beginner's mistakes. To compare two values and test their equality, you use the == operator. This operator is deceptively similar to the assignment operator = that we have used many times before, when assigning a value to a variable.

Now back to our coin tossing example - the code should be easy to understand now. It basically says that if `Math.random()` returns a number that is smaller than or equal to `0.5`, the counter for `heads` will increase by `1`. Otherwise, the counter for `tails` will increase by `1`.

At the moment, our code simply tosses a coin ten times and then prints how often it came up heads or tails. What we want to do, however, is to figure out whether my original claim - that the probability of 80% heads 20% tails is 4.39% - is correct. What we need to do, then, is to repeat our code thousands of times, and count how often we get exactly 8 heads and 2 tails. So the first thing I'm going to do is to wrap the coin tossing functionality into a function that I'm going to name `throw_coins()`:

```
function throw_coins() {
    var coins = 10;
    var heads = 0;
    var tails = 0;
    for (var i = 0; i < coins; i = i + 1) {
        if (Math.random() <= 0.5) {
            heads = heads + 1;
        }
        else {
            tails = tails + 1;
        }
    }
    if (heads == 8) {
        return true;
    }
    else {
        return false;
    }
}
```

You'll recognize the first part of the function - it's an exact copy of the code that we developed above. But then I've removed the line with the `console.log` method, since there is no need to print the outcome every time we throw the coin 10 times. However, what we need instead is for the function to somehow tell us what happened. So what I added there at the end simply says if heads comes up 8 times, then return `true`, otherwise, return `false`.

Now I can call this function however many times I want to, and then count how many times the function returns `true`, which indicates that the 10 coin tosses resulted in exactly 8 heads and 2 tails. Here's how I do this:

```
var repeats = 10000000;
var counter = 0;
for (var i = 0; i < repeats; i = i + 1) {
    var desired_outcome = throw_coins();
    if (desired_outcome) {
        counter = counter + 1;
    }
}
console.log("Getting 8 heads, 2 tails " + (counter / repeats) * 100 + "% of the \
time")
```

Be sure to set the `repeats` variable to a value that allows your computer to execute the code in just a few seconds. It's best to start with a small value like 1000 and then increase it by an order of magnitude, as we have done before. In my case, the sweet spot seems to be 10 million.

The setup here should look familiar - you initialize the number of repeats in a variable called `repeats`, then initialize a variable `counter` at `0`, and then iterate using a `for` loop. In the loop, I call the function `throw_coins()`, and whatever it returns gets stored in the variable `desired_outcome`. If that happens to be `true` - which means that the 10 coin tosses resulted in 8 heads and 2 tails - I am going to increase the `counter` by 1. Once the loop has run its course, I'm simply printing how often, in percentage, 10 coin tosses resulted in 8 heads and 2 tails.

Thus, our full code is as follows:

```
function throw_coins() {
    var coins = 10;
    var heads = 0;
    var tails = 0;
    for (var i = 0; i < coins; i = i + 1) {
        if (Math.random() <= 0.5) {
            heads = heads + 1;
        }
        else {
            tails = tails + 1;
        }
    }
    if (heads == 8) {
        return true;
    }
    else {
        return false;
    }
}

var repeats = 10000000;
var counter = 0;
for (var i = 0; i < repeats; i = i + 1) {
    var desired_outcome = throw_coins();
    if (desired_outcome) {
        counter = counter + 1;
    }
}
console.log("Getting 8 heads, 2 tails " + (counter / repeats) * 100 + "% of the \
time");
```

When I run this code three times (i.e. reload the page three times) I'm getting the following output:

```
Getting 8 heads, 2 tails 4.39729% of the time
Getting 8 heads, 2 tails 4.39629% of the time
Getting 8 heads, 2 tails 4.38968% of the time
```

Thus, 4.39% is indeed correct.

The Randomness of Finite Populations

So how is life random, in the biological sense? Think about the circle of life in most animals - new life is conceived, a single fertilized egg (the zygote) multiplies and the animal grows to the reproductive age, reproduces, and eventually dies. Its offspring goes through the same cycle, and so does its offspring, and so on. Randomness can hit the animal at any stage from the fertilized egg (the zygote) to the moment it reproduces, and a new zygote is created. Which genes get passed on will be largely random. When a single cell grows into billions of cells forming an adult capable of reproduction, a lot can go wrong due to randomness. At any time, an animal can be struck by a deadly disease, be eaten by a predator, die of hunger and thirst, etc.

Life is complicated, and if we would set out to simulate life in all its details, we would end up with a very complicated model. But just as we have done in the previous chapter, we can again reduce the complexity into a simple model that captures the essence of these random processes. Once again, the model is named after the two originators, Sewall Wright and Ronald Fisher: it's called the *Wright-Fisher model*.

Before we go on, it's important to remind ourselves that the only assumption we are relaxing from the Hardy-Weinberg model is that of the infinite population size. We still have random mating, we still have no selection, etc. *We are only interested in the effect of a finite population size.*

The complexity of the scenario described above can be reduced to a simple sampling process. Imagine that we have a finite population of *N* individuals. Once again, let's zone in on one gene, with two alleles. If we have *N* diploid individuals, that means we have *2N* copies of the alleles in the population (because every individual has two copies). Once again, we assume that individuals produce infinitely many allele copies from which the next generation is formed. However, this time, we don't form infinitely many offspring individuals - we stick to our population size of *N* individuals.

This process can be thought of as having a jar with *2N* marbles in it (the marbles being the actual allele copies of the *N* individuals). To generate the next generation, we *sample* - we simply pick a marble at random (representing a randomly picked allele), and then put it back into the jar. This "replacement" step ensures that we are sampling from an infinitely large pool of allele copies. To generate an individual, we need to repeat this process, because each individual consists of two alleles. Once we have picked two marbles in that fashion, we will have generated one offspring genotype. Great! But in order to produce the next generation, which should have a population size of *N*, we need to keep repeating this procedure, until we have picked (or sampled) 2N allele copies, representing the next generation.

This random sampling process can have profound consequences. Let's go through an example, with N = 10. This is arguably an extremely small population, but it helps us keep the example manageable. Let's also assume that we start with p = q = 0.5, which means that the frequencies of both alleles, A1 and A2, are equally 50%. Starting with this population, what will the next generation look like? Let us remind ourselves here what we have established in the previous chapter: in an infinite population under the same conditions (i.e. Hardy Weinberg conditions), the allele frequencies would not change - they would stay at 50% for ever. In other words, no evolution would occur.

Starting from 10 individuals, we have 20 allelic copies in the population, half of them are A1, and half of them are A2. We don't care about the genotype frequencies at the moment - this chapter is all about allele frequencies. In order to generate the next generation of 10 individuals, or 20 allelic copies, we need to randomly sample, with replacement, from the marble jar representing the infinite pool of allele copies.

Let's get started. We grab the first allele - it's an A1! (Recall that the chance is 50/50, given the allele frequencies in the parent generation.) Ok, let's copy this A1 allele, put it back in the jar, and grab another one. Again an A1! Let's copy it again, put it back in the jar, and grab another one. An A2! And so on.

After we have done this 20 times, we assess our new allele pool. Let's say we have drawn allele A1 12 times, and allele A2 8 times. In other words, p, the frequency of the A1 allele, is now 0.6, and q, the frequency of the A2 allele, is now 0.4.

Let me repeat this. p is now 0.6, and q is now 0.4.

Ok, let me repeat this again. *p is now 0.6, and q is now 0.4.*

Mind blown? It should be.

Let's consider what just happened. In a single generation, we went from *p = q = 0.5* to *p = 0.6* and *q = 0.4*. That is, in absolute terms, a dramatic change of allele frequencies, in a single generation. And since you now know that a change of allele frequencies is pretty much the definition of evolution, another way of saying this is that we have just observed a dramatic evolutionary change in a single generation. "Fair enough", I hear you say, "but we are talking about a very small population of 10 animals. Certainly these effects are much weaker in larger populations." That is true - but the effect is still there. And keep in mind, we have observed only a single generation. What happens over evolutionary timescales? Thousands, hundreds of thousands, millions of generations? Even small changes in allele frequencies will add up.

And here is where we go back to code. We can, in fact, simulate the dynamics of larger populations over many generations. We will be using the exact same approach that we've taken above in the coin tossing example. Our goal is to simulate the allele frequencies over time, starting from *p = q = 0.5*, in a population of 1,000 individuals. Here is the code:

```
var p = 0.5;
var N = 1000;
var generations = 1000;

function next_generation() {
    var draws = 2 * N;
    var a1 = 0;
    var a2 = 0;
    for (var i = 0; i < draws; i = i + 1) {
        if (Math.random() <= p) {
            a1 = a1 + 1;
        }
        else {
            a2 = a2 + 1;
        }
    }
    p = a1/draws;
}

for (var i = 0; i < generations; i = i + 1) {
    next_generation();
    console.log("generation "+i+":\tp = " + round_number(p,3) + "\tq = " + round\
_number(1-p,3));
}
```

(Note that I'm omitting the `round_number()` method for brevity - you can just copy it from the previous chapter).

The function `next_generation()` is almost identical to the function `throw_coins()` from above, with a small but important difference. Instead of saying

```
if (Math.random() <= 0.5)
```

we are using

```
if (Math.random() <= p)
```

The first line is correct if you want to do something 50% of the time. However, when the frequency of allele A1 is `p`, then the second line is correct. Imagine for example that the frequency of allele A1 is `0.8`. This means that when you randomly sample from the allele copy pool, you will pick an A1 allele 80% of the time. Because `p` will be `0.8`, the line

```
if (Math.random() <= p)
```

is equivalent to

```
if (Math.random() <= 0.8)
```

which is saying "80% of the time" (because indeed, 80% of the time, `Math.random()` will return a number smaller than or equal to `0.8`).

This is an important line, make sure you understand it. It's your key to programming stochastic events.

With the code saved in a new HTML document, reload that document and look at the output in the JavaScript console. It will output one thousand lines, representing the allele frequencies over 1000 generations (i.e. evolution). When I run this a couple of times, I'm getting the following frequencies at generation 999:

```
generation 999:         p = 0.886        q = 0.114

generation 999:         p = 0.585        q = 0.415

generation 999:         p = 0        q = 1

generation 999:         p = 0.953        q = 0.047

generation 999:         p = 0.124        q = 0.876
```

and so on. If you scroll through the generations, you can see that the allele frequencies are changing wildly. In other words, there is a lot of evolution going on, despite the complete lack of natural selection. This is genetic drift - evolution due to chance.

The third simulation from my examples above has a pretty remarkable outcome: the allele A1 has completely disappeared from the population, and all alleles are A2. In fact, when I scroll back through time in this simulation, I find the following:

```
...
generation 771:         p = 0.003        q = 0.997
generation 772:         p = 0.002        q = 0.998
generation 773:         p = 0.001        q = 0.999
generation 774:         p = 0        q = 1
generation 775:         p = 0        q = 1
generation 776:         p = 0        q = 1
generation 777:         p = 0        q = 1
generation 778:         p = 0        q = 1
generation 779:         p = 0        q = 1
generation 780:         p = 0        q = 1
generation 781:         p = 0        q = 1
...
```

At generation 774, the allele A1 is completely lost from the population, and after that, it will never come back into the population again (because our assumptions say we have no mutation and no migration). This is what loss of genetic diversity looks like in mathematical terms.

Now go ahead and change your code to say that the simulation should run for ten thousand generations, not just for a thousand generations. That is, change this line

```
var generations = 1000;
```

to

```
var generations = 10000;
```

and run the simulation again. Because we increased the number of generations by an order of magnitude, the simulation will now take longer, but did you notice the pattern in the results? I'm sure you did, because it's really hard to miss: in (almost) all cases, one of the two alleles will disappear entirely from the population. There might be the rare simulation where you still have both alleles after 10,000 generations, but in my 50 runs or so, one of the alleles had always disappeared by the end of the simulation.

This is a remarkable result, and it is perhaps one of the key insights about genetic drift, which is this: *genetic drift reduces genetic variation.* This may sound a little counterintuitive at first, because we're accustomed to think that randomness, or stochasticity, leads to more variation, not less. But you can observe the pattern very clearly in the results of your simulations. Why is that?

Fundamentally, genetic drift cannot add more variation - it is simply a random sampling process, unable to generate variation on its own. At the same time, because it is a random sampling process, alleles can eventually be removed, thus reducing variation.

As we have established in the beginning of this chapter, stochastic effects are strongest when population sizes are smaller. Extreme results, like throwing only heads, are much more likely when you are throwing the coin only a few times. In the same vein, when the population size is small, loss of an allele is much more likely, and will thus occur much sooner, than when the population size is big. Recall from above that at population size `N=1000`, only rarely was an allele lost within `1000` generations, typically. Go ahead and change your population size to `100` (also be sure to set the simulation to run for `1000` generations only), and you will see that in practically all cases, one of the alleles will be lost.

Visualizing Drift

These simulations are a great tool to examine the dynamics of genetic drift, but it's a bit cumbersome to scroll though hundreds or thousands of lines of allele frequencies. Wouldn't it be nice if we had a way to visualize the allele dynamics? Let's start visualizing things to get a better overview about what's going on here.

Before we get to it, a word of warning: Data plotting in the browser isn't trivial. What I'm going to do here is to give you some plotting code that I'm simply asking you to copy and paste into your documents. The code that we will continue to write focusses on generating the data, and we then hand the data over to a plotting function. There's no need for you at this stage to understand how plotting works, and it would be a significant distraction at the moment. Feel free to examine the plotting code, of course, but I won't explain it, nor do I expect you to understand it.

Copying Code

Just as a reminder: all the code, including the plotting code below, is available at the book website www.natureincode.com[2]. I recommend copying the plotting code from the website - some ebooks readers introduce weird characters that the browsers don't like.

First, add this line to your code, at the beginning of your `<head>` element:

```
<script src="http://d3js.org/d3.v4.js"></script>
```

This line loads an external JavaScript library called `D3.js`, which is the most advanced data plotting library available for the browser to date. Note that you need to be connected to the internet for this library to be loaded!

A note on D3 versions

D3 is an external library, which means things can change outside of our control. The current version is 4, as you can see in the URL above (d3.v4.js). However, many people still use version 3, and the "Nature, in Code" videos (see website www.natureincode.com[3]) also were created when version 3 was the latest version. But no need to worry - the plotting code below works in both versions.

Next, you will need to add the plotting code to your document.

[2]http://www.natureincode.com
[3]http://www.natureincode.com

Careful!

So far, it didn't matter whether your JavaScript code was located in the `<head>` or the `<body>` of the document. However, for the visualizations in this book to work, the JavaScript code that generates the visualizations needs to be located in the <body>. I recommend you use the following HTML structure as a template:

```
<!DOCTYPE html>
<html>
   <head>
       <title>Nature, In Code</title>
       <script src="http://d3js.org/d3.v4.js"></script>
   </head>
   <body>
       <script type="text/javascript">
           // your code
       </script>
   </body>
</html>
```

You will need to have the following plotting code in your document:

```
function draw_line_chart(data,x_label,y_label,legend_values,x_max,y_max_flex) {
    var margin = {top: 20, right: 20, bottom: 50, left: 50},
        width = 700 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;

    var version = d3.scale ? 3 : 4;
    var color = (version == 3 ? d3.scale.category10() : d3.scaleOrdinal(d3.schem\
eCategory10));

    if (!x_max) {
        x_max = data[0].length > 0 ? data[0].length : data.length
    }

    var y_max = data[0].length > 0 ? d3.max(data, function(array) {
            return d3.max(array);
        }) : d3.max(data);

    var x = (version == 3 ? d3.scale.linear() : d3.scaleLinear())
        .domain([0,x_max])
        .range([0, width]);
    var y = y_max_flex ? (version == 3 ? d3.scale.linear() : d3.scaleLinear())
        .domain([0, 1.1 * y_max])
        .range([height, 0]) : (version == 3 ? d3.scale.linear() : d3.scaleLinear\
())
        .range([height, 0]);

    var xAxis = (version == 3 ? d3.svg.axis().scale(x).orient("bottom") :
            d3.axisBottom().scale(x));

    var yAxis = (version == 3 ? d3.svg.axis().scale(y).orient("left") :
            d3.axisLeft().scale(y));

    var line = (version == 3 ? d3.svg.line() : d3.line())
        .x(function (d, i) {
            var dat = (data[0].length > 0 ? data[0] : data);
            return x((i/(dat.length-1)) * x_max);
        })
        .y(function (d) {
            return y(d);
        });

    var svg = d3.select("body").append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis)
        .append("text")
        .style("text-anchor", "middle")
        .attr("x", width / 2)
        .attr("y", 6)
        .attr("dy", "3em")
        .style("fill", "#000")
        .text(x_label);

    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("x", -height / 2)
        .attr("dy", "-3.5em")
        .style("text-anchor", "middle")
        .style("fill", "#000")
        .text(y_label);

    if (legend_values.length > 0) {
        var legend = svg.append("text")
            .attr("text-anchor", "star")
            .attr("y", 30)
            .attr("x", width-100)
            .append("tspan").attr("class", "legend_title")
            .text(legend_values[0])
            .append("tspan").attr("class", "legend_text")
            .attr("x", width-100).attr("dy", 20).text(legend_values[1])
            .append("tspan").attr("class", "legend_title")
            .attr("x", width-100).attr("dy", 20).text(legend_values[2])
            .append("tspan").attr("class", "legend_text")
            .attr("x", width-100).attr("dy", 20).text(legend_values[3]);
    }
    else {
        svg.selectAll("line.horizontalGridY")
            .data(y.ticks(10)).enter()
            .append("line")
            .attr("x1", 1)
            .attr("x2", width)
            .attr("y1", function(d){ return y(d);})
            .attr("y2", function(d){ return y(d);})
            .style("fill", "none")
            .style("shape-rendering", "crispEdges")
            .style("stroke", "#f5f5f5")
            .style("stroke-width", "1px");

        svg.selectAll("line.horizontalGridX")
            .data(x.ticks(10)).enter()
            .append("line")
            .attr("x1", function(d,i){ return x(d);})
            .attr("x2", function(d,i){ return x(d);})
            .attr("y1", 1)
            .attr("y2", height)
            .style("fill", "none")
            .style("shape-rendering", "crispEdges")
            .style("stroke", "#f5f5f5")
            .style("stroke-width", "1px");
    }

    d3.select("body").style("font","10px sans-serif");
    d3.selectAll(".axis line").style("stroke","#000");
    d3.selectAll(".y.axis path").style("display","none");
    d3.selectAll(".x.axis path").style("display","none");
    d3.selectAll(".legend_title")
        .style("font-size","12px").style("fill","#555").style("font-weight","400\
");
    d3.selectAll(".legend_text")
        .style("font-size","20px").style("fill","#bbb").style("font-weight","700\
");

    if (data[0].length > 0) {
        var simulation = svg.selectAll(".simulation")
            .data(data)
            .enter().append("g")
            .attr("class", "simulation");

        simulation.append("path")
            .attr("class", "line")
            .attr("fill", "none")
            .attr("d", function(d) { return line(d); })
            .style("stroke", function(d,i) { return color(i); });
    }
    else {
        svg.append("path")
            .datum(data)
            .attr("class", "line")
            .attr("fill", "none")
            .attr("d", line)
            .style("stroke","steelblue");
    }
    d3.selectAll(".line").style("fill", "none").style("stroke-width","1.5px");  \

}
```

Like I said, let's not bother to understand this code. All we care about is that it defines a function called `draw_line_chart()` with a number of parameters: `data`, `x_label`, `y_label`, `legend_values`, `x_max`, and `y_max_flex`. The `data` corresponds to the data that we generate in the simulation, i.e. the frequencies of the A1 allele over time. `x_label` and `y_label` simply correspond to the labels that we want on the axes. The parameter `legend_values` contains the values that we want to show in the legend. We will use the two remaining parameters `x_max`, and `y_max_flex`, later. At the moment, simply note that if you don't provide these last two parameters when calling the function, their value in the function will be `undefined`.

The function `draw_line_chart()` is quite generic: indeed, it will be the only function you'll need to use to plot line charts throughout the book. The only other plotting function that we'll use later in the book will be used to plot spatial dynamics.

With the plotting code in place, we are now ready to plot our simulation results.

Let's take our simulation code form above, and modify it slightly:

```
var p = 0.5;
var N = 500;
var generations = 1000;
var data = [];

function next_generation() {
    var draws = 2 * N;
    var a1 = 0;
    var a2 = 0;
    for (var i = 0; i < draws; i = i + 1) {
        if (Math.random() <= p) {
            a1 = a1 + 1;
        }
        else {
            a2 = a2 + 1;
        }
    }
    p = a1/draws;
    data.push(p)
}

for (var i = 0; i < generations; i = i + 1) {
    next_generation();
}
draw_line_chart(data,"Generation","p",["Population Size:",N,"Generations:",gener\
ations]);
```

I’ve made a few changes, which I’m going to explain next. First, after I’ve played around with various values for `N` and `generations` above, I’ve set them to `500` and `1000`, respectively.

Next, I initialized an empty *array* called `data`. with the following line:

```
var data = [];
```

Arrays are the most important data structures in JavaScript, and I’ll talk more about arrays below. For now, simply note that you can store multiple values (e.g. multiple numbers) in an array. We’ll use it to store the values of p over time.

In the function `next_generation()`, I’ve added this line at the end:

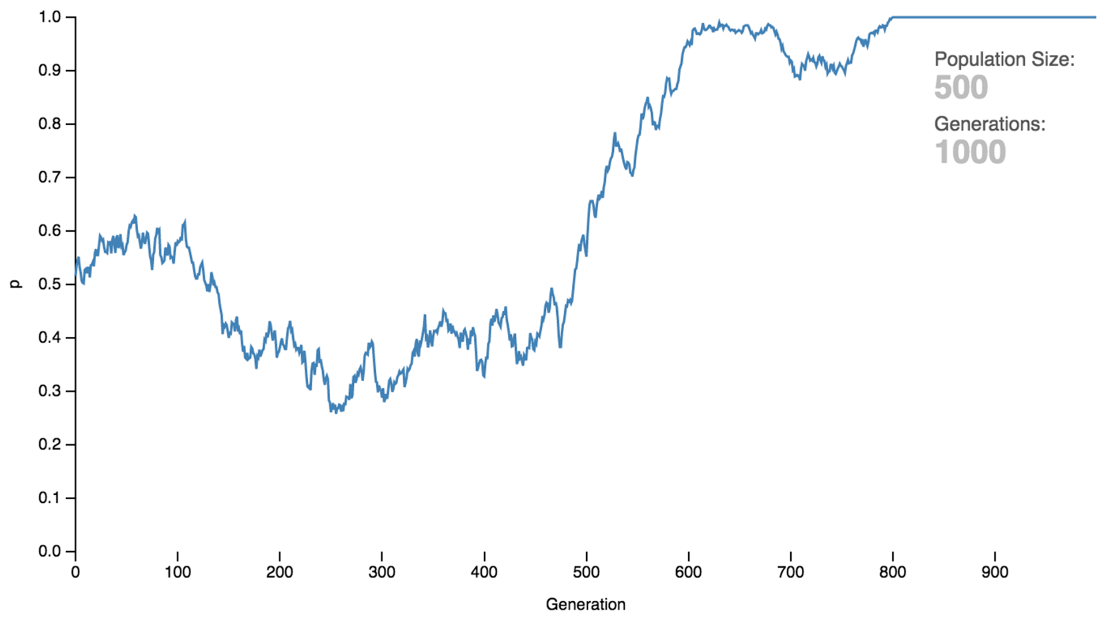
```
data.push(p)
```

which simply adds `p` to the `data` array. Next, I’ve removed the line that prints the values of `p` and `q` to the JavaScript console, but you can leave it in there if you want to. Finally, I’ve added this line:

```
draw_line_chart(data,"Generation","p",["Population Size:",N,"Generations:",gener\
ations]);
```

This is the line that calls the function `draw_line_chart()` to plot the data. Note that I’m calling the function at the end of the code - at that point, the array `data` is not empty anymore, but in fact contains all `p` values over the `1000` generations. It’s that set of values that we hand over to the function, and the function then plots the data visually.

Save the document and reload the browser. You should see a nice line chart showing the change of `p` over time:

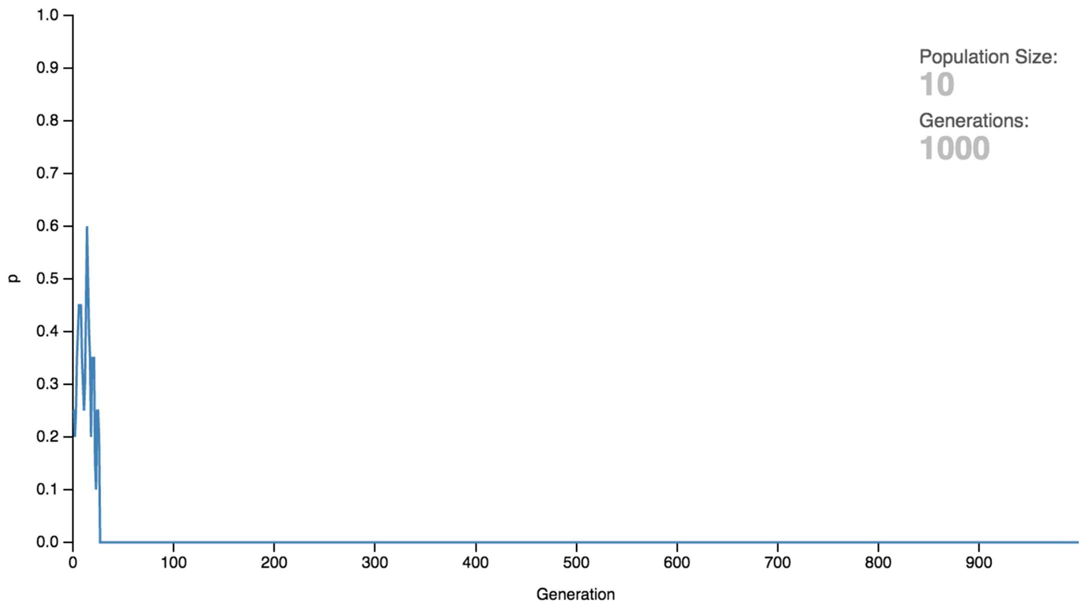


In this particular simulation run, the allele A1 went to fixation around generation 800, which is another way of saying that its frequency reached 100%. This also means that the allele A2 was lost at the same time (whenever an allele goes to fixation, all other alleles are lost). Reload the page and watch different evolutionary dynamics unravel. As you can observe, the dynamics are completely random, as we expected. Sometimes, one of the alleles goes relatively straight to fixation. Other times, the frequencies fluctuate widely, with one allele almost driven to extinction, only to make a dramatic comeback, and then still being lost 200 generations later.

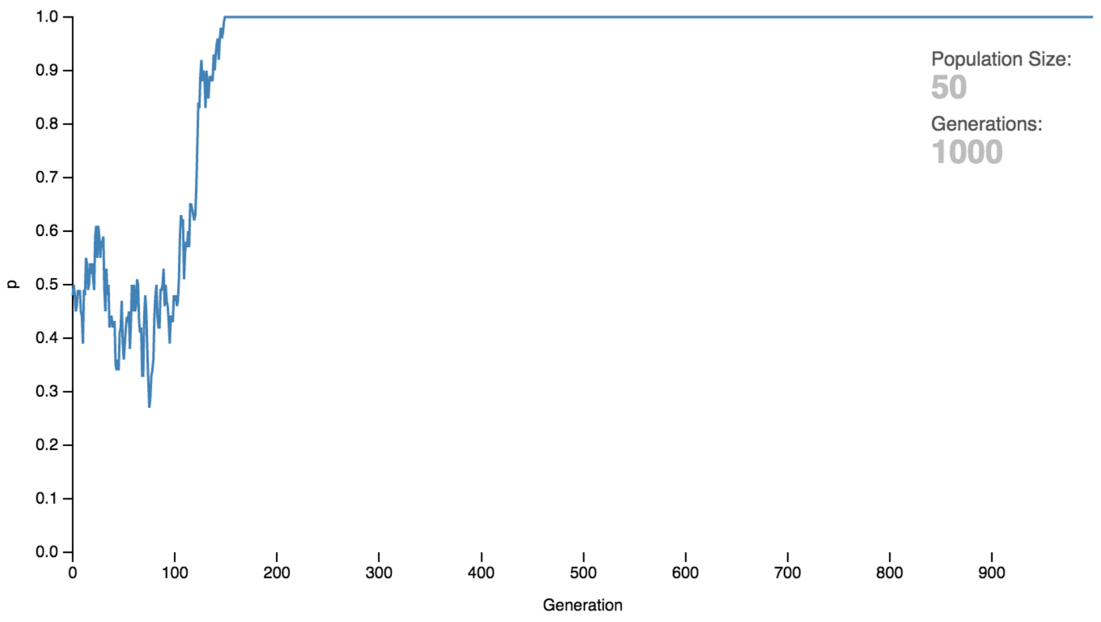
With this code in place, let's focus on the effect of the population size *N*. Change the population size to 10 by setting the corresponding variable to `10`:

```
var N = 10;
```

and reload the page. You should see something like this:



Again, yours will look different every time you reload the page, but you'll note that one of the alleles will go to fixation very rapidly, sometimes within just a few generations. Now increase the population size to 50, and you'll get a result like this:



With `N = 50`, the time to loss / fixation is a little longer, but it's still shorter than with `N = 500`. If you play around with different numbers, you'll notice the general pattern that we already observed earlier: that the random effects are much stronger when the population is smaller, and as a consequence, allele loss / fixation happens sooner.

Rather than reloading the page and seeing a single simulation run its course, I want to modify the code so that we can run a number of simulations and plot them all at once. In order to do this though, I first need to explain in more detail the concept of an array. I mentioned above that we are using a JavaScript array that we call `data`, and that we initialize it as an empty array, like so:

```
var data = [];
```

So what is an array? An array is simply a list-like object that contains any number of so-called *elements*. For example, this is an array with five elements:

```
var arr = [4,2,6,8,3];
```

The `data` array that we used above contained zero elements - it is an empty array. Arrays use brackets, i.e. `[` and `]`, to wrap the elements which are themselves separated by commas.

Why are arrays important? The variable types that we encountered so far - numbers, strings, and booleans - can only ever hold one single value, like `123`, `"hello there"`, or `true`. But whenever you are dealing with data, you have many values, and you need a way to store them somehow in code. That's where arrays come in. The array `arr` in the example above holds 5 values, which happen to be numbers. An array can store any other type, and even mix them, so you could have an array like this:

```
var b = [4,"hello there",6,true,false];
```

Now that you know what an array is, how to do you manage it? How do you add data to it, and how do you retrieve the data later on?

There are numerous ways by which you can add data to an array. One way is by initializing the array, as we've done with array `arr` above. Another way is to *push* data into an existing array, as we've done in the genetic drift code example above, like so:

```
arr.push(5);
```

This line adds the number five to the list of numbers in the array. Importantly, it will be added *at the end* of the list, so that the new array will contain the values 4, 3, 6, 8, 3 and 5, in that order.

You can retrieve elements by using the concept of an *index*. If you want the nth element of the array `arr`, you retrieve like so:

```
arr[n]
```

However, there is one major gotcha: like in almost any data structure in almost any programming language, JavaScript arrays are zero-indexed. This means that the first element has index 0, not index 1. If you are new to programming, this is going to be a major source of bugs, and probably hours of desperation. Sorry to be so blunt, but it is quite true. Perhaps this dramatic warning will at least remind you what to look for when your code behaves strangely.

Thus, in order to retrieve the first element in the array `arr`, you would use `arr[0]`; in order to retrieve the second element, you would use `arr[1]`; and so on.

Here's where it gets interesting: I mentioned above that you can store any type of variable in an array. What that means is that you can also store *other arrays*, allowing you to create *multi-dimensional arrays*. Imagine you want to store three sets of five numbers in an array. You could do it as follows:

```
var set1 = [1,2,3,4,5];
var set2 = [10,11,12,13,14];
var set3 = [33,44,55,66,77];

var data = [set1, set2, set3];
```

Alternatively, you could initialize the data array directly, like so:

```
var data = [[1,2,3,4,5], [10,11,12,13,14], [33,44,55,66,77]];
```

If you need to access the subsets, you would then do it as before: `data[0]` accesses the first element in the array, which is the array `[1,2,3,4,5]`.

If you want to access a specific element in one of the arrays, you would first have to access the corresponding array, and then access the element that you want from that array. For example, in order to get the number 33 from the `data` array, you would have to write `data[2][0]`.

Multidimensional arrays sound complicated, but they are very simple, as you've just learned. They are also very handy when you deal with more complex datasets. For example, when we want to store the dynamics of multiple simulation runs, rather than just one, we can simply use a multidimensional array. The main array would then contain all the arrays for each simulation run, and those in turn would contain the allele frequencies for each generation.

There's one more thing I want to mention about arrays. If you want to know how many elements an array currently holds, you can just use the following syntax:

```
data.length
```

This often comes in handy when you need to set up a loop to iterate over an array, and you need to know how many elements you have in the array.

Going back to our code, let's go ahead and change our simulation slightly to allow for multiple simulation runs.

Here is the new code:

```
var p;
var N = 20;
var generations = 200;
var data = [];
var simulations = 10;


function next_generation(simulation_data) {
    var draws = 2 * N;
    var a1 = 0;
    var a2 = 0;
    for (var i = 0; i < draws; i = i + 1) {
        if (Math.random() <= p) {
            a1 = a1 + 1;
        }
        else {
            a2 = a2 + 1;
        }
    }
    p = a1/draws;
    simulation_data.push(p);
}

function simulation(simulation_counter) {
    p = 0.5;
    for (var i = 0; i < generations; i = i + 1) {
        next_generation(data[simulation_counter]);
    }
}

for (var i = 0; i < simulations; i = i + 1) {
    data.push([]);
    simulation(i);
}

draw_line_chart(data,"Generation","p",["Population Size:",N,"Generations:",gener\
ations]);
```

Let's go through the changes here. First, I define the global variable `p`, but don't assign it a value any more. That's because every simulation should start again with a new value, so it'll be the responsibility of the `simulation()` function to set `p` to `0.5` every time a new simulation begins. I'm also introducing a new variable `simulations` that defines how many simulations we're going to run.

The function `next_generation()` has one subtle but important change. It now has an argument called `simulation_data`, and in the last line, the allele frequency `p` is added to `simulation_data`. (i.e. `simulation_data` is expected to be an array). I'll get back to that in a second.

The loop where we call the `next_generation()` function is now encapsulated in a function called `simulation()`. This function is also where we make sure to (re)set the value of `p` to `0.5` every time we start a new simulation. It also has one argument, `simulation_counter`, which we use as an index for the data array.

All of this should become clear on the next few lines, where we have a loop that iterates as many times as we have simulations. In each iteration, we add an empty array to the `data` array. This empty array is the array that will contain all the A1 allele frequency values for one simulation run. Then, we call the function `simulation()`, with the value `i` as an argument, which is our current counter for the simulations (starting at `0`). So when the function `simulation()` is called, `simulation_counter` will be set to whatever `i` was, and `data[simulation_counter]` thus corresponds to the array for a given simulation. It's that array that we pass as an argument to the function `next_generation()`.

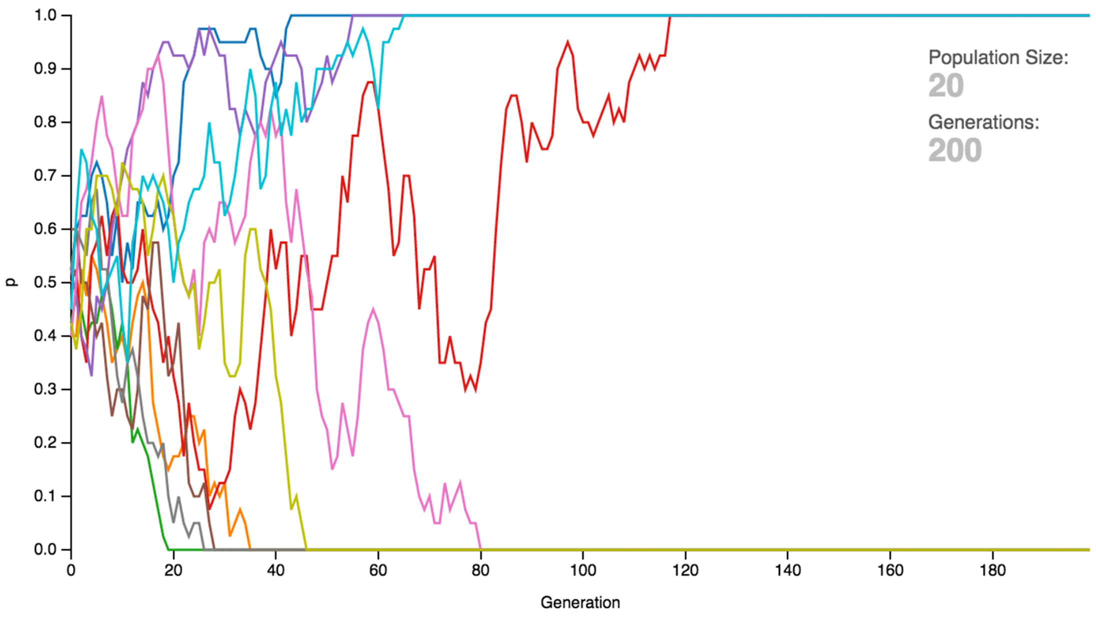
At the end, we call the `draw_line_chart()` function as before - but importantly, this time we will be passing it a multidimensional array (unlike in the example before, where we passed it a onedimensional array containing only the results of a single simulation run). Thankfully, `draw_line_chart()` is already prepared to handle two-dimensional data, so no need to change anything there!

Go ahead and save your HTML file, and then reload the page. You will see 10 simulation runs at the same time.

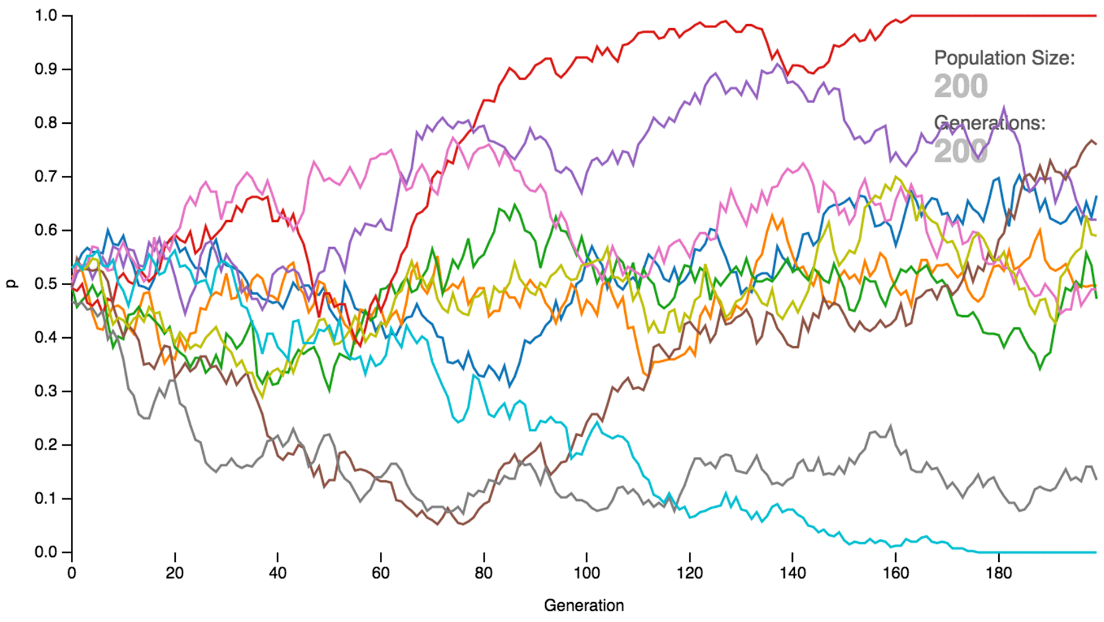
With that in place, let us run this code three times for 200 generations, with three different population sizes: `N = 20`, `N = 200`, and `N = 2000`.

Here is what these three sets of simulation will look like:

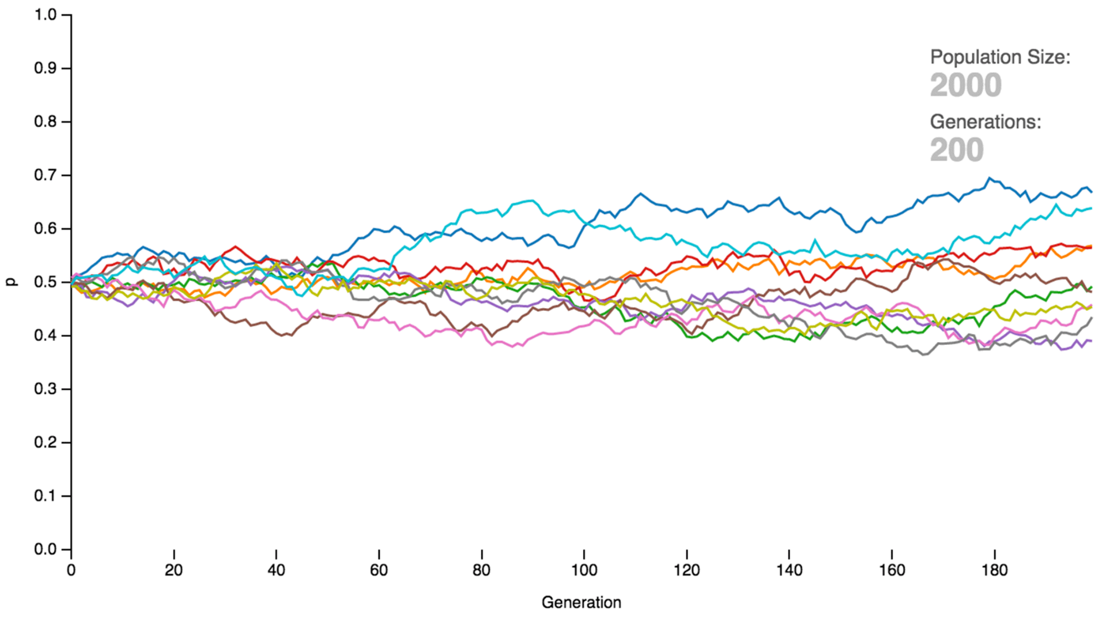
N=20:



N=200:



N=2000:



These figures are telling. They show that the smaller the population, the more pronounced the random evolutionary swings in each generation. Because of that, alleles will go to fixation in smaller populations much faster.

Now that we have a good intuition from the simulations, let's see if we can develop a mathematical model that captures this process. Don't worry, the math is going to be easy, and I will be explaining every step in great detail.

You've reached the end of this sample chapter (part of chapter 3 in the book). Please be sure to check out the website www.natureincode.com[4] where you can find all code examples that are developed in the book.

I hope you have enjoyed this sample chapter, and would be delighted if you decided to purchase the full book for $19.95 at the Leanpub website[5]. If you buy a Leanpub book you get all the updates to the book for free. All books are available in PDF, EPUB (for iPad) and MOBI (for Kindle), and there is no DRM on any of the formats.

[4]http://www.natureincode.com

[5]https://leanpub.com/natureincode/