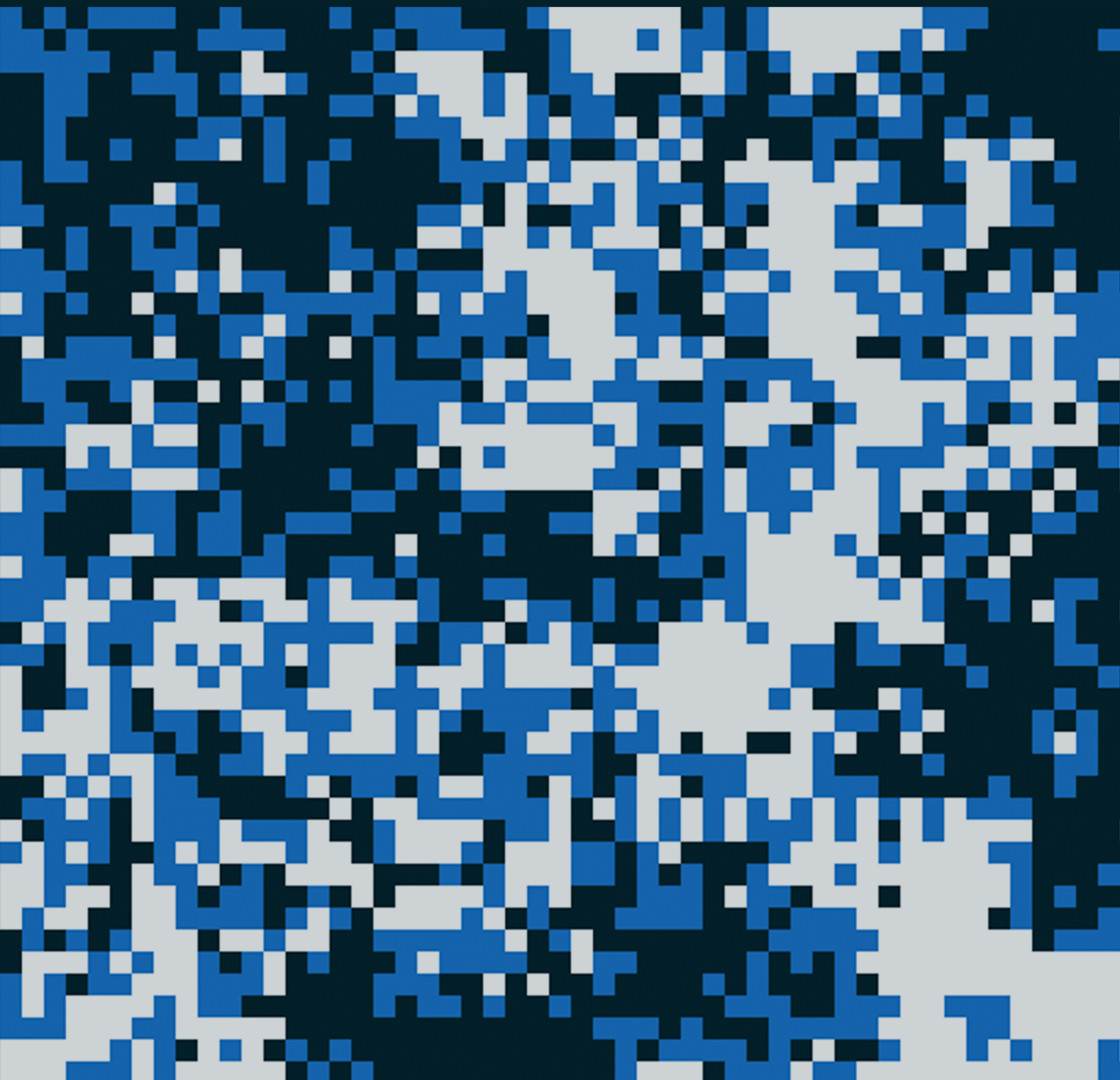
Nature, In Code

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 2016-11-13

To Rahel, Jonas, and Elina

Contents

1. Introduction

This is a book about biology. It is a also a book about programming. And it's probably unlike any book you've ever read.

I have two goals in mind for the reader of this book. The first goal is that after reading this book, you will know quite a bit more about nature than you did before, and perhaps in unexpected ways. If I succeed, you will understand some of the processes that shape our lives, the lives of everything that has ever lived, and of everything that will ever live on this planet.

The second goal is that you learn to use code to re-create these processes in your computer.

I assume very little about what you should know in order to read this book, but I still want to spell it out to avoid disappointments. You should be able to think critically. You should be able to think logically. You should be willing to be inspired. And you should have a computer. By that, I mean anything that allows you to write emails on. Your smartphone, amazingly, has the computational power to run the programs in this book without running a sweat. But typing on smartphones is usually a hassle, and I would recommend something with a bigger screen and a decent keyboard for best results. An internet connection will be helpful most of the times. I also assume that your computer has a modern browser.

This book covers quite a bit of material, not least because it's a book about two completely different topics (programming and biology). It is nevertheless a short book. It's short because it is dense, and correspondingly, you should give it your full attention, one page at a time. Don't rush it - this is not a book that you can read when you are already mentally exhausted from the day. I suggest you read it when you are at your most alert, take one page at a time, be sure you fully understand every single detail of what is said, and most importantly, **implement the code examples**. I cannot emphasize this enough. If you manage to implement all the code examples, you will have succeeded and understood the material, and you will have taken the most important step to learning programming.

Having learned programming myself - and continuing to do so - I am perfectly aware that things will not always go smoothly, and that there will be bumps in the road. To help you through these, all examples, with the full code, are available at the website www.natureincode.com[1]. Go there if you just can't make your code work, and be inspired by - but don't copy - a working solution.

I've also created an free online course based on this material, a so called MOOC (massive open online course). You can join the course at any time on the edX platform[2] - the course is self-paced, and you'll find lots of coding exercises that will help you assess your learning progress.

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

[2] https://www.edx.org/course/nature-code-biology-javascript-epflx-nic1-0x

To keep updated on anything related to the book, you can follow the Twitter account @natureincode[3]. If you feel adventurous, you can also follow my personal account @marcelsalathe[4] where I post about anything under the sun, including topics related to biology and programming.

All set up? Let's get started.

Acknowledgements

I'd like to thank everyone who has pointed out mistakes or passages that were unclear - they have ultimately made the book much better than what I could have done on my own. They are, in alphabetic order:

- Alessio Andronico
- Todd Bodnar
- Daryl Branford
- Michael Collins
- Boris Conforty
- Melissa Hicks
- Alex Mesoudi
- Sharada Mohanty
- Martin Mueller
- Rahel Salathe

If you find a mistake, or something that is unclear, please get in touch - I will be very glad to add your name here.

[3] https://twitter.com/natureincode
[4] https://twitter.com/marcelsalathe

2. The Genes, They Are a-Changin'

Life is change. It is also beautiful. The process responsible for these two outcomes is evolution.

Evolution is the grand unifying process underlying all of biology. At its core, evolution is a seemingly simple concept. It is defined as the change of allele frequencies in a population over time. Don't worry if that sentence doesn't makes sense to you right now - it will in a few minutes.

Living organisms pass on their genes from generation to generation. Physical bodies don't survive for long, but the genes that built them do. Genes are the hereditary material of all living organisms. But if all we would ever do is to pass on the same genes, generation after generation, there would be no evolution. For reasons we will talk about later, however, genes don't just come in one version. For every gene, there are multiple versions, and these versions of a gene are called *alleles*. For various reasons, alleles might be rare in a population, or might be common. They might be common at one time point, and become rarer over time, and vice versa.

A well-known example is lactose intolerance. Being lactose intolerant means not being able to fully digest lactose, a sugar that is found in milk and dairy products. If you are lactose intolerant, you don't produce enough lactase, the molecule responsible for digesting lactose. This is not a disease by any definition - indeed, roughly two thirds of the human population are lactose intolerant as adults. When a child is born, her main source of nutrition is milk, and like other mammals, she produces enough lactase to digest the lactose in the milk. After weaning, typically around 2 to 3 years of age, the child loses the ability to digest lactose and becomes lactose intolerant, a process which is usually complete around age 10.

Lactase is encoded by a single gene called LCT. This gene exists in many different versions, which means that there are many different alleles of the gene. One allele, LCT*P, allows lactase to persist after the weaning phase, making its bearer able to digest lactose. In some regions of the world, this allele is very common. In northern Europe for example, this allele is the most frequent allele of the LCT gene. In other regions of the world, it is practically absent.

Now here is the most interesting aspect of this story: there is strong evidence that the allele that causes lactase production to taper off after weaning is the original, ancestral allele. In other words, being lactose intolerant was common almost anywhere in the world until about 9,000 years ago, when humans started domesticating livestock and practicing milk-based pastoralism. At some point, a mutant version of the ancestral allele, LCT*P, must have arisen, and it has since then increased in frequency in the human population - from 0% to about 35% - until today.

This is evolution in action, precisely as we have defined it above: the change of allele frequencies in a population over time. What caused this change? There could be many different explanations, but all of them fall into four basic categories: natural selection, mutation, migration, and random drift. These are the four basic forces of evolution, and we will encounter them time and time again in this book. Of these forces, natural selection may be the most famous of them all. It's the only

force capable of producing adaptations. It's the one described by Charles Darwin, although he, like any scientist in the mid-19th century, had a very different - and wrong - understanding of genetic heredity than we have today, which makes his insights all the more stunning. But evolution can occur even in the absence of natural selection; as long as allele frequencies change, for whatever reason, that is evolution.

The Hardy Weinberg Principle

Consider these two conversations between two friends:

Friend 1: "The other day, on my way back home, I saw a car driving next to me!"
Friend 2: "And?"

Friend 1: "The other day, on my way back home, I saw a car flying right over me!"
Friend 2: "SAY WHAT???"

In science, just as in life, a phenomenon is interesting to the extent that it is unexpected. (Coincidentally, this is also the reason why life gets boring as you get older unless you continuously immerse yourself in new experiences, and learn new things.) Thus, for a scientific phenomenon to be called interesting, you first need to know what to expect so that you know when you see something unexpected.

In science, this idea is conceptualized in the *null model*. The null model states how a system behaves, given what you know about the system. This is an enormously important concept. Your brain is constantly scanning the world and comparing it to the null models you hold in your head. In fact, you do this subconsciously, which is why you would run away from a tiger in a supermarket without having to think about it: your observation ("there is a tiger staring at me from the dairy section") is very different from your null model ("there should be no animal, and certainly no tiger, in any section of a supermarket"), raising your mental red flags.

The null model in evolution is called *the Hardy-Weinberg principle*, named after two mathematicians G.H. Hardy and Wilhelm Weinberg. On its surface, it's quite simple - so simple in fact, that Hardy was a bit embarrassed for biologists that they hadn't figured this out themselves. In 1908, he wrote:

> "To the Editor of Science: I am reluctant to intrude in a discussion concerning matters of which I have no expert knowledge, and I should have expected the very simple point which I wish to make to have been familiar to biologists. However, some remarks of Mr. Udny Yule, to which Mr. R. C. Punnett has called my attention, suggest that it may still be worth making…"

(The modern equivalent, tweet-sized version would be: I can't believe you haven't figured this out yet, so here we go…. ht @udny_yule, via @rc_punnet)

The Hardy-Weinberg principle states that under certain assumptions - which we will get to in a moment - both the frequencies of alleles, as well as the frequencies of genotypes (which are simply

combinations of alleles), will remain constant from generation to generation. It then goes on to state what those frequencies are.

Before we dig into the details, let's remind ourselves why the Hardy-Weinberg principle is important. It's important because it is the evolutionary null model against which you can compare your observations. If your observations don't fit with what Hardy-Weinberg predicts, something interesting could be going on that warrants further investigation. It's for this very reason that practically every single evolutionary or ecological study compares their finding to what is expected given the Hardy-Weinberg principle.

The normal version of the Hardy-Weinberg principle assumes diploid organisms (most animals, including humans, are diploid). Being diploid means having two copies of each gene, one from the mother and one from the father. You, just like any human being, have about 21,000 genes. Every single cell of your roughly 1,000,000,000,000,000 cells contains two copies of each of these genes - the one set you inherited from your mother's egg, and the other set you inherited from your father's sperm cell that managed to inseminate the egg.

Now consider a single gene. If there is only one allele - i.e. only one version - of that gene in the population, nothing interesting can happen. You, just like everyone else, would have two copies of that allele, and so would your children, and their children, and so on. However, let's assume there are two alleles in the population, which we denote as A1 and A2. You could have inherited A1 from both parents, in which case you would have two copies of A1. In that case, we would say your genotype (for this particular gene) is A1/A1, and because both of your copies are identical, you'd be *homozygous* for the A1 allele. You could have inherited A2 from both parents, in which case your genotype is A2/A2, and you're homozygous for the A2 allele. Or, you could have inherited A1 from one parent, and A2 from the other, in which case your genotype is A1/A2, and you are *heterozygous*. What I just described is the simplest model of evolutionary genetics - the so-called *one locus, two alleles* model, where locus just stands in as a fancy term for gene (think of it as the address of the gene in the genome). Given that you have 21,000 genes, and most genes have a lot more than two alleles, things are usually a bit more complicated than this simple model would predict. However, you'll be surprised just how much one can deduce and learn from this simple one gene, two alleles model.

If you would randomly sample a few individuals from a population, and look at their genotype for the A gene, you could easily calculate allele and genotype frequencies. Say you sampled 100 individuals, and you found that 15 of those individuals have genotype A1/A1, 35 have genotype A2/A2, and the remaining 50 have genotype A1/A2, then you could deduce the following genotype frequencies:

A1/A1: 15% (15 out of 100 genotypes)
A1/A2: 50% (50 out of 100 genotypes)
A2/A2: 35% (35 out of 100 genotypes)

Because you know each genotype, you can also deduce the following allele frequencies (knowing that there are 200 alleles in 100 diploid individuals):

A1: 40% (80 out of 200 alleles - 30 from the 15 A1/A1 individuals which each have two A1 alleles,

and 50 from the 50 A1/A2 individuals which each have one A1 allele).

A2: 60% (120 out of 200 alleles - 70 from the 35 A2/A2 individuals which each have two A2 alleles, and 50 from the 50 A1/A2 individuals which each have one A2 allele).

Since you know there are only two alleles in this population - A1 and A2 - it would have been enough to calculate just the A1 frequency based on the genotypes: the A2 frequency must be 100% minus the A1 frequency (and vice versa).

To simplify matters, we'll assume that your sample perfectly captures the make up of the entire population. In fact, let's just not care about absolute numbers anymore, and simply assume an infinite population size, so that we can go on making our calculations with just the frequencies alone.

In fact, what I've just done is to establish one of the key assumptions of the Hardy-Weinberg principle - that of an infinite population size. We make this assumption not only because it makes the math easier, but also because we'll see later that finite population sizes can have quite dramatic effects on evolution.

At this point, you might shake your head and say "that doesn't make much sense - all populations are finite in size", and you would of course be absolutely right. But the purpose of the Hardy-Weinberg model - or any model, really - is not to perfectly capture reality. It is to capture a few essential features of reality into a simple model, in the hope that you can understand why the model behaves the way it behaves. From there, you can go on and add more realistic features one at a time, in order to understand their effects individually. The best analogy I can offer is that of a map. A good model is like a good map - *of course* it does not capture all the details, and *of course* it is a caricature of the real world, but that is the point. A good map captures the essence of your surroundings so that you can easily find your way around. A perfect map would be a 1:1 replica of the real world, and thus not perfect at all (in fact it would be completely useless, as is beautifully illustrated in Lewis Carroll's *Sylvie and Bruno Concluded*, which features a fictional map that had "the scale of a mile to the mile").

Let's continue our population study. In the Hardy-Weinberg world, what would the genetic makeup of the population look like in the next generation? To calculate this, we introduce a few more simplifying assumptions: first, the population has non-overlapping generations, which means that as one generation is born, the old generation dies. There is only ever one generation alive. Second, reproduction is sexual (i.e. no cloning), but there are no males or females, just hermaphrodites (individuals with both sexes). Third, mating is completely random, just like molecules, randomly bouncing into each other. Also, no other process (mutation, selection, etc.) is in place.

Arguably, this is very strange world these Hardy-Weinberg creatures live in - a world of infinite, random nothingness, other than the genotypes. But once again, this is a map-like model, where we reduce everything down to the necessary elements.

The next generation can now be calculated as follows: each of the infinite number of individuals will make infinitely many copies of their alleles, and since none makes less or more than the other, the frequencies of alleles will remain the same: the A1 allele will be just as frequent as before (40%).

The A2 allele will be just as frequent as before (60%). In short, no evolution will occur, by our simple definition of evolution (the change of allele frequencies in a population over time). But what will happen with the genotype frequencies? Since mating is random, you can imagine the mating process to be like picking marbles from a infinitely large jar of black and white marbles, with the colors corresponding to the alleles. This jar contains 40% black marbles, and 60% white marbles. To create a new offspring individual, you have to pick two marbles (because individuals are diploid). The chance that you pick two black marbles in a row is 0.4 * 0.4 = 0.16. Thus, 16 out of every 100 individuals will have the A1/A1 genotype. The chance that you pick two white marbles in a row is 0.6 * 0.6 = 0.36, which means 36 out of every 100 individuals will have the A2/A2 genotype. Because there is only one genotype left (A1/A2), you know that 100 - (16 + 36) = 48 out of every 100 individuals will have the A1/A2 genotype. But you could also have calculated this result with probabilities. A black and white marble combination can come about by either picking a black marble first and a white marble second (0.4 * 0.6) or the other way around, i.e. a white marble first and black marble second (0.6 * 0.4). Each of these probabilities is 0.24, and summed up they are 0.48, as expected.

We've just established that the genetic makeup of the next generation is going to be 16% of genotype A1/A1, 48% of genotype A1/A2, and 36% of genotype A2/A2. That's a bit different from our previous generation, which is interesting. Wonder what will happen in the next generation? Let's find out! (Spoiler alert: it's going to be short and interesting).

We said before that the allele frequencies haven't changed. But yet, we have slightly different frequencies for the genotypes. Is that possible? Let's double check the allele frequencies which we would calculate based on the new genotype frequencies. To make this more intuitive, let's again assume you collect a perfectly random and representative sample of 100 individuals:

A1: 40% (80 out of 200 alleles - 32 from the 16 A1/A1 individuals which each have two A1 alleles, and 48 from the 48 A1/A2 individuals which each have one A1 allele).

A2: 60% (120 out of 200 alleles - 72 from the 36 A2/A2 individuals which each have two A2 alleles, and 48 from the 48 A1/A2 individuals which each have one A2 allele).

Check. Ok, so let's go ahead and calculate the next generation. Let's use the jar analogy again. Again, the jar contains 40% black marbles, and 60% white marbles. Again, to create a new offspring individual, you have to pick two marbles (because individuals are diploid). Again, the chance that you pick two black marbles in a row is 0.4 * 0.4 = 0.16.

Wait a minute. Didn't we just do that before? Oh yeah, we did. We calculated genotype frequencies based on allele frequencies. And since the allele frequencies did not change, we will get the exact same numbers for the genotype frequencies as before. Not only for the next generation, and the generation after the next, and the generation after that, but - *forever*.

Whoa.

You just reached an equilibrium - the Hardy-Weinberg equilibrium. Here, both allele frequencies and genotype frequencies stay the same, forever. But everything went so fast! It just took *one generation* to get into this equilibrium. Is that normal? What if we had started out at the same allele frequencies (40% and 60%), but with different genotype frequencies? Say we had started at 20% A1/A1, 40%

A1/A2, and 40% A2/A2 genotype frequencies. Well, it does't matter - since the allele frequencies are the same (40% and 60%), the genotypes we will calculate will follow the same logic as described above with the marble jars. No matter where you start from, you will reach Hardy-Weinberg equilibrium in a single generation.

This is an important insight from a biological perspective. From a mathematical perspective, this was simple (or trivial, as mathematicians like to call everything), which is why Hardy was so high-nosed about the whole affair. The reason why it is important from a biological perspective is because of what it says: that in the absence of evolution (which means no change in allele frequencies), the genotype frequencies must be in Hardy-Weinberg equilibrium. Even if they aren't at present, they will be *in the next generation*, which is incredibly fast.

From these considerations follows an interesting insight, bringing us back full circle to our null model: if the genotype frequencies you find in a population are not at Hardy-Weinberg equilibrium, something interesting must be going on. Something must be different. Some of the assumptions that we have established above must have been violated.

What Does Boring Look Like?

We have just established that things are boring at Hardy-Weinberg equilibrium (maybe that was why biologists hadn't figured this out earlier - nothing in nature is ever boring, so why bother about this seemingly boring stuff?). Things are boring in the sense that genotype frequencies stay the same, forever. But why is that boring? It's boring because everything you see in the living world around you is an expression of a genotype. It would be a mistake to think that genes are all there is, and that everything is just genes, and nothing more. We know very well that the environment and other non-genetic factors play a big role in determining the development, shape, form, and behavior of an organism. But living organisms are ultimately built with the blueprint of a genotype, and different genotypes will generally result in different organisms. (The technical term here is phenotype: a phenotype is the organism that a genotype produces in its given environment.) Thus, the boundless diversity and beauty of nature, with its endless flow of forms, shapes and colors, can only come about if there is genetic diversity. In the Hardy-Weinberg world, nothing ever changes, everything stays predictably constant for eternity. If that doesn't qualify as boring, I don't know what would.

But boring has it's benefits too - it's predictable, and easy to quantify, which is what we will do next. Let's formalize the Hardy-Weinberg principle in mathematical terms, so that we can develop a short program that helps us understand what's going on.

We are going to denote the allele frequency of A1 as p, and the allele frequency of A2 as $q = 1-p$. The expected frequency of the genotypes is then as follows:

Genotype	Frequency
A1/A1	$p * p = p^2$
A1/A2	$p * q + p * q = 2pq$
A2/A2	$q * q = q^2$

Since these are all the genotypes there are, these three frequencies need to add up to 1. Let's verify that real quick by looking at the sum of these three terms:

$$p^2 + 2pq + q^2$$

Because *q = 1-p*, we can write

$$p^2 + 2p(1 - p) + (1 - p)^2$$

after distribution, we get

$$p^2 + 2p - 2p^2 + (1 - 2p + p^2)$$

which equals 1. QED.

Code

The title of this book is "Nature, in Code". We just described a natural process (however artificial), so let's simulate that process in code. Because this is the first time we do this, this section is a little longer than future code sections. If you don't understand everything completely on the first read, don't worry - the material is dense, and if you have never programmed before, many concepts will seem alien. In that case, I recommend re-reading this section now and again, until it truly clicks.

We are going to use JavaScript throughout this book. If you know anything about programming languages at all, you might be surprised at this choice. JavaScript is typically associated with dynamic webpages, which it was originally designed for, and it isn't very popular yet in scientific computing - but that doesn't really matter. What matters is that it's a modern, blazingly fast, object-oriented language, and it's easy to get started with, which is a substantial benefit. In addition, every browser runs JavaScript out of the box, and since every personal computer and every smartphone has a browser installed, JavaScript is ubiquitous. There are an estimated 2 billion personal computers in the world, and about 2 billion smartphones. In short, the code in this book - and JavaScript in general - runs on about 4 billion devices, today, without anyone needing to install anything. That's pretty neat.

If you think about getting serious with scientific computing in the future, you'll find it very easy to switch from JavaScript to other languages like Python, C++, or Java. On the other hand, if you later decide to get more serious about web or app development, JavaScript is the most important language to learn.

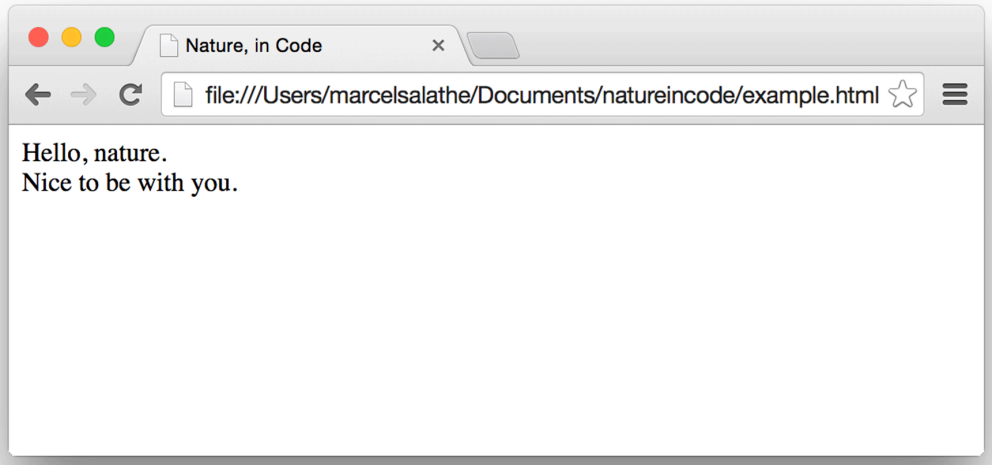
The first thing you need to create is a document that holds your code. JavaScript is typically embedded in an HTML document, and we'll follow that convention, although it isn't strictly necessary. An HTML document is simply a plaint text file with the file ending `.html` or `.htm`. So go ahead, and create a file called `example.html` in the text editor of your choice. You can use a plain

vanilla text editor, just be sure it is in plain text mode. If you are on a Mac, I highly recommend TextWrangler[5], which is a free, excellent text editor.

Then, enter the following HTML code:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Nature, in Code</title>
    </head>
    <body>
        Hello, nature.<br>Nice to be with you.
    </body>
</html>
```

The indentations aren't required, but they make the code easy to read. Save this document, and open it in the browser. You should see the following:



You've just created a simple HTML document, which you can use as a template for the rest of this book. HTML stands for Hypertext Markup Language, and can be interpreted by any browser. In fact, every webpage you've ever looked at (many, I'm sure) is written in HTML.

HTML is a set of so-called tags, like `<body>` and `<title>`. If you load an HTML document in a browser, the browser will read the HTML, and render a page according to the HTML specifications (if

[5] http://www.barebones.com/products/textwrangler/

you're curious, here is the latest official specification: http://www.w3.org/TR/html401/[6]). We won't be spending much time talking about HTML, but it's worthwhile understanding the basics. It is, after all, the markup language that the entire web is based on.

Let's go quickly through the HTML document above. It starts with a so-called DOCTYPE declaration: `<!DOCTYPE html>`. This tells the browser "Hey, I'm an HTML document". Truth be told, the browser could live without it, but it's good practice. (In fact, the browser is the most forgiving piece of software you'll ever encounter - it renders even the most broken HTML. Let's be thankful.). The first tag, the `<html>` tag, contains all the content of the document. Notice how it closes at the end of the document using `</html>`. This pattern - `<tag>content</tag>` - is very common in HTML, and the content can be other tags as well, which themselves can contain other tags, and so on.

The next tag in our example is the `<head>` tag. It opens with `<head>`, contains some content, and later closes with `</head>`. The head specifies a number of things that are not directly rendered on the screen, such as the title of the document, which itself is surrounded by the `<title>` tag. Further below, the `<body>` tag contains the content that should be rendered on the screen. In our case, we just want to render two lines of text. The line break is forced with the `<br>` tag, and because the `<br>` tag does not contain any content, no closing tag is necessary.

Now - how does one program in HTML? The simple answer is one doesn't: HTML is a markup language, not a programming language. The makers of the first commercially successful browser, Netscape, realized that the ability to have a programming language running in a browser could be quite useful - it would allow for dynamic content, which wasn't possible with pure HTML. They added JavaScript, and the rest is history. It's somewhat ironic that a language that was implemented in a few weeks, with the goal of making webpages a little interactive, is now becoming the most common programming language on the planet. (Note also that JavaScript has nothing to do with the programming language Java - the name was just a marketing play on the rising popularity of Java at the time).

In order to use JavaScript, you need to put your JavaScript code within a `<script>` tag, like so:

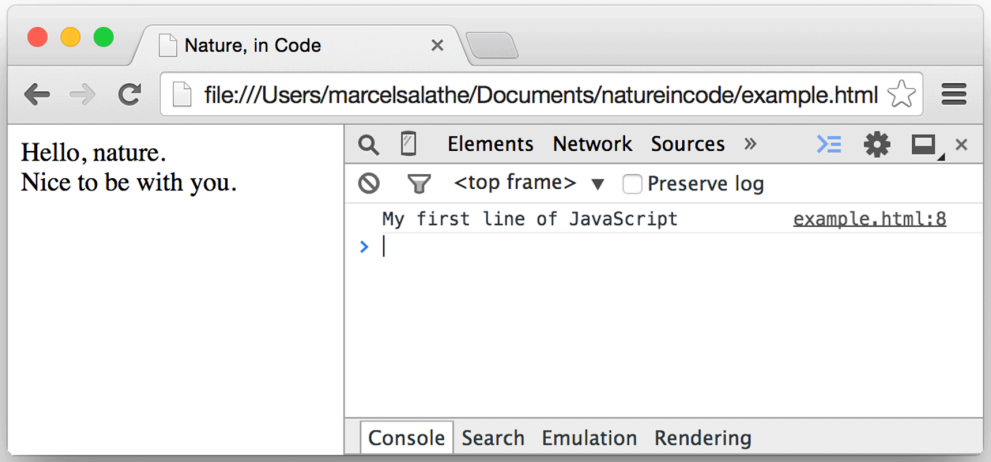
```
<script type="text/javascript">
    console.log("My first line of JavaScript");
</script>
```

It doesn't really matter where you add the script tag, but it's mostly found within the `<head>` tag. We'll talk about these lines of code in a second, but first, go ahead and put them in the `<head>` tag of your document, like so:

[6] http://www.w3.org/TR/html401/

```
<!DOCTYPE html>
<html>
    <head>
        <title>Nature, in Code</title>
        <script type="text/javascript">
            console.log("My first line of JavaScript");
        </script>
    </head>
    <body>
        Hello, nature.<br>Nice to be with you.
    </body>
</html>
```

If you save your document and reload it in the browser, you won't see any change. In order to see the effect of this single line of JavaScript, you need to open the JavaScript console in the browser. In Chrome, select View - Developer - JavaScript Console, and you will see the following (in Firefox, select Tools - Web Developer - Web Console, and in Safari, select Develop - Show Error Console):



Nice! So what just happened?

We added a `<script>` tag, which is the tag that contains our JavaScript code. Before we get to that, take a close look at the opening tag, which reads

```
<script type="text/javascript">
```

What we've done here is to add a so-called *attribute* to the tag (`type` in this case) with a value of `"text/javascript"`. Generally, HTML tags can have many different attributes, and in case you've glanced over the HTML documentation above (don't worry if you haven't), you've noticed that attributes are a major part of the HTML specifications. You could almost go as far as saying that HTML without the attributes is practically useless, since the effect and behavior of most tags depends completely on their attributes and their values. In the present example, the `type` attribute specifies the type of scripting language. In principle, browsers could be able to interpret multiple scripting languages, but currently JavaScript is the only one that's supported by all modern browsers. If you wouldn't add the `type` attribute to the `script` tag, everything would still work - that's because `"text/javascript"` is the default value for the `type` attribute.

Let's move on to the actual JavaScript code, which reads

```
console.log("My first line of JavaScript");
```

This simple line already contains a lot of essential features of any programming language, such as objects, methods, and parameters. If none of these things make any sense to you right now, that's absolutely fine - we'll be getting there soon. In any case, what this line does is to print the text `My first line of JavaScript` in the console. The console is simply a somewhat hidden JavaScript output in the browser that is very helpful for debugging (i.e. fixing errors or just generally trying to find out what's going on with the code).

I'd like to add a second line to our code as follows:

```
console.log("My first line of JavaScript");
// this is a comment - it won't be executed
```

It's sometimes important to leave a clarifying comment in the code - either for yourself, or for someone else who might work on the code later on. In JavaScript, everything after a double slash (`//`) on a line is interpreted as a comment, rather than as executable code.

So how would we implement the Hardy-Weinberg principle in code? Let's start with declaring three variables that will store the genotype frequencies, and initialize them with values that we used in the example above:

```
var a1a1 = 0.15;
var a2a2 = 0.35;
var a1a2 = 1 - (a1a1 + a2a2);
```

Let's walk through this line by line. On the first line, we define a variable `a1a1`, and assign it the value `0.15`. Variables are declared using the `var` keyword (keywords are simply words predefined by the language, and since these words have very specific meanings, you shouldn't use them for anything other than what they were intended for). Your variable name can be anything you like, but it needs

to start with a letter or an underscore, and be followed by other letters, numbers, or underscore. You're not allowed to use any of these keywords for variable names: `break`, `case`, `class`, `catch`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `export`, `extends`, `finally`, `for`, `function`, `if`, `import`, `in`, `instanceof`, `let`, `new`, `return`, `super`, `switch`, `this`, `throw`, `try`, `typeof`, `var`, `void`, `while`, `with`, `yield`.

The assignment itself is done using the `=` operator. Finally, the line ends with a semicolon, which terminates the statement. You would be forgiven for not using semicolons, and your code would still execute correctly, but it is dangerous practice that might introduce unexpected errors, so always use them. Line two does the same thing, except we declare another variable, `a2a2`, and assign it the value `0.35`. As you surely have guessed, these two variables denote the genotype frequencies of the two homozygotes A1A1 and A2A2 (I'll use this shortcut notation from now on, instead of writing A1/A1 and A2/A2). The genotype frequency for the heterozygotes is stored in the variable `a1a2`, and rather than assigning its value directly, we assign it by calculating it, using the previously defined variables `a1a1` and `a2a2`. These three lines of code shouldn't look all that different from regular algebra.

Having just set up the genotype frequencies, let's calculate the allele frequencies:

```
var p = a1a1 + (a1a2 / 2);
var q = 1 - p;
```

The variable `p` denotes the frequency of A1 alleles, which is simply the frequency of the A1A1 genotype (since they only carry A1 alleles) plus half of the frequency of the A1A2 genotype (since half of their alleles are A1). We then calculate `q` by simply subtracting `p` from `1`; `p` and `q` need to add up to one because there are only two alleles in the population.

I should highlight two subtleties in the first line of code. The first is that I am using parentheses to group the equation. JavaScript uses the standard algebraic rules and would first divide `a1a2` by `2` before adding the result to `a1a1`, even without the parentheses - just as algebraic rules dictate. However, I prefer to make these rules explicit with parentheses. Even more than the semicolons, explicit parentheses are just good style that makes your code clearer and prevents subtle errors in the future, when your code becomes more complex (which it most certainly will). The second subtlety is that I divided `a1a2` by `2`, rather than by `2.0`. If you have any experience with other programming languages, that might raise a red flag. That's because most programming languages have two numeric types, integer and floating point numbers, but JavaScript doesn't (and that's a good thing). All numbers are floating point numbers. `2` is equivalent to `2.0`, which avoids many numerical errors that plague other languages.

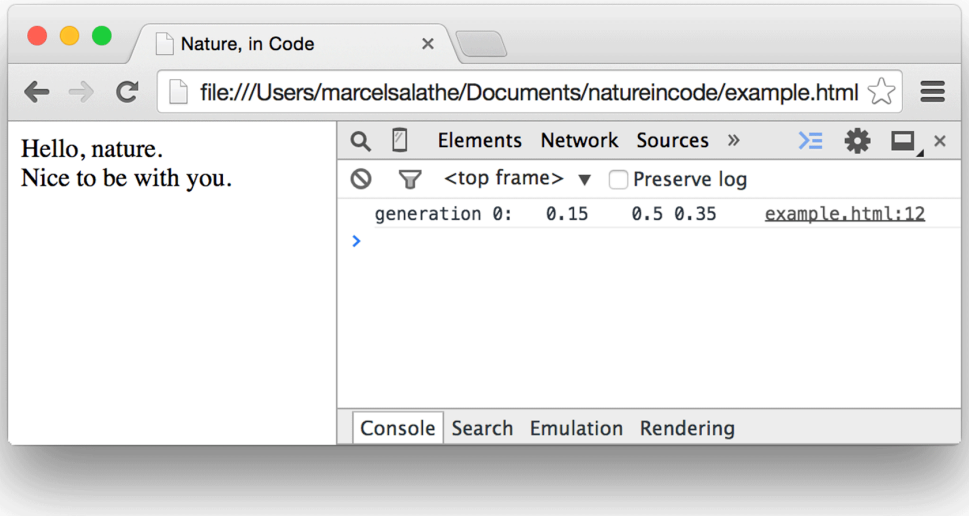
Combine these five lines and copy them into your `script` tag in your HTML file, like so:

```
<script type="text/javascript">
    var a1a1 = 0.15;
    var a2a2 = 0.35;
    var a1a2 = 1 - (a1a1 + a2a2);

    var p = a1a1 + (a1a2 / 2);
    var q = 1 - p;
</script>
```

Save the file, reload the page (be sure to have your console open), and you should see... nothing. That's because we are not writing anything into the output yet! Your code is executed all right, but since we didn't instruct JavaScript to write anything into the output, it won't do it. So go ahead and add the following line:

```
console.log("generation 0:\t" + a1a1 + "\t" + a1a2 + "\t" + a2a2);
```

Using `console.log` as before, we're writing text into our output. We'll talk about this line in a second, but first, make sure that it works and save the file and reload the browser:



Excellent, worked like a charm. The simple line above wrote our three genotype frequencies at generation 0 into the output. This line can teach us a few new ideas about JavaScript. The first thing to recognize is that we are calling a method, or a function (the two are not technically equivalent, but for almost all practical purposes, they can be treated as equivalent). A method is invoked, or called, by using its name and by supplying a number of arguments in parentheses, separated by commas. A method can take any number of arguments, including none, in which case your parentheses would be empty. So any of these examples are valid JavaScript:

```
some_method();
some_method(argument1);
some_method(argument1, argument2);
some_method(argument1, argument2, argument3);
```

Somewhere, a method must be defined before you can call it. JavaScript comes with a whole range of predefined methods (like `console.log`) that you will be able to use, and we'll learn quite a few throughout this book. Oftentimes, however, you define your own methods. In any case, by its definition, the method is usually quite clear about how many arguments it normally expects.

Calling Functions

The code within a function definition is not executed until you call the function. As an example, this code:

```
function say_hi() {
   console.log("Hi");
}
```

wouldn't do anything, whereas this code:

```
function say_hi() {
   console.log("Hi");
}
// now calling the function:
say_hi();
```

would print "Hi" into the console.

Above, we used

```
console.log("My first line of JavaScript");
```

We were passing one argument to the method, namely `"My first line of JavaScript"`. In JavaScript, anything wrapped in single or double quotes is a so-called *string* which is simply a sequence of textual characters. This is an important tool that allows us to differentiate text from numbers and variables. For example, `a1a1` (without any quote) refers to the variable `a1a1`, while `"a1a1"` is simply text. Similarly, `"100"` is simply text, and has no numerical meaning whatsoever, while `100` represents the actual number one hundred (remember why the number `100` isn't at risk to be confused with the variable `100`? That's right, because there is no variable `100` - all variable names must begin with a letter).

In line with the example above, let's try to pass one string as an argument to `console.log`. Let's say we'd like to output

```
generation 0:        0.15        0.5        0.35
```

(where the values are separated by tabs).

The problem is that some of this information is stored in variables (such as `a1a1`), rather than as text. String *concatenation* to the rescue! In JavaScript, you can add string together using the + operator, like so:

```
"hello " + "world"
```

which would result in the string "`hello world`" (don't forget the space character). But what happens when you this?:

```
"hello " + a1a1
```

What happens is that you get the string "`hello 0.15`". That's because JavaScript coerces numeric values to string values if one the operands is a string. This is actually a pretty handy feature that let's you do things like

```
"generation 0:\t" + a1a1 + "\t" + a1a2 + "\t" + a2a2
```

Which creates the string "`generation 0: 0.15 0.5 0.35`" (note that the string "`\t`" is interpreted as the tabular character).

I wanted to show this example because we'll be using string concatenation quite a bit throughout the book. However, it turns out that the method `console.log` takes any number of arguments, and will simply output their string representations, separated by spaces (rather than tabs), so let's just replace our `console.log` line with

```
console.log("generation 0:", a1a1, a1a2, a2a2);
```

and we'll get the same quantitative output, but with more concise code.

Now, let's move on to the next generation. Let's calculate the next generation, using the allele frequencies. They are easy to calculate:

```
a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;
```

This follows the logic described a few pages above, and it terms of JavaScript, there is nothing new here, with the exception perhaps that we are now reusing previously defined variables, and overwriting their values. We can now go ahead and output the new genotype frequencies using our line above (slightly modified to take into account that we're at a new generation):

```
console.log("generation 1:", a1a1, a1a2, a2a2);
```

Your full code should now look like this (for the sake of brevity I will omit the `<script>` tag from now on):

```
var a1a1 = 0.15;
var a2a2 = 0.35;
var a1a2 = 1 - (a1a1 + a2a2);

var p = a1a1 + (a1a2 / 2);
var q = 1 - p;

console.log("generation 0:", a1a1, a1a2, a2a2);

a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;

console.log("generation 1:", a1a1, a1a2, a2a2);
```

and the output is

```
generation 0: 0.15 0.5 0.35
generation 1: 0.16000000000000003 0.48 0.36
>
```

Everything looking great, so let's move on and... wait, what is this? `0.16000000000000003`? This should just be `0.16` - what's going on here? Well, you have just entered the world of floating point rounding errors. This is not specific to JavaScript - all languages know this phenomenon, and it's a consequence of the binary nature of computers. Fundamentally, anything in a computer is stored in bits, represented by 0 and 1, or off and on. Floating numbers are represented by 64 bits in JavaScript, which means there is a limited count of numbers that can be stored, and they are all stored in binary format. As a consequence, some numbers need to be rounded at a (hopefully) insignificant place. Think about it this way: what is 1/3? It's 0.33333333333333... and so on. The decimal system cannot perfectly capture 1/3, and we're usually fine to round it to 0.33 or 0.333. But then, even though we know that 1/3 + 1/3 + 1/3 = 1, if we added 0.333 + 0.333 + 0.333, we would get 0.999, which isn't the same as 1. The same logic applies when it comes to computers - some numbers cannot be captured perfectly in the binary system, and some rounding is necessary. As a consequence, when we perform calculations with these rounded numbers, we may get some rounding errors. Because JavaScript uses 64 bits to store numbers, the rounding usually occurs at an insignificant place, as it does in `0.16000000000000003`. However, you need to keep the issue in mind because it can introduce subtle errors. Most of the time though, it's just a matter of aesthetics, which is easily solvable - we'll get to it in a minute.

Back to our genotypes - we already know what will happen in the next generation, but just to be sure, let's code it up. Once again, we will calculate the genotype frequencies using the following code:

```
a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;

console.log("generation 2:", a1a1, a1a2, a2a2);
```

Go ahead and copy the lines from generation 1 and paste it four times, changing only the generation count. Your code will then look like this:

```
var a1a1 = 0.15;
var a2a2 = 0.35;
var a1a2 = 1 - (a1a1 + a2a2);

var p = a1a1 + (a1a2 / 2);
var q = 1 - p;

console.log("generation 0:", a1a1, a1a2, a2a2);

a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;

console.log("generation 1:", a1a1, a1a2, a2a2);

a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;

console.log("generation 2:", a1a1, a1a2, a2a2);

a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;

console.log("generation 3:", a1a1, a1a2, a2a2);

a1a1 = p * p;
a1a2 = 2 * p * q;
```

```
a2a2 = q * q;

console.log("generation 4:", a1a1, a1a2, a2a2);

a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;

console.log("generation 5:", a1a1, a1a2, a2a2);
```

and your output will be:

```
generation 0: 0.15 0.5 0.35
generation 1: 0.16000000000000003 0.48 0.36
generation 2: 0.16000000000000003 0.48 0.36
generation 3: 0.16000000000000003 0.48 0.36
generation 4: 0.16000000000000003 0.48 0.36
generation 5: 0.16000000000000003 0.48 0.36
> |
```

As expected, your genotype frequencies won't change - under Hardy-Weinberg conditions, they are predetermined by allele frequencies which don't change.

What we have just done, however, is considered very bad practice: copying and pasting code. This isn't an issue of copyright - it is your code, after all. The issue is that you repeat yourself, and the exact same pieces of code are repeated time and time again. This is bad practice, because once you decide that something about this code needs to change, you will need to change it in all the places where you pasted the code, and it's almost guaranteed that you will forget some places in the code, leading to nasty errors. Your code should follow the DRY principle: Don't Repeat Yourself. Thankfully, JavaScript, like all other programming languages, provides a simple but powerful tool that helps you solve this problem: *functions*.

We have already encountered functions, and have talked a little bit about their syntax. But now it's our turn to write our own functions. In JavaScript, a function is defined as follows:

```
function name(parameters) {
    body
}
```

The indentation and the line breaks are optional, but it's good practice and I highly recommend that you use this style. It makes the code more readable and, like so often, avoids subtle yet nasty errors.

The keyword `function` is predefined by the JavaScript language. The name of the function is up to you - function names follow the same rules as variable names, and it's best to name the function according to what it does. The *parameters* are zero, one, or many variables for which values can be

passed to the function when it is invoked (and as we've seen before, we will refer to these values as *arguments*). Thus, a function is defined using parameters, and when it is invoked, it is given arguments. The body is the code that is executed when the function is invoked. Once that code is done executing, your program will continue executing the code where it was before it hit that function:

```
// some code will be executed here before the function
some_function(); // jump to the function and execute the code in the function
// some code will be executed here after the function
```

Functions, as their name suggests, encapsulate functionality. Almost all of your code will be inside of functions. Ultimately, programming is nothing more than manipulating data, and functions are the places where the manipulations happen. They are the bread and butter of programming.

In our code, we copy-pasted the code

```
a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;
```

many times, so we should probably encapsulate this code in a function. What does this code do? It creates the next generation using allele frequencies, so let's call it `create_next_generation()`:

```
function create_next_generation() {
    a1a1 = p * p;
    a1a2 = 2 * p * q;
    a2a2 = q * q;
}
```

This function doesn't take any parameters at the moment. As you can see, its body makes use of variables that were defined somewhere else. We'll talk about this in a moment. You can also see that I used a fairly long name for the function name. The benefit is that anyone who reads your code, including your future self, can instantly guess what the function does from its name alone. There are two widespread styles to name functions that contain multiple words. You can either separate the words by underscore, like I prefer to do, or you can use the so-called camelCase notation, where you don't use underscores, but rather capitalize each new word like so: `createNextGeneration`. It's up to you which style you choose. JavaScript programmers have historically often used camelCase, but I simply find the underscore notation more readable, and will stick to it.

Having encapsulated this functionality, we can now rewrite the code. We add the function, and then simply replace the five occurrences of

```
a1a1 = p * p;
a1a2 = 2 * p * q;
a2a2 = q * q;
```

with calls to the function:

```
create_next_generation();
```

which will lead to:

```
var a1a1 = 0.15;
var a2a2 = 0.35;
var a1a2 = 1 - (a1a1 + a2a2);

var p = a1a1 + (a1a2 / 2);
var q = 1 - p;

function create_next_generation() {
    a1a1 = p * p;
    a1a2 = 2 * p * q;
    a2a2 = q * q;
}

console.log("generation 0:", a1a1, a1a2, a2a2);

create_next_generation();
console.log("generation 1:", a1a1, a1a2, a2a2);

create_next_generation();
console.log("generation 2:", a1a1, a1a2, a2a2);

create_next_generation();
console.log("generation 3:", a1a1, a1a2, a2a2);

create_next_generation();
console.log("generation 4:", a1a1, a1a2, a2a2);

create_next_generation();
console.log("generation 5:", a1a1, a1a2, a2a2);
```

Much better, but not quite perfect yet. What if we would want to do this a hundred times, rather than just five times? Time for yet another concept (I did warn you this was going to be a dense

chapter - but we're almost done): *iterations*. Iterations can be used to repeat code a certain number of times, or until certain conditions are met. The most common iterator is the `for` loop. Its formal syntax is as follows:

```
for (initialization; condition; final_expression) {
    code to be iterated
}
```

The `initialization` is executed once, in the beginning. Then, as long as the `condition` is met, the `code to be iterated` is executed. After each iteration, the `final_expression` is executed. In practice, the `for` loop often looks like this:

```
for (var i = 0; i < 10; i = i + 1) {
    code to be iterated
}
```

You'll see this construct over and over again, and after a while, it will become second nature. You will simply look at the `for` loop, and in a split second will realize that this code will be executed exactly ten times. Why is that?

The initialization is

```
var i = 0
```

Ok, nothing unusual here. Just think of `i` as a counter. Then, while the condition

```
i < 10
```

is true, the code to be iterated will be executed. How often is that? Well, if you wouldn't touch `i` at all again, you would be stuck in an infinite loop - `i` would always be `0`, and hence the condition would always be true. (If you ever visit the original Apple campus in Silicon Valley, note the street address: 1 Infinite Loop). However, after every code iteration, the final-expression is executed, which is

```
i = i + 1
```

What this means is that `i` is incremented by `1`, and thus, after the tenth iteration, `i` is `10`, at which point the condition `i < 10` is not true anymore, and the loop will stop.

Back to our code - taking a close look, we can see that the code

```
create_next_generation();
console.log("generation 1:", a1a1, a1a2, a2a2);
```

is repeated 5 times. The only change in each iteration that we need to take care of is the generation count. So let's first set up a `for` loop that executes this code 5 times, like so:

```
for (var i = 0; i < 5; i = i + 1) {
    create_next_generation();
    console.log("generation 1:", a1a1, a1a2, a2a2);
}
```

This works, but will always print `generation 1`, which is not what we want. Thankfully, we have access to the counter `i` in the loop, and using string concatenation, we can write the iteration as follows:

```
for (var i = 0; i < 5; i = i + 1) {
    create_next_generation();
    console.log("generation "+(i+1)+":", a1a1, a1a2, a2a2);
}
```

Our full code now reads:

```
var a1a1 = 0.15;
var a2a2 = 0.35;
var a1a2 = 1 - (a1a1 + a2a2);

var p = a1a1 + (a1a2 / 2);
var q = 1 - p;

function create_next_generation() {
    a1a1 = p * p;
    a1a2 = 2 * p * q;
    a2a2 = q * q;
}

console.log("generation 0:", a1a1, a1a2, a2a2);

for (var i = 0; i < 5; i = i + 1) {
    create_next_generation();
    console.log("generation "+(i+1)+":", a1a1, a1a2, a2a2);
}
```

Wow, look at you! Fancy code you've got there! Using variables, algebra, functions, string concatenation and iterations to show that Hardy and Weinberg were correct. And here, almost at the end of chapter 1, I want to share a deep truth about science and programming: The deepest understanding of a scientific principle comes from putting it in code. When you express a scientific idea in code, you have to be crystal clear about your assumptions, and how the idea should work exactly. You can't fool yourself or others with code, and there will never be any misunderstandings. Code provides the ultimate clarity.

You may now say, hang on a second. All you are showing here is that if your allele frequencies don't change, of course your genotype frequencies won't change either. But who says that allele frequencies won't change? Excellent point, I would answer. Let's modify our code to take this into account. Let us indeed calculate the allele frequencies too, from genotype frequencies, rather than assuming that they are fixed. I'm going to create a function `calculate_allele_frequencies()` as follows:

```
function calculate_allele_frequencies() {
    p = a1a1 + (a1a2 / 2);
    q = 1 - p;
}
```

And then I'm going to call that function right before generating the genotype frequencies of the next generation:

```
function create_next_generation() {
    calculate_allele_frequencies();
    a1a1 = p * p;
    a1a2 = 2 * p * q;
    a2a2 = q * q;
}
```

(yes, you can call functions from within other functions). Our code is now a little longer, but clearer about our assumptions. As you will see in your console, the results will still be the same.

While everything works as it should, the output is still a bit ugly. Let's clean up these rounding errors, and round the numbers to two decimal places. As I mentioned above, JavaScript in a browser comes with a whole package of functionalities encapsulated in methods. One such method is `Math.round(x)` which returns the value of `x` rounded to the nearest integer. That's not what we want, though - we want to round the number to two decimals of precision. Unfortunately, there is no built-in function that takes a number, and the number of decimals of precision you'd want it to return, so we will have to write the function ourselves:

```
function round_number(value) {
    return Math.round(value *100) / 100;
}
```

This function takes one argument, `value`, and returns the number rounded to two decimal places. Before explaining how it works, we need to talk about a new concept: that of a return value. You've certainly noticed the `return` keyword in the function. What does it do? The keyword `return` has two effects: first, it does indeed "return" the value that immediately follows it - in this case, `Math.round(value *100) / 100`. But to whom does it return it? To whomever called the function in the first place. This will make more sense in a moment. Second, `return` terminates the function. Most of the time, you will find the return statement at the end of the function - but sometimes, you'll find it much earlier. Whenever and wherever your function hits a `return` statement, it will return the value following the `return` statement, and terminate the function's execution, jumping back to where the function was called. A `return` statement isn't always necessary (you might have nothing to return) - indeed, as you can see, this is the first time I've used it in this chapter.

Going back to our `round_number(value)` function, how does this work? It's a simple idea. Let's say you have a number, like `3.4567`, which you want to round to two decimals. First, multiply it by hundred, shifting the decimal point to the right by two: `345.67`. Now, round the number using the built-in `Math.round` method: `346`. Great, you just got rid of the decimals you did not care about. Now, shift the decimal point back to the left by two, which you can do by dividing by `100` again: `3.46`. Done! Now, we could stop here and say our function does what it is supposed to do, namely to round a number to `2` decimals. But what if we want to round it to `3` decimals, or to `4`, later on, either in this or another project? Let's go ahead and make the function more flexible, by adding a second parameter that I'm going to name `decimals`:

```
function round_number(value, decimals) {
    var shifter = Math.pow(10,decimals);
    return Math.round(value * shifter) / shifter;
}
```

What I'm doing here is simply to replace `100`, which was the number to shift the decimal point two positions to the right and then to the left (because `10^2 = 100`), with the more generic version `10^decimals`, using the built in method `Math.pow`. The method `Math.pow(base, exponent)` calculates the `base` to the `exponent`th power. Thus, if you pass in `3` as an argument for decimals, the shifter will be `1000`, shifting the decimal point three positions to the right and then three to the left.

There is always a danger to make your method too generic, and to end up with super flexible functions that take a lot of arguments. Don't be tempted - whenever you think you'll keep reusing a function, but with slightly different values, making a function more generic is a great idea. But generality often makes your code needlessly complicated, and you'll need to find a good balance. A general rule of thumb is that when you start copying parts of functions around, you should think about making the function more generic and reusable.

Now, all we need to do is to call this function whenever we want to print a rounded number, which happens in our `for` loop:

```
console.log("generation "+(i+1)+":", round_number(a1a1,2), round_number(a1a2,2),\
 round_number(a2a2,2));
```

Our final code of this chapter looks as follows:

```
var a1a1 = 0.15;
var a2a2 = 0.35;
var a1a2 = 1 - (a1a1 + a2a2);

var p = a1a1 + (a1a2 / 2);
var q = 1 - p;

function calculate_allele_frequencies() {
    p = a1a1 + (a1a2 / 2);
    q = 1 - p;
}

function create_next_generation() {
    calculate_allele_frequencies();
    a1a1 = p * p;
    a1a2 = 2 * p * q;
    a2a2 = q * q;
}

console.log("generation 0:", a1a1, a1a2, a2a2);

for (var i = 0; i < 5; i = i + 1) {
    create_next_generation();
    console.log("generation "+(i+1)+":", round_number(a1a1,2), round_number(a1a2\
,2), round_number(a2a2,2));
}

function round_number(value, decimals) {
    var shifter = Math.pow(10,decimals);
    return Math.round(value * shifter) / shifter;
}
```

leading to the following result:

```
generation 0: 0.15 0.5 0.35
generation 1: 0.16 0.48 0.36
generation 2: 0.16 0.48 0.36
generation 3: 0.16 0.48 0.36
generation 4: 0.16 0.48 0.36
generation 5: 0.16 0.48 0.36
>
```

which is exactly what we wanted. (For the record, we could have used the `round_number(value, decimals)` function in the first `console.log` statement too - I didn't bother because it was simply printing the values I had defined above in the code).

I want to introduce one more concept before ending this chapter: that of a scope. If you take a look at the `calculate_allele_frequencies()` and `create_next_generation()` functions, you notice that we use the variables that store the genotype frequencies (`a1a1`, `a1a2`, and `a2a2`) and the allele frequencies (`p` and `q`). The reason why we can do this is that the variables have been defined in the global scope - we can access them globally, i.e. from anywhere in our program. Any variable that is defined outside a function is in the global scope. This is very convenient - so convenient, that you probably haven't even noticed it. It just feels natural. However, be warned that as programs get more complex, the global scope is a dangerous concept. Some key variables could be overwritten from anywhere in the code. Since the programs in this book are going to be relatively short, this won't be a big issue, but I want you to be aware of the danger.

As you have probably guessed, there is also a local scope. Variables that are defined with the keyword `var` inside a function are in the local scope of that function. You can use them in the function, but nowhere else. For example, the variable `shifter` in the `round_number(value, decimals)` function is a local variable. You can't use it anywhere else, other than in that particular function. However, here's the danger zone: Variables that are defined *without* the keyword `var` inside a function are still in the global scope!

Consider this short code snippet:

```
function test1() {
    foo = 4;
}

function test2() {
    console.log(foo);
}

test1();
test2();
```

What happens when you run this code? Because the variable `foo` has been defined without the

keyword `var`, it is in the global scope, which means that you have access to it from the function `test2()`. The program will output `4`.

If you add the `var` keyword when you declare `foo`, like so:

```
function test1() {
  var foo = 4;
}

function test2() {
  console.log(foo);
}

test1();
test2();
```

then `foo` will be in the local scope of the function in which it is declared (i.e. `test1()`), and upon trying to access it anywhere outside that function - e.g. in `test2()` - you would see this error:

What does all of this mean for you in practice? It means you should always define your variable with keyword `var`. Variables declared outside functions are then in the global scope, and variables defined inside functions are in the local scope of that function. Follow this simple rule, and you'll likely never have to think about scope problems again.

And with that, the chapter is finished. If you feel like your head is spinning a little, don't worry. I suggest you take a break, and come back and read the chapter again. Both the biological concepts and the code concepts introduced in this chapter are of fundamental importance throughout the rest of the book, so be sure to understand them before moving on.

What you've learned in this chapter:

- The definition of evolution
- The concept of alleles and genotypes
- The Hardy-Weinberg principle
- How to write simple JavaScript programs using variable, functions, and iterations.

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[7]). 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:

[7] 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[8]. 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[9]) 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.

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

[9] 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:

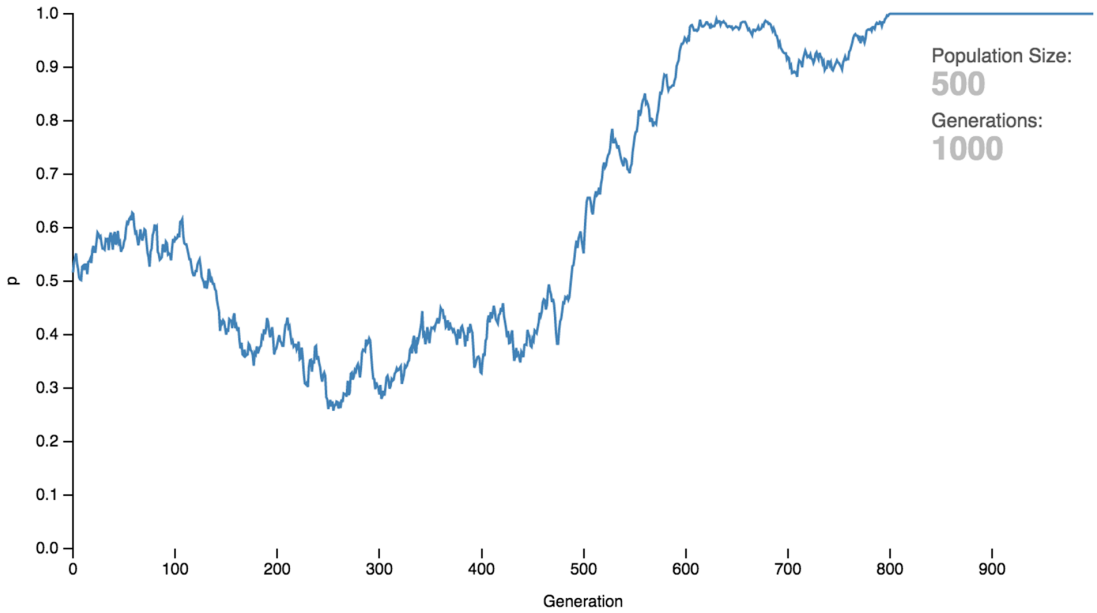
```
data.push(p)
```

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:

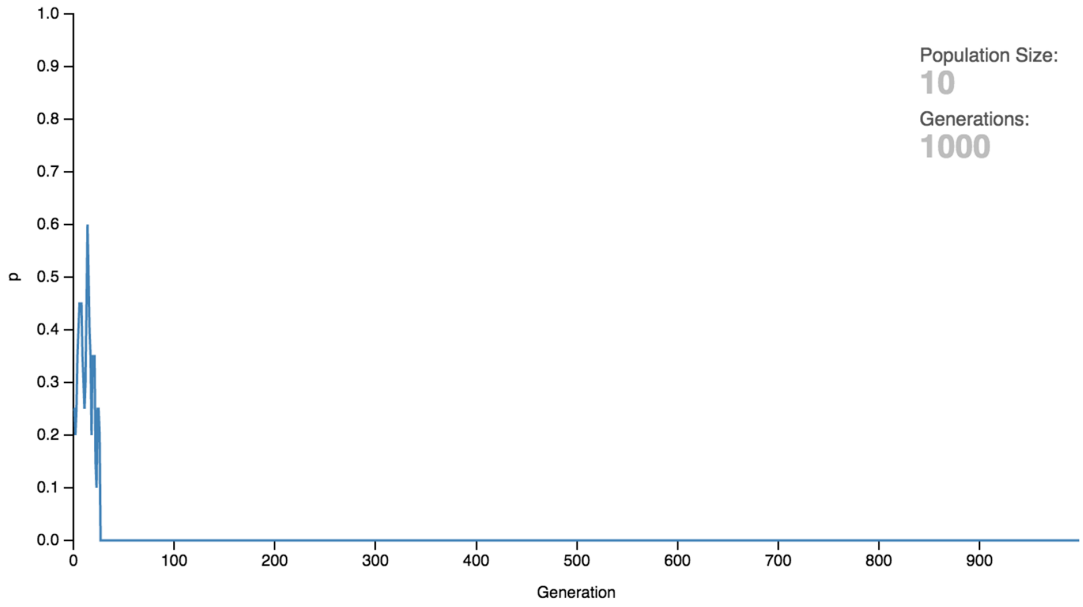


In this particular simulation run, the allele A1 went to fixation around generation 800, which is another way of saying that its frequency reached 100%. This also means that the allele A2 was lost at the same time (whenever an allele goes to fixation, all other alleles are lost). Reload the page and 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.

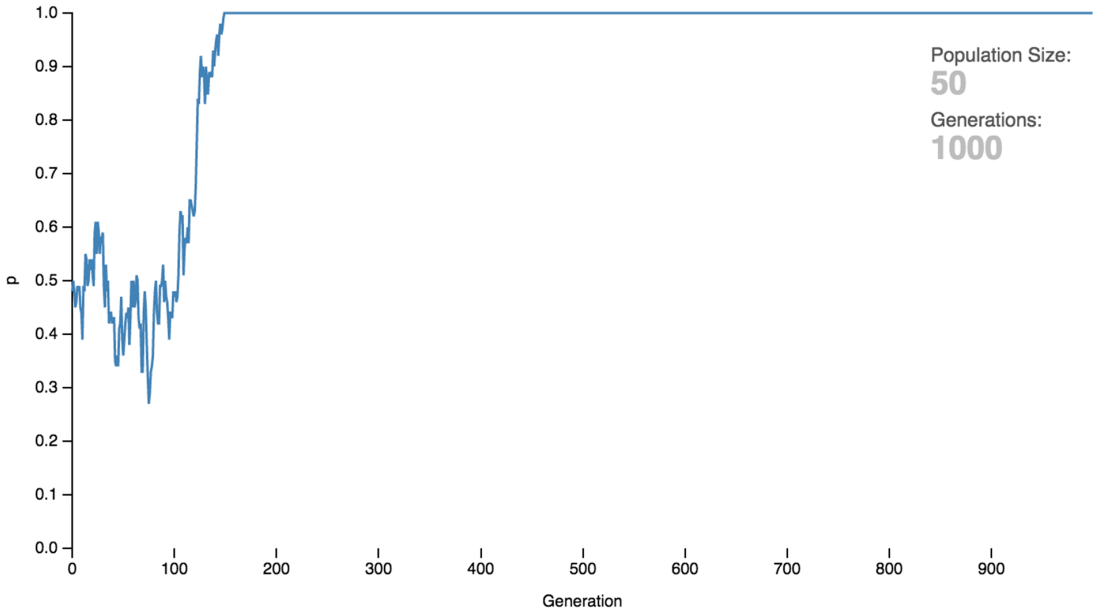
With this code in place, let's focus on the effect of the population size *N*. Change the population size to 10 by setting the corresponding variable to `10`:

```
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 n^{th} 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 and 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 one-

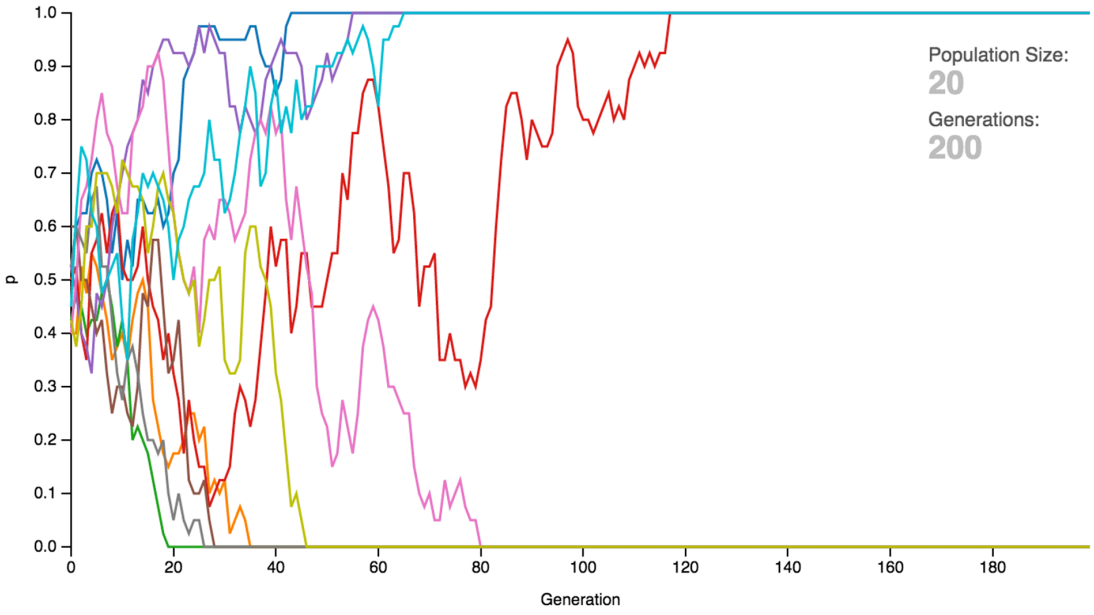
dimensional array containing only the results of a single simulation run). Thankfully, `draw_line_chart()` is already prepared to handle two-dimensional data, so no need to change anything there!

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

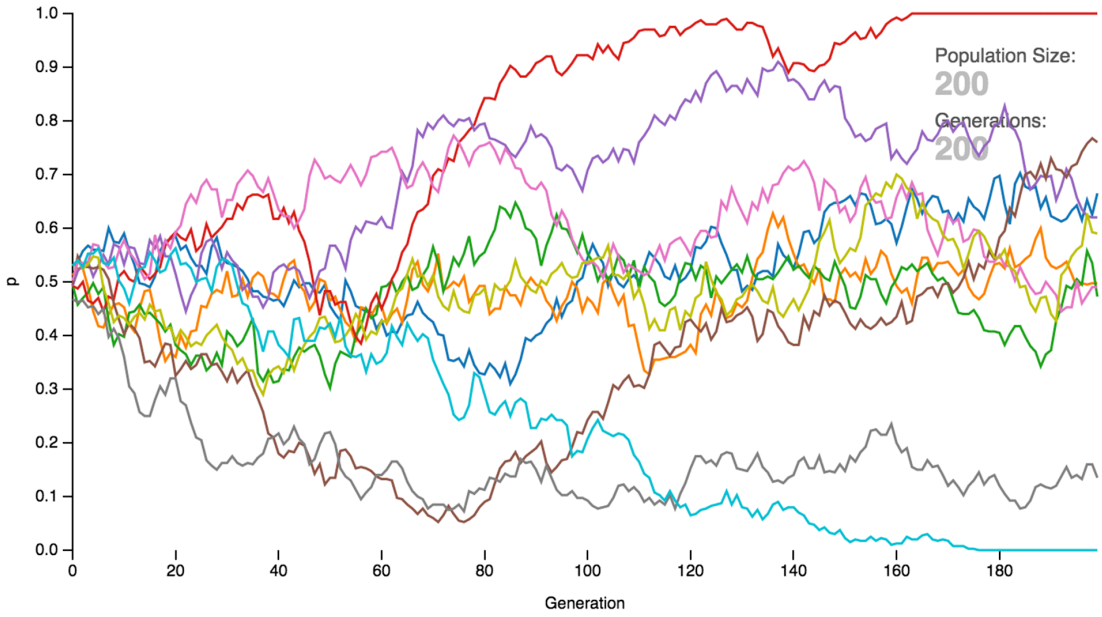
With that in place, let us run this code three times for 200 generations, with three different population sizes: `N = 20`, `N = 200`, and `N = 2000`.

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

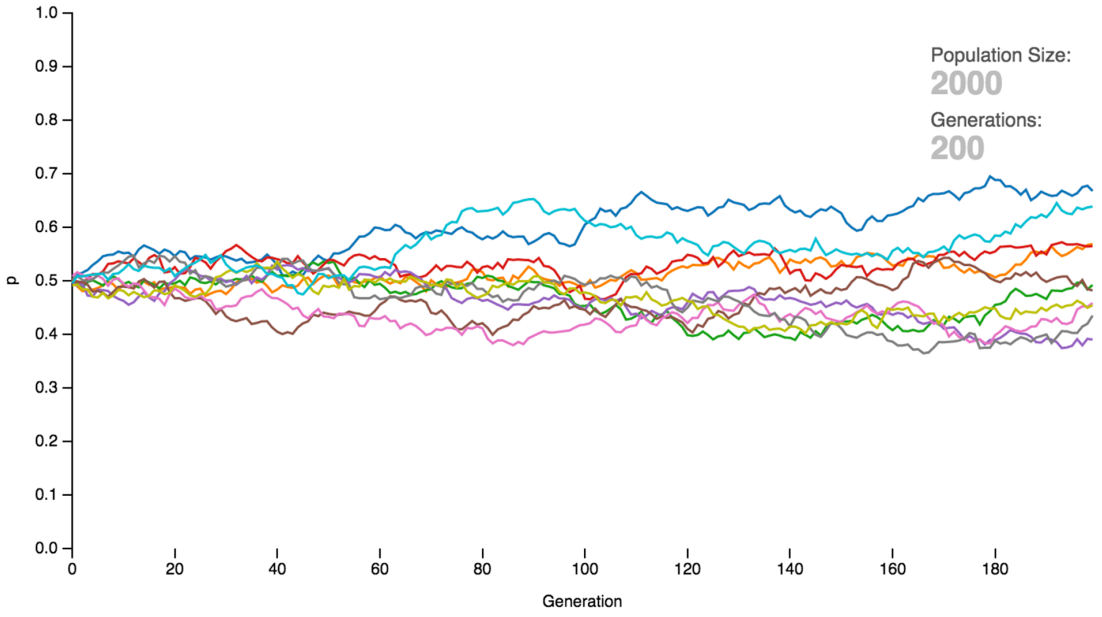
N=20:



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.

A Mathematical Formulation of Genetic Drift

We have just established, through stochastic simulations, that the main effect of genetic drift is the reduction of genetic variance. We are now looking to find a way to capture this process mathematically.

Let's go back to our marble jar example. Recall that we had all the allele copies of the population in a jar, and in order to generate the next generation, we would pick a random marble, copy it, and put it back into the jar. We would do this *2N* times in total.

Let's define *G* as the probability that two alleles picked randomly from the population of *2N* alleles are identical by state (in our example, that would mean they are both A1s, or both A2s). Thus *G* is the probability that if you pick any two marbles from the jar, at the same time, they would both be A1s, or both be A2s.

We are now in a position to ask ourselves the following question: given that the current generation has a certain value of *G*, what is *G* going to be in the next generation? We're going to denote *G* in the next generation simply as *G'*, to avoid confusion with *G* in this generation.

There are two ways by which two randomly picked marbles are identical by state. It could be that they had the same ancestor marble that was simply picked twice. The probability of this is $\frac{1}{2N}$. Why? Because once you pick a certain marble from a jar out of *2N* marbles, and put it back again, the possibility that you pick that exact same marble again next time is $\frac{1}{2N}$ (since there are *2N* marbles you can pick from).

The second way by which two randomly picked marbles are identical by state is that they did *not* have the same ancestor marble, but their ancestors just happened to be the same type (i.e. either both ancestors were A1 or both were A2). The probability of that is $(1 - \frac{1}{2N})G$, and here's why. First, the probability that two marbles did not have the same ancestor is $1 - \frac{1}{2N}$. We know this because we have just established above that the probability that they do have the same ancestor is $\frac{1}{2N}$, so the probability that they *don't* have the same ancestor is 1 minus that, i.e. $1 - \frac{1}{2N}$. But that's not the end of the story - the two alleles also need to be identical in state (i.e. of the same type), which is *G* by definition. The total probability, then, is the product of these two probabilities, i.e. $(1 - \frac{1}{2N})G$.

Having established the two (mutually exclusive) ways by which two randomly picked marbles can be identical, considering what *G* is in this generation, we simply need to add them up to get the total probability, which is

$$\frac{1}{2N} + (1 - \frac{1}{2N})G$$

which corresponds to *G'* in the next generation.

What we have just done is the following: we have started from the probability that two randomly picked alleles in this generation are the same (*G*), and then established that in the next generation,

G' will be $\frac{1}{2N} + (1 - \frac{1}{2N})G$. This is the most important part of the story - everything that follows now is just algebraic reformulation.

Let's start by defining H as *H = 1 - G*. Thus, H is the probability that two alleles picked randomly from the population of *2N* alleles are different (because *G* was the probability that they are the same). This is a good measure for genetic variance. If you have high genetic variation, then the chance that two randomly picked alleles are different is high. Conversely, if you have low genetic variation, then the chance that two randomly picked alleles are different is low. In the worst case, you have no genetic variation (*H = 0*) - all alleles are the same type, and the chance that two randomly picked alleles are different is zero.

Because *H = 1 - G*, the same must also be true in the next generation, i.e.

$$H' = 1 - G'$$

Since we know that

$$G' = \frac{1}{2N} + (1 - \frac{1}{2N})G$$

we can replace this term in the equation above and write

$$H' = 1 - (\frac{1}{2N} + (1 - \frac{1}{2N})G)$$

Because *H = 1 - G*, it's also true that *G = 1 - H*, and we can thus replace the *G* in the previous equation as well, and write

$$H' = 1 - (\frac{1}{2N} + (1 - \frac{1}{2N})(1 - H))$$

If we expand the terms in the equation, we get

$$H' = 1 - (\frac{1}{2N} + 1 - H - \frac{1}{2N} + \frac{H}{2N})$$

removing the parentheses yields

$$H' = 1 - \frac{1}{2N} - 1 + H + \frac{1}{2N} - \frac{H}{2N}$$

which makes various terms fall out of the equation, leaving only

$$H' = H - \frac{H}{2N}$$

Let's factor out H:

$$H' = (1 - \frac{1}{2N})H$$

which is an equation that looks unassuming - but is indeed one of the most profound equations of evolutionary biology, immediately revealing two major aspects about genetic drift. First, it tells us that no matter what H is in this generation, in the next generation it is going to be smaller. That's because $(1 - \frac{1}{2N})$ will always be smaller than 1, and in order to get H', we need to multiply H by something that is smaller than 1, which means that $H' < H$. Boom - right there, we've shown mathematically that the effect of genetic drift is to reduce genetic variation. But this formula is not done yet! It reveals another truth, which is that the smaller the population size N, the stronger the reduction in genetic variation per generation. Why? Well, we just said that in order to get H', we need to multiply H by something that is smaller than 1 - but how much smaller? As you can easily see, the term $(1 - \frac{1}{2N})$ will be smaller if N is small. And it will be closer to 1 if N is large.

Two major insights based on a simple equation that we were able to derive from simple principles, with simple algebra. Not too shabby! But wait, there's more!

Let's denote the genetic variance and generation 0 with H_0, and thus

$$H_1 = (1 - \frac{1}{2N})H_0$$

For the next generation, we have

$$H_2 = (1 - \frac{1}{2N})H_1$$

Since we know the formula for H_1, we can just replace it and write

$$H_2 = (1 - \frac{1}{2N})(1 - \frac{1}{2N})H_0$$

Similarly, in generation 3, we'll have

$$H_3 = (1 - \frac{1}{2N})H_2$$

Again, since we know the formula for H_2, we can just replace it and write

$$H_3 = (1 - \frac{1}{2N})(1 - \frac{1}{2N})(1 - \frac{1}{2N})H_0$$

and so on. For every generation, you simply have to multiply by $(1 - \frac{1}{2N})$ again, leading to the more generic form

$$H_t = (1 - \frac{1}{2N})^t H_0$$

What this means is that if *t* is large enough - in other words, over sufficiently long time - the term $(1 - \frac{1}{2N})^t$ will approach 0, which means genetic variation will approach 0, and all alleles but one will be lost. This is another significant insight from just this one formula. Unless other forces counter it, genetic drift will reduce all genetic variation in every population.

And as if those three insights were not enough, let's develop a fourth insight. We know that drift reduces genetic variation; we know that the effect of drift is strongest in smaller populations; and we know that the ultimate outcome of drift would be the fixation of one allele, and hence a total destruction of genetic variance, unless other processes counter it. One question remains open, though - how long does all of this take?

As with any process of decay, the best way to address this problem is by asking how long it takes to reduce the current genetic variance by half. You may have come across this concept as that of half-life - i.e. the amount of time required for a quantity to fall to half of its initial value - in physics or chemistry.

We can turn this question into a mathematical equation as follows:

$$\frac{H_0}{2} = H_t$$

which is to say that we want to know at which point the current genetic variation, H_t, is exactly half of that of the one we had initially, H_0. Because we have established above the equation for H_t, we can simply replace it and write

$$\frac{H_0}{2} = (1 - \frac{1}{2N})^t H_0$$

Now all we have to do is solve this equation for *t*. First, divide by H_0 on both sides:

$$\frac{1}{2} = (1 - \frac{1}{2N})^t$$

Now, using the natural logarithm, we can get *t* down from the exponent:

$$ln(\frac{1}{2}) = ln(1 - \frac{1}{2N})t$$

That's as far as we would get with regular algebra - but we can use an approximation to simplify this equation even more. The approximation is the following:

$$ln(1 + x) \approx x$$

and it works only when *x* is small. So in our case, we can say that

$$ln(1 - \frac{1}{2N}) \approx \frac{-1}{2N}$$

which simplifies our equation above to

$$ln(\frac{1}{2}) \approx \frac{-1}{2N}t$$

Let's multiply both sides by *-2N*, and we'll get

$$-2Nln(\frac{1}{2}) \approx t$$

You may remember that $ln(\frac{1}{x}) = -ln(x)$. We'll make use of this and replace $ln(\frac{1}{2})$ with *-ln(2)*:

$$-2N(-ln(2)) \approx t$$

which simplifies to

$$t \approx 2Nln(2)$$

Since *ln(2)* is roughly 0.7, the equation finally is

$$t \approx 1.4N$$

That's a pretty simple result. What it says is that the time it takes to halve the genetic variance in a population by drift is roughly *1.4N* generations. So for a population of size 100, it takes about 140 generations, for a population of size 10,000, it takes about 14,000 generations, and so on. As expected, drift reduces genetic variation faster in small populations. We already established this in the equation above. The new insight we gained here is how long this process takes, and the answer is, a rather long time. Recall from the previous chapter how fast genotype frequencies go into Hardy-Weinberg equilibrium: it just takes a single generation. In contrast, drift takes thousands of generations to just half the genetic variance in even relatively small populations. Drift, it seems, is a rather slow process.

Let's briefly summarize what this simple mathematical exercise has revealed:

- Genetic drift acts to reduce genetic variation.
- The effect of drift is proportional to the population size: the smaller the population, the larger the effect of drift.
- Unless other forces counter it, genetic drift will reduce all genetic variation in every population.

- Drift is a slow process - it takes about $1.4N$ generation to reduce the genetic variation of a population by half.

These are fundamental insights into the biology of populations. They are also in agreement with our stochastic simulations.

I hope to have shown you that both approaches - that of the computational simulation, and that of the mathematical analysis - are valuable and correct approaches to address a problem. They're certainly the most powerful when they're combined, but it's important to note that each has its unique strengths.

Effective Population Size

In the remainder of this chapter, I want to introduce an important concept that should always be on your radar when talking about drift: that of effective population size. We have until now assumed a finite population size of N. We have also assumed that the population size is constant over time (i.e. N in every generation). Both the simulations and the mathematical equations were based on this assumption. In reality, the size of a population can fluctuate over generations, sometimes strongly so.

Another assumption we've made so far is that we did not differentiate between sexes, and simply assumed that individuals are *hermaphroditic*, which is to say that they are both male and female at the same time. That assumption is violated in many species, although hermaphroditism is widespread among plants, and certain groups of animals such as snails and slugs.

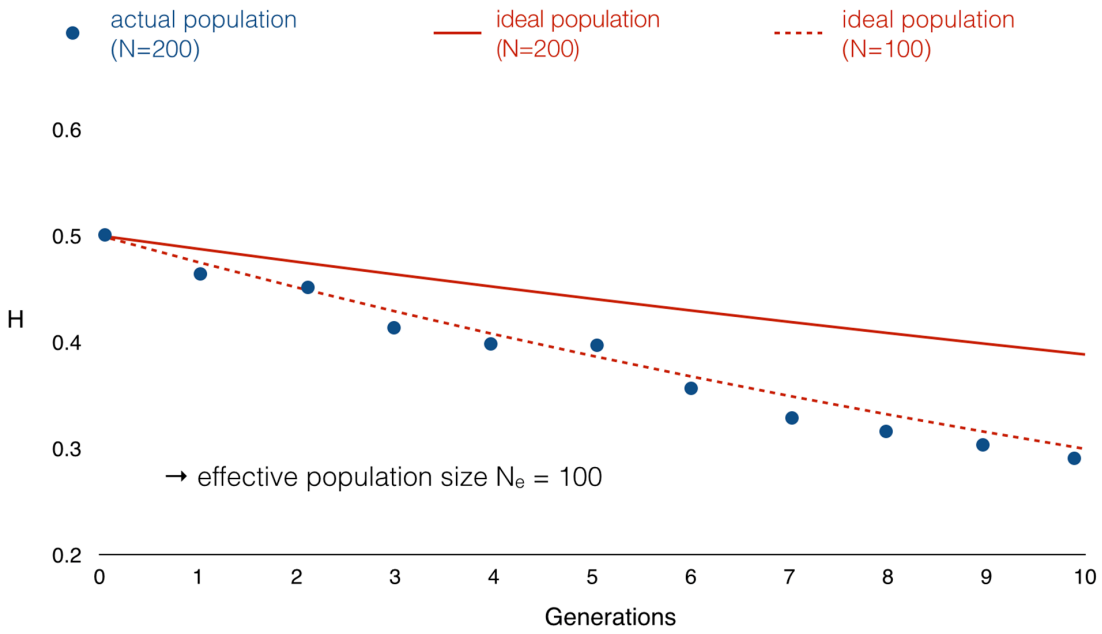
There are other assumptions that we may have violated, so the question naturally emerges whether the simplified Wright-Fisher model is useful at all. The answer to this question is a resounding yes. As you have seen, the simplifying assumptions of the Wright-Fisher model allowed us to derive many important effects of a finite population size. The problem is, each of the violated assumptions, when taken into account, may have different effects on evolutionary dynamics, so how are we going to be able to reconcile all these differences? How can we compare different populations that have different specific biological characteristics? What we need is a way for us to translate what we observe in nature into something we can all agree on when talking about models. That something is the *effective population size*.

Before I go into the details, let's make sure we really understand this idea. When you talk about the effect of drift in your population, your focus should be on the effective population size, not the actual size of your population (commonly referred to as the *census* population size). Another way of saying this is as follows: from the perspective of genetic drift, it is not really important how large your population is in reality, i.e. if you were to count up the individuals - what matters is *as how large a population it behaves if it were a Wright-Fisher population.*

This is useful for a number of reasons. I have already mentioned one of the reasons above, which is that it allows for much more sensible comparisons of populations when you are interested in

the effect of genetic drift. The second reason is that the effective population size, as it turns out, is usually *smaller* than the census population size, for reasons we'll go into in a minute. Thus, you would usually underestimate the force of genetic drift if you would make your calculations based on the actual number of individuals in your population.

Here is a way of showing this graphically:



As you can see, the effect of genetic drift, which is to reduce genetic variation, follows the pattern that you would expect in a Wright-Fisher of population of size 100. In other words, the effective population size of this population is 100, whereas the census population size is 200. Or put differently, even though this population consists of 200 individuals, it behaves - from a genetic drift perspective - as if it consisted of 100 individuals in a Wright-Fisher population. And this brings us to the definition of the effective population size, which is *the size of an ideal (Wright-Fisher) population that shows the same decay of genetic variation as the actual population of interest.*

Now that we have established the concept of the effective population size, let's take a look at why the effective population size is usually smaller than the census population size. There can be a number of reasons, but I will focus on two of the most important ones.

The first reason for a substantially reduced effective population size is the occurrence of *population bottlenecks*. Sometimes, populations that are usually stable in population size may experience a substantial reduction in population size. Perhaps the population has been infected by a very deadly pathogen, or predation pressure has been unusually high. In that case, even just one generation of a strong population size reduction can have a very strong effect on the effective population size.

Let's look at a simple numerical example. Let's assume that the size of a population is usually 1,000 individuals. However, for reasons unknown, the population size collapses to 100 in one generation, before going back up to 1,000 again in the following generation.

If we look at just three generations in a row, where the population sizes were 1000, 100, and 1000 individuals, respectively, we could be forgiven for thinking that the effective population size in these three generations would be the average, which is 700. It turns out that the effective population size is the *harmonic mean* of the population sizes, which is 250. This is quite remarkable - this single generation bottleneck has slashed the effective population size to a number that's almost three times smaller than what you would have expected by simply calculating the average. To make an even more extreme example, let's say the population sizes in these three generations are 1000, 10, and 1000. The average population size over the three generations is 670, but the effective population size is 29.4!

Let's have a quick look at the definition of the harmonic mean - for any set of numbers x_1, x_2, x_3 ... x_n, the harmonic mean is

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \frac{1}{x_3} + ... + \frac{1}{x_n}}$$

Because this term is a reciprocal of a sum of reciprocals, small numbers will have very strong effects on the mean. For the numbers 1000, 10, and 1000, the mean is

$$\frac{3}{\frac{1}{1000} + \frac{1}{10} + \frac{1}{1000}} = \frac{3}{0.102}$$

The reason why the harmonic mean is so small (29.4) is that the denominator is rather big, and it itself is almost fully determined by the value $\frac{1}{10}$. Even if you had population sizes of 1,000,000 before and after the bottleneck, the formula shows that they wouldn't have mattered the slightest bit, because then the mean would simply be equal to $\frac{3}{0.100002}$, which is still about 30. The other important insight that the formula reveals is that even when you have longer time periods of large, stable population sizes, the one generation with population size 10 can still dominate the outcome, and the effective population size would still be much smaller than expected.

The important message here is obvious: because the effective population size is given by the harmonic mean, population bottlenecks, even if over very short periods of time, will dramatically reduce the effective population size.

Let's take a look at this claim with code.

First, if we want to display the effective population size, we need to calculate it. In order to do this, we need to keep track of the population sizes over time. To do this, I first define a new variable which will hold the population sizes over the course of a simulation:

```
var population_sizes = [];
```

This variable holds an empty array into which I'm going to add the population sizes. In order to do that, I will change my simulation function to

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

All I'm changing here is the line

```
population_sizes.push(N);
```

which will add the population size `N` to the array. "But wait", I hear you say, "since `N` currently remains constant, we are always pushing the same value into the array!" That's right, and we'll change that in a little bit. For now I just want to make sure we have all the pieces set up.

Now that we have our array with all the population sizes over the course of a simulation, we need to write a function that can actually calculate the effective population size based on the values in the array. Here's how we can do this:

```
function effective_population_size(all_Ns) {
    var denominator = 0;
    for (var i = 0; i < all_Ns.length; i = i + 1) {
        denominator = denominator + (1 / all_Ns[i]);
    }
    return Math.round(all_Ns.length / denominator);
}
```

This function takes an array as an argument, iterates through its elements, and calculates the harmonic mean of its values. In order to avoid rounding issues, we simply return the number rounded to the closest integer by using the method `Math.round()`.

Add the function somewhere in your code, and call it like so (after you've run the simulations):

```
Ne = effective_population_size(population_sizes);
```

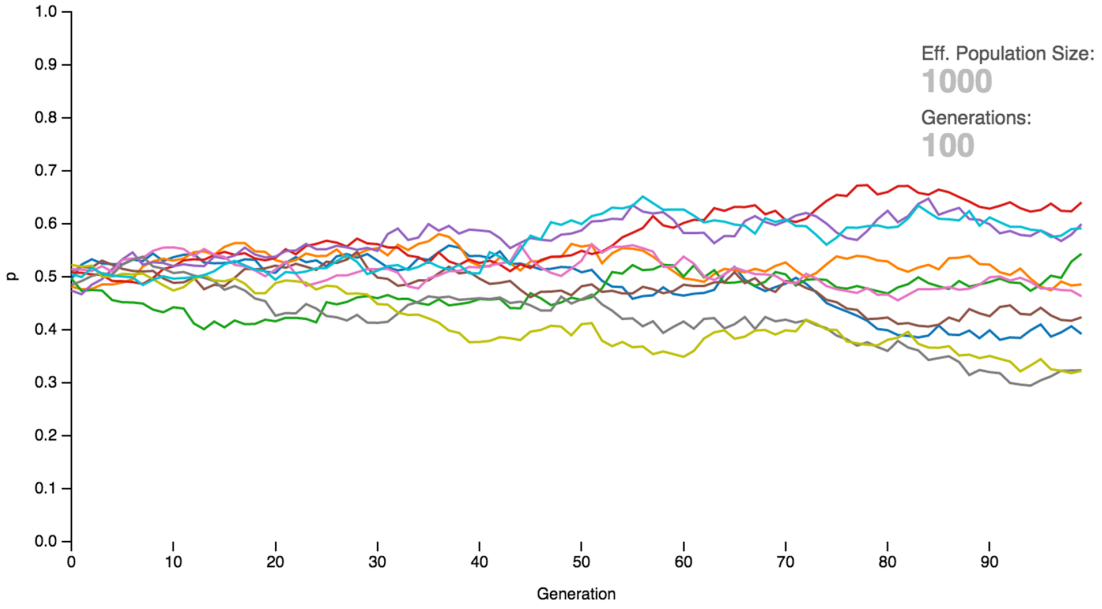
Then, when calling the `draw_line_chart()` function, update the legend accordingly: we want the legend to say "Eff. Population Size", and we want it to display `Ne`, rather than `N`:

```
draw_line_chart(data,"Generation","p",["Eff. Population Size:",Ne,"Generations:"\
,generations]);
```

Set the population size `N` to `1000`, and the number of generations to `100`:

```
var N = 1000;
var generations = 100;
```

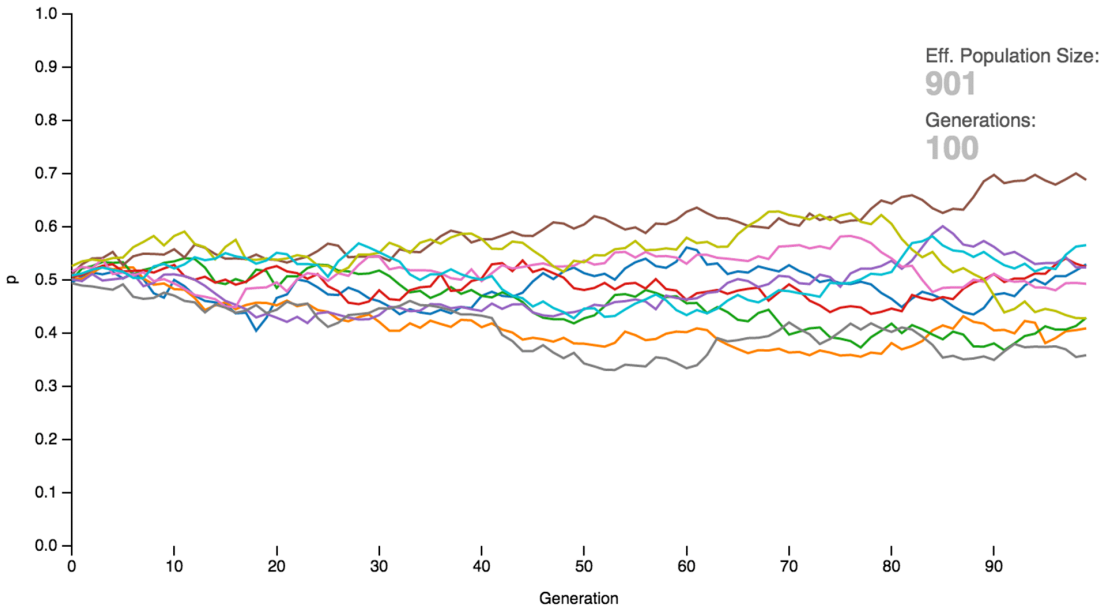
Save the document and load it in the browser. You should see something like this:



Which is exactly what we expected. Since `N` remains constant throughout the `100` generations, the effective population size will be `1000`.

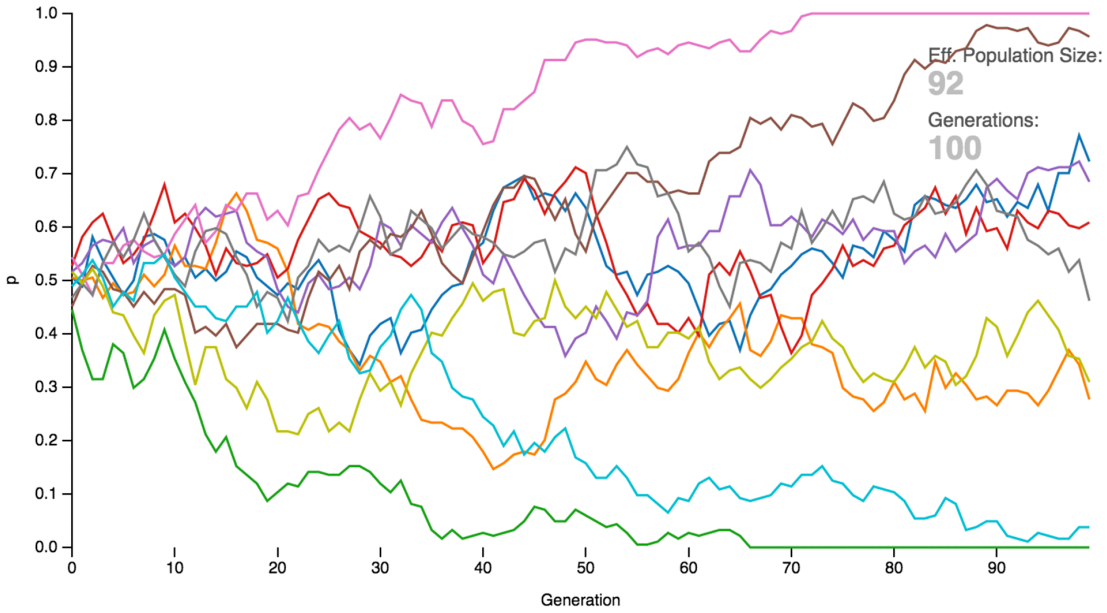
Now let's adapt the code further to allow for a change in population size. We are going to add a slight modification that allows us to introduce a bottleneck of `N = 10` every `10` generations.

First, let's briefly do the math. If we did not know about the harmonic mean, we might have thought that the average population size would be a good predictor of the evolutionary dynamics. The average population size of 90 generations at `N = 1000` and `10` generations at `N = 10` is 901. If we run ten simulations, using our current code with `N = 901`, we get the following outcome:



As expected, because of the reduced population size, the allele fluctuations are a bit more pronounced, but by and large the effect is small.

On the other hand, I have just claimed above that the population will behave much more like a Wright-Fisher population of a size that corresponds to the effective population size, which in this case would be 92. How does a Wright-Fisher population of size 92 behave? Let's find out by setting `N = 92`, and here is what we get:



Obvisouly quite different - a substantial reduction in genetic variation, with some populations having lost all variation by generation 100 already.

Let me repeat this - if the claim about effective population size and harmonic mean is indeed correct, then the dynamics of a population with `N = 1000` that goes through `N = 10` bottlenecks every 10^{th} generation should be much more like a population that has a constant population size of `N = 92`, rather than `N = 901` (and as the two previous images have shown, the dynamics would be quite different).

Only one way to find out! Let's go into the code and implement the actual bottlenecks.

First, just to be sure, set `N` back to `1000`.

Then, since we want each generation to be able to have a different population size, we will pass the population size as a parameter to the `next_generation()` function. To do that, we need to change the function slightly to read

```
function next_generation(simulation_data, current_N) {
    var draws = 2 * current_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);
}
```

All that has changed is the number of arguments, and the first line of the function body. Before, when the function did not have a second argument `current_N` for the population size, we used the global variable `N` in the function body when referring to the population size. Now the function doesn't care about the global variable `N` anymore - it simply works with the population size that it is given (via the parameter `current_N`).

Now let's implement the occasional bottleneck. Here's how we will do it:

```
function simulation(simulation_counter) {
    p = 0.5;
    var population_size;
    for (var i = 0; i < generations; i = i + 1) {
        if (i%10 == 9) {
            population_size = 10;
        }
        else {
            population_size = N;
        }
        population_sizes.push(population_size);
        next_generation(data[simulation_counter],population_size);
    }
}
```

Here's what has changed. First, we define a local variable called `population_size`. We don't assign it any value yet - that's about to happen in the `for` loop. In the loop, I've added the following lines of code:

```
if (i%10 == 9) {
    population_size = 10;
}
else {
    population_size = N;
}
```

But wait - what is this `i%10 == 9` business? The `%` operator is called the *modulus* operator. It returns the integer remainder of a division.

Thus,

`0 % 10` equals `0`
`1 % 10` equals `1`
`2 % 10` equals `2`
`3 % 10` equals `3`
`4 % 10` equals `4`
`5 % 10` equals `5`
`6 % 10` equals `6`
`7 % 10` equals `7`
`8 % 10` equals `8`
`9 % 10` equals `9`
`10 % 10` equals `0`
`11 % 10` equals `1`
etc.

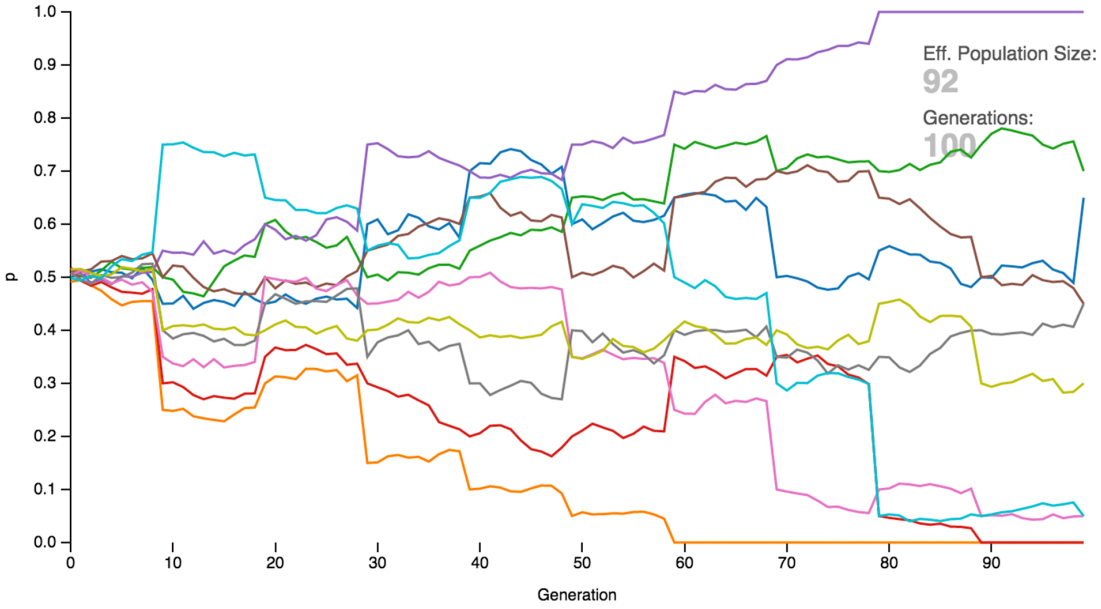
In other words, with `i` starting at `0`, when `i%10` equals `9`, we are in the 10th, 20th, 30th etc. generation. And whenever that is the case, we set the population size to `10`, otherwise we set it to `N`.

In the following two lines,

```
population_sizes.push(population_size);
next_generation(data[simulation_counter],population_size);
```

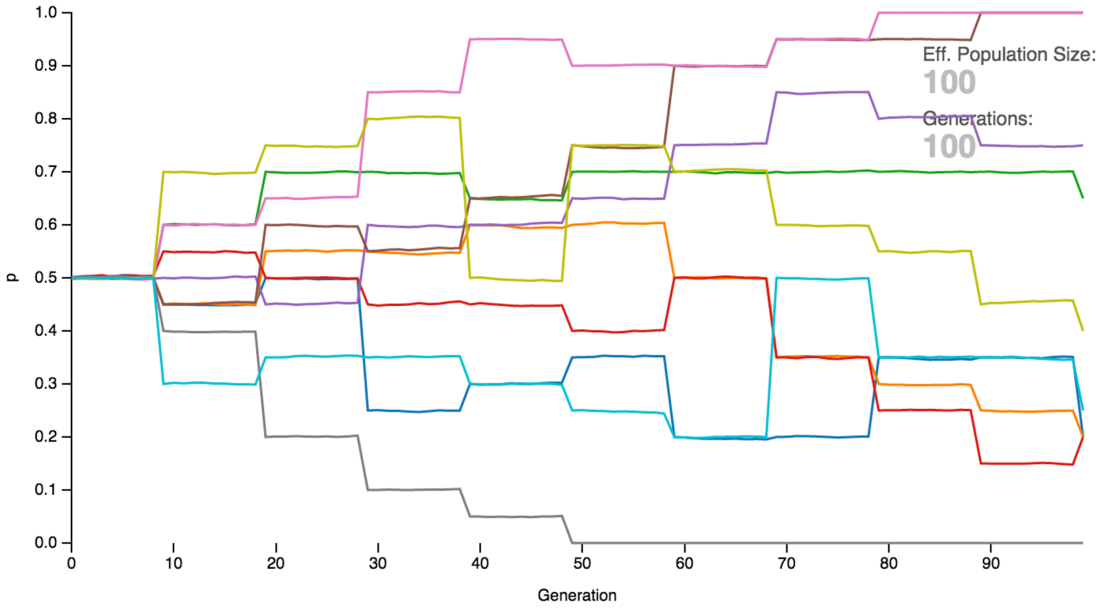
I simply changed `N` to `population_size`, because it is the variable `population_size` that holds the value of the population size in the current generation.

That's it! Save the document and load it in a browser, and here's what you'll get:



Wow - that does look much more like the Wright-Fisher population at a constant `N = 92`! The effective population size is indeed `92` - substantially lower than the average in the census population size, which is `901`. You can clearly see when the population goes through the bottlenecks, but if you flatten out those jumps visually in your mind, you can see that the population behaves pretty much like a Wright-Fisher population at `N = 92`, rather than at `N = 901`.

To make an even more extreme case, set `N = 100000`, but keep the bottlenecks in the code. Your simulations will now of course run a bit slower (they take a few seconds on my laptop computer), but here's what you will see when they are done:



As you can see, the effective population size is still very small (`100`), despite the fact that this population has a size of `100000` during `90%` of the time! You can also see that the dynamics are completely driven by the bottlenecks.

To sum up: changing census population sizes can have a strong effect on the effective population size, especially when the population goes though severe bottlenecks. The effect of bottlenecks is to reduce the effective population size, which will result in a much faster reduction of genetic variation.

The second reason for a substantially reduced effective population size is an *unbalanced sex ratio*. Recall that the Wright-Fisher model assumes hermaphroditic individuals. Because many species are not hermaphroditic, but rather have individuals with distinct sexes, the sex ratio can have a big impact on the effective population size. Imagine a population of 100 individuals, where 99 are female, and only one is male. The next generation will all have alleles from the one male, and it is intuitively clear that this offspring generation will have a much lower genetic variance than it would have had, if the parent generation had had a more balanced sex ratio.

As it turns out, there is a simple formula to calculate the effective population size if we know the number of females, N_f, and the number of males, N_m, in a population ($N = N_f + N_m$). The formula is

$$N_e = \frac{4 N_m N_f}{N}$$

If the sex ratio is balanced such that $N_m = N_f$, then it's easy to see that the census population size and the effective population size are equal: because $N_m = N_f$, it follows that $N_m = N_f = \frac{N}{2}$, and so

$$\frac{4N_mN_f}{N}$$

becomes

$$\frac{4\frac{N}{2}\frac{N}{2}}{N}$$

which is

$$\frac{4\frac{N^2}{4}}{N}$$

which is

$$\frac{N^2}{N}$$

which is *N*, and thus $N_e = N$ in that case.

However, let's look at an example where the ratio is very unequal. Unequal sex ratio may be due to sexual selection (where females prefer certain males over others, and vice versa), or due to artificial breeding (e.g. where a farmer has just a few male bulls but many female cows), or due to other reasons. Let's look at the case where we have a population of 10 male bulls and 90 female cows. Thus, $N_f = 90$, and $N_m = 10$. If you plug these numbers into the equation above, you get

$$N_e = \frac{4 * 90 * 10}{100} = 36$$

Thus, the effective population size is only slightly more than a third of the actual census population size.

We can get a good feeling for how strong this effect is going to be by plotting the ratio $\frac{N_e}{N}$ against the sex ratio. $\frac{N_e}{N}$ tells us how large the effective population size is, relative to the census population size. In the example above, the ratio would be $\frac{36}{100} = 0.36$, which is quite low. On the other hand, our sex ratio was very distorted. Let's plot the two quantities against each other.

First, we need a little math. I would like $\frac{N_e}{N}$ plotted as a function of the sex ratio. I will chose the ratio of $\frac{N_m}{N}$ as my sex ratio. So how can I transform this equation

$$N_e = \frac{4N_mN_f}{N}$$

into one that expresses $\frac{N_e}{N}$ as a function of $\frac{N_m}{N}$? A few easy steps! First, divide both sides by *N*:

$$\frac{N_e}{N} = \frac{4N_mN_f}{N^2}$$

We can rewrite this as

$$\frac{N_e}{N} = 4\frac{N_m}{N}\frac{N_f}{N}$$

and since $\frac{N_f}{N} = 1 - \frac{N_m}{N}$, we can write

$$\frac{N_e}{N} = 4(\frac{N_m}{N})(1 - \frac{N_m}{N})$$

That's it: $\frac{N_e}{N}$ is now expressed as a function of $\frac{N_m}{N}$. Now, let's write the JavaScript code that calculates $\frac{N_e}{N}$ for values of the male proportion ranging from 0 to 1, in increments of 0.01:

```
var data=[];
var x_max = 1;

for (var i = 0; i <= x_max + 0.005; i = i + 0.01) {
    var male_to_female_ratio = i;
    Ne_to_N_ratio = 4 * male_to_female_ratio * (1 - male_to_female_ratio);
    data.push(Ne_to_N_ratio);
}

draw_line_chart(data,"Proportion Males","Ne / N",[],x_max);
```

As you can see, I'm implementing the equation above, and push the result into an array that we'll hand over to the plotting function (which we will talk about in a second). But there are two subtleties in the `for` loop that I want to mention. First, I'm using the smaller than or equal operator (`<=`), because I want to include the last value, `x_max` (which I'm setting to `1`). Second, I'm not actually writing

```
i <= x_max;
```

but rather

```
i <= x_max + 0.005;
```

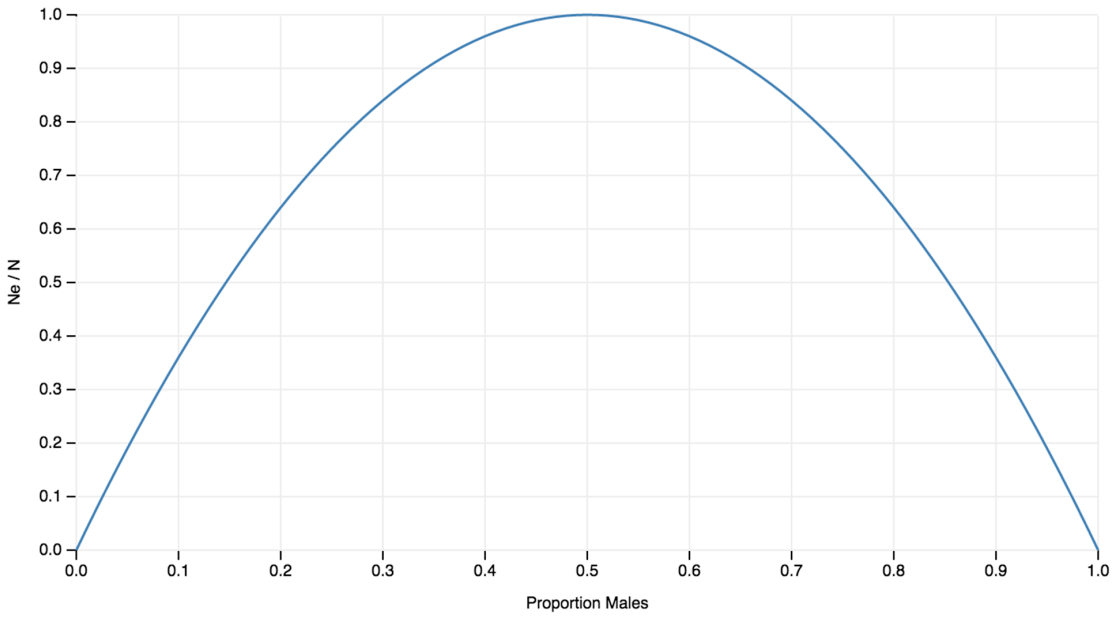
why is that? It's because the rounding error would kick in again otherwise. Feel free to output the value of `i` to the console - you would see that after you've added the value `0.01` a hundred times to `0`, your final value will be `1.0000000000000007`, rather than `1`. To be sure that the for loop doesn't end prematurely, I'm adding a small amount as an error margin to `1` when making the comparison.

What about plotting? We're passing the data to the plotting function like so:

```
draw_line_chart(data,"Proportion Males","Ne / N",[],x_max);
```

The first three parameters are straightforward: the data, and the axis labels. The fourth parameter is an empty array - empty because we don't want a legend in the figure. The last parameter defines the maximum range for the values of `x`, `x_max` (which is `1` in our code example). If we don't provide that value, the x axis labels would be set according to the number of elements in the `data` array. In the previous plotting examples, that was fine, because each element in the `data` array corresponded to one generation. But for this plot, each point on the x axis should represent an increase of 0.01 (rather than 1), with a maximum value of 1. That's why we pass `x_max` (i.e. 1) as a parameter.

And here is the outcome:



As you can see, the effect is very strong when the sex ratio is strongly skewed. A quick check at a proportion of males at 10% shows that the graph correctly predicts that the effective population size will be 36% of the census population size, as our farm example above has shown.

Overall, the message here is that unless the sex ratio is severely skewed (less than 30% of one sex), the overall relative reduction in effective population size is small. However, beyond that value, the effect can be quite severe.

Boy, what a chapter! We're done! Congratulations for following me all the way through here. Let's briefly wrap up what we've learned in this chapter.

- Finite population size introduces randomness into evolutionary dynamics (genetic drift).

- Genetic drift acts to reduce genetic variation.
- The effect of drift is proportional to the population size: the smaller the population, the larger the effect of drift.
- Unless other forces counter it, genetic drift will reduce all genetic variation in every population.
- Drift is a slow process - it takes about $1.4N$ generations to reduce the genetic variation of a population by half.
- The effective population size is a key concept - it is the size of an ideal (Wright-Fisher) population that shows the same decay of genetic variation as the actual population of interest.
- The effective population size is often smaller than the actual population size, due to various reasons such as population size bottlenecks, or unequal sex ratios.
- We've also covered some key programming concepts such as arrays and conditional flows.

I hope you feel empowered to move on to the next chapter. You should be - you've mastered a lot of ground already. Think about where we started - at the very beginning, and yet at this point, you are already running stochastic simulations and plotting them in your browser, all the while learning key ideas about evolution.

4. Mutation: The Power of Mistakes

In the previous chapter, we've learned a number of highly interesting aspects about evolution by genetic drift. One of the rather depressing consequences of drift, we learned, is the total elimination of genetic variation in the long run. Clearly, something else must be going on, for the world is not void of genetic variation. That something else is *mutation*.

Mutation is simply the random change in a genetic sequence. Most mutations occur when the genetic sequence (the DNA in nearly all living species) is replicated. The replication process, even though highly accurate, is not quite 100% error-free. Now and then, an error will occur, and the new copy will not be identical base for base to the original.

In animals like ourselves, almost all mutations are so-called *somatic mutations*. That is, they occur in cells that are not passed on to future generations. These mutations are often contributing to cancer, but from an evolutionary perspective, they are a dead end. The mutations that do matter from an evolutionary perspective are the so-called *germ line mutations*. They occur in the germ line, i.e. the cells that will be passed on to future generations in the form of sperm or egg.

The ultimate effect of mutation is to increase genetic variation. Let's write a short JavaScript program to illustrate that. Rather than using the two alleles model from earlier chapters, we'll just use a simple DNA model. As you know, DNA is made up of four base pairs (Adenine, Guanine, Cytosine, and Thymine) that are usually abbreviated by their first letters A, G, C, and T.

First, let's set up a few key parameters and data structures that we'll be using:

```
var sequences = [];
var original_sequence = [];
var number_of_sequences = 100;
var sequence_length = 20;
var number_of_generations = 100;
var mutation_rate = 0.0001; // per base and generation

var BASES = ['A','G','C','T'];
```

We'll be storing a number of sequences in the array `sequences` - each sequence will itself be an array of the characters `'A'`, `'G'`, `'C'` and `'T'`. We'll store the original sequence in the array `original_sequence`. We would like to have `100` sequences in total (defined by `number_of_sequences`), and each sequence should consist of `20` base pairs (defined by `sequence_length`). Starting from an originally uniform population (i.e. no genetic variation - all sequences will be exact copies of the `original_sequence` initially), we'll run `100` generations (defined by `number_of_generations`) during which we will have mutations occur with a probability of `0.0001` per base and generation (defined by

`mutation_rate`). That is, in each generation, each base pair of each sequence has a `1` in `10,000` chance of "switching" to another base pair.

If you are wondering why the variable `BASES` is in capital letters: I'm simply following the established practice of using all capital letters in variables that are expected to be constant and never changing. JavaScript doesn't have a built in mechanism to make a variable immutable, and thus making it clear to others that this variable shouldn't change by using all capital letters is considered the best practice.

Let's first establish the code for generating the first generation. Here it is:

```
function random_base(current_base) {
    var new_base;
    do {
        var index = Math.floor(Math.random() * 4);
        new_base = BASES[index];
    } while (new_base == current_base);
    return new_base;
}

function generate_original_sequence() {
    for (var i = 0; i < sequence_length; i = i + 1) {
        original_sequence.push(random_base(""));
    }
}

function generate_first_generation() {
    generate_original_sequence();
    for (var i = 0; i < number_of_sequences; i = i + 1) {
        sequences.push(original_sequence.slice());
    }
}

function print_sequences(title) {
    console.log(title);
    for (var i = 0; i < number_of_sequences; i = i + 1) {
        print_sequence(sequences[i]);
    }
    console.log('');
}

function print_sequence(sequence) {
    var sequence_string = "";
    for (var i = 0; i < sequence.length; i = i + 1) {
```

```
        sequence_string = sequence_string + sequence[i];
    }
    console.log(sequence_string);
}


generate_first_generation();
print_sequences("Generation 0");
```

Even though this is quite a bit of code, you should be able to read through it and understand most of it. I'll walk you through it in a second, but before we do that, take a look at the overall structure here. As you can see, the code is more or less packaged into functions that have a very specific task. At the end, we call two functions directly, and that's it. This should make it much easier to read and understand the code.

So rather than going from top to bottom of the code, I'll start where the first function is called: `generate_first_generation()`. The first thing that the function does is call another function, `generate_original_sequence()`, so let's go and take a look at that first. The function `generate_original_sequence()` simply contains a `for` loop that will add a random base to the `original_sequence` array. How does it get a random base? By calling the function `random_base()` with an empty string argument, so let's go and take a look at that function.

The function `random_base()` should return a new base that is different from a given base. It should be different from a given base because we want to use this function later on when we mutate a DNA sequence. If we happen to get to the base `'A'`, say, and that function would return `'A'` also, then we wouldn't have a mutation - the base is still `'A'`. This is why we added an argument to the function - so that we can essentially tell the function to give us a new base that is *different* from a given base.

We do this by using a `do while` loop, which is something you haven't seen before. The only loop you've encountered so far was the `for` loop, which is very practical in general, although not for all uses. The `for` loop is a straightforward construct when you know how often you want to iterate. However, in this case, our requirements are a little different from that. What we want to express, in code, is the following: draw a random letter from the letters `'A'`, `'G'`, `'C'` and `'T'`, and continue to do that as long as the drawn letter is the same as the one I have given you. The `do while` loop is ideal for this kind of situation. It says "do something while a certain condition is met".

Its structure is simple:

```
do {
    code to be iterated
} while (condition);
```

One of the side effects of this construct is that the code to be iterated *is guaranteed to be executed at least once*. It may be executed many more times, as long as the condition is met, but it will be

executed at least once. And this is exactly what we need: we need to draw at least one base pair, and if that base pair does not meet our condition, we will continue to draw a base pair until that condition is met. Let's take a detailed look at how we do this:

```
do {
    var index = Math.floor(Math.random() * 4);
    new_base = BASES[index];
} while (new_base == current_base);
```

In other words, we draw a random integer between `0` and `3` (note that `Math.floor()` always rounds down to the next lower integer), which we will use as our index in the `BASES` array. We then store that selected base in the variable `new_base`, and compare it to the base that we passed to the function as the parameter `current_base`. If the two are identical, we'll repeat the process until they're not identical anymore.

Note that in the function `generate_original_sequence()`, we pass an empty string as a parameter to the function `random_base()` - you hopefully now understand why: the randomly drawn base will be compared to an empty string, and thus the function will always return the first randomly drawn base, which is what we want when we generate the first random sequence.

After the function `generate_original_sequence()` has finished executing, we're going back to the function `generate_first_generation()`, where we find another `for` loop. This time, the loop makes copies of the original sequence and saves those copies in the `sequences` array. The copying is done using the `slice()` method, which copies an array (another handy built-in method of the array object that you might want to use in the future).

Once that's done, we're done with the function `generate_first_generation()`, and can go to the next function we're calling, which is `print_sequences()`. This function, and the function it calls, is very simple and has no new concepts, so I will trust that you understand what's happening here. It is simply printing all the sequences into the console.

If you save this code in a new file and load it in a browser, you should see the following in your console (depending on the browser):

The console window in my browser (Chrome) is being nice here, and rather than printing the same output a hundred times, it is printing it once, and letting me know that it was asked to print the same output `100` times. If you reload, you should see the same thing, but with an other sequence.

The code seems to be working well. You now have a uniform population of `100` DNA strings, and no variation whatsoever. Now let's introduce mutation.

I'm going to add one method:

```
function run_generations() {
    for (var i = 0; i < number_of_generations; i = i + 1) {
        for (var ii = 0; ii < sequences.length; ii = ii + 1) {
            for (var iii = 0; iii < sequences[ii].length; iii = iii + 1) {
                if (Math.random() < mutation_rate) {
                    sequences[ii][iii] = random_base(sequences[ii][iii]);
                }
            }
        }
    }
}
```

and also add these two method calls after `print_sequences("Generation 0")`:

```
run_generations();
print_sequences("After 100 generations");
```

Granted, the function `run_generations()` looks a little intimidating. But we'll walk through it line by line. Notice that there is no new concept here - just loops and arrays. The only unusual thing is that we have three loops at once, and they are *nested*. The first loop iterates over the generation counter. The second loop iterates over all sequences. Because the two loops are nested, we can say that in every generation, we loop over all sequences. But it doesn't stop there - we have a third loop which iterates over all bases. In other words: in every generation (first loop), we go through each sequence (second loop) and iterate over each base (third loop).

Notice that each loop has a different counter variable. I use `i`, `ii`, `iii` etc. Other people use `i`, `j`, `k`, etc. I prefer my style because it instantly tells me which nesting level I'm at, but feel free to use whichever variable names you prefer.

Once we are in the third loop that iterates over each single base pair in a sequence, we can trigger a mutation event with probability `0.0001`. As before, we are implementing this stochastic event by using the built-in random number generator `Math.random()`. Let's take a look at how we do this:

```
if (Math.random() < mutation_rate) {
    sequences[ii][iii] = random_base(sequences[ii][iii]);
}
```

The expression `sequences[ii][iii]` is accessing an element from a two-dimensional array. If this doesn't ring a bell, be sure to re-read the discussion about arrays in the previous chapter. The array `sequences` contains all sequences, which are themselves arrays of base pairs. Thus, `sequences[ii]` is itself an array (an array of bases), and the expression `sequences[ii][iii]` means the iii^{th} element (the base pair) in the ii^{th} sequence, where `iii` and `ii` are the counters of the two innermost `for` loops.

Now, the complete code is as follows:

```
var sequences = [];
var original_sequence = [];
var number_of_sequences = 100;
var sequence_length = 20;
var number_of_generations = 100;
var mutation_rate = 0.0001; // per base and generation

var BASES = ['A','G','C','T'];

function random_base(current_base) {
    var new_base;
    do {
        var index = Math.floor(Math.random() * 4);
        new_base = BASES[index];
    } while (new_base == current_base);
    return new_base;
}

function print_sequence(sequence) {
    var sequence_string = "";
    for (var i = 0; i < sequence.length; i = i + 1) {
        sequence_string = sequence_string + sequence[i];
    }
    console.log(sequence_string);
}

function generate_original_sequence() {
    for (var i = 0; i < sequence_length; i = i + 1) {
        original_sequence.push(random_base(""));
    }
}

function generate_first_generation() {
    generate_original_sequence();
    for (var i = 0; i < number_of_sequences; i = i + 1) {
        sequences.push(original_sequence.slice());
    }
}

function run_generations() {
    for (var i = 0; i < number_of_generations; i = i + 1) {
        for (var ii = 0; ii < sequences.length; ii = ii + 1) {
```

```
            for (var iii = 0; iii < sequences[ii].length; iii = iii + 1) {
                if (Math.random() < mutation_rate) {
                    sequences[ii][iii] = random_base(sequences[ii][iii]);
                }
            }
        }
    }
}

function print_sequences(title) {
    console.log(title);
    for (var i = 0; i < number_of_sequences; i = i + 1) {
        print_sequence(sequences[i]);
    }
    console.log('\n');
}

generate_first_generation();
print_sequences("Generation 0");
run_generations();
print_sequences("After 100 generations");
```

If you save the file and load in the computer, you'll see something like this:

```
Generation 0
100 GGAGTCAGCGCCTAGACTTG

After 100 generations
CGAGTCGGCGCCTAGACTTG
GGAGTCAGCGTCTAGACTTG
12 GGAGTCAGCGCCTAGACTTG
GGAGTCAGGGCCTAGACTTG
2 GGAGTCAGCGCCTAGACTTG
GGAGTCAGCGCCTAGGCTTG
5 GGAGTCAGCGCCTAGACTTG
GGAGTCAGCGCCTAGACTAG
GTAGTCAGCGCCTAGACTTG
GGAGTCAGCGCCTAGACTTG
```

In the interest of space, I'm only showing a fraction of what you'll see, but the point is as follows: mutation has introduced a substantial amount of genetic variation. In an example run on my machine, there are now 34 unique genetic sequences, and the population is dominated by 4 sequences that each occur more than 10 times. Of course, this will be different every time you re-run the

simulation, but it's nevertheless remarkable. We are often tempted to think that mutations rates as low as 1 in 10,000 don't have a strong effect, simply because 1 in 10,000 is an unlikely event. Nevertheless, if you have lots of base pairs in lots of individuals over many generations, even much lower mutation rates will produce a substantial amount of genetic variation.

Feel free to play around with the value of the mutation rate. If you increase the mutation rate by a factor of ten, to `0.001`, you'll notice that you get even more variation: almost all sequences will be unique. Conversely, if you decrease the mutation rate by a factor of ten, to `0.00001`, you'll notice that you get very little variation, and sometimes even none at all in a single simulation run.

Mutation rates in nature are quite variable. In humans, the estimates are somewhere around 10^{-10} per base pair and replication, and 10^{-8} per base pair and generation - that is, around 1 in 10 billion per base pair and replication, and 1 in 100 million per base pair and generation (human germ cells go through many replications per single human generation). At the lower end, some viruses have mutation rates as low as 1 in 1,000 per base pair and replication. It's important to be careful about what exactly the mutation rate denotes. Mutation rates are always per some unit, and per some time interval. They can be per base pair, per gene / allele, or per genome (the totality of all base pairs). They can be per replication or per generation.

Having established that mutations can generate genetic variation, we should notice something very interesting: that the consequences of mutation are the exact opposite of the consequences of drift. This is the central idea in this chapter, and one of the central ideas in evolution in general. Mutation creates genetic variation; drift eliminates it.

We'll have plenty to code later, but in this chapter I would first like to establish the mathematical foundations of this battle of forces, mutation vs. drift. We'll use the same type of math that we used in the previous chapters, and I'll be again very careful to walk you slowly through each step.

Let's start again at the beginning. Last time, when were just looking at drift, we started with the observation that

$$G' = \frac{1}{2N} + (1 - \frac{1}{2N})G$$

Let's recall what that equation means. It simply means that the probability that two randomly picked alleles in the next generation will be the same (G') depends on that very probability in the current generation (G) and the population size N. From this equation, we derived all the other insights about genetic drift in the previous chapter. Let us just revisit very briefly how we obtained this initial equation.

How can two randomly picked alleles be the same type of allele? There are two possibilities. Either, they happen to be descendants of the same parental allele, which occurs with probability $\frac{1}{2N}$. Or, they are not descendants of the same parental allele (which happens with probability $1 - \frac{1}{2N}$), but their parental alleles are nonetheless of the same type (which is the definition of G). Notice what is missing here? Mutation! We have not considered the effect of mutation yet. The two alleles could in fact be different, simply because of a mutation that occurred during the copying process. So in order for the two alleles to be identical, it must be that no mutation occurred - otherwise they would be

different (we're ignoring here the astronomically small chance that they both mutated to the same new type). What is the probability that no mutation occurs in either of the two alleles?

If we define the mutation rate per allele per generation as μ, then the probability that no mutation occurs in either of the two alleles is $(1-\mu)^2$. Why is that? The probability that no mutation occurs in the first allele is $1-\mu$. Equally, the probability that no mutation occurs in the second allele is also $1-\mu$. Both of these non-events need to occur, and therefore we need to multiply their probabilities. You can verify this by thinking about throwing dices. The probability that you throw a 6 with one dice is $\frac{1}{6}$. The probability that you don't throw a 6 is $1 - \frac{1}{6} = \frac{5}{6}$. Now if you throw two dice at the same time, what would be the probability that none of them would show a 6? The first dice must not show a 6 (probability $\frac{5}{6}$) and the second dice must not show a 6 (probability $\frac{5}{6}$), and therefore the total probability of not throwing a 6 with two dice is $\frac{5}{6} * \frac{5}{6} = \frac{25}{36}$ which is roughly 70%.

About that astronomically small chance...

But wait, I hear you say, what about both alleles mutating at the same time? Wouldn't they be the same again? Or what if we start with an A1 and an A2, and one of them mutates? Wouldn't they then be the same again? No - so far, we've made the simplifying assumption that there are currently two alleles in the population, A1 and A2. But that should not be taken to mean that there are only two *possible* alleles - there are in fact a great number of possible alleles (consider that the average gene contains thousands of base pairs, each of which could mutate). So when A1 mutates, it becomes some other allele - the chance that it mutates into exactly A2 is extremely small; and the chance that both alleles mutate into the same type is even smaller. So we can safely ignore it here.

Going back to alleles, we can adjust our equation to take into account that no mutation happened in either two alleles, by writing

$$G' = (1-\mu)^2(\frac{1}{2N} + (1 - \frac{1}{2N})G)$$

We simply multiply the entire term by the probability that no mutation occurs in any of the two alleles, which is $(1-\mu)^2$. Let's expand the mutation term, and we get

$$G' = (1 - 2\mu + \mu^2)(\frac{1}{2N} + (1 - \frac{1}{2N})G)$$

This can be simplified if we make the reasonable assumption that the mutation rate μ is small, and therefore, the square mutation rate will be extremely small to the point where we can ignore it in an addition. That is, we apply the approximation

$$1 - 2\mu + \mu^2 \approx 1 - 2\mu$$

when μ is small, giving us

$$G' \approx (1 - 2\mu)(\frac{1}{2N} + (1 - \frac{1}{2N})G)$$

Let's expand this further to

$$G' \approx \frac{1}{2N} + (1 - \frac{1}{2N})G - \frac{2\mu}{2N} - 2\mu(1 - \frac{1}{2N})G$$

The last term can be expanded a little further:

$$G' \approx \frac{1}{2N} + (1 - \frac{1}{2N})G - \frac{2\mu}{2N} - 2\mu G + \frac{2\mu G}{2N}$$

The next simplifying approximation is similar to the previous one. As you can see, the right hand side is now an addition of terms, and some of the terms are so small, relatively to the others, that we can effectively ignore them. Which terms are those? It's the terms that have both μ in the numerator, and N in the denominator. These terms are small because μ is small, and is made even smaller by the division by N. If we ignore these very small terms, we'll get

$$G' \approx \frac{1}{2N} + (1 - \frac{1}{2N})G - 2\mu G$$

Before we move on, let's bring H back into the scene. Recall that H is simply the opposite of G, and thus $1 - G$, and thus $G = 1 - H$. Starting from

$$H' = 1 - G'$$

we can now insert the previous term for G':

$$H' \approx 1 - (\frac{1}{2N} + (1 - \frac{1}{2N})G - 2\mu G)$$

and replace all occurrences of G with $1 - H$ to obtain

$$H' \approx 1 - \frac{1}{2N} - (1 - \frac{1}{2N})(1 - H) + 2\mu(1 - H)$$

If we fullly expand this equation, we'll get

$$H' \approx 1 - \frac{1}{2N} - 1 + \frac{1}{2N} + H - \frac{H}{2N} + 2\mu - 2\mu H$$

Since the first four terms of the rights side of the equation cancel each other, we get

$$H' \approx H - \frac{H}{2N} + 2\mu - 2\mu H$$

Let's pause here for a moment. Let's recall what we are trying to do there. We've established above that mutation and drift are two opposing forces - one increases genetic variation, the other decreases it. Since we have now established a value for *H'*, we can take a look at the difference in *H* in each generation, which we denote by *ΔH*:

$$\Delta H = H' - H$$

Thus *ΔH* is simply the change in genetic variation per generation. By replacing the above established value for *H'*, we get

$$\Delta H = H - \frac{H}{2N} + 2\mu - 2\mu H - H$$

Because the *H* is canceling itself, we can simplify this to

$$\Delta H = -\frac{H}{2N} + 2\mu - 2\mu H$$

which can be rewritten as

$$\Delta H = -(\frac{1}{2N})H + 2\mu(1 - H)$$

Note that this equation is a sum of a negative term and a positive term. Indeed, this is quite a beautiful equation: the negative term, $-(\frac{1}{2N})H$, denotes the loss of genetic variation due to drift, while the positive term, *2μ(1 - H)*, denotes the gain of genetic variation due to mutation.

From this simple equation, which captures both processes elegantly, we can deduce another interesting observation. First, it confirms what we already know, which is that the loss of genetic variation depends on the population size *N*, in such a way that the loss is larger if *N* is smaller. But second, it also shows quite clearly that the gain of genetic variation depends on the mutation rate *μ*, such that the gain is higher if the mutation rate is higher, as we would expect.

Now, there is going to be a point where in each generation, the increase through mutation will be exactly offset by the decrease through genetic drift, i.e. *ΔH* will be zero. This is an an interesting point, because it represents an equilibrium: the genetic variation will neither go up nor go down, but simply remain the same forever. What stable genetic variation can we expect?

We've just established that

$$\Delta H = -(\frac{1}{2N})H + 2\mu(1 - H)$$

The equilibrium will be reached when ΔH is zero, i.e.

$$-(\frac{1}{2N})H + 2\mu(1-H) = 0$$

If we bring the first term on the right hand side of the equation, we get

$$2\mu(1-H) = (\frac{1}{2N})H$$

Let's expand the left side of the equation:

$$2\mu - 2\mu H = (\frac{1}{2N})H$$

If we add $2\mu H$ to both sides, we'll get

$$2\mu = (\frac{1}{2N})H + 2\mu H$$

Factoring out H will yield

$$2\mu = H(\frac{1}{2N} + 2\mu)$$

Dividing both sides by $(\frac{1}{2N} + 2\mu)$ gives us H

$$\frac{2\mu}{(\frac{1}{2N} + 2\mu)} = H$$

This looks a little unwieldy with a double fraction. Let's simplify this by multiplying the term by $\frac{2N}{2N}$, and we'll get

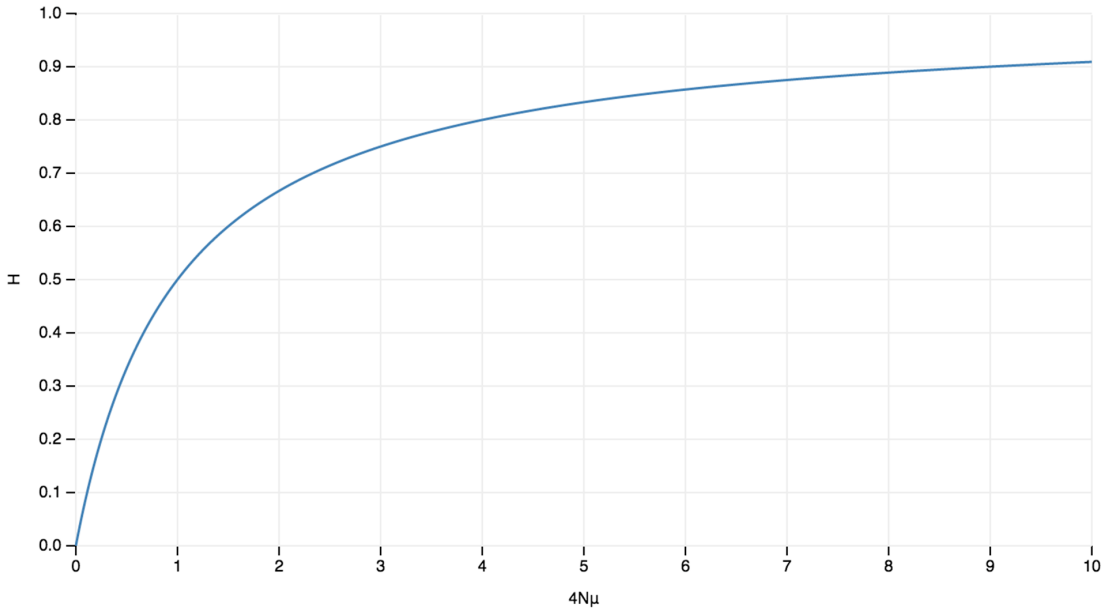
$$H = \frac{4N\mu}{(1 + 4N\mu)}$$

This is much nicer indeed. It is also a rather profound result. It tells us how much genetic variation we should expect in a population subject to drift and mutation.

Let's plot this relationship between H and the term $4N\mu$. We can do this fairly simply by filling up a `data` array:

```
var data=[];
var x_max = 10;

for (var i = 0; i <= x_max+0.005; i = i + 0.01) {
    data.push(i/(1+i));
}

draw_line_chart(data,"4N\u03BC","H",[],x_max);
```

and then pass that to the `draw_line_chart()` function that we've used above (note that `\u03BC` is simply the unicode character for μ). Here's what you'll see:



As you can see, H is quite sensitive in the 0 - 1 range for $4N\mu$, and thereafter starts to asymptotically approach 1. The way to think about this as follows. If $4N\mu$ is small (i.e. less than 1), the population will have little genetic variation at equilibrium. $4N\mu$ can be small if either N is small, in which case the variance-reducing effect of drift would be strong, and / or the mutation rate μ can be low, which generates little genetic variation. On the other hand, if $4N\mu$ is large (i.e. larger than 1), the population will have a lot of genetic variation at equilibrium. $4N\mu$ can be large if either N is large, in which case there would be very little reduction of genetic variation by drift, and / or the mutation rate μ can be high, which continuously generates a lot of genetic variation.

The overall take-home message is the following - just as genetic drift reduces genetic variation, mutation increases variation. Over time, the two processes will reach an equilibrium which is given by both the population size N, and the mutation rate μ.

The Fixation of Mutations

Let's move on to another interesting dynamic. We know that mutation introduces new genetic variants. But what does the future of such a mutation look like? In the beginning, the mutation exists as a single copy. Recall that in our world, everything is selectively neutral - we still assume no form of selection whatsoever. It's quite possible that such a rare mutation will be lost, due to random sampling, in the next generation already. It might stay around for a few generations, even manage to make a few copies of itself, and then disappear from the population. Or, it might actually take over the population and go to fixation!

You may think "yeah ok, but that's probably rare" and you would be right, as we'll see in a minute. Nevertheless, consider this: every allele that you see in a population, even the most common alleles, once started as a single copy. Granted, some of them might have been very beneficial to its bearer from the outside, which would have made its ride to domination over time a little easier. Nevertheless, even neutral alleles can go from a single copy to fixation in a population. The question is, how likely is that?

It turns out that the answer is very simple. The probability that a neutral allele will go to fixation is - drum roll... *its frequency*. Simple as that! If an allele has a frequency of 50%, the probability that it will go to fixation is 50%. If the allele has a frequency of 10%, the probability that it will go to fixation is 10%, and so on. What is the frequency of a single allele in a populate of size N? Since we're talking about diploid individuals, each individual has two copies of each gene, and thus there are $2N$ alleles, as we know. Thus, the frequency of the initial mutant allele is $\frac{1}{2N}$, which is also the probability that it will go to fixation.

Let's use some of the code that we already wrote to verify that this claim is actually true. In the previous chapter, all stochastic simulations that we ran started at `p = 0.5`, which meant that both alleles, A1 and A2, had a frequency of 50%. We can easily modify our code to ensure that `p` equals $\frac{1}{2N}$ - thus assuming that A1 is the mutant allele which has just come into existence in a population previously completely composed of A2 alleles. We will then run many, many simulations that we will stop when one of the two alleles, A1 or A2, has gone to fixation, and we'll count the number of times that A1 has gone to fixation (i.e. where `p==1`). That number, divided by the total number of simulations, will be the probability of fixation.

I invite you to try and write this code on your own. Don't worry if you don't manage to do it, I don't expect you to. But it's important to get your feet wet early - programming is all about practice. Feel free to copy from what we developed in the previous chapter.

I'm going to go ahead and show you how I would do it. The code is as follows:

```
var N = 100;
var p;
var simulations = 10000;
var fixations_of_mutant = 0;

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;
}

function run_until_fixation() {
    p = 1 / (2*N);
    do {
        next_generation();
    } while (p > 0 && p < 1);
    if (p == 1) {
        fixations_of_mutant = fixations_of_mutant + 1;
    }
}

for (var i = 0; i < simulations; i = i + 1) {
    run_until_fixation();
}
console.log(fixations_of_mutant / simulations);
```

You should now be in good shape understanding almost everything this code does. The function `next_generation()` is identical to the one we used in the previous chapter. I've defined an additional function called `run_until_fixation()`, that runs a simulation until one of the two alleles has become fixed. Its code is relatively simple - first, we (re)set `p` to `1 / (2*N)`. Forgetting to do this is a typical beginner's mistake. If you don't reset `p` for every simulation run, only the first simulation run will be meaningful - all the following simulation runs would start with `p` either being `0` or `1`, depending on the outcome of the very first simulation. Next, we call the `next_generation()`

function however often we need to, as long as `p` is neither `0` nor `1`. But what's this strange-looking operator `&&` doing there?

You've encountered the `do while` loop before. It says "do something while a certain condition is true". However, so far in this book, we have only ever encountered simple conditions, where we tested for either a boolean value to be true, or where we tested for a number to be smaller, equal to, or greater than a certain value. But sometimes, our conditions need to be a bit more complex. For example, we may require that multiple expressions are true. Or, we may require that at least one of multiple expressions is true. This is where so-called *logical operators* come in handy.

The most frequent logical operators in JavaScript are

```
&& // the logical and
|| // the logical or
!  // the logical not
```

The first, `&&`, is the *logical and*. If you need two expressions to be true at the same time, you should use this operator. For example, the condition

```
(p > 0 && p < 1)
```

is true if `p` is greater than `0` *and* `p` is smaller than `1`.

The second operator is the *logical or*. If you have multiple expressions and you need at least one of them to be true, you should use this operator like so, for example:

```
(p == 0 || p == 1)
```

Finally, there is the *logical not* operator. It flips a boolean value on its head, like this

```
(!something)
```

If `something` is true, this condition is false. If `something` is false, this condition is true.

Going back to the code example, we run the generations while `p` is greater than `0` and smaller than `1` - which means that neither of the two alleles has gone to fixation yet. Once A1 goes to fixation - at which point `p` will be `1` - or A2 goes to fixation - at which point `p` will be `0` - the loop stops. If A1 has fixed, we'll record that by increasing the counter `fixations_of_mutant` by `1`.

In the last loop in the code, we simply repeat this process a number of times (defined by the variable `simulations`) and then calculate and log the fraction of simulation runs in which `p` has gone from $\frac{1}{2N}$ to 1.

If you run this with the values that I'm using (i.e. `N = 100` and `10000` simulations), your results should look like this (reload the page a couple of times to get multiple values):

```
0.0054
0.0053
0.0049
0.0045
0.0049
```

etc.

Does this fit our prediction? Indeed it does. With `N=100`, $\frac{1}{2N}$ equals 0.005, and that's exactly what we get here.

Now go to the line in the code where you (re)set `p` to `1 / (2*N)`, and set it to, say, `0.1`, and run the simulation again. Your results should now look something like this:

```
0.0976
0.103
0.1009
0.969
0.1038
```

etc.

Again, the prediction is verified - in 10% of all simulations, the allele went from its initial frequency of 10% to 100% - i.e. the probability of the allele at `p = 0.1` to go to fixation is indeed 0.1.

By the way, you may have noticed that the simulations that started at `p = 0.1` were slower than the simulations that started at `p = 1 / (2*N)`. Why is that? Recall that we only run a simulation until `p` becomes `0` or `1`. When we start the simulation at `p = 1 / (2*N)`, it will generally go to `0` very rapidly - there is only one copy of the mutation in the population, and it will often be lost immediately, terminating the simulation. On the other hand, if we start the simulation at `p = 0.1`, then `p` will also go to `0` in most of the simulations (in 90%, as you now know), but getting there takes a little longer because you are starting at a much larger value for `p` than just `1 / (2*N)`.

Ok, so we have established that the probability of a new mutation going to fixation is $\frac{1}{2N}$. How many mutations are there per generation? Let's see - we have $2N$ alleles in the population, and the mutation rate per allele per generation is μ. Thus, there are $2N\mu$ new mutants per generation. And as we now know, each of those mutations will go to fixation with probability $\frac{1}{2N}$. Thus, each generation, $2N\mu$ times $\frac{1}{2N}$ mutations will go to fixation, which is μ.

Let's think about what this means. Let's assume μ is 0.0001. Thus, each generation, 0.0001 mutations will go to fixation. Per se, that is a weird statement. A mutation either goes to fixation or it doesn't. What we should rather say is that the probability that a mutation will go to fixation in a given generation is 0.0001. In the next generation, it is 0.0001 as well. And so on. In other words, every $\frac{1}{0.0001} = 10,000$ generations on average, a new mutation will become fixed in the population. Population geneticists refer to this idea as the replacement rate. In our case, we would say the replacement rate is 0.0001 per generation.

But have you noticed what happened here? The replacement rate corresponds exactly to the mutation rate μ! But where has the population size, N, gone? It turns out that the replacement rate is independent of the population size. Recall where we "lost" N: we said the replacement rate is the number of mutations per generation (which is $2N\mu$), times the probability of fixation of each of this mutations (which is $\frac{1}{2N}$):

$$\text{replacement rate} = 2N\mu * \frac{1}{2N} = \mu$$

This a remarkable result. Most of us would have guessed that population size must play a role. You may have thought that in large populations, there would be more replacement than in small populations, because there are simply more mutations. On the other hand, your intuition may have told you - correctly - that it's harder for mutations to go to fixation in larger populations, which may have led you to believe that the replacement rate would be lower. But as it turns out, the two effects - more mutations, but it's harder for them to make it all the way to fixation - are both dependent on N in such a way that N falls out of the equation.

We now have a good feeling for how often a new mutant allele fixes in a population. But *how long* does it take for a successful mutant allele to do that, on average? Before I give you the answer to that, I would like you to adapt the code above to find out. You already have more or less everything in place, and all you need to do is to keep track of the number of generations in those runs where `p` actually goes to `1`.

Tried it? Ok, here is how I would do it:

```
var N = 100;
var p;
var simulations = 100000;
var fixations_of_mutant = 0;
var total_generations = 0;

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;
```

```
}

function run_until_fixation() {
    p = 1 / (2*N);
    var generations = 0;
    do {
        next_generation();
        generations = generations + 1;
    } while (p > 0 && p < 1);
    if (p==1) {
        fixations_of_mutant = fixations_of_mutant + 1;
        total_generations = total_generations + generations;
    }
}

for (var i = 0; i < simulations; i = i + 1) {
    run_until_fixation();
}
console.log(total_generations / fixations_of_mutant);
```

As you can see, this code is almost identical to the one we used above. We've simply added a new global variable called `total_generations` that keeps track of the generations spent in those simulations where `p` goes to `1`. However, since we don't know from the outset which simulation will actually result in `p==1`, we will have to keep track of the generations in each simulation. To do that, we introduce the local variable `generations` in the function `run_until_fixation()`, and we increase it by `1` in every generation. If the simulation ends up resulting in `p ==1`, we will add that generation count to the global variable `total_generations`. Thus, at the end of all simulations runs, `total_generations` will be the sum of all generations from all simulations where `p` went to `1`, and all that is left to do is to divide that number by the number of those successful fixation simulations, `fixations_of_mutant`.

By the way, the other change I've made is to increase the number of simulations from `10000` to `100000` - the simulations still run within a few seconds on my laptop, but you may have to adjust these numbers if it's too slow for you.

If you run this a number of times, what are your results? With the sets of parameters shown in the code above, I get the following results (yours will vary, of course):

```
405.17102615694165
401.9110671936759
401.19109461966605
392.43110236220474
396.681640625
```

See the pattern? The answer is $4N$ - if a mutant allele manages to go to fixation in a population of size N, it will take $4N$ generations to do so on average.

And that concludes this chapter on mutation. We learned a number of really interesting things. The main points were the following:

- Mutation is the source of genetic variation.
- Mutation acts to increase genetic variation, while drift acts to reduce it. When the effect of drift is exactly offset by the effect of mutation, the genetic variation in a population will remain in equilibrium.
- If a population is small, and / or has a low mutation rate, the genetic variation at equilibrium will be low. Conversely, if a population is large, and / or has a high mutation rate, the genetic variation at equilibrium will be high.
- The chance that a new mutation will go to fixation is $\frac{1}{2N}$, its frequency. If it does go to fixation, the process will take $4N$ generations on average.
- The replacement rate is given by the mutation rate, and does not depend on the population size.
- We have also covered a few new JavaScript concepts and methods: the `do while` loop, `Math.floor()`, and logical operators.

In this chapter, we have written quite a bit of code, and we have developed great intuitions about mutation and drift with simple algebra. However, we haven't really had too much fun this time with visualizations. But you'll be more than compensated for that in the next chapter, when we are going to create spatial, individual-based models. When I teach this material in a class, this is usually when students are most excited about the resulting animations.

Ready for the next step? Let's roll!

5. Migration: Spatial Models

In this chapter, we'll be addressing an assumption that we introduced early on when we first introduced the Hardy-Weinberg principle. The assumption was that mating is completely random. In fact, in our code, we went one step further and didn't even assume that there were any individuals - we just simulated a gene pool, with alleles floating around almost like molecules in a gas. In addition to not having a concept of individuality, we also had no concept of space, and location.

Such a world obviously does not exist. Nevertheless, it has made it possible for us to develop a few key insights, using some very simple math. Once again, it would be easy to question those insights: if they were developed using assumptions that are clearly not met in the real world, then how could they possibly be relevant in the real world? It's intellectually healthy to keep asking this question. However, it's equally healthy to keep in mind the map metaphor that we developed earlier in the book. Just as we develop our models by making very strong simplifying assumptions, a map is a model of the world that makes very strong simplifying assumptions. The best map is not the one that is the most detailed - indeed, as we have seen, this would be a completely useless map. The best map is the one that simplifies the world as much as possible while still being a helpful guide. As Albert Einstein supposedly said, everything should be made as simple as possible, but no simpler. (If you are as wary about quote attributions as I am, you can read more on the origin of the quote here: http://quoteinvestigator.com/2011/05/13/einstein-simple/[10])

When mathematical models start to take space and location into account, they tend to get extremely complex - so complex that you often need very advanced math to even get started. On the other hand, expressing a spatial model in code is not that complicated. You already possess all the tools to do that, and this chapter will show you how to use these tools - and expand on them - to build spatial models. We'll be developing quite a bit of code - don't be intimated by it. The result is absolutely worth it.

As we recalled above, our assumption so far has been that mating is completely random. In reality, there are two major forces that violate that assumption. The first is sexual selection, by which we mean that some individuals are better at securing access to mates, through whatever means. For example, in some birds, the male's tail length is known to be sexually attractive. In a particularly intriguing study of long-tailed widowbirds, it has been shown that males in which the tail was experimentally elongated had higher mating success than other males. Sexual selection is a fascinating topic, but it's not the one we're concerned with here. Even in the absence of sexual selection, there is major force that prohibits random mating: space.

Animals can only migrate so far. Plants are sessile. Most organisms, in one way or another, are limited in their dispersal. Because of that, individuals tend to mate with other individuals nearby. It's easy to see that this must have enormous genetic consequences. The individuals who are nearby

[10] http://quoteinvestigator.com/2011/05/13/einstein-simple/

must themselves also have parents who have lived nearby, and thus even their grandparents used to be nearby. That is, there is fairly high chance that the individuals nearby a given individual are genetically much closer than any random individual anywhere in the population.

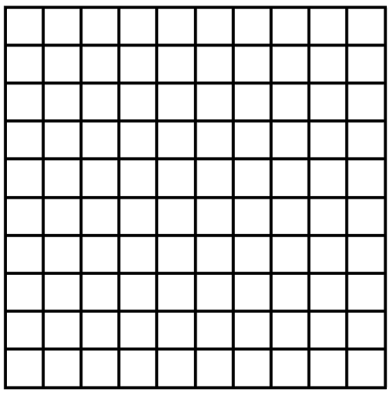
This phenomenon is called *inbreeding*. Inbreeding is mating between genetic relatives. The direct consequence of inbreeding is an increase in homozygosity, relative to what would be expected if mating were random.

In what follows, we will develop a rather sophisticated simulation of individuals mating with individuals nearby. In order to have a concept of "nearby", our simulation will need to incorporate the concept of space. If a simulation incorporates the concept of space, it is called a *spatial* simulation (or a spatial model). In addition, we are going to explicitly model individuals in our simulation. Of course, our individuals will be extremely simple, represented by just a diploid genotype with one locus (with two possible alleles). Nevertheless, we are going to represent each genotype individually, giving rise to a so-called *individual-based* simulation (or *agent-based* simulation). This is in contrast to *frequency-based* simulations like the ones we've developed above. In frequency-based simulations, we don't care about the fate of individuals, but rather about the overall frequency of a trait of interest (e.g. allele frequency).

What we're developing, then, is a spatial, individual-based model. This is going to be a fairly complex model, but I think it is the best way to demonstrate the effect of limited dispersal, and of inbreeding. Let's get started.

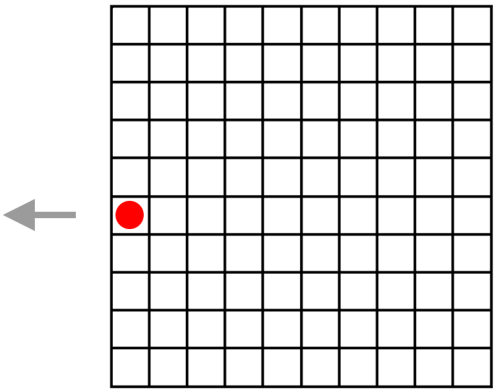
Because the model is a bit more sophisticated than previous models in this book, I will build it up step by step, rather than showing you the full code in advance and walk you through it. The first thing we have to think about is how we should model space. In the real world, your spatial position is given by coordinates. For example, as I'm writing this, my exact geographic coordinates are 37° 25' 48.1'' N, 122° 10' 18.2'' W. We could use exact coordinates like these, but that would seem like too much detail for a simple model (remember we always want to find the simplest way to model something). In fact, since this is just a conceptual model, not a real-world model, we should probably abandon the idea of actual geography altogether. We could for example say that our individuals live on an abstract sphere, and they have an x and y coordinate that places them on the sphere. This is better, but still quite complicated. How about we just flatten the sphere into a square, and just assume it's a grid? Let's do that. Indeed, this grid-based structure is very common in spatial models.

We could assume that our grid contains e.g. 10 times 10 cells, and each of the cells is inhabited by one individual, which would give us a population size of 100 individuals. Our world would thus look like this:

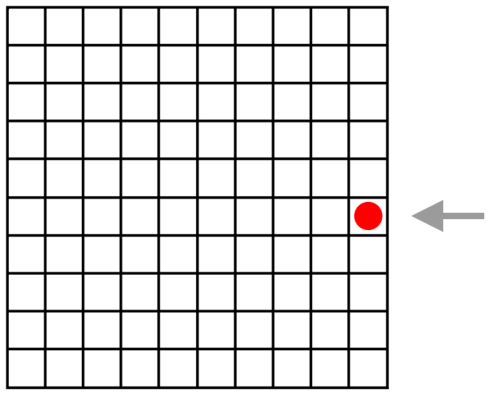


A typical assumption in these types of model is that there are no borders. If an individual would exit the grid on the left side, it would simply re-enter the grid on the right hand side, as if the grid were wrapped.

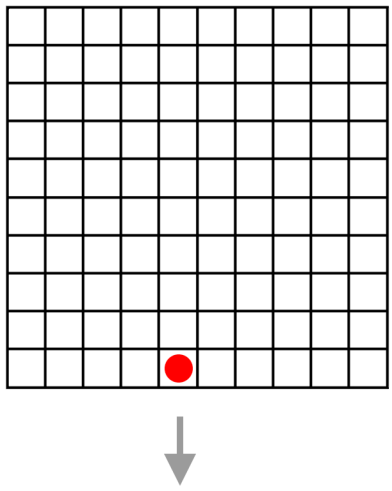
For example, if the individual denoted by the red dot below would move one cell to the left at time step t=1:



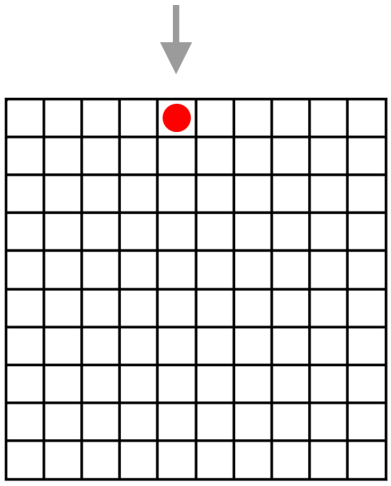
it would find itself at the following position at time step t=2:



Similarly, if the individual denoted by the red dot below would move one cell down at time step t=1:

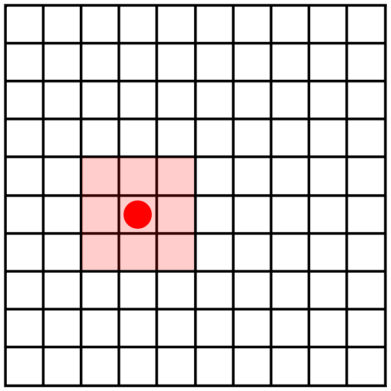


it would find itself at the following position at time step t=2:



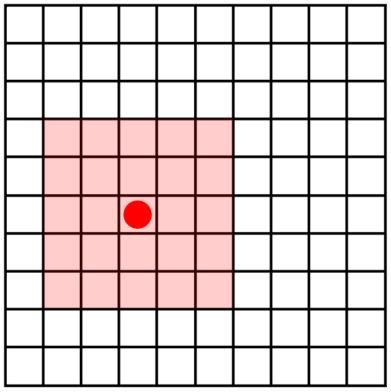
The great thing about this spatial structure is that we will be able to implement it with a data structure that we already know: a two-dimensional array.

We can now define a concept of neighborhood, or of "nearby". We don't need to simulate the individuals as moving entities - but their choice of mating should be restricted by their neighborhood. For example, we could define a distance measure, *d*, that defines the range of mates (i.e. the mating neighborhood). Concretely, *d* means that an individual can choose to mate with any other individual that is maximally *d* cells away from its own cell. Thus, for *d=1*, the neighborhood of the red individual would be given by the pink square:

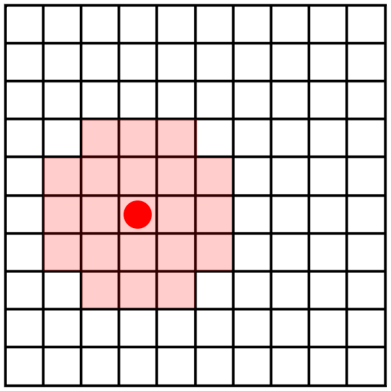


Note that we could exclude the current cell of the individual, if we wanted to. To keep things simple, we won't do this in our simulation, and simply assume that individuals can in principle mate with themselves.

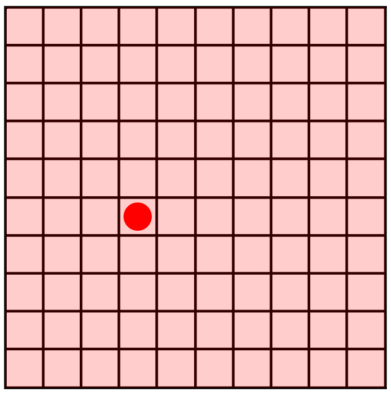
If *d=2*, the mating neighborhood would be as follows:



Note that it is up to you how you define the neighborhood. You could for example say that you won't include the uttermost diagonal cells in the example above, so that the neighborhood would look more like a circle:



In our code, we will stick to the simpler case of considering the entire rectangle. Finally, what would happen if d were half of the grid length, i.e. $d=5$? The neighborhood would now look like this:



This means that the neighborhood corresponds to the entire population, which, in the absence of sexual selection, is the equivalent of random mating (because any individual in the entire population could be randomly picked as a mate). Thus, by setting d to at least half of the grid length, we can simulate the case of completely random mating that doesn't care about spatial considerations. The model would still be spatial, but space would have no effect. Of course, if random mating was all you cared about, you wouldn't implement a spatial model in the first place! But when we do implement a spatial model, we can easily compare the case of highly localized processes such as local mating - which is what we're really interested in - to the case of fully random mating simply by setting a parameter to a specific value, rather than having to implement a second, separate (non-spatial) model.

Ok, let's go ahead and start implementing the grid, and the population. For our simulation, we want a bigger grid than in the example above - one that is a hundred by a hundred cells, and thus has space for a population of 10,000 individuals. We'll start by setting up a few variables:

```
var grid_length = 100;
var p = 0.5;
var grid = [];
var max_mating_distance = 1;
var A1A1 = 0;
var A1A2 = 0;
var A2A2 = 0;
var generation_counter = 0;
```

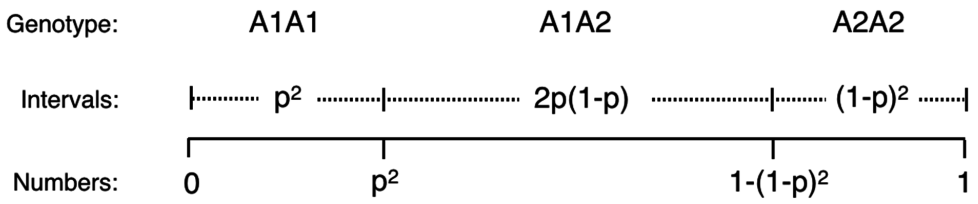
The `grid_length` variable defines the length of the grid, and since we are assuming a rectangular grid, the population size `N` will be the squared value of `grid_length`. As usual, we'll start with a given allele frequency (assuming once again A1 and A2 alleles) of `p = 0.5`. The `grid` array will be the data structure in which we store the individuals (i.e. their genotypes). The variable `max_mating_distance` corresponds to the maximum mating distance that we discussed above (where we referred to it as d). Next, the variables `a1a1`, `a1a2`, and `a2a2` are simple genotype counters that allow us to keep track

of the exact genotype numbers. Finally, `generation_counter` allows us to keep track of the number of generations.

Let's first populate the grid. To do that, we define the following function, and then call it immediately:

```
function init_grid() {
    for (var i = 0; i < grid_length; i = i + 1) {
        grid[i] = [];
        for (var ii = 0; ii < grid_length; ii = ii + 1) {
            var random_number = Math.random();
            if (random_number < p*p) {
                grid[i][ii] = "A1A1";
                a1a1 = a11a + 1;
            }
            else if (random_number > 1 - (1-p) * (1-p)) {
                grid[i][ii] = "A2A2";
                a2a2 = a2a2 + 1;
            }
            else {
                grid[i][ii] = "A1A2";
                a1a2 = a1a2 + 1;
            }
        }
    }
    console.log(a1a1, a1a2, a2a2);
}

init_grid();
```

What we're doing here is to iterate over the `grid` array, and add new empty arrays as elements - that is, each `grid[i]` will be a new empty array, and `grid` will thus be a two-dimensional array representing our spatial world. Then, we start another `for` loop that sets up the elements for each cell. We said we would have exactly one individual per cell, and the only individual trait we care about is the genotype, which are represented by the strings `"A1A1"`, `"A1A2"`, and `"A2A2"`. Since we know the desired initial `p`, the frequency of the A1 allele, we can easily calculate the desired initial Hardy-Weinberg genotype frequencies of A1A1, A1A2, and A2A2, which are p^2, $2p(1-p)$, and $(1-p)^2$. We can randomly assign these genotypes to cells by first drawing a random number between `0` and `1`, using our old friend `Math.random()`. How do we do this? Think of the values from `0` to `1` as a horizontal line. `Math.random()` will randomly pick a point on that line. When the value is smaller than p^2, then we're assigning the A1A1 genotype to the cell. If the value happens to be larger than $1-(1-p)^2$, then we're assigning the A2A2 genotype to the cell. Otherwise, we're assigning the A1A2 genotype to the cell. You can visual this line as follows:



Finally, we also keep count of how many A1A1, A1A2, and A2A2 genotypes we're generating - at this point, this is simply a sanity check, to make sure our code does what we want it to do.

Since we have a square grid of length 100, our population size is $N = 10{,}000$. With `p = 0.5`, we should get about 2,500 A1A1 genotypes, 5,000 A1A2 genotypes, and 2,500 A2A2 genotypes. Note that since we're creating the genotypes stochastically, we do not expect these numbers to be matched exactly. But the final tally should be close enough. If you now run this code, you should see something like:

```
2527 4975 2498
>
```

The numbers are close enough, so it looks like our code is working. The population is initialized on a grid, with 10,000 individuals at Hardy-Weinberg equilibrium.

After having initialized the population, let's run the generations. Right after the call to `init_grid()`, we'll add the following code to run 100 generations:

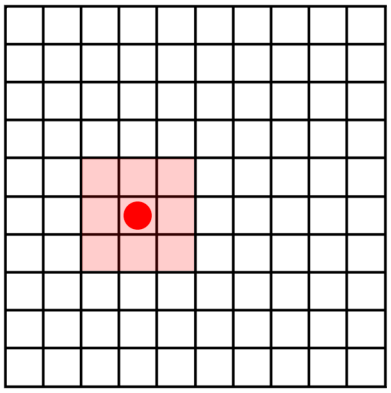
```
for (var i = 0; i < 100; i = i + 1) {
    run_generation();
}
```

Next, we need to go ahead and implement the function `run_generation()`.

Before we get to the code, let's think about what should happen during each generation. Basically, for each individual, we'll want to pick a random mating partner in the local neighborhood, and then create an offspring genotype based on the genotypes of the two parental individuals. The new offspring genotype should then simply replace the old focal individual. Simple enough.

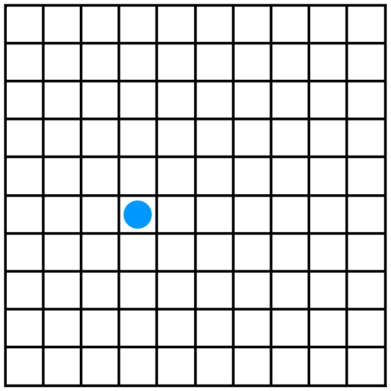
There is a potential pitfall in these spatial, individual-based models that would be easy to fall into if you implemented this by yourself for the first time. In principle, we want the individuals to reproduce - and the offspring individuals to replace the parental individuals - *at the exact same time*, in parallel. However, as you know by now, things don't generally happen at the exact same time in programming - everything occurs in sequence, because one statement will be executed after the other. However, what happens with one individual will depend on the individuals nearby, and if we update individuals sequentially, we would potentially run into strange dynamical issues. Let me explain this with an example.

Let's say we're iterating through all the individuals one at a time. Let's assume we're now at the individual that's colored in red (each cell on the grid contains one individual - they're not shown here for visual clarity):

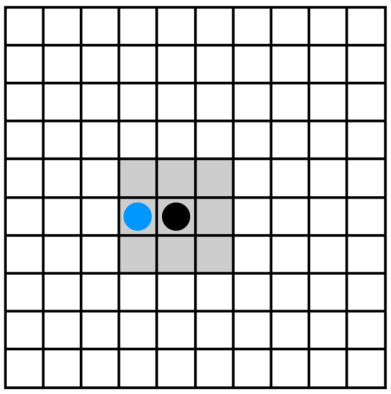


After this individual has picked a random mate in its local neighborhood, we generate an offspring genotype and replace the red individual.

Let's say, just for the sake of the argument, that the offspring individual is blue:



This seems fine at first sight, but consider what happens next. Our iterator will move on to the next individual, the individual on the right of the blue individual (colored in black):



As you can see, the local neighborhood of the black individual now contains a blue individual. However, if we had replaced all individuals in parallel at the same time, that individual left of the black individual would still have been the original red, as we've seen above.

This may seem like a small detail, but it could have pretty dramatic effects. We should aim to replace all individuals at once, not in a sequential fashion, because the sequential fashion could introduce strange dynamics.

Thus, the almost correct, but dangerous way to implement the `run_generation()` method would be as follows:

```
function run_generation() {
    for (var i = 0; i < grid_length; i = i + 1) {
        for (var ii = 0; ii < grid_length; ii = ii + 1) {
            var mating_partner = pick_mating_partner(i,ii);
            grid[i][ii] = get_offspring(grid[i][ii],mating_partner);
        }
    }
    generation_counter = generation_counter + 1;
}
```

What we are doing here is to iterate over the two-dimensional array, using two (nested) `for` loops. Inside the second `for` loop, the variables `i` and `ii` are the coordinates on the grid. Then, we're picking a mating partner and produce an offspring (note that we haven't implemented the functions `pick_mating_partner()` and `get_offspring yet()`). Then, we place the new offspring individual on the cell at the current location. Finally, we update the generation counter.

It's the replacement step (i.e. immediately replacing the current individual with the new offspring individual) that is problematic, for the reasons outlined above. But how can we solve this problem? It turns out that there is a simple solution. The answer is to place the offspring at the correct cells in a *temporary* placeholder grid, rather than on the real grid. Thus, a correct implementation of the function would look like this:

```
function run_generation() {
    var temp_grid = [];
    for (var i = 0; i < grid_length; i = i + 1) {
        temp_grid[i] = [];
        for (var ii = 0; ii < grid_length; ii = ii + 1) {
            var mating_partner = pick_mating_partner(i,ii);
            temp_grid[i][ii] = get_offspring(grid[i][ii],mating_partner);
        }
    }
    for (i = 0; i < grid_length; i = i + 1) {
        for (ii = 0; ii < grid_length; ii = ii + 1) {
            grid[i][ii] = temp_grid[i][ii];
        }
    }
    generation_counter = generation_counter + 1;
}
```

There are a few changes that we've made. First, we set up a temporary grid, using the local array `temp_grid`. Then, we're placing the new offspring on the corresponding cell on the temporary grid. Once the `temp_grid` is full, we iterate over the `grid` array once more, and copy all offspring individuals from `temp_grid` into `grid`.

Ok, let's go ahead an start implementing the two functions that we're calling from within `run_generation()`:`pick_mating_partner()`, and `get_offspring()`. Let's start with `pick_mating_partner()`.

We'll be using the neighborhood model that I described above. In formal terms, given the maximum distance *d*, the neighborhood of an individual at position *i, ii* is *j, jj*, such that

$$i - d \leq j \leq i + d$$

and equally

$$ii - d \leq jj \leq ii + d$$

For *d = 1*, the local neighborhood of a cell are thus the 8 cells surrounding the cell, and also the cell itself.

Because we need to pick a random cell of the neighborhood, we will simply pick a random `j` and a random `jj` in the allowable range. For example, if *d = 1*, *i = 5* and *j = 10*, then the allowable range for `j` would be 4 to 6 (both inclusive), and the allowable range for `jj` would be 9 to 11 (both inclusive).

The only out-of-the box method for random numbers in JavaScript is `Math.random()`, which returns a floating point number between 0 and 1 (recall that all numbers are floating point numbers in

JavaScript). What we need is a method that returns a random integer in a given range. Here's how we do that:

```
function get_random_int(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

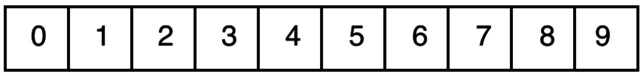
How does this work? First, we generate a random number between 0 and 1. However, the range of our desired values is not 1 (from 0 to 1), but something else. For example, if we want a random integer number between 7 (`min`) and 13 (`max`), the range would be 13 - 7 = 6. In order to increase the range of our random number generator, we must therefore multiply the result of `Math.random()` by 6. However, since we only want integer values, we will round the numbers down using `Math.floor()`. Because of this rounding, we would loose the largest value of the range (no value will ever be 6), which is why we will need to add 1 to the range. Finally, our range does not start at 0, as does `Math.random()`'s range, but at `min`, so we need to add `min` in order to move the range to its correct location on the number line.

Now, with this function in place, we can get a new coordinate `j`, given `i` and `d`, like so:

```
var j = get_random_int(i-d, i+d);
```

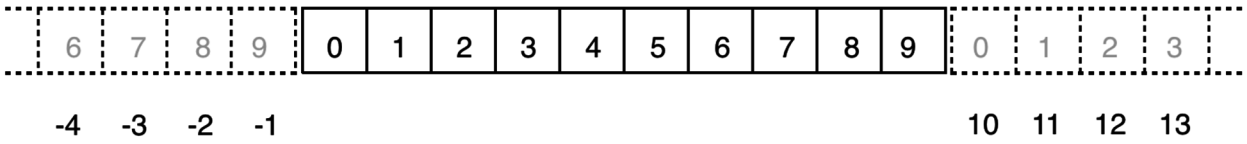
So far so good. But you may have noticed an issue. What happens if our cell is near the border of the grid? We don't want our spatial model to have any borders (if you exit on the right, you immediately re-enter on the left), but our array does not know about this idea yet. Its elements begin at index `0` and end at index `length-1`. Imagine that you are currently looking at the cell with an `i` coordinate of `0`. What if the random number that you draw is `-1`? In JavaScript, when you try to access an array element at index `-1`, you would get an error. Similarly, if you have 10 elements in an array, and you would try to access the element at index `10` (or higher), you would get an error too. This is an issue that we need to deal with.

Let's think about how we would implement this in a regular, one-dimensional array. Let's say we have an array with ten elements. We could visualize the array as follows (the numbers are the corresponding indexes):



Since the array has length 10, the index starts at `0` and ends at `9`. Now imagine you are at index `9`, and you would like to move one to the right. The index `10` does not exist, but since we want to re-enter the array from the left, we can just imagine placing a virtual copy of the array on the right hand side of the array, and the index `10` would simply correspond to the actual index `0`, `11` would

correspond to index `1`, `12` to index `2`, and so on. In the same spirit, if we are at index `0` and would like to move one to the left, we would be at index `-1`, which doesn't exist. Once again, we can simply imagine placing a virtual copy of the array on the left side of the array, and index `-1` would then correspond to the actual index `9`, `-2` would correspond to index `8`, `-3` to index `7`, and so on:



As you can see, the trick is simply to transform incorrect indexes. When the index is incorrect because it is *negative*, all you need to do is add the length of the array (`10` in our example), so that `-1` becomes `9`, `-2` becomes `8`, and so on. When the index is incorrect because it is *larger* than the maximum index, `9` (which is `length-1`), all you need to do is to subtract the length of the array, so that `10` becomes `0`, `11` becomes `1`, and so on.

It's probably a good idea to implement a method for this. Let's call the method `get_bounded_index()` - it takes an index and returns the transformed index (if a transformation is necessary at all):

```
function get_bounded_index(index) {
    var bounded_index = index;
    if (index < 0) {
        bounded_index = index + grid_length;
    }
    if (index >= grid_length) {
        bounded_index = index - grid_length;
    }
    return bounded_index;
}
```

Note that this solution is not very generic - it would fail if our indexes are either very negative (e.g. `-15`) or very large (e.g. `25`). But the indexes we'll be passing to this function will be in the range where the function works well, and we won't bother with more extreme cases for now.

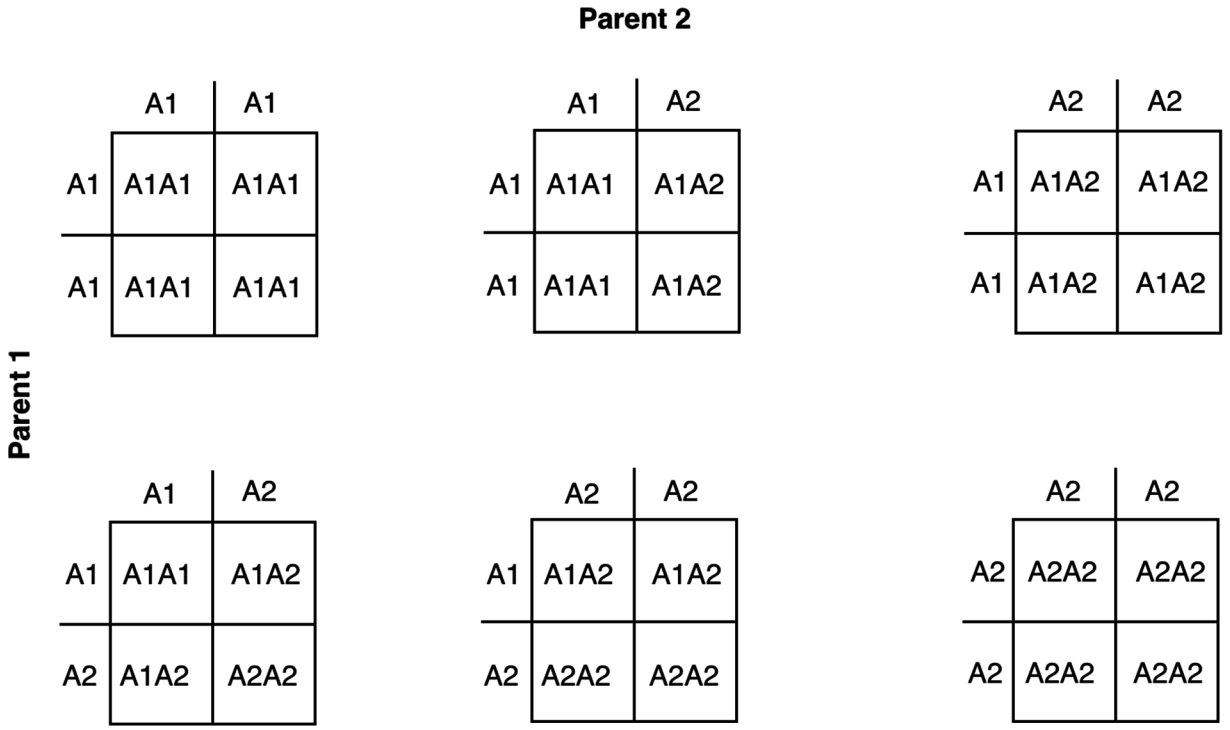
Now that we have all these helper functions in place, we can write the function `pick_mating_partner()`:

```
function pick_mating_partner(i, ii) {
    var j   = get_random_int(i-max_mating_distance, i+max_mating_distance);
    var jj  = get_random_int(ii-max_mating_distance, ii+max_mating_distance);
    j  = get_bounded_index(j);
    jj = get_bounded_index(jj);
    return grid[j][jj];
}
```

Well done! Now on to the other missing function, `get_offspring()`. Now that we have a mating partner (which is just a genotype, as you may recall), we need to create an offspring individual (again, simply a genotype). How do we do that? Mendel to rescue!

You may recall from biology class back in school that Gregor Mendel, an Austrian monk, discovered the basic laws of genetics around 150 years ago (even though nobody, not even he himself, realized it at that point). The first law of Mendel states that alleles segregate during the formation of gametes, and that each parent passes on one allele to the offspring individual. This is not news to us, of course - this concept has formed the basis of our assumptions. Nevertheless, we now need to implement it on an individual-level basis, rather than on a population-level basis as we have done before.

When we pick two parental individuals, there are six combinatorial possibilities: A1A1 and A1A1; A1A1 and A1A2; A1A1 and A2A2; A1A2 and A1A2; A1A2 and A2A2; A2A2 and A2A2. These possibilities have different outcomes with respect to offspring production by mendelian inheritance, as outlined in these diagrams of mendelian inheritance:



Of course you can flip parent 1 and parent 2, the outcome will be the same. Now, we will have to implement this in code. Looks like a bit of work, but the good news is, you'll only have to implement this once, and you'll be all set for all individuals and all generations. Also, keep in mind that the probability of each of the four boxes (i.e. each of the four possible offspring genotypes) is 25% for each paring of parental genotypes, as per the laws of Mendel. Here's how we would implement this:

```
function get_offspring(parent1, parent2) {
    var p1 = parent1;
    var p2 = parent2;
    if (p1 == "A1A1" && p2 == "A1A1") {
        return "A1A1";
    }
    else if ((p1 == "A1A1" && p2 == "A1A2") || (p1 == "A1A2" && p2 == "A1A1")) {
        if (Math.random() < 0.5) {
            return "A1A1";
        }
        else {
            return "A1A2";
        }
    }
    else if ((p1 == "A1A1" && p2 == "A2A2") || (p1 == "A2A2" && p2 == "A1A1")) {
```

```
        return "A1A2";
    }
    else if (p1 == "A1A2" && p2 == "A1A2") {
        var random_number = Math.random();
        if (random_number < 0.25) {
            return "A1A1";
        }
        else if (random_number > 0.75){
            return "A2A2";
        }
        else {
            return "A1A2";
        }
    }
    else if ((p1 == "A1A2" && p2 == "A2A2") || (p1 == "A2A2" && p2 == "A1A2")) {
        if (Math.random() < 0.5) {
            return "A1A2";
        }
        else {
            return "A2A2";
        }
    }
    else if (p2 == "A2A2" && p1 == "A2A2") {
        return "A2A2";
    }
}
```

This looks a little complicated, but if you've followed the diagrams of mendelian inheritance above, this method is actually extremely simple. It just checks for the genotypes of the parents, and then randomly returns one of the 4 possible offspring genotypes.

Great work so far! So, what would happen if we would put these functions together, and run the code? Well, the code would run all right, but we wouldn't know what's going on, since we're not logging any information. So let's go ahead and do this. In the function `run_generation()`, I'm going to add a call to another function, `print_data()`. In this function, we can log whatever we want. At the moment, all we care about is the corresponding numbers of genotypes.

First, let's add the call to function in the code:

```
function run_generation() {
    var temp_grid = [];
    …
    for (i = 0; i < grid_length; i = i + 1) {
      for (ii = 0; ii < grid_length; ii = ii + 1) {
        grid[i][ii] = temp_grid[i][ii];
      }
    }
    print_data();
    generation_counter = generation_counter + 1;
  }
```

Then, we can go ahead and implement the `print_data()` function:

```
function print_data() {
    a1a1 = 0;
    a1a2 = 0;
    a2a2 = 0;
    for (var i = 0; i < grid_length; i = i + 1) {
        for (var ii = 0; ii < grid_length; ii = ii + 1) {
            if (grid[i][ii] == "A1A1") {
                a1a1 = a1a1 + 1;
            }
            else if (grid[i][ii] == "A1A2") {
                a1a2 = a1a2 + 1;
            }
            else {
                a2a2 = a2a2 + 1;
            }
        }
    }
    console.log("generation " + generation_counter + ":");
    console.log(a1a1, a1a2, a2a2);
}
```

And now, if we run this, we should see something like this:

```
2457 4978 2565
generation 0:
2621 4739 2640
generation 1:
2699 4518 2783
generation 2:
2758 4441 2801
generation 3:
2783 4325 2892
generation 4:
2799 4299 2902
generation 5:
2829 4288 2883
generation 6:
2885 4180 2935
generation 7:
2856 4230 2914
generation 8:
2852 4126 3022
generation 9:
2825 4101 3074
generation 10:
```

As always with stochastic simulations, your numbers will look slightly different. The important part here however is that our code seems to be working fine. But just looking at these numbers isn't really all that insightful yet. After all, we have just implemented a stochastic, individual-based, spatial simulation. How about we visualize what is going on in our two-dimensional world?

Like before, I am going to give you the corresponding code to visualize the spatial dynamics for you to copy and paste. There is no point at this stage for us to dig into the rather complex visualization code. But it will be helpful to understand the basic principle behind it. What we would like to do is to take our two-dimensional array, and color-code the cells according to the genotype that lives on the cell. At each generation, we would like to update the visualization in order to reflect to new state of the population.

As it turns out, the visualization itself is much more computation-intensive than our simulation. Thus, the limiting factor will be the visualization. In order to have some control over the timing, I will introduce a new method in JavaScript.

First, you will need to copy and past the spatial drawing code - this time, it's made up of two function: `draw_grid()`, and `update_grid()`

As before when we used code to plot things visually, be sure to place this code within the `<body>`. And once again, you can copy and paste this code from the book website, www.natureincode.com[11].

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

```
function draw_grid(data,colors) {
        var width = 600;
        var height = 600;
        var grid_length = data.length;

        var svg = d3.select('body').append('svg')
                  .attr('width', width)
                  .attr('height', height);

        var rw = Math.floor(width/grid_length);
        var rh = Math.floor(height/grid_length);

        var g = svg.selectAll('g')
                        .data(data)
                        .enter()
                        .append('g')
                        .attr('transform', function (d, i) {
                          return 'translate(0, ' + (width/grid_length) * i + ')';
                        });

        g.selectAll('rect')
                        .data(function (d) {
                          return d;
                        })
                        .enter()
                        .append('rect')
                        .attr('x', function (d, i) {
                          return (width/grid_length) * i;
                        })
                        .attr('width', rw)
                        .attr('height', rh)
                        .attr('class',function(d) {
                          return d;
                        });
        if (!colors) {
            d3.selectAll(".A1A1").style("fill","#fff");
            d3.selectAll(".A1A2").style("fill","#2176c9");
            d3.selectAll(".A2A2").style("fill","#042029");
        }
        else {
                for (var i = 0; i < colors.length; i = i + 2) {
                        d3.selectAll("."+colors[i]).style("fill",colors[i+1]);
```

```
                }
        }
}

function update_grid(data,colors){
    var grid_length = data.length;
        d3.select('svg').selectAll('g')
                .data(data)
                .selectAll('rect')
                .data(function (d) {
                  return d;
                })
                .attr('class',function(d) {
                  return d;
                });
        if (!colors) {
            d3.selectAll(".A1A1").style("fill","#fff");
            d3.selectAll(".A1A2").style("fill","#2176c9");
            d3.selectAll(".A2A2").style("fill","#042029");
        }
        else {
                for (var i = 0; i < colors.length; i = i + 2) {
                        d3.selectAll("."+colors[i]).style("fill",colors[i+1]);
                }
        }
}
```

Because this the drawing code uses the D3.js library, also be sure to include the corresponding tag that loads the library. As before, I would suggest to put the library-loading `<script>` tag in the head:

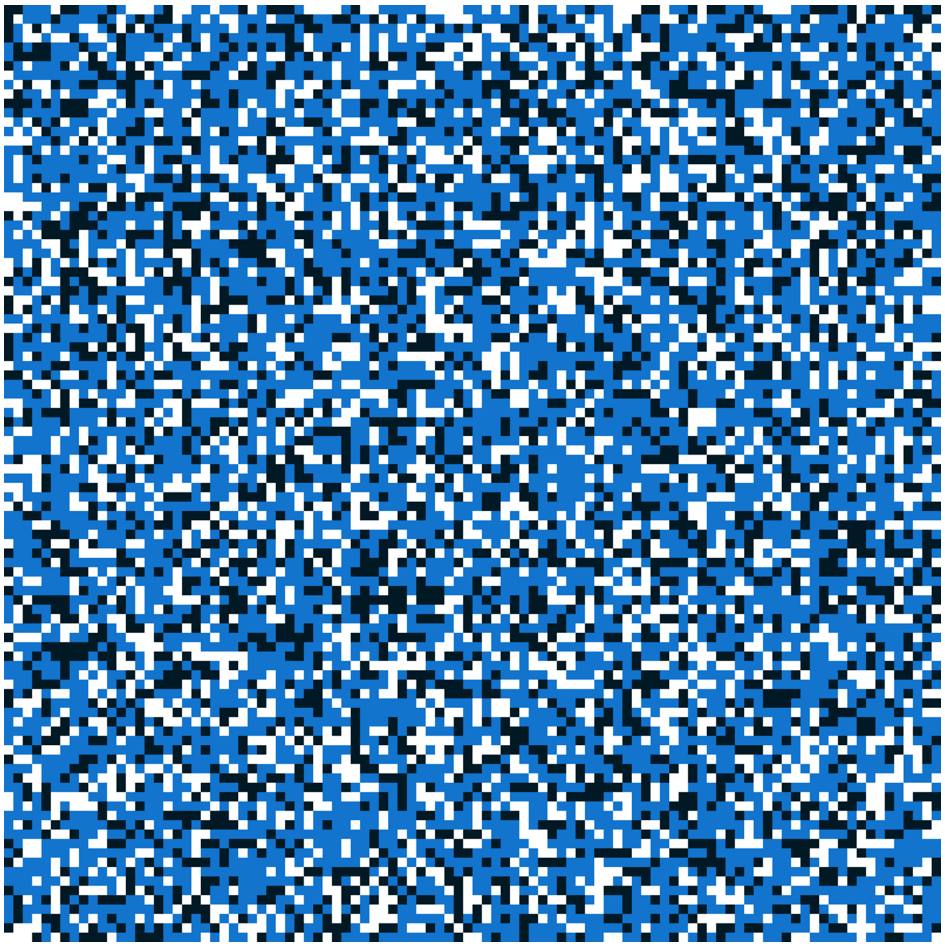
```
<!DOCTYPE html>
<html>
    <head>
        <title>Nature, In Code</title>
        <script src="http://d3js.org/d3.v4.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            // the drawing code and the simulation code
        </script>
    </body>
</html>
```

Great - everything is in place, let's put it to use!

First call the function `draw_grid()` right before you run your generation loop, i.e.:

```
draw_grid(grid);

for (var i = 0; i < 100; i = i + 1) {
    run_generation();
}
```

Now, if you run this code, you will see your simulations run in the logs like before - but in addition, you should also see something like this in the browser:



Wow! That is your initial population, color-coded to show A1A1 genotypes in white, A1A2 genotypes in bright blue, and A2A2 genotypes in dark blue. Isn't this amazing? But it's going to get much, much better. We're currently only showing the initial population, and as expected, the genotypes are randomly distributed in space. Now, let's go ahead and update the grid when the population is updated.

We are currently using the following code to run the generations:

```
for (var i = 0; i < 100; i = i + 1) {
    run_generation();
}
```

As mentioned above, the visualization will not be fast enough to keep in pace with the simulations - nor should it. Running one generation barely takes one millisecond, and our eye wouldn't be able to see 100 visual changes in 100 milliseconds anyways. So the code needs to be replaced with something that runs the generations - and updates the visualization of the grid - at specific time intervals. That code is as follows:

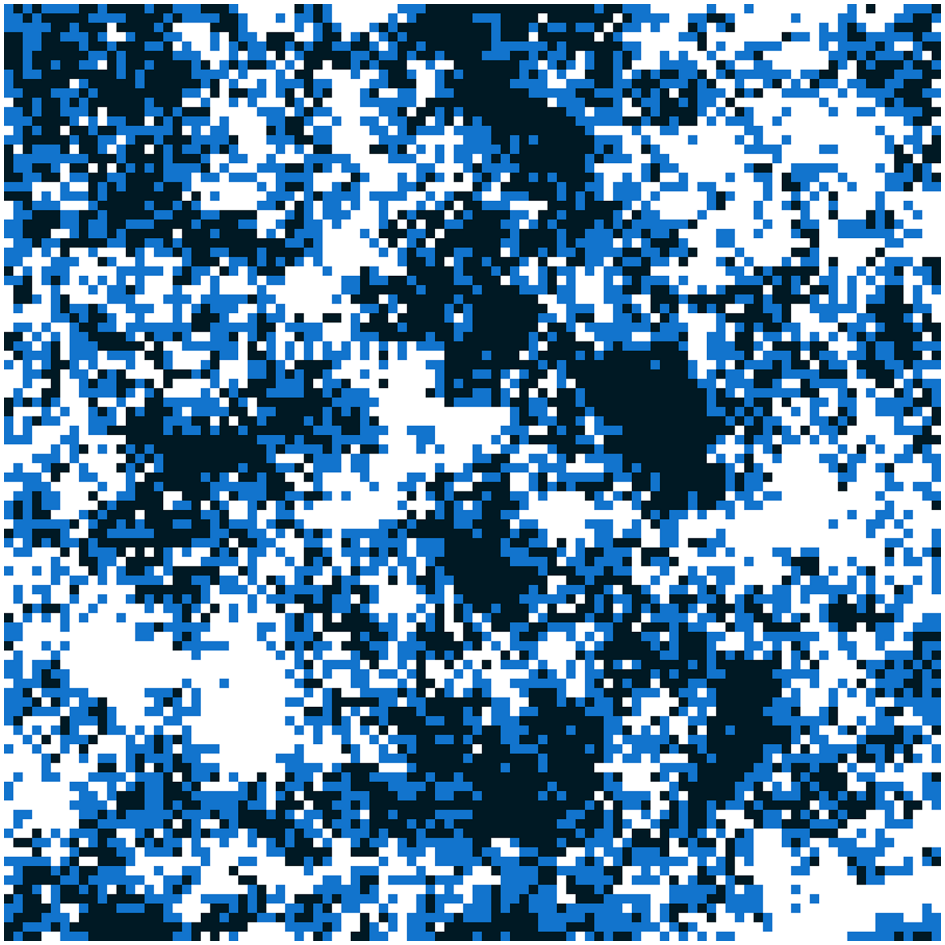
```
function simulate_and_visualize() {
    run_generation();
    update_grid(grid);
}
setInterval(simulate_and_visualize, 100);
```

Go ahead and remove the old `for` loop running the generations, and place the code above right after your call to `draw_grid()`. The method `setInterval()` is provided out of the box by JavaScript and allows you to execute any function, specified by the first parameter, at an interval of some number of milliseconds, defined by the second parameter. Thus,

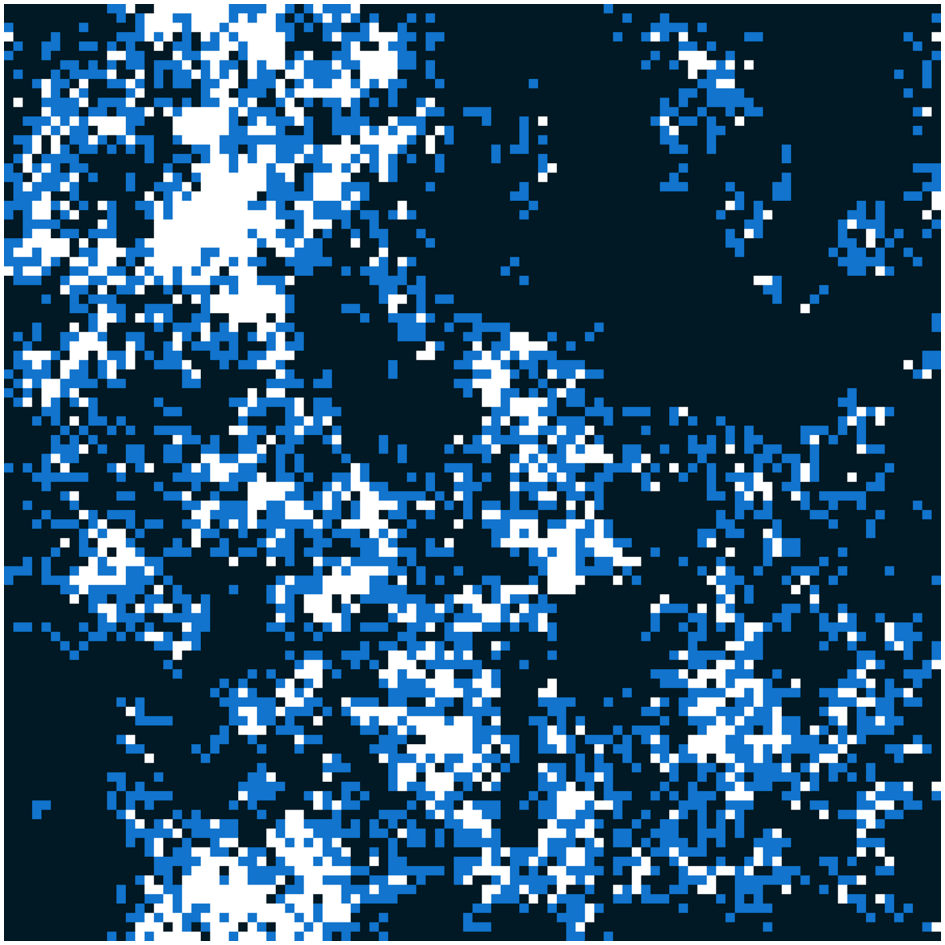
```
setInterval(simulate_and_visualize, 100);
```

will execute the function `simulate_and_visualize()` every `100` milliseconds.

Now save and reload the page, and - voilà! We can now see the population evolving! After the first few generations, your population may already look like this:

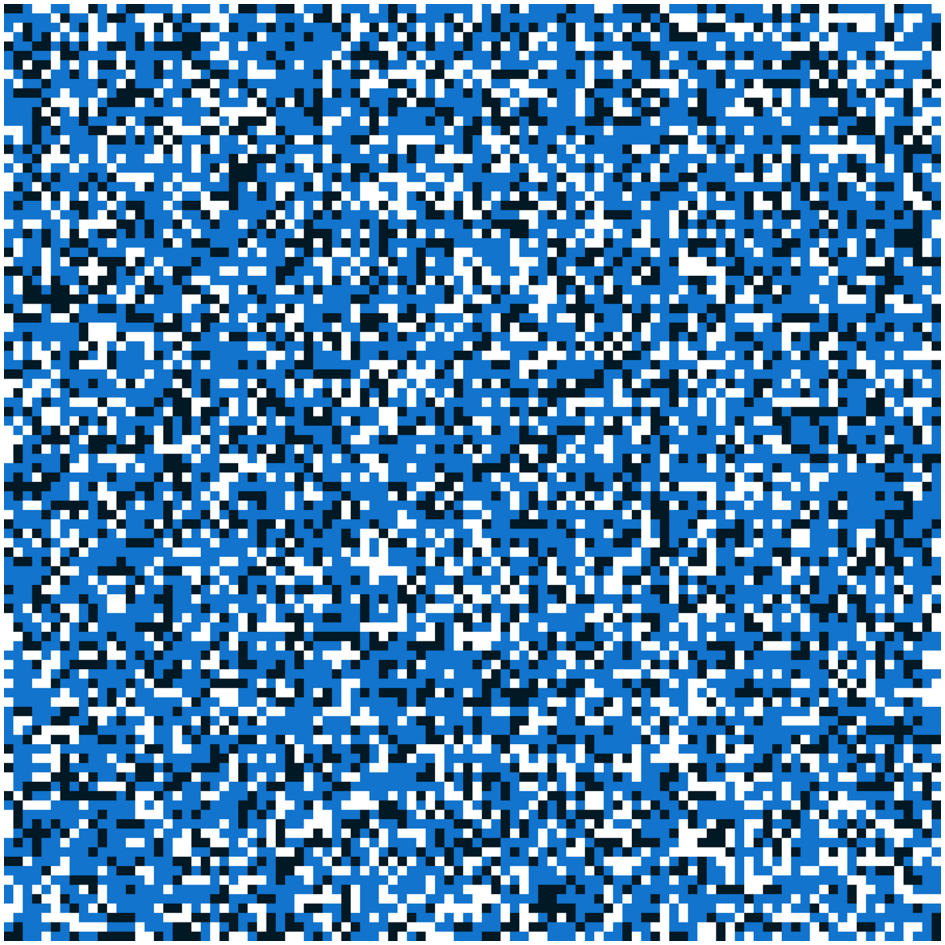


You can already see clear clusters of homozygotes (both A1A1, in white, and A2A2, in dark blue). Let the simulation run for a while and you'll see that the clusters will get bigger:



Why is that? Imagine you are an A1A1 genotype, and you find yourself in one of those A1A1 clusters. Your entire neighborhood is made up of A1A1 genotypes, and thus your offspring will be A1A1 too. The same is true for A2A2 genotypes in A2A2 clusters. However, this is not true for A1A2 genotypes. When two heterozygotes reproduce, there is only a 50% chance that the offspring will be a heterozygote too - that's why you won't find any big clusters in bright blue.

These clusters of homozygotes are the consequence of inbreeding, and there is inbreeding in this simulation because the mating neighborhood is spatially limited. What would happen if that were not the case? Well, we have the code in place to see what happens in that case: simply change the value of `max_mating_distance` to `50`, to simulate a "non-spatial" model, as discussed above (or rather a model with global mating, if you will). When you reload the page, you will see this:



The random pattern that we observed right upon population initialization remains (in different permutations) - no clusters emerge. There is simply no inbreeding, because mating is completely random over the entire population.

Quantifying Inbreeding

The effect of inbreeding due to limited spatial mating neighborhoods is strikingly obvious when plotted visually, as we have done here. Is there also a more formal way to measure the effect? It turns out there is, and we're going to develop it based on very simple mathematics first, after which we're going to measure it in our computational simulations.

Let's start again at the definition of inbreeding: the mating between genetically similar individuals. The consequence of mating between genetically similar individuals is an excess of homozygotes in the population - compared to the level of homozygosity that we would expect if mating was random. The flip side of this observation is that there will be a reduction in heterozygosity, compared to the level of heterozygosity that we would expect under random mating.

But what level of heterozygosity would we expect under random mating? Hardy-Weinberg to the

rescue! Given allele frequencies *p* and *q*, the expected heterozygosity is *2pq*. If we denote the expected heterozygosity with the label H_e, we can write

$$H_e = 2pq$$

The observed heterozygosity, H_o, is simply what we observe in the field (or in our simulations). Now, we said that inbreeding leads to less observed heterozygosity than we would expect. We can quantify this as follows

$$F = \frac{H_e - H_o}{H_e}$$

This quantity is called the *inbreeding coefficient*. In words, this quantity measures the difference between what you should have and what you actually do have, relative to the what you should have (in terms of heterozygosity). This type of equation is very useful anywhere where you have an expected value and an observed value.

Consider, for example, a payment of $100 that you are waiting to receive - in other words, $100 is the expected value. Now assume you are actually getting a payment of just $80. You will automatically make the following calculation in your head: "I should have gotten $100, but I only got $80. That's a difference of $20." Now consider a bank that expects a major monetary transfer of $1,000,000, but ends up receiving just $999,980. The bank makes the same calculation and ends up concluding, "we got $20 less than we expected". In both cases, the difference is equal in absolute terms - $20. However, we need to put that difference in the context of what we expected in the first place. In your case, you expected $100, but only got $80 - a 20% loss. The bank, on the other hand, expected $1,000,000 and got $999,980 - a mere 0.002% loss. In relative terms, what the bank got was very, very close to what it expected, but what you got was substantially reduced from what you expected. (I bet the bank would still try to get the $20 back, but I hope you get my point).

Equipped with this quantity, we can now go back to our code and measure the inbreeding coefficient.

We can simply extend the `print_data()` function like so:

```
function print_data() {
    A1A1 = 0;
    A1A2 = 0;
    A2A2 = 0;
    for (var i = 0; i < grid_length; i = i + 1) {
      for (var ii = 0; ii < grid_length; ii = ii + 1) {
        if (grid[i][ii] == "A1A1") {
          A1A1 = A1A1 + 1;
        }
        else if (grid[i][ii] == "A1A2") {
          A1A2 = A1A2 + 1;
```

```
        }
        else {
          A2A2 = A2A2 + 1;
        }
      }
    }
    console.log("generation " + generation_counter + ":");
    console.log(A1A1, A1A2, A2A2);
    var N = A1A1 + A1A2 + A2A2;
    var h_o = A1A2 / N;
    var p = ((2 * A1A1) + A1A2) / (2 * N);
    var h_e = 2 * p * (1-p);
    var F = (h_e - h_o) / h_e;
    console.log("F = " + F);
}
```

All of the added new code is at the end of the function. The observed heterozygosity H_o is simply the fraction of heterozygotes in the population. In order to calculate the expected heterozygosity H_e, we need to know the allele frequencies, which we can easily calculate based on the genotype counts (keep in mind that there are $2N$ alleles). Finally, we calculate F based on the definition that we introduced above.

First, let's run the code for the case of global mating - i.e. be sure to set the mating distance to `50`:

```
var max_mating_distance = 50;
```

Now, if you run the simulation, you will see something like this in the logs:

```
generation 1000:
3588 4764 1648
F = 0.009938027607017737
generation 1001:
3548 4794 1658
F = 0.005681965489241197
generation 1002:
3496 4840 1664
F = -0.0016164931304007998
generation 1003:
3467 4870 1663
F = -0.006764298409734039
generation 1004:
3425 4907 1668
F = -0.012661352144771675
generation 1005:
3474 4860 1666
F = -0.0048470846036585595
generation 1006:
3433 4917 1650
F = -0.015689678854602047
generation 1007:
3455 4837 1708
F = 0.0021453874170532967
```

As you can see, I've let the simulation run for 1,000 generations, and the A1A1 genotype is already much more frequent (around 3,500 individuals) than the A2A2 genotype (around 1,700 individuals) in this particular run. That's the normal pattern of genetic drift that we would expect: allele frequencies change over time in populations with finite size, simply because of random chance. However, the genotypes are almost perfectly in Hardy-Weinberg equilibrium, as evidenced by the *F* value hovering around 0 (which means that the observed heterozygosity is practically identical to the expected heterozygosity).

If you now switch back to local mating by setting the mating distance back to `1`:

```
var max_mating_distance = 1;
```

you should see a rather different outcome:

```
generation 100:
2890 3535 3575
F = 0.28966693969779705
generation 101:
2914 3515 3571
F = 0.29395235238956613
generation 102:
3015 3441 3544
F = 0.3098687297520253
generation 103:
2982 3450 3568
F = 0.30762240302709903
generation 104:
2993 3469 3538
F = 0.30413310134426785
generation 105:
3045 3347 3608
F = 0.3284714627106393
generation 106:
3041 3408 3551
F = 0.31662253521409195
generation 107:
3035 3375 3590
F = 0.32291440710247726
```

As you can see, already after 100 generations, the *F* value is around `0.3`, which corresponds to a reduction of heterozygosity of 30% relative to what we would expect! This confirms our visual suspicion that the heterozygosity is substantially reduced due to inbreeding. Only this time, we actually know by how much.

And with that, we conclude this chapter. Just to remind you of your achievement here - you have just developed an individual-based, spatial, and stochastic simulation of population genetics that actually runs in a browser! We have truly come a long way. What we learned was:

- How to implement a spatially explicit model.
- Mating is not always random - sexual selection or limited mating distances can substantially reduce the range of mates.
- When offspring doesn't disperse very far from the parental generation, there is a much higher chance of inbreeding.
- The effect of inbreeding is a reduction in heterozygosity, which can be measured by the inbreeding coefficient *F*.
- We have also learned about a new method, `setInterval()`, that allows you to execute a function repeatedly at certain time intervals, which can be very handy for visual simulations.

In the next chapter, we'll be looking at the queen of ideas in all of evolutionary biology; the idea that some people have called the best idea anyone's ever had; the idea that has made Charles Darwin the most famous natural scientist in history: natural selection.

An the best part? We won't just be talking about it - we'll be implementing natural selection in code. That's right - you will have evolution by natural selection running on your own computer. Let's go!

6. Natural Selection: The Best Idea Anyone's Ever Had

Natural selection is often equated with evolution. But as you now know, evolution is simply the change in allele frequencies. Natural selection is just one of the forces affecting these allele frequencies. Thus, natural selection is not evolution - rather, natural selection is a cause of evolution.

What sets natural selection apart from the other forces that shape evolution (drift, mutation, and migration) is that it is the only force capable of producing adaptive traits. An adaptive trait is a trait of an individual - such as the color of a flower, the shape of a hand, the efficiency of a lung - that makes the individual well adapted to the environment it lives in. But what does "well adapted" mean? It simply means that all else being equal, the individual is more likely to survive and produce offspring than it would without that specific trait.

Some of the most visually stunning examples of adaptation are those of animal mimicry. Mimicry is the imitation of another species. If you have a hard time seeing the leaf katydid (an insect) in the image below, it's because of the spectacular visual imitation of the plant that it rests on. In nature, you would probably have missed it entirely - just like most birds and other predators who would love nothing more than to eat the insect for lunch.

(Photo by Geoff Gallice[12] via Wikimedia Commons[13]). Darwin's insight was the following. In a world where natural resources are limited, there will be a struggle for survival and reproduction. Those individuals who are better adapted to survive, and to produce many offspring individuals, will contribute more offspring individuals to the next generation than those who are less well adapted. Because the offspring individuals inherit the traits that made their parents better adapted, those traits will increase in frequency over the generations.

[12] http://bit.ly/1vU1Y0W

[13] https://commons.wikimedia.org/wiki/File:Flickr_-_ggallice_-_Leaf_katydid_(1).jpg

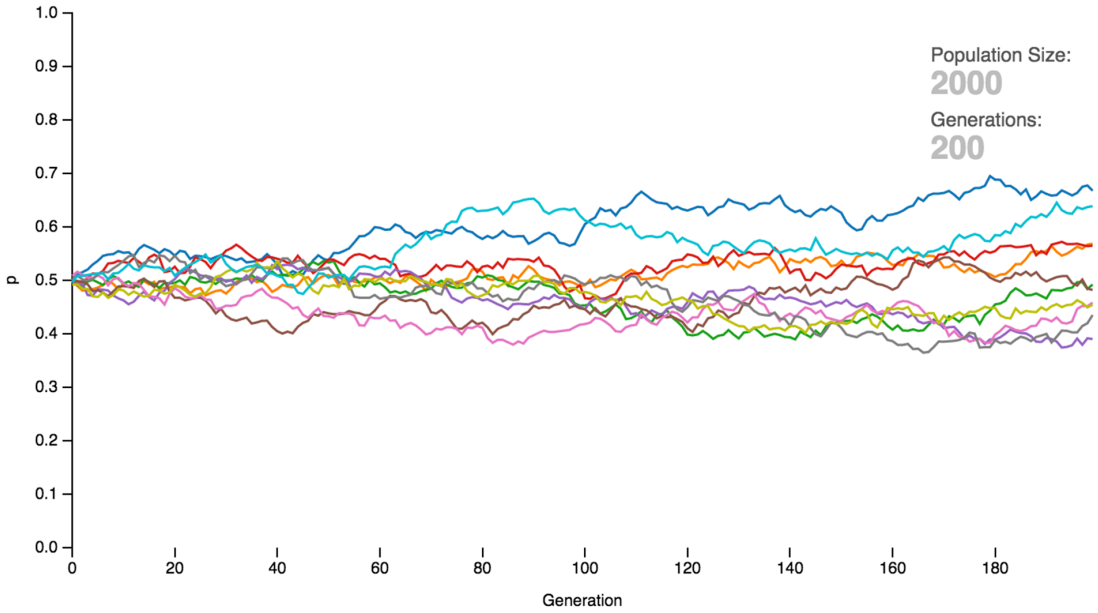
We have so far in this book avoided the issue of the struggle for survival and reproduction. In all of our models so far, we have assumed that in terms of the probabilities of making it to the next generation, all alleles, and all genotypes, are equal. *Natural selection occurs when that assumption is not true anymore.*

Another way of saying that all alleles and genotypes are equal with respect to making it into the next generation is to say that they have the same *evolutionary (or darwinian) fitness*. This is a key concept in evolutionary biology. Evolutionary fitness is the ability to survive and reproduce. Notice that fitness is not an absolute value, but rather a relative one. If a certain allele has a higher fitness than other alleles, that means that it is more likely than the other alleles to survive and make copies of itself (of course, alleles don't survive themselves, but the individuals carrying the alleles do). As a consequence of this, the allele is expected to increase in frequency in the population. This is evolution by natural selection. It is *evolution* because the allele frequencies change; and it is *by natural selection* because the allele frequencies change as a consequence of a higher fitness.

What we have assumed so far throughout the book is that all alleles and genotypes have the same fitness - without actually invoking the concept of fitness. Because all alleles or genotypes had the same fitness, they were *selectively neutral*. As we have now seen many times, just because alleles are selectively neutral does *not* mean that we don't get evolution as an outcome! Finite population size, mutations, and migration alone will lead to evolution, even if all alleles have the same fitness and are thus selectively neutral. The only difference when alleles do have differential fitness (i.e. not all of them have the exact same fitness) is that evolution becomes predictable in one regard: the alleles with higher fitness will become more common, and the ones with lower fitness will become less common. That's very different from the situation when alleles are selectively neutral - which alleles will go to fixation, and which ones will be lost, is completely due to chance, and thus impossible to predict.

I want to give you a visual example straight away. Remember in chapter 3, when we ran multiple simulations of genetic drift? We had our simple one locus, two alleles model, and we initiated the population at `p = 0.5`, which is to say half of the alleles were A1 and the other half A2, and then we observed the change of allele frequencies over time.

For a population of size `N = 2,000`, the typical graph for ten simulations looked something like this:



In some simulations, the frequency of A1 would go up, in others, it would go down, and we learned that in about half of simulations, A1 would eventually go to fixation, and in the other half, A2 would eventually go to fixation.

The crucial function in the code underlying this graph was the following:

```
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);
}
```

As you may recall, we are picking alleles randomly to produce the next generation, and we do this simply in accordance to their frequencies - allele A1 is picked with probability `p` (i.e. the frequency of A1), and allele A2 is picked with probability `1-p` (i.e. the frequency of A2).

Now let's introduce a seemingly minor change. Let's the change this line in the function:

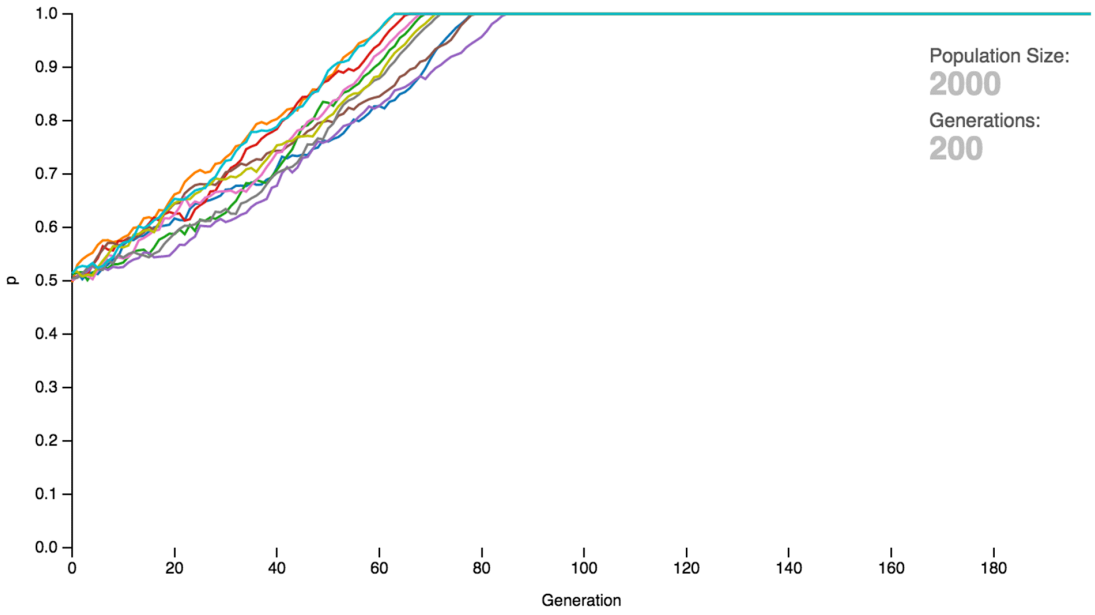
```
if (Math.random() <= p) {
```

to this

```
if (Math.random() <= p*1.01) {
```

What this means is that we're not picking allele A1 with a probability that corresponds to its frequency anymore. Rather, we are picking it with a probability that is a mere 1% higher than its frequency. Consequently, A2 is picked with a probability that is slightly less than its frequency.

Now let's take a look at the outcome with this minor change in the code:



Quite a different outcome! What is happening here? Natural selection, that's what's happening!

By increasing the probability of picking allele A1 to make it into the next generation from `p` to `1.01*p`, we are giving the allele a slight selective benefit over allele A2. In other words, allele A1 has a slightly higher fitness than allele A2.

This example has hopefully brought home the main point about fitness, natural selection, and its consequences. Because the example is borrowed from chapter 3, it uses a finite population size, and thus the effects of drift are still acting. Indeed, in real populations, both drift and natural selection are generally acting at the same time, and depending on the situation, one force can be much stronger than the other. In the example above, the dynamics were dominated by the force of natural selection.

However, you can still see that the lines zigzag a little bit - that's the force of chance that's still acting there.

This is important in the beginning, when a new mutation arises - even one that has a higher fitness than its competing alleles. We can take the code of the example above, and start the simulation at `p=0.01`, rather than at `p=0.5`. To do that, change the first line in the `simulation()` function from

```
p = 0.5;
```

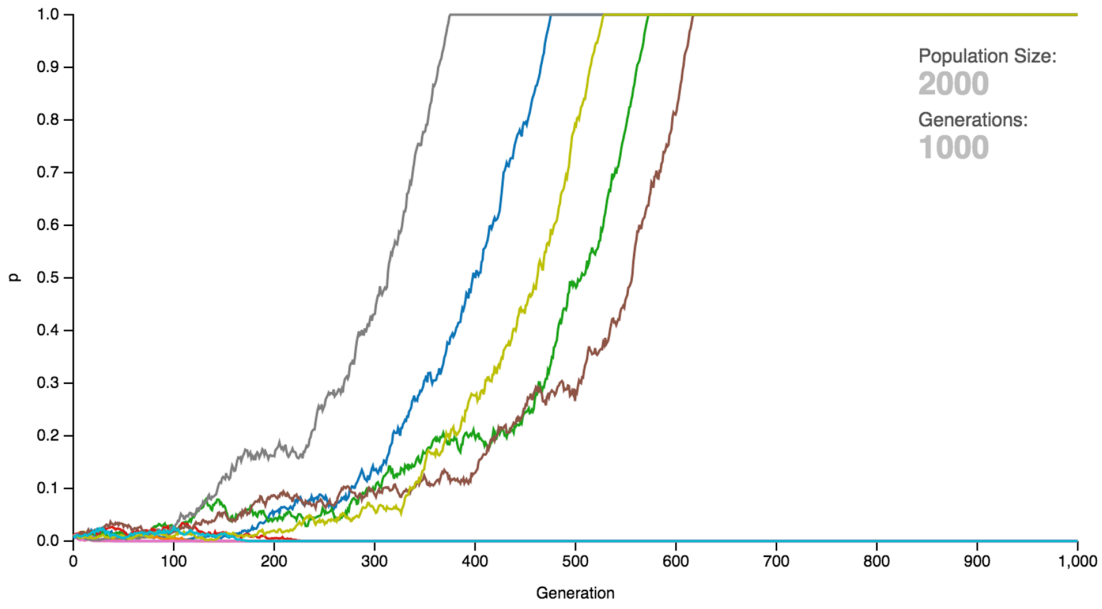
to

```
p = 0.01;
```

In order to see the dynamics in full, you will also need to increase the number of generations by setting `generations` to `1000`:

```
var generations = 1000;
```

When you run this code, you'll see the following dynamic play out:



As you can see, the allele will go to fixation in only a fraction of the simulations. In the other simulations, genetic drift has pushed it to extinction *before* natural selection could sweep the allele to fixation. And consider that in reality, mutations don't start at *p=0.01*, but at the generally lower frequency of $p = \frac{1}{2N}$, which means that the initial hump to overcome is even higher. It's sobering

to think about all the highly adaptive mutations that have occurred in the history of life, but that have never made it across the threshold where natural selection could take over and bring them to fixation.

With that in mind, in this chapter, we are going to investigate the effects of natural selection in isolation - that is, in populations of infinite size, where drift does not come into play at all.

We are going to start by formalizing the concept of fitness. We have mentioned above that an allele, or a genotype, has a fitness. Mathematically, we can capture the genotypic fitness as follows. We can simply assign the following fitness values to the three possible genotypes:

	A1A1	**A1A2**	**A2A2**
Fitness:	w_{11}	w_{12}	w_{22}

Let us assume, for simplicity, that the genotypes are at Hardy-Weinberg equilibrium. That is, their frequencies are:

	A1A1	**A1A2**	**A2A2**
Fitness:	w_{11}	w_{12}	w_{22}
Frequency at generation t:	p^2	$2pq$	q^2

Given these fitness values, and the frequencies in the current generation t, what are the genotype frequencies in the next generation t+1? Here they are:

	A1A1	**A1A2**	**A2A2**
Fitness:	w_{11}	w_{12}	w_{22}
Frequency at generation t:	p^2	$2pq$	q^2
Frequency at generation t+1:	$p^2 w_{11}/\bar{w}$	$2pqw_{12}/\bar{w}$	$q^2 w_{22}/\bar{w}$

where $\bar{w}$ is the average fitness in the population, which can be calculated as

$$\bar{w} = p^2 w_{11} + 2pqw_{12} + q^2 w_{22}$$

These few formulas form the basis for a lot that we will be doing in this chapter, so I want to make sure that we understand them well. Let's go through an example with actual numbers.

Assume the genotype frequencies are 20%, 50% and 30% for genotypes A1A1, A1A2, A2A2, respectively. In other words, $p^2 = 0.2$, $2pq = 0.5$, and $q^2 = 0.3$. Let's also assume that the fitness values of the three genotypes are $w_{11} = 1$, $w_{12} = 1.2$, and $w_{22} = 1.5$. That is, A2A2 has the highest fitness, A1A2 has the second highest fitness, and A1A1 has the lowest fitness.

Now, if we would simply multiply the fitness values with the frequencies, we would get the following frequencies:

$$p^2 w_{11} = 1 * 0.2 = 0.2$$

$$2pqw_{12} = 1.2 * 0.5 = 0.6$$

$$q^2w_{22} = 1.5 * 0.3 = 0.45$$

The problem is that these new genotype frequencies (0.2, 0.6, and 0.45) don't add up to 1, but of course they have to. So what we need to do is to normalize these values, so that they do indeed add up to 1 again. Normalizing here means to reduce each of the values by the same factor in such a way that the numbers add up to 1. It's very easy to do that: simply add the three numbers up (0.2 + 0.6 + 0.45 = 1.25), and then divide each of the three numbers by that sum. We call the sum of the three numbers the average fitness in the population, $\bar{w}$, and we get the normalized frequencies by dividing by it:

$$\frac{p^2w_{11}}{\bar{w}} = \frac{0.2}{1.25} = 0.16$$

$$\frac{2pqw_{12}}{\bar{w}} = \frac{0.6}{1.25} = 0.48$$

$$\frac{q^2w_{22}}{\bar{w}} = \frac{0.45}{1.25} = 0.36$$

These numbers were rounded a bit - but I hope you get the idea. So overall, here is our example, in numbers:

	A1A1	**A1A2**	**A2A2**
Fitness:	1	1.2	1.5
Frequency at generation t:	0.2	0.5	0.3
Frequency at generation t+1:	0.16	0.48	0.36

This is quite interesting. As you can see, the frequency of the A1A1 genotype has gone down. That happened because it had the lowest fitness, relative to the others. However, the frequency of genotype A1A2 has also gone down. The only genotype that has not gone down in frequency is genotype A2A2 - indeed, its frequency has gone up.

It's important to note that the absolute fitness values don't matter. Instead of 1, 1.2 and 1.5, we could have chosen 10, 12, and 15, and we would have obtained the exact same results. What matters is only how the fitness values compare to each other - that is, their relative value.

What would happen if all three fitness values were exactly the same? (Think about this before reading on.) Recall that the average fitness is

$$\bar{w} = p^2w_{11} + 2pqw_{12} + q^2w_{22}$$

If $w_{11} = w_{12} = w_{22}$ then

$$\bar{w} = p^2 w_{11} + 2pqw_{11} + q^2 w_{11}$$

from which we can factor out w_{11} and obtain

$$\bar{w} = w_{11}(p^2 + 2pq + q^2)$$

Since $p^2 + 2pq + q^2 = 1$, we get

$$\bar{w} = w_{11}$$

Thus, our next generation table will be:

	A1A1	**A1A2**	**A2A2**
Fitness:	w_{11}	w_{11}	w_{11}
Frequency at generation t:	p^2	$2pq$	q^2
Frequency at generation t+1:	$p^2 w_{11}/w_{11} = p^2$	$2pqw_{11}/w_{11} = 2pq$	$q^2 w_{11}/w_{11} = q^2$

Our frequencies don't change at all! Since all genotypes have the same fitness, there is no natural selection - and we are back in the Hardy-Weinberg world. This is an important insight - in order to have evolution by natural selection, there needs to be something for natural selection to act on. That something is a difference in fitness, or *variance in fitness.*

Given the three fitness values w_{11}, w_{12}, and w_{22}, under what circumstances will an allele increase or decrease in frequency? Because we know what genotype frequencies are at generation *t+1*, we can easily calculate the frequency of allele A1 at that generation:

$$p' = \frac{p^2 w_{11}}{\bar{w}} + \frac{2pqw_{12}}{2\bar{w}}$$

which is simply adding the frequency of genotype A1A1 to half of the genotype frequency of A1A2. We can simplify this a bit by removing the 2's on the right hand side of the equation:

$$p' = \frac{p^2 w_{11}}{\bar{w}} + \frac{pqw_{12}}{\bar{w}}$$

which can be rewritten as

$$p' = \frac{p^2 w_{11} + pqw_{12}}{\bar{w}}$$

The difference per generation is simply what we have at generation *t+1*, minus what we had at generation *t*:

$$\Delta p = p' - p$$

Having just established the equation for *p'*, we can rewrite this to get

$$\Delta p = \frac{p^2 w_{11} + pq w_{12}}{\bar{w}} - p$$

If we replace $\bar{w}$ with $p^2 w_{11} + 2pq w_{12} + q^2 w_{22}$, we will get quite a beast of an equation, which can be simplified to this equation:

$$\Delta p = \frac{pq(p(w_{11} - w_{12}) + q(w_{12} - w_{22}))}{p^2 w_{11} + 2pq w_{12} + q^2 w_{22}}$$

Now, I am the first to grant you that this is not a very memorable equation. However, it is the fundamental equation of evolution by natural selection. What it describes is the change in allele frequency, given the current allele frequencies, and the three fitness values. As we will see soon, this equation can explain a whole range of interesting phenomena.

If Δp is positive, the allele will increase in frequency. If Δp is negative, the allele will decrease in frequency. Note that in this world of an infinite population size, there is no randomness anymore - everything is determined. That's why these models are called *deterministic* models (as opposed to the stochastic models that we worked with in some of the chapters above, where randomness played a big role).

Now you may think "fine - if an allele has a high fitness, it will increase in frequency; if it has a low fitness, it will decrease in frequency. So far so good. But is that all there is? Because it sounds almost too simple".

Half of the answer is yes - natural selection really is quite simple. An allele can confer an advantage or a disadvantage to the individuals carrying the allele. If it is an advantage, then on average, these individuals will survive better, and reproduce more, than the individuals without that allele. Because of genetic inheritance (offspring have the genes of their parents), that allele will become more frequent in the population, eventually being the only allele in the population. This is evolution by natural selection.

The other half of the answer is no - there is one aspect that is incredibly important when thinking about natural selection. That aspect is that fitness must be assigned to a genotype, not to an allele. Each diploid individual has two alleles, and in the case where we have two different alleles, there are three different genotypes: A1A1, A1A2, and A2A2. The different genotypes may have different fitness values. Sometimes, the two ways of assigning fitness (either to the allele or to the genotype) are equivalent. For example, if A1A1 has the lowest fitness, A1A2 has a higher fitness than A1A1, and A2A2 has the highest fitness of them all, then you might as well say that the allele A2 confers a fitness benefit: the more A2s you have, so to speak, the fitter you are:

$$w_{11} < w_{12} < w_{22}$$

In that case, natural selection is very easy to predict: the allele A2 will spread in the population. However, there are two cases where things are more interesting. The first case is when the fitness of the A1A2 genotype is higher than the other two fitnesses. The second case is when the fitness of the A1A2 genotype is lower than the other two fitnesses.

Let us find a good way to formalize these three cases. First, we start by arbitrarily setting the fitness value of A1A1 to 1. As we have seen before, the absolute values of fitness don't matter - what matters is the relative fitness values, i.e. how they compare to each other. Thus, we can reduce the complexity of the system by simply setting one of them to a fixed value. The question is now, how do the other two fitness values compare to 1?

Let's simplify this a bit further and say that no matter what, we assume that the A2A2 fitness is lower than 1, i.e. the genotype A2A2 is always less fit than the genotype A1A1. We could of course also assume that it is always higher, but you might as well just replace A1 with A2, and you would get the same system. It really doesn't matter in this simple model: we are simply saying that one of the homozygotes has fitness 1, and that the other homozygote has a fitness that is lower than 1. It's up to you to decide which of the homozygotes is the one with fitness 1. For the purpose of our discussion here, I'm setting A1A1 to be the genotype with fitness value 1, and A2A2 to be the genotype with the lower fitness.

But lower by how much? For this, let's introduce a new, important parameter: the *selection coefficient, s*. This coefficient defines by how much the fitness of A2A2 is reduced, compared to the fitness of A1A1. In mathematical terms, $w_{22} = 1 - s$.

What about the fitness of the heterozygote, A1A2? That is where things get interesting. To capture the fitness for A1A2, we will now introduce another new parameter: the *heterozygous effect, h*. We simply define the fitness of A1A2 to be $w_{22} = 1 - hs$. Depending on the value of *h*, we can get all three different types of selection. Let me visualize this so that it becomes easier to understand.

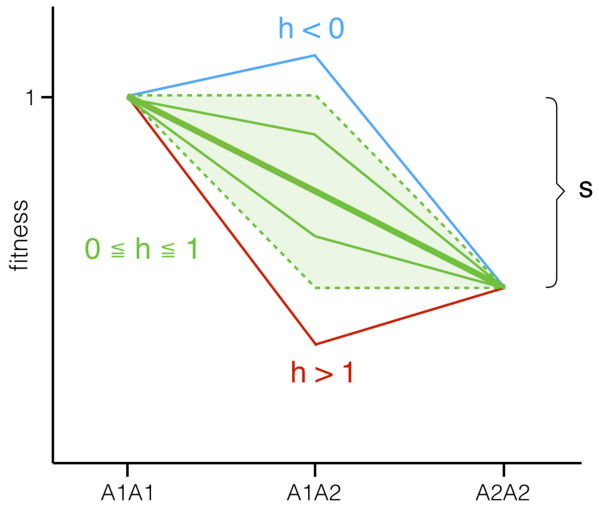
First, our new fitness table is as follows:

	A1A1	**A1A2**	**A2A2**
Fitness:	1	$1 - hs$	$1 - s$

As you can see, we now need to work with only two values (*h* and *s*), rather than with three (w_{11}, w_{12} and w_{22}). The first effect of that reduction in parameters is that it makes our equation for Δp a little simpler:

$$\Delta p = \frac{pqs(ph + q(1-h))}{(1 - 2pqhs - q2s)}$$

If we would plot the genotypes fitnesses visually, we would get one of the following graphs:



Any line in the green area, where *h* is some value between 0 and 1 (including 0 and 1), represents the case where the fitness of the heterozygote is an intermediate value between the fitness of the two homozygotes, or equal to one of them (think *fitness slope*). This type of selection is called *directional selection.*

The blue line, or really any line you can think of above the green area, represents the case where the fitness of the heterozygote is higher than the fitness of both homozygotes (think *fitness peak*). This type of selection is called *balancing selection.*

The red line, or really any line you can think of below the green area, represents the case where the fitness of the heterozygote is lower than the fitness of both homozygotes (think *fitness valley*). This type of selection is called *disruptive selection.*

As we'll see below, these three types of fitness "landscapes" have very different dynamics.

Directional Selection

Let's start with directional selection. Let's go ahead and plot Δp, the change in the frequency of allele A1, as a function of p (i.e. the allele's current frequency). We've used very similar code before, and we're also reusing our `draw_line_chart()` function for the plotting. All that needs to be implemented here is the actual equation:

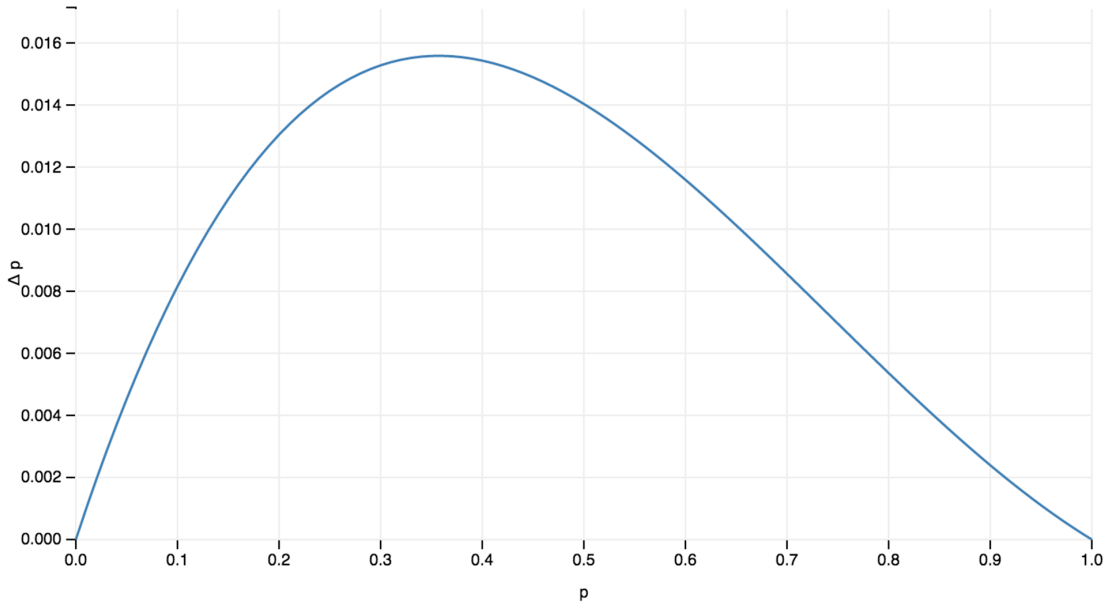
```
var s = 0.1;
var h = 0.2;

var data=[];
var x_max = 1;

for (var i = 0; i <= x_max + 0.005; i = i + 0.01) {
    var p = i;
    var q = 1-p;
    var delta_p = (p*q*s * ( p*h + q*(1-h))) / (1 - 2*p*q*h*s -q*2*s);
    data.push(delta_p);
}

draw_line_chart(data,"p","\u0394 p",[],x_max,true);
```

(Note that '\u0394' stands for Δ).

You may have noticed that I have passed a sixth parameter to the function `draw_line_chart()`. As indicated in chapter 3 where I introduced it, the function has been defined with 6 arguments. I simply never supplied values for all 6 arguments, until now. If a value is not supplied for an argument, the corresponding local variable in the function will be `undefined`. Thankfully, the function has always been able to deal with this situation.

So what is this argument for? The argument name for the parameter in the function is called `y_max_flex` - if it is not `true` (which is the case if you don't pass 6 values to the function in the first place), then the y axis will span from 0 to 1 (check some of the earlier figures that we generated with this function to verify that this is indeed the case). On the other hand, if it is `true`, the y axis will span from 0 to a little bit more than the maximum value in the data set - making sure that the entire vertical space in the figure is utilized.

Setting the values `s` and `h` to `0.1` and `0.2`, respectively, we get the following plot:



Let's examine this plot. The first, and most important thing to notice is that Δp is always positive (except when p is either 0 or 1). What that means for the allele A1 is that unless it is lost (i.e. $p = 0$) or fixed (i.e. $p = 1$), its frequency is going to be higher in the next generation. This is why this type of selection is called directional selection: the allele frequency knows only one direction, and that is up in this case. The A1 allele will go to fixation.

The second thing to notice is that Δp is largest around $p = 0.35$. In other words, when the frequency of the allele A1 is around one third, the per generation increase will be maximal. To understand this a little better, let's go ahead an run a (deterministic) simulation, starting a `p = 0.01`.

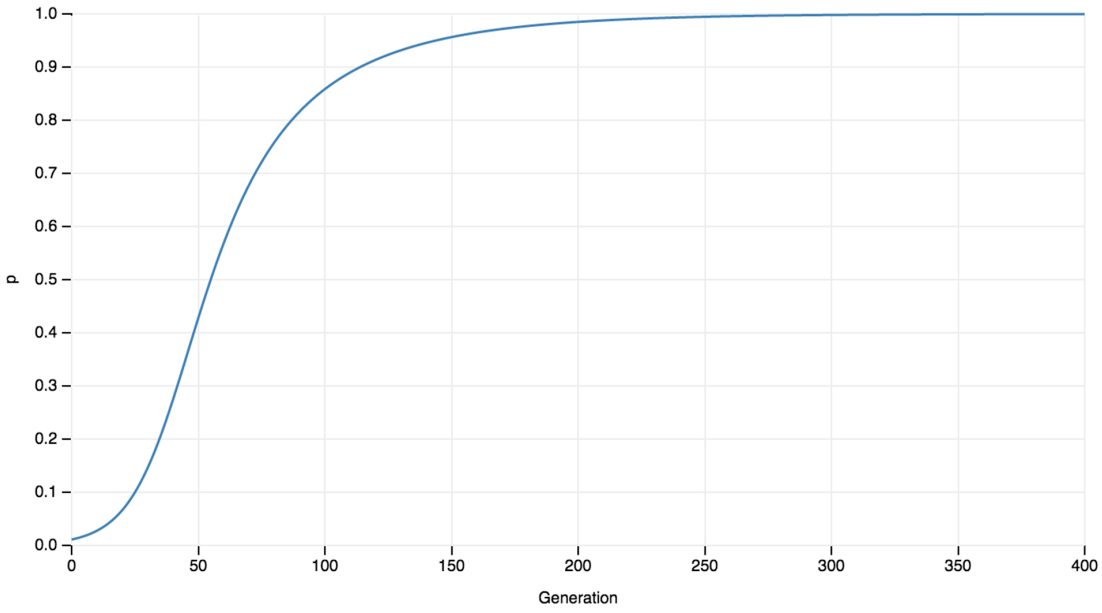
I'm going to be using the following code:

```
var s = 0.1;
var h = 0.2;
var p = 0.01;

var data=[];
var generations = 400;

for (var i = 0; i < generations; i = i + 1) {
    var q = 1-p;
    var delta_p = (p*q*s * ( p*h + q*(1-h))) / (1 - 2*p*q*h*s -q*2*s);
    p = p + delta_p;
    data.push(p);
}
```

```
draw_line_chart(data,"Generation","p",[],generations);
```

This code is very similar to the one we used above. Given `s`, `h`, and `p`, we simply run a number of generations where we calculate Δp, and then add that to the previous value of `p` in order to get the new value of `p`.

What we will see is the following plot:



As you can see, p starts increasing right from the start, even if a little slow. Then, around generation 25, when p is roughly 0.1, the increase starts accelerating noticeably. We know from the previous plot that the per generation increase is maximal at around $p = 0.35$, after which the increase starts slowing down again, leading to the classical observed S-shaped curve.

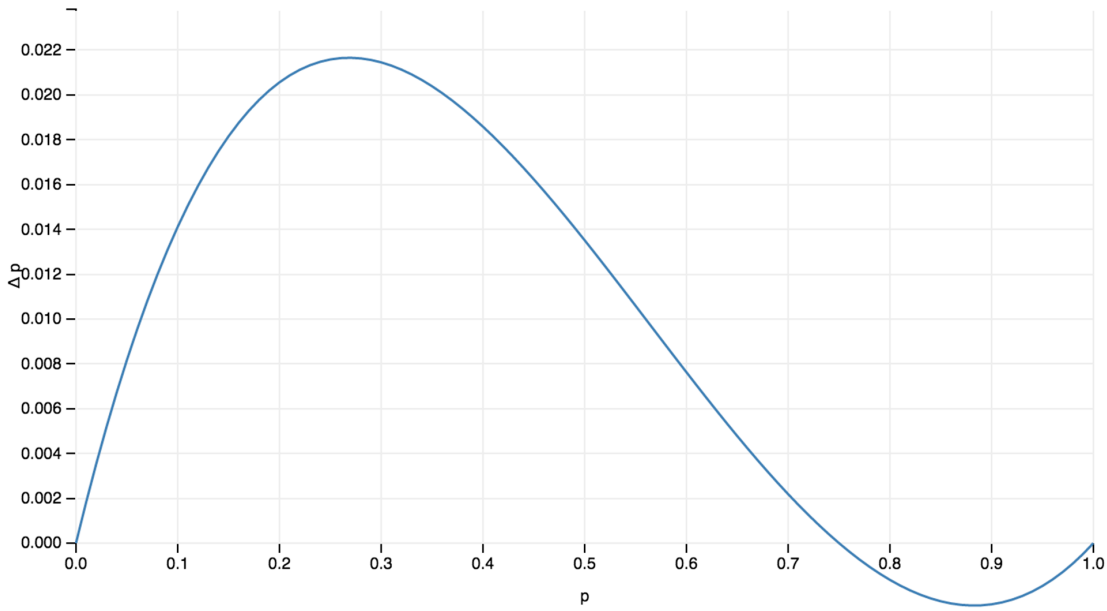
If you play around with the `s` and `h` values (making sure that they remain between 0 and 1), you will notice that the shapes of the curves will change a bit. Depending on the value of `s`, the fixation of the allele will be slower or faster. Depending on the value of `h`, the allele will increase rapidly in the beginning, and then slow down as it approaches fixation, or it will increase slowly in the beginning, and then rapidly go to fixation at the end. Nevertheless, the ultimate outcome is always the same - fixation. This is the consequence of directional selection.

Balancing Selection

Let us now move on to the next type of selection: balancing selection. Recall that balancing selection means that the fitness of the heterozygote is higher than that of both homozygotes (resulting in

the fitness peak in the fitness landscape figure above). In order to achieve that, we need to set the heterozygote effect, *h*, to a negative value.

We can reuse the code from above, and plot Δp as a function of p with the values `s = 0.1` and `h = -0.5`. If we do that, we get the following plot:



Interesting! As you can see, Δp is mostly positive, but for values of p higher than 0.75, it starts to become negative. What does that mean for the evolutionary dynamics?

Let's start again at a low p value, and use this plot to understand what will happen. Let's say we start at a low p value, where Δp is positive. A positive Δp means that p is going to be even higher in the next generation. We'll keep on moving to the right on the x axis, to higher values of p, until we get to the point where Δp is zero (at $p = 0.75$). What happens then? Nothing, really - the allele frequency has arrived at an equilibrium.

Imagine now for a moment that some event will bump p up to 0.9. At this value, Δp is negative, which means that p will be lower in the next generation. If we follow the same procedure, we'll keep on moving to the left on the x axis, to lower values of p, until we once again get to the point where Δp is zero.

In other words, $p = 0.75$ is a stable equilibrium. If the p value is somehow reduced, it will go right back up to 0.75. If it is somehow increased, it will go right back down to 0.75.

This is an intriguing finding. It means that the allele A1 will *not* go to fixation, even though the genotypes A1A1 and A1A2 have higher fitness values than A2A2. What is going on here? Well, it is true that the fitness of A1A1 is higher than the fitness of A2A2. However, the heterozygote A1A2 has the highest fitness. This will keep the A2 allele around, and is the reason why A1 won't go to fixation.

Another way to think about this is to imagine that your entire population is A1A1. Now, imagine that you introduce an A1A2 genotype. Because this heterozygote has a higher fitness than the other A1A1 genotypes, it will increase in frequency, and thus the A2 allele will increase in frequency too (and the A1 frequency will decrease). However, this process can't go on forever, because A1A2 genotypes will produce both A1A1 and A2A2 offspring genotypes, which have lower fitness values.

Let's see if our reasoning is correct, and plot the result of a deterministic simulation, using a slightly adapted version of our previous code, with `s = 0.1` and `h = -0.5`. This time, however, we run multiple simulations, starting at `p = 0.01` all the way up to `p = 0.99`, in `0.01` increments. We can adapt the code like so:

```
var s = 0.1;
var h = -0.5;
var initial_p = 0.01;
var p_values = [];

var data = [];
var generations = 300;

while (initial_p < 1) {
    p_values.push(initial_p);
    initial_p = initial_p + 0.01;
}

for (var i = 0; i < p_values.length; i = i + 1) {
    run_simulation(p_values[i]);
}

function run_simulation(p) {
    var results = [];
    for (var i = 0; i < generations; i = i + 1) {
        var q = 1-p;
        var delta_p = (p*q*s * ( p*h + q*(1-h))) / (1 - 2*p*q*h*s -q*2*s);
        p = p + delta_p;
        results.push(p);
    }
    data.push(results);
}

draw_line_chart(data,"Generation","p",[],generations);
```

This code makes use of the same concepts that we used in chapter 3, where we plotted multiple simulation runs at the same time. Since it's been a while since that chapter, let's revisit what's going

here. The key idea is that rather than storing the resulting values of a single simulation run in one array, we now have to store multiple such arrays in another array (i.e. a two-dimensional array). I've encapsulated the actual simulation code in a function `run_simulation()`, to which I'm passing an initial value for `p`. For each simulation, I'm creating a new `results` array, in which I will store the results of this particular simulation run. Once the simulation has run its course, I store the `results` array in the `data` array. It is this two-dimensional `data` array that I'm passing on to the plotting function.

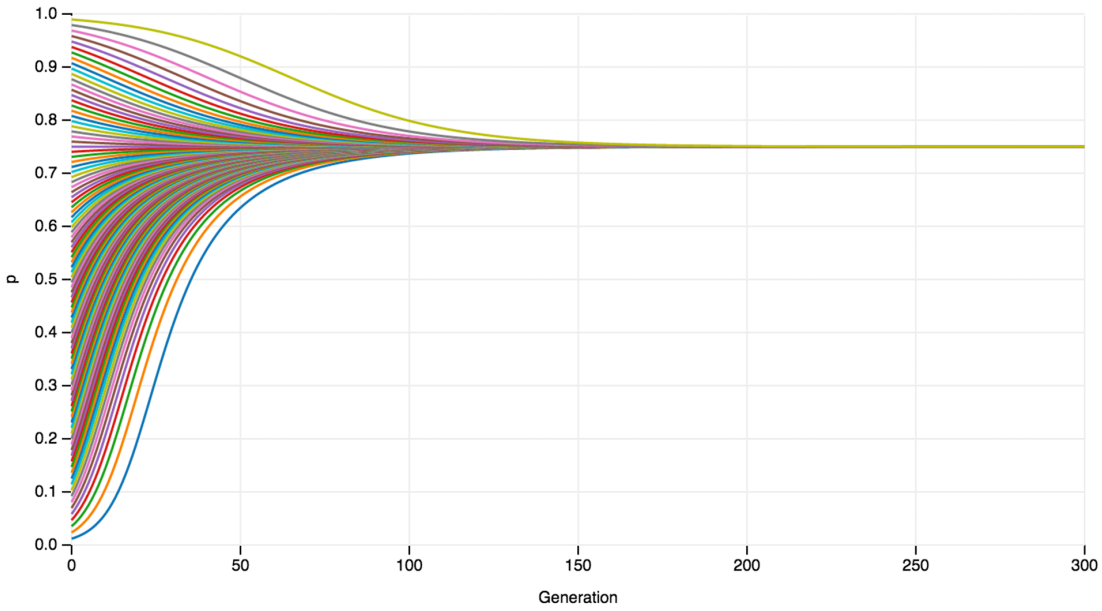
Because I have numerous initial values of `p`, I will store all of them in an array called `p_values`. I've already set the first initial value of `p`, `initial_p`, to `0.01`. Starting from that, I then create all the other initial values for `p` and store them in the array with this snippet of code:

```
while (initial_p < 1) {
    p_values.push(initial_p);
    initial_p = initial_p + 0.01;
}
```

This is a new loop construct that we haven't met before. We've met the `for` loop, and the `do while` loop. The `while` loop is a simplified version of the `do while` loop. The `do while` loop, as you may recall, executes some code at least once, and then repeats executing it while a given condition is true. The `while` loop simply executes some code while a given condition is true. It may be that the condition is false already from the get go, and the code would never be executed.

For our purpose here, the `while` loop is sufficient. It fills up the `p_values` array with values starting at `0.01`, up to `0.99`, in `0.01` increments. Once the array with initial values for `p` is set up, we simply iterate over it and call the `run_simulation()` function, passing along the corresponding initial value of `p`.

This produces the following plot:



As this plot clearly shows, all simulation runs will go the equilibrium allele frequency of $p = 0.75$, independent of the initial value of p. Thus, our reasoning was correct.

You probably understand now why this type of selection is called balancing selection. It balances the allele frequencies at an equilibrium value, because both homozygotes are less fit than the heterozygote. Balancing selection is important because it maintains genetic variation. As we've seen above, that's not true with directional selection, which will get rid of genetic variation.

There's one more thing about balancing selection that I want to mention. It turns out that to you can very easily calculate the equilibrium frequency under balancing selection. It is simply

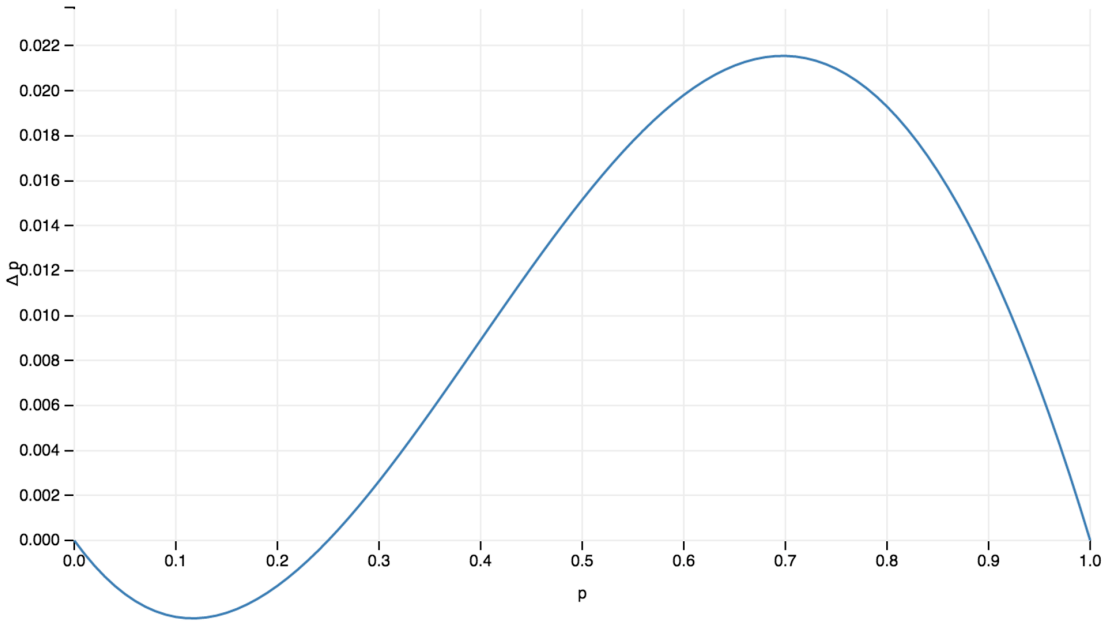
$$p* = \frac{h-1}{2h-1}$$

In our code above, we used `h = -0.5`. And if you plug $h = -0.5$ into the equation above, you'll get $p* = 0.75$. It's also interesting to note that the equilibrium value does not depend on the selection coefficient, *s*. A higher selection coefficient will lead to faster dynamics, i.e. the population will reach the equilibrium value of *p* faster - but it will still be the same value. You can experiment with the value of `s` in your code to verify this.

Disruptive Selection

Last, but certainly not least, let's take a look at disruptive selection. Disruptive selection occurs when the fitness value of the heterozygote genotype A1A2 is lower than that of both homozygotes (resulting in the fitness valley in the figure above). Disruptive selection occurs when the value of the heterozygote effect, *h*, is larger than 1.

As before, let's go ahead and plot the value Δp as a function of p, but this time with the values `s = 0.1` and `h = 1.5`. This gives us the following plot:



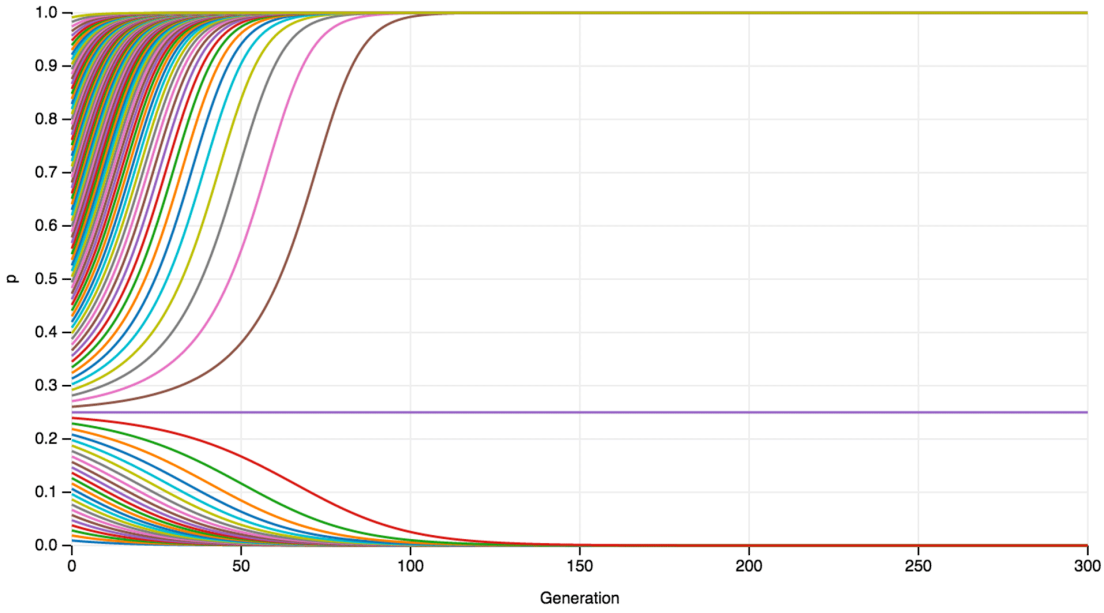
This plot looks rather similar to the plot that we saw in the case of balancing selection above. For most of p, Δp is positive, but sometimes, it is negative. The only difference here is that the negative values of Δp are found of the left side of the graph, where p is small.

Let's follow our method again and start with a low value of p, and see where this takes us. With a low value of p, Δp is negative, which means that p will be even lower in the next generation. Because Δp will again be negative (and remain there until $p = 0$), p will go down to 0, and the A1 allele will be lost.

Now let's start with a higher value for p, for example $p = 0.5$. At that value, Δp is positive, which means that p will be higher in the next generation. Because Δp will again be positive (and remain there until $p = 1$), p will go all the way up to 1, and the A1 allele will be fixed.

This is another astounding finding. It turns out that in the case of disruptive selection, evolutionary dynamics can go either way - the A1 allele can either go to fixation, or it can be lost altogether! It all depends on the initial conditions - i.e. the initial value of p. This is quite remarkable, and somehow violates our intuition. Shouldn't a deterministic system always go to the same equilibrium value?

The answer to this question is no. Some systems have multiple equilibria. The system of disruptive selection, it turns out, has three such equilibria. Why three? Let's go ahead an run a few simulations, plotting the evolutionary dynamics of disruptive selection. We can reuse the code from above, and simply set the heterozygous effect, `h`, to `1.5`. Here's what we'll find:



As predicted, depending on the initial conditions, p will either go to 0 or to 1. These are two of the three equilibria. However, there seems to be another equilibria at $p = 0.25$. What's up with that?

If we revisit the plot above, we can see that when $p = 0.25$, Δp is 0. That is why $p = 0.25$ is an equilibrium; the value of p won't change. However, there is something special about this equilibrium. Let's say your population is exactly at $p = 0.25$. Now assume that for some reason, the value of p is pushed very slightly upward. Now, $\Delta p > 0$, and the allele will go to fixation. On the other hand, if you assume that the value of p is pushed very slightly downward, to a value that is less than 0.25, you'll find that $\Delta p < 0$, and the allele will start its downward path until it is lost from the population. In other words, $p = 0.25$ is not a stable equilibrium. Quite the opposite - it is a so-called *unstable* equilibrium. An unstable equilibrium is an equilibrium where, if even just slightly pushed away from the equilibrium value, the system will go to another equilibrium value (0 or 1 in this case).

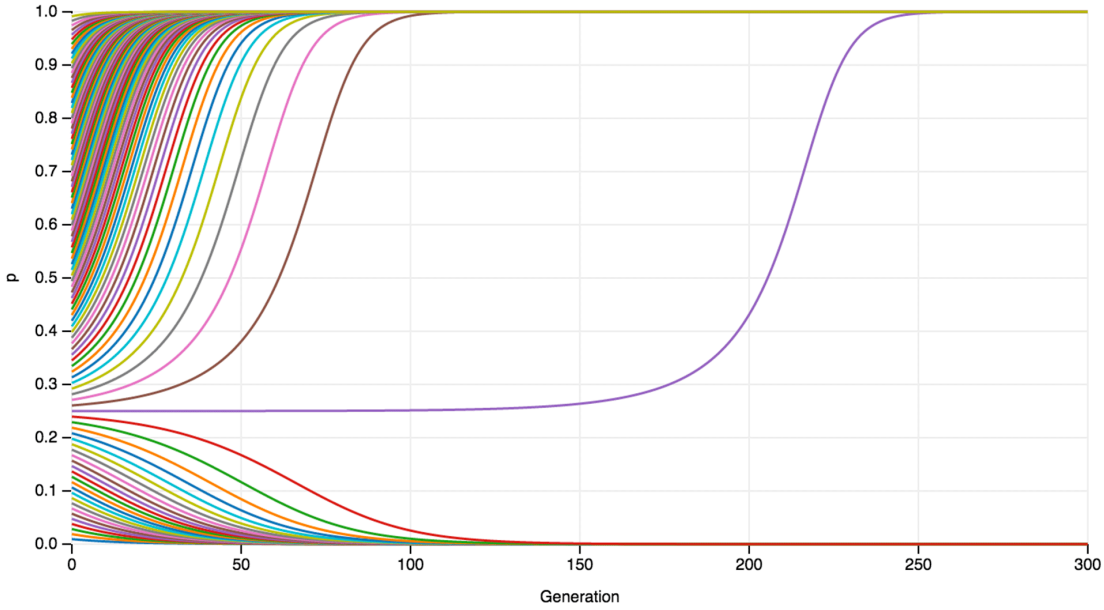
Thus, the unstable equilibrium is not important with respect to the population remaining there - it won't. Even if it happens to be there for a moment, the slightest deviation from that value will make the system go to one of the other two states. We can demonstrate this using our code, simply by changing this line in the `while` loop:

```
initial_p = initial_p + 0.01;
```

to this

```
initial_p = initial_p + 0.01 + (Math.random() * 0.000001);
```

What we are doing here is to add a tiny value (between 0 and a millionth) to the initial value of p. If you now rerun the code, you will see the following:



As you can see, the purple line will not remain at its initial value anymore. It will, as predicted, increase in frequency and go to fixation. Initially, the values of Δp are very, very small in the vicinity of $p = 0.25$, which is why it takes quite some time for the line to visibly begin its upward trajectory.

It is nevertheless important to know this equilibrium value, because of its importance in predicting the dynamics of the system. Once again, it turns out that it's very easy to calculate this value. Indeed, it is the exact same value as in the case of balancing selection:

$$p* = \frac{h-1}{2h-1}$$

The crucial difference is that in the case of balancing selection, the system will converge to $p*$, while in the case of disruptive selection, the system will go away from $p*$, to either 0 or 1. And once again, the value of the selection coefficient, *s*, has no effect on this value. It will simply determine at what speed the system will find its stable equilibrium.

To sum up - evolution by natural selection, even in a simple model of one gene with two alleles, can result in very different dynamics, depending on the fitness landscape. In the case of directional selection, the same allele will always go to fixation. In the case of balancing selection, both alleles will always remain in the population at an equilibrium. In the case of disruptive selection, one of the alleles will go to fixation, but which one will depend on the initial conditions.

Coevolution

You now understand how evolution by natural selection works. Genotypes with higher fitness have a higher probability of surviving and reproducing, and thus a higher chance of passing on their alleles to the next generation. Over the generations, those alleles and genotypes become more frequent.

We've haven't talked much about fitness until now. We simply mentioned earlier that fitness is the ability to survive and reproduce. But why are certain genotypes better able to survive and reproduce? Let's think about some examples. An animal may survive better because it is harder to be spotted by a predator. Another animal may survive better because it has better senses to locate its prey. Another animal may survive better because it can deal with harsher climates. What all of these examples have in common is that the animals are simply better adapted to their current environment. It is the natural environment that is doing the selecting - hence the term natural selection (as opposed to artificial selection in animal and plant breeding, for example).

We have seen above that interesting dynamics can result from this simple logic. But things get even more interesting when the environment is not just the abiotic (non-living) environment (such as the climate), but rather the biotic environment - in particular when the environment consists of other species who are subject to natural selection themselves. When the fitness of one species depends on the fitness of another species, co-evolution is often a direct consequence.

A good example of co-evolution is the interaction between hosts and parasites. A parasite has a high fitness if it can infect a host, evade the host's immune responses (i.e. survive), and create many offspring parasites. The problem is that the host's fitness will be reduced by the success of the parasite: many parasitic infections make a host sick, reducing the host's ability to survive and reproduce. Thus, hosts that can avoid or kill off parasites without getting sick have a higher fitness than those who can't. In other words, natural selection will favor those hosts that can avoid getting infected. In turn, the parasites that can't infect a host will have a very low fitness, and natural selection will favor those parasites that find new ways to infect the hosts. Thus, an endless coevolutionary arms race begins, where the host tries to build defense systems against parasite infection, and the parasites try to find ways around the defense systems. Both species are playing catch up all the time, which can lead to very interesting evolutionary dynamics.

In this section, we will build a simple coevolutionary arms race between a host and a parasite species. We will do this by implementing a simple conceptual model of genetic interactions between hosts and parasites. We are going to assume that the host has one gene with two alleles (A1 and A2) that represent a simple immune system. Equally, we are going to assume that the parasite has a gene with two alleles, A1 and A2. To keep things simple, we will assume haploid hosts and parasites - that is, both hosts and parasites have only one allele, not two. Thus, there are only two possible genotypes, A1 and A2, and we can use the term allele and genotype interchangeably.

You can think of the genetic host-parasite interaction system as a key-lock mechanism: in order to get into the host, the parasite allele (the key) needs to match the host's allele (the lock).

In reality, the genetics of parasites and hosts are much more complicated, but this simple system captures a key idea. That key idea is *negative frequency-dependent selection*. This is quite a mouthful,

but it is actually a rather easy idea to grasp. Frequency-dependent selection simply means that an individual's fitness value does not only depend on its genotype, but also on the *frequency* of its genotype in a population. This can work in two ways:

1. A genotype could be more fit when it is more common in the population (this is called *positive* frequency-dependent selection).
2. A genotype could be less fit when it is more common (this is called *negative* frequency-dependent selection).

Negative frequency-dependent selection is much more common in nature than positive frequency-dependent selection. Almost all antagonistic species interactions such as host-parasite or predator-pray interactions result in negative frequency-dependent selection. The key-lock model above is a great example. Imagine that almost every host has the lock A1. Because only parasites with the key A1 can infect those hosts, there is a huge selective benefits for A1 parasites, and they will increase rapidly in frequency. However, that means that there are increasingly fewer parasites with the (non-matching) A2 keys. This in turn means that the hosts with the A2 locks are becoming less infected. In this case, the A2 host genotype benefited from being rare, because the parasite population adapted to attack the very frequent A1 host genotype.

However, this blissful safety that the A2 hosts enjoy is only of limited duration. Because A2 hosts get less often infected, they are at a selective advantage, and will thus increase in frequency. As they increase in frequency, the parasite population will catch up on them, driven by natural selection. As you can guess, this leads to a never ending cycling of genotypes, where the hosts try to escape the parasites (genetically speaking), and the parasites try to catch up with the hosts. This type of dynamic is often called "Red Queen" dynamic, inspired by the fictitious character of the Red Queen in Lewis Carroll's famous book 'Through the Looking-Glass', who at one point in the story explains: "Now, here, you see, it takes all the running you can do, to keep in the same place!".

Let's go ahead and implement this simple host-parasite model, and see if we really get those cyclical dynamics!

The first thing we're going to do is set up a few key variables:

```
var data = [];
var generations = 400;

var host_frequencies = [0,0];
var parasite_frequencies = [0,0];

var sh = 0.2;
var sp = 0.5;
```

The `data` array, as always, will store the data that we want to plot, i.e. the host and parasite genotype frequencies. The `generations` variable defines the duration of the simulation, expressed as the

number of generations. The two arrays `host_frequencies` and `parasite_frequencies` will keep track of the frequencies of the two genotypes in each species throughout the simulation. We initialize the values at `0`, but we'll change that in a minute. Note that both arrays have two elements, since there are only two possible genotypes in each of the haploid species (A1 and A2).

Finally, the last two variables, `sh` and `sp`, define the fitness reduction (selection coefficient *s*) if the host gets infected by a a parasite, or if the parasite cannot infect a host, respectively.

There are four combinations of possible interactions between the two host and parasite genotypes, with the following outcomes:

	Host A1	**Host A2**
Parasite A1	**Infection** Fitness Host: 1-sh Fitness Parasite: 1	**No Infection** Fitness Host: 1 Fitness Parasite: 1-sp
Parasite A2	**No Infection** Fitness Host: 1 Fitness Parasite: 1-sp	**Infection** Fitness Host: 1-sh Fitness Parasite: 1

We will implement this fitness model in a minute. The important thing to note is that if the genotypes don't match, the parasite won't be able to infect the host. That's great for the host, who will have the maximal fitness value `1`. It's bad for the parasite though, as parasites generally need a host to reproduce. The parasite's fitness value is reduced by the parasite-specific selection coefficient `sp`, to `1-sp`. On the other hand, if the genotypes do match, the parasite will be able to infect the host. In this case, the parasite's fitness is maximal (i.e. `1`), and the host's fitness is reduced by the host-specific selection coefficient `sh`, to `1-sh`, due to the infection.

To start the simulation, let's initiate the genotype frequencies and push to the `data` array accordingly:

```
function initialize_frequencies() {
    host_frequencies[0] = Math.random();
    data.push([host_frequencies[0]]);
    host_frequencies[1] = 1 - host_frequencies[0];
    data.push([host_frequencies[1]]);
    parasite_frequencies[0] = Math.random();
    data.push([parasite_frequencies[0]]);
    parasite_frequencies[1] = 1 - parasite_frequencies[0];
    data.push([parasite_frequencies[1]]);
}

initialize_frequencies();
```

The first line in this method sets the first element of the `host_frequencies` array (which holds the

frequency of host genotype A1) to a random number between 0 and 1. The second line adds an array to the `data` array, initialized with a single value - that of the host genotype A1 frequency which we have just set.

Keep in mind that our `data` array will end up being a two-dimensional array, which holds the four arrays storing the four genotype frequencies over the generations. Because the data array is initially empty, the line

```
data.push([]);
```

would simply add an empty array `[]` to the `data` array, making it an empty two-dimensional array. Instead of pushing an empty array, we can push an array that has values in it already. For example, we could write

```
data.push([1,4,7]);
```

In this case, we would now have a two-dimensional array where the first element of `data` (i.e. `data[0]`), would contain the array `[1,4,7]`. Our actual code reads

```
data.push([host_frequencies[0]]);
```

which means that the first element in `data` is now an array with one value, namely the host genotype A1 frequency, `host_frequencies[0]`.

The next two lines are similar:

```
host_frequencies[1] = 1 - host_frequencies[0];
data.push([host_frequencies[1]]);
```

In the first line, we simply set the frequency of host genotype A2, which is 1 minus the frequency of host genotype A1 (since they must add up to 1). The second line pushes another, new array into `data`, this time with the initial frequency of the host genotype A2.

The following four lines are more or less identical, simply setting the parasite frequencies in the exact same manner. After this function has finished executing, we have all initial four genotype frequencies set to random values, and our data array is initialized with the four arrays keeping track of the four genotype frequencies over time, and the first value in each array is already set. We're good to move on!

After calling the function `initialize_frequencies()`, we we will run the following code:

```
for (var i = 0; i < generations; i = i + 1) {
    host_selection();
    parasite_selection();
    data[0].push(host_frequencies[0]);
    data[1].push(host_frequencies[1]);
    data[2].push(parasite_frequencies[0]);
    data[3].push(parasite_frequencies[1]);
}
draw_line_chart(data,"Generation","p",[]);
```

This loop runs over a given number of generations, and in each generation, calls the two methods `host_selection()` and `parasite_selection()`. These methods will calculate the new genotype frequencies, based on the current genotype frequencies and the fitness values. We'll implement them right away. Once they have finished executing, we will store the new genotype frequencies in the corresponding array in `data`. And finally, after the generations have passed, we'll plot the results stored in `data` visually.

Let's now turn to the key methods in our simulation, `host_selection()` and `parasite_selection()`. Before we develop the code, let's think about how they should work.

At the beginning of this chapter, we calculated frequencies of alleles over time, given certain fitness values of genotypes. These fitness values were assumed to be constant. Then, we simply developed the following logic for a given genotype: The frequency of a genotype at time *t+1*, f_{t+1}, equals its frequency at *t*, f_t, times its fitness value, *w*, normalized by the total population fitness, $\bar{w}$. In mathematical terms, we had

$$f_{t+1} = \frac{w f_t}{\bar{w}}$$

The same logic applies in frequency dependent selection, with one important difference: the fitness value *w* is not a constant anymore, but rather a function of other genotype frequencies. In our case, the fitness of a given host genotype depends on the parasite frequencies.

As a concrete example, the fitness of host genotype A1, w_{H1}, depends on both parasite frequencies f_{P1} and f_{P2}, as well as the fitness values for host A1 when interacting with parasite A1 (w_{H1P1}), and when interacting with parasite A2 (w_{H1P2}):

$$w_{H1} = w_{H1P1} f_{P1} + w_{H1P2} f_{P2}$$

Think about it as follows. As a host with genotype A1, you can either meet a parasite with genotype A1 (which will happen with probability f_{P1}, the frequency of parasite genotype A1), or you can meet a parasite with genotype A2 (which will happen with probability f_{P2}, the frequency of parasite genotype A2). In the first case, the fitness of the host A1 when meeting parasite A1, w_{H1P1}, will be *1-sh*. In the second case, the fitness of the host A1 when meeting parasite A2, w_{H1P2}, will be 1. The

total fitness is simply the sum of these two possible cases. The exact same logic applies for all the other genotypes.

Note that there is no normalizing happening at this stage, since we aren't calculating frequencies yet. Now, if we want to know the frequency of host genotype A1 in the next generation, we will do the exact same thing as we have before: multiply its fitness, w_{H1}, with its frequency. Then, we do the same thing for host genotype A2. Once we have these two frequencies, we need to normalize them by their sum, since they might add up to a value other than 1.

In code, this looks as follows:

```
function host_selection() {
    var sum_host_frequencies = 0;
    for (var i = 0; i < host_frequencies.length; i = i + 1) {
        var host_fitness = 0;
        for (var ii = 0; ii < parasite_frequencies.length; ii = ii + 1) {
            host_fitness = host_fitness + get_host_w_f(i, ii);
        }
        host_frequencies[i] = host_frequencies[i] * host_fitness;
        sum_host_frequencies = sum_host_frequencies + host_frequencies[i];
    }
    for (i = 0; i < host_frequencies.length; i = i + 1) {
        host_frequencies[i] = host_frequencies[i] / sum_host_frequencies;
    }
}

function get_host_w_f(i,ii){
    return (i == ii ? 1-sh : 1) * parasite_frequencies[ii]
}
```

Let's walk through this line by line. At first, we initialized the variable `sum_host_frequencies` at `0`. This variable will be necessary for the normalizing at the end. Then, in the first loop, we go through all host frequencies, and calculate their new frequencies, using the formula above.

Notice how the first four lines in that loop,

```
var host_fitness = 0;
for (var ii = 0; ii < parasite_frequencies.length; ii = ii + 1) {
    host_fitness = current_host_fitness + get_host_w_f(i, ii);
}
```

are the programmed equivalent of

$$w_{H1} = w_{H1P1} f_{P1} + w_{H1P2} f_{P2}$$

The function `get_host_w_f(i, ii)` simply returns the product of the frequency and the fitness:

```
function get_host_w_f(i,ii){
    return (i == ii ? 1-sh : 1) * parasite_frequencies[ii]
}
```

There is something interesting in that function that we haven't seen before, which is this expression:

```
i == ii ? 1-sh : 1
```

This is the so-called *ternary* operator `?:` which is a short cut for an `if else` expression. It works as follows. The following `if else` expression

```
if (condition) {
   code1
}
else {
   code2
}
```

Can be simplified with the ternary operator `?:` as follows:

```
condition ? code1 : code2
```

Note that these two pieces of code do the exact same thing - they're just different versions of expressing an idea. This is very common - there are infinitely many ways to achieve the same thing in code. Don't worry if you would implement an idea differently than I have here throughout the book. As long as your code does the same thing, it's perfectly fine. As you become more comfortable with programming, you will learn that some ways can be better than others to express the same idea - either because they're easier to read, easier to extend, or for other reasons. At this stage in your path to learning programming, you don't need to worry about this too much. Shorter code is not automatically better code. What is most important at the moment is that you understand what you're doing, and that you will find it easy to read your code when you go back to it in the future.

Back to our function, where the next two lines read

```
host_frequencies[i] = host_frequencies[i] * host_fitness;
sum_host_frequencies = sum_host_frequencies + host_frequencies[i];
```

Now that we've calculated the current host fitness (given the parasite frequencies), we can multiply it with the current host frequency in order to get the new host frequency. Then, we add this value to `sum_host_frequencies` for later normalization.

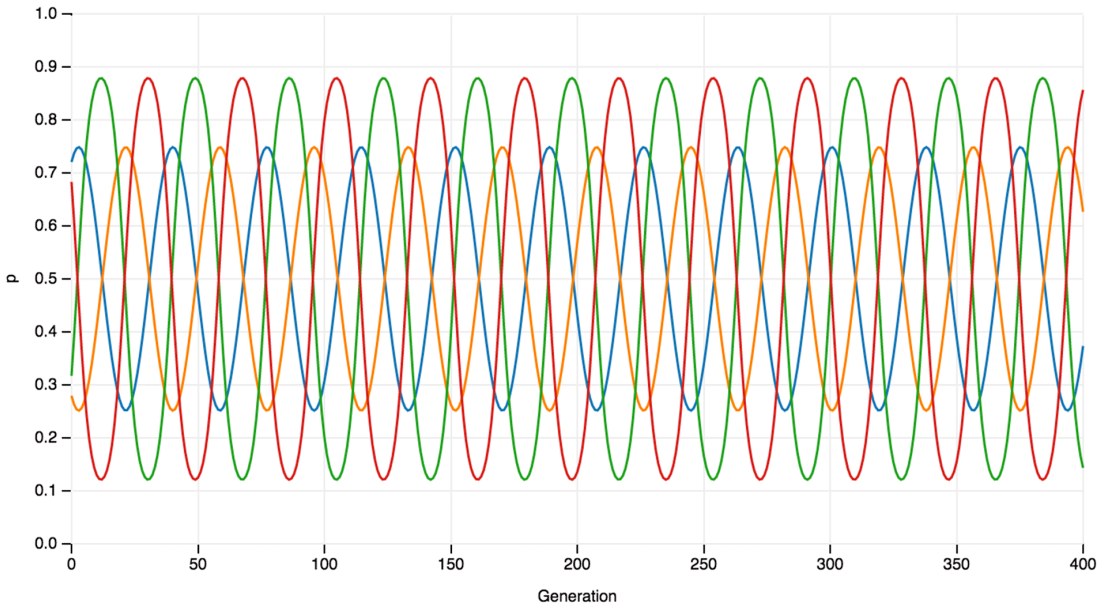
After the loop has gone through all hosts genotypes, and calculated the new frequencies, we need to normalize them so that they add up to 1 again. The last three lines in the function take care of this:

```
for (i = 0; i < host_frequencies.length; i = i + 1) {
    host_frequencies[i] = host_frequencies[i] / sum_host_frequencies;
}
```

And that's it. The function `parasite_selection()` follows the exact same logic, except that we now do everything from the perspective of the parasite:

```
function parasite_selection() {
    var sum_parasite_frequencies = 0;
    for (var i = 0; i < parasite_frequencies.length; i = i + 1) {
        var parasite_fitness = 0;
        for (var ii = 0; ii < host_frequencies.length; ii = ii + 1) {
            parasite_fitness = parasite_fitness + get_parasite_w_f(i, ii);
        }
        parasite_frequencies[i] = parasite_frequencies[i] * parasite_fitness;
        sum_parasite_frequencies = sum_parasite_frequencies + parasite_frequenci\
es[i];
    }
    for (i = 0; i < parasite_frequencies.length; i = i + 1) {
        parasite_frequencies[i] = parasite_frequencies[i] / sum_parasite_frequen\
cies;
    }
}

function get_parasite_w_f(i,ii) {
    return (i == ii ? 1 : 1-sp) * host_frequencies[ii]
}
```

And that's it! Now that have put everything together, we can run the code and look at the output. Here's what I see, the first time I run this code:



Wow - cyclical dynamics indeed! This is the Red Queen, in action.

Go ahead and reload the page a number of times. You will see that the dynamics are very much dependent on initial conditions. In addition, try changing the values of the selection coefficients `sh` and `sp`, and see how that changes the dynamics.

Before we conclude, I'd like to make one very small modification to the code. In your function `host_selection()`, go ahead and add this line

```
host_frequencies[i] = host_frequencies[i] + Math.random() * 0.01;
```

right after this one

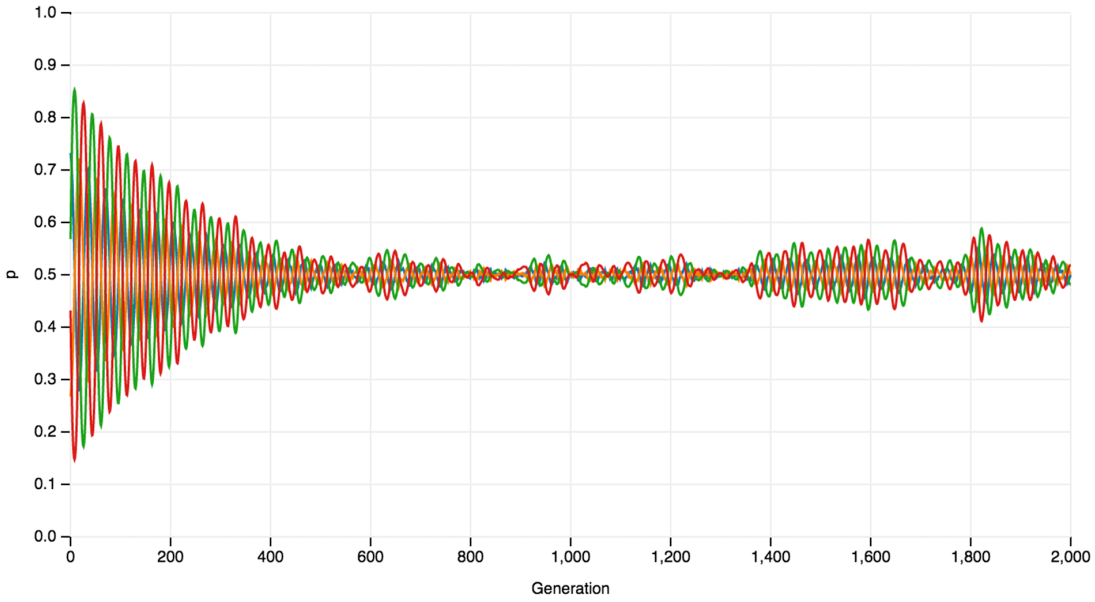
```
host_frequencies[i] = host_frequencies[i] * host_fitness;
```

What the new line does is to add a very small value - a random number between 0 and 0.01 - to the new host frequencies. At this stage, the host frequencies are not yet normalized, so the net effect of this addition is that the normalized host frequencies will just be slightly disturbed.

Before you reload the page, set the number of generations to `2000` - you'll see why in a second:

```
var generations = 2000;
```

Now reload the page. Here's what I see the first time I run this code:



Look at that - that's a very different system from what we just observed. What is going on here?

It turns out that the very strong and regular cyclical dynamics are an artifact of a purely deterministic system. In our original code, everything was deterministic - there was no randomness to anything. As this small code change demonstrates, adding a little bit of randomness - or noise, if you will - to the host frequencies dramatically changes the dynamics of the system. In other words, the dynamics that we observed are not robust. When slightly perturbed, the system behaves very differently.

Why does this matter from a biological perspective? By adding a little bit of noise to the host genotype frequencies, we've essentially introduced the effects of random genetic drift. It's very reasonable to assume that host populations are not always extremely large, and that drift will therefore affect genotype frequencies. What this shows is that even a little bit of genetic drift will change the evolutionary dynamics considerably. We still get cycling dynamics, but they are much less pronounced than in the purely deterministic system. In addition, the dynamics seem to be more complex, with larger changes occurring over hundreds of generations (this is why I asked you to increase the number of generations to 2,000).

And with that, we conclude the chapter on natural selection. Natural selection is an enormously potent force, being responsible for all adaptations that we see in nature. As you can imagine, we have only scratched the surface here. But we have learned some key insights into how natural selection works, and how it can affect evolution.

Let's briefly wrap up what we learned in this chapter.

- Genotypes and alleles that give individuals a higher average chance to survive and reproduce will over time increase in frequency - this process is called evolution by natural selection.
- Fitness is a measure of the ability to survive and reproduce.

- In diploid individuals, there are three possible types of natural selection: directional selection, balancing selection, and disruptive selection.
- Directional selection will drive alleles to fixation.
- Balancing selection will maintain genetic diversity in a population.
- Disruptive selection will drive alleles to fixation, but which ones depends on initial conditions.
- Negative frequency-dependent selection (coevolution) leads to cyclical evolutionary dynamics (red queen dynamics).
- Red queen dynamics can be strongly affected by genetic drift.
- We've also introduced the ternary operator in JavaScript.

We've now covered all four forces affecting evolution: drift, mutation, migration, and natural selection. You've come a long way. Not only do you now have an overview on the major forces affecting the living world, but you also have the ability to implement these ideas in code. This is a wonderful achievement - congratulations. In the next chapters, we'll be looking at some other, highly interesting dynamic systems in biology: we'll look at how infectious disease spread, and how cooperation evolves.

7. Epidemics: The Spread of Infectious Diseases

Have you ever had a cold, or the flu? I'm sure you have - especially common colds, with the runny nose and the coughs, are hard to avoid, and a reoccurring nuisance mostly during the winter months. Hopefully, you have never had the flu, or only rarely - unlike the common cold, the flu is caused by the influenza virus, which can cause very nasty infections that can put you out for multiple weeks. In fact, every year, tens of thousands of people die from complications caused by the flu.

The common cold and the flu are two examples of infectious diseases. An infectious disease is a disease that is caused by an infectious agent - typically a microorganism such as a virus, a bacterium, or a fungus. When these microorganisms enter our body and cause an infection that makes us sick, we call them pathogens. While it is bad enough for us as individuals that pathogens make us sick, what is really bad is that they make us pass them on to other individuals through infection.

Thus, the term infectious disease is slightly incorrect - you are not actually passing on a disease. Instead, you are passing on the pathogens that cause the disease in the first place.

But not all microorganisms have the same effect. In fact, most microorganisms in your body do not cause disease. Did you know that you have more bacterial cells in your body than human cells? That's right - your human cells are outnumbered by about factor 10! In other words, for every single human cell, there are about 10 bacterial cells in your body. Most of them live in your gut, where they don't cause disease, and indeed, they even keep you healthy.

However, some microorganisms do cause disease, and when they do, the disease can range from hardly noticeable to deadly. The common cold certainly ranges among the mildest infectious diseases. It's typically a little annoying, sometimes causes a fever, but never kills anyone (although people can die of other complications following a common cold, especially when they are already very weak). On the other hand of the spectrum, we have diseases like Ebola, where untreated individuals have an up to 90% chance of dying because of the disease.

Only a little more than a century ago, infectious diseases were a major killer everywhere in the world. Many children didn't make it past age 5, and would die of a childhood disease against which there were no vaccines. Nowadays, modern public health systems and medicine have helped us keep many of the killer diseases in check. Nevertheless, infectious disease remain a major threat. In developing countries, they are still a major cause of death. But nobody in the world is safe from new infectious diseases that emerge every year, typically jumping from animals to humans.

In this chapter, we are going to look at the fascinating process of infectious disease spread through a population. We will program a virtual world where a single infectious individual starts infecting others and thereby starts an epidemic that spreads through the population. We will then investigate

how a few contacts between random individuals, rather than local neighbors, can completely change the dynamics of an epidemic, leading to much larger outbreaks.

Ready to enter the world of germs? Let's go!

The SIR Model

The most common way to model an infectious disease outbreak in a population is to assume that individuals are in one of three possible states:

1. Susceptible: This means that the individual has not been infected yet, and could get infected.
2. Infected: This means that the individual is currently infected, and can infect other individuals who are susceptible.
3. Recovered: This means that the individual has recovered from the infection. The individual cannot infect others anymore, and can't be infected either because it developed immunity.

Because of the first letters of the three states (susceptible, infected, and recovered), a model implementing these states is called an *SIR model*. The SIR model captures some key aspects about many infectious diseases, but simplifies the reality down to three states. This follows the same idea that we have encountered earlier in the book: to make a model as simple as possible, but not simpler.

We are going to assume that initially, everybody is susceptible to an infection. In reality, that is not always true: Some people may have some background immunity because they were infected by a similar pathogen earlier in their life. Some people may have some genetic mutations that make them resistant to some infections. Some people may be vaccinated. But for a completely new disease that has never circulated in humans, the assumption is probably quite close to the truth.

So how do susceptible individuals get infected? At first, we are simply going to assume that one individual - patient 0 - becomes infected. We are not going to bother about how exactly that happens. In nature, many emerging diseases in humans are of zoonotic origin, which means that a pathogen made the jump from an animal to humans. For example, flu circulates in birds, pigs, horses, and other animal species. The bird flu that most epidemiologists are so worried about is caused by an influenza virus that circulates in birds. Sometimes, a human gets infected by being exposed to an infected bird (for example when working on a poultry market). Ebola, as another example, circulates in fruit bats, and the devastating Ebola outbreak that emerged in West Africa in 2014 is thought to have originated in a person who got infected by a fruit bat (perhaps by accidentally eating bat droppings on fruit).

But once an individual is infected, we assume that the pathogen can jump from person to person. For that to happen, susceptible individuals need to come in contact with an infected person. Whenever such a contact happens, there's some probability that the pathogen will jump from the infected person to the susceptible person, and as a consequence, the susceptible person becomes infected.

Finally, an infected individual can recover from infection. This also happens with a certain probability per time. As we will see below, much of the dynamics of infectious disease spread in an

SIR model is explained by these two probabilities: the probability of getting infected upon exposure, and the probability of recovering.

We could go ahead and implement a non-spatial model as we have done most of the time throughout the book, with the exception of chapter 5. However, I think epidemic dynamics are best understood when we can observe the spatial spread of the disease through a population. Thankfully, we have already developed most of the code for spatial models in chapter 5, and will be able to reuse almost all of it. In fact, our code is going to be substantially simpler, because we don't need to deal with genetics in this chapter.

In the first iteration of the code, we are going to make the simplifying assumption that there is no recovery from the disease. We'll change that right after we implemented the code. In a sense, we will start with an *SI* model, rather than an *SIR* model, and see what happens. From that, we'll add the code for recovery in order to get an *SIR* model. You will see at the end why this is a good way to proceed.

Let's get started and set up a few key variables:

```
var grid_length = 100;
var grid = [];
var temp_grid = [];
var beta = 0.05;
var gamma = 0.1;
```

The variable `grid_length` once again defines the side length of our two-dimensional world. As in chapter 5, we will store the individuals in a two-dimensional array named `grid`, and we'll have a `temp_grid` (also a two-dimensional array) to store transient values for proper updating (more about that later). Finally, the variable `beta` defines the infection probability per contact and unit time. Notice that our time steps are not generations anymore. Instead, they are arbitrary units of time (in our case, we can perhaps think of them as days). `beta` essentially defines the probability that a susceptible individual becomes infected, per time step, when exposed to an infected individual. The variable `gamma`, on the other hand, defines the probability of recovery of an infected individual, per unit time.

Next, we'll go ahead and initialize the grid:

```
function get_random_int(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

function init_grid() {
    for (var i = 0; i < grid_length; i = i + 1) {
        grid[i] = [];
        for (var ii = 0; ii < grid_length; ii = ii + 1) {
           grid[i][ii] = "S";
        }
    }
    grid[get_random_int(0,grid_length-1)][get_random_int(0,grid_length-1)] = "I";
}

init_grid();
```

The function `init_grid()` is almost identical to the one we used in chapter 5, but rather than assigning random values to the individual cells, we'll simply set all values to `"S"` (which stands for susceptible). After that, we pick a random cell on the grid - using the function `get_random_int(min, max)` from chapter 5 - and set its value to `"I"` (for infected). And with that, the population is initialized. There are 9,999 susceptible and 1 infected individual: patient 0.

We'll be reusing the plotting functions `draw_grid()` and `update_grid()` from chapter 5 to draw the population in the browse, so go ahead and copy them over from chapter 5.

This time, we'll use a slightly different coloring scheme for the different types of individuals: gray for susceptible, red for infected, and green for recovered. I have developed the plotting functions in such a way that you can pass the colors to the function, as the second argument.

We're now going to call the `draw_grid()` function, and then use the method `setInterval()` that we've learned about in chapter 5 to run a time step und update the population, like so:

```
draw_grid(grid,["S","#dcdcdc","I","#c82605","R","#6fc041"]);

function simulate_and_visualize() {
    run_time_step();
    update_grid(grid,["S","#dcdcdc","I","#c82605","R","#6fc041"]);
}

setInterval(simulate_and_visualize, 50);
```

As you can see, we are passing a second parameter, the array

```
["S","#dcdcdc","I","#c82605","R","#6fc041"]
```

to the plotting functions. The values after "S", "I", and "R" are color-codes; but like before, let's not worry about the drawing functions, but rather about the simulation code.

I've changed one more thing, even if only slightly: in comparison to chapter 5, we reduced the interval duration from `100` milliseconds to `50` milliseconds, so that the simulations run a little faster. If you feel things are too fast, or that your computer can't quite keep up with the speed, simply change the milliseconds parameter to whatever is appropriate in your browser.

We haven't defined the function `run_time_step()` yet - let's go ahead and implement an empty function so that we won't get any JavaScript errors:

```
function run_time_step() {
}
```

With that in place, save the document and load it in the browser. You should see the following:

See that little red square in the bottom left corner? That's our patient 0, the first infected individual in the population.

Now, because we haven't implemented a function body for `run_time_step()` yet, nothing happens after this initialization. Let's go ahead and implement the full SIR model logic now.

What we want to do at each time step is to take each individual, check if it is infected, and then, if it is indeed infected, expose all susceptible individuals who are in contact with that infected individual. *We are going to assume that an individual is only in contact with its eight adjacent neighbors on the grid.*

You may recall that in chapter 5, we've had a short discussion about the correct way of updating cells on the grid in a simulation like this. We argued that we needed to go through the individuals, apply whatever rules we want to apply, and then update a temporary cell on the grid (or a cell on a temporary grid), rather than updating the actual cell on the grid. The reason why we did this was to simulate a situation where all grid cells would update simultaneously. Since we can't really do things simultaneously in a computer (code is executed one statement after the other), we needed to resort to this trick.

In this simulation, we will resort again to the same trick. We'll go through all the susceptible individuals that are exposed by an infected individual, and infect them with a certain probability; but we won't infect the individuals (i.e. the grid cells) right away - rather, we'll infect the same cell on a temporary grid, and then, once we've gone through all cells, we copy the contents of the temporary grid over to the real grid.

Here's how we will do that:

```
function run_time_step() {
    for (var i = 0; i < grid_length; i = i + 1) {
        temp_grid[i] = [];
        for (var ii = 0; ii < grid_length; ii = ii + 1) {
            temp_grid[i][ii] = grid[i][ii];
        }
    }
    for (i = 0; i < grid_length; i = i + 1) {
        for (ii = 0; ii < grid_length; ii = ii + 1) {
            if (grid[i][ii] == "I") {
                expose_neighbors(i,ii);
            }
        }
    }
    for (i = 0; i < grid_length; i = i + 1) {
        for (ii = 0; ii < grid_length; ii = ii + 1) {
            grid[i][ii] = temp_grid[i][ii];
        }
```

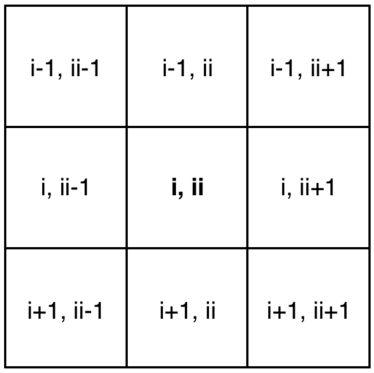
```
    }
}
```

This looks intimidating, but is actually fairly straightforward. This function iterates over the two-dimensional `grid` (using a nested `for` loop) three times in a row. The first time, it simply sets the `temp_grid` array to be an exact copy of the current `grid`. The second time, it goes through all cells, and if the individual on the cell is infected, it calls the function `expose_neighbors(i,ii)`, which we will implement below. The third time, we simply copy all the values from `temp_grid` back to `grid`.

Let's implement the function `expose_neighbors(i,ii)`:

```
function expose_neighbors(i,ii) {
    for (var n_i = i-1; n_i <= i+1; n_i = n_i + 1) {
        for (var n_ii = ii-1; n_ii <= ii+1; n_ii = n_ii + 1) {
            if (n_i == i && n_ii == ii) {
                continue;
            }
            try_infection(get_bounded_index(n_i),get_bounded_index(n_ii));
        }
    }
}

function get_bounded_index(index) {
    var bounded_index = index;
    if (index < 0) {
        bounded_index = index + grid_length;
    }
    if (index >= grid_length) {
        bounded_index = index - grid_length;
    }
    return bounded_index;
}

function try_infection(i,ii) {
    if (grid[i][ii] == "S") {
        if (Math.random() < beta) {
            temp_grid[i][ii] = "I";
        }
    }
}
```

The function `expose_neighbors(i,ii)` has two parameters: the coordinates `i` and `ii` on the grid. What we want to do is to expose all eight neighborhood cells of that one cell specified by the coordinates. Here are the coordinates of these eight cells:



In order to get access to all these cells, we need to loop over the two-dimensional space that these cells occupy. In the function `expose_neighbors(i,ii)`, we do that by using the coordinates `i`, and `ii`, like so:

```
for (var n_i = i-1; n_i <= i+1; n_i = n_i + 1) {
    for (var n_ii = ii-1; n_ii <= ii+1; n_ii = n_ii + 1) {
        // do something with cell at coordinates n_i, n_ii
    }
}
```

As you can see, we are using local variables named `n_i` and `n_ii` as "neighbor" placeholders for the variables `i` and `ii`. Why do we do that? Well, the function has two arguments, `i` and `ii`, and as we've learned earlier in this book, arguments are local variables in the function scope. Therefore, `i` and `ii` are already taken as variable names within the function body, and we don't want to override them - they represent the original coordinates of the infected cell. That's why we introduce the new variables in the function body.

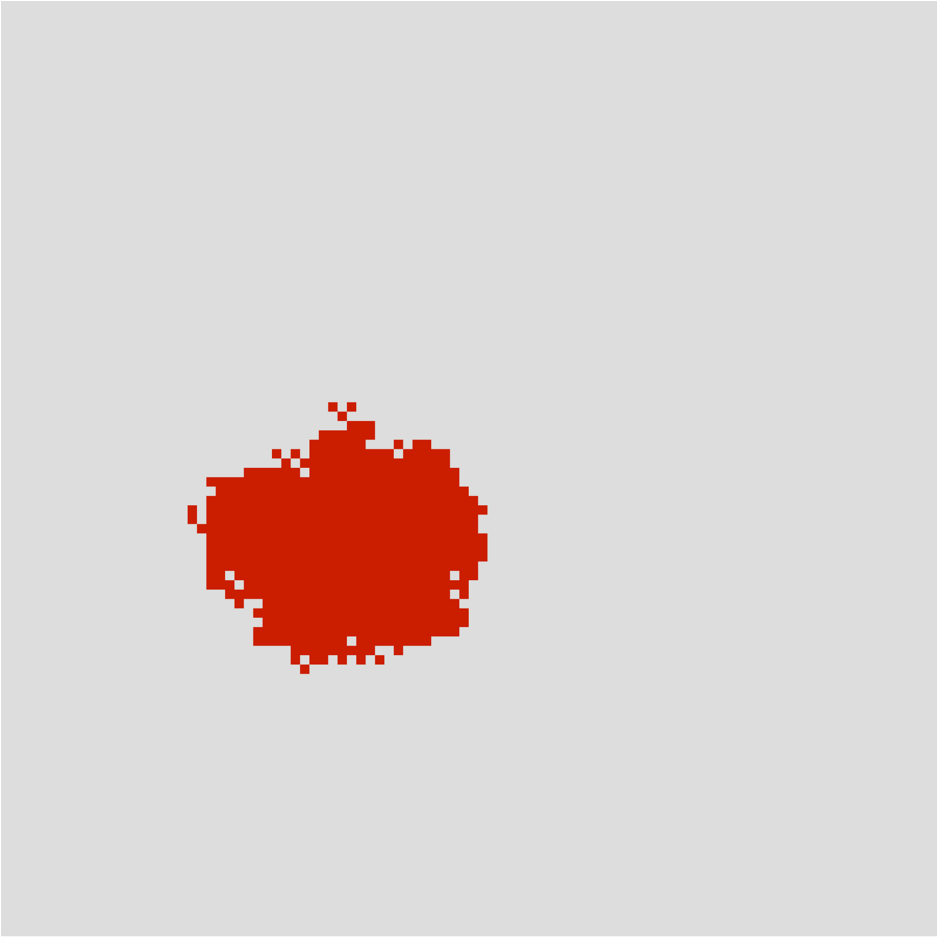
The nested loop iterates over all nine cells shown above. The outer loop starts at `i-1`, and stops after `i+1`. The inner loop starts at `ii-1`, and stops after `ii+1`. You've probably realized immediately that there is a slight problem with this: By looping through all nine cells, we are also visiting the original cell at coordinates `i`, `ii`. We don't want to expose the cell to itself, of course, so once we're at that cell, we simply need to instruct our loop to move on immediately. The key expression here is the `continue` statement:

```
if (n_i == i && n_ii == ii) {
   continue;
}
try_infection(get_bounded_index(n_i),get_bounded_index(n_ii));
```

We know that we have arrived at the focal cell when the value of `n_i` is equal to `i`, *and* when the value of `n_ii` is equal to `ii`. In this case, the code will hit the `continue` statement, and immediately jump to the next iteration, without executing the remaining code in the loop body.

For all other cells, we will call the function `try_infection(i,ii)`. Notice that when the function receives the coordinates in the form of the two parameters `i` and `ii`, those coordinates now represent a neighboring cell, and we have already taken into account the grid bounding issues by passing them through the function `get_bounded_index(index)` which we developed in chapter 5. In the function `try_infection(i,ii)`, we first check to make sure that the exposed individual is actually susceptible. If so, we infect it with probability `beta`.

Now, with this code in place, let's go ahead and reload the page. You should see an outbreak starting from a single patient 0, expanding rapidly outward:



After a while, your entire world should be infected, and the grid should be completely red. Because we haven't built in recovery yet, infected individuals continue to expose susceptible individuals indefinitely, and each susceptible individual will eventually become infected. This is why 100% of the population will end up being infected.

100% infection - what a terrible outcome! Thankfully, that is rarely the case, for two reasons. The "good" reason is that we have an immune system that in many cases eventually manages to get the infection under control and clear it. Because of immune memory, we can't get infected anymore by the exact same pathogen, at least for some time. The "bad" reason why people would eventually

stop infecting others is that they could succumb to the disease. For example, influenza strains that circulate in birds (bird flu) have occasionally managed to jump the species barrier and infect a human. Unfortunately for the infected person, bird flu in humans is very deadly, and the virus often kills its host before transmitting itself on to the next person.

The standard SIR model assumes that all hosts recover and retain lifelong immunity. So let's go ahead and implement that. This is actually very simple: first, let's just add a line in the `run_time_step()` function, right underneath `expose_neighbors(i,ii)`:

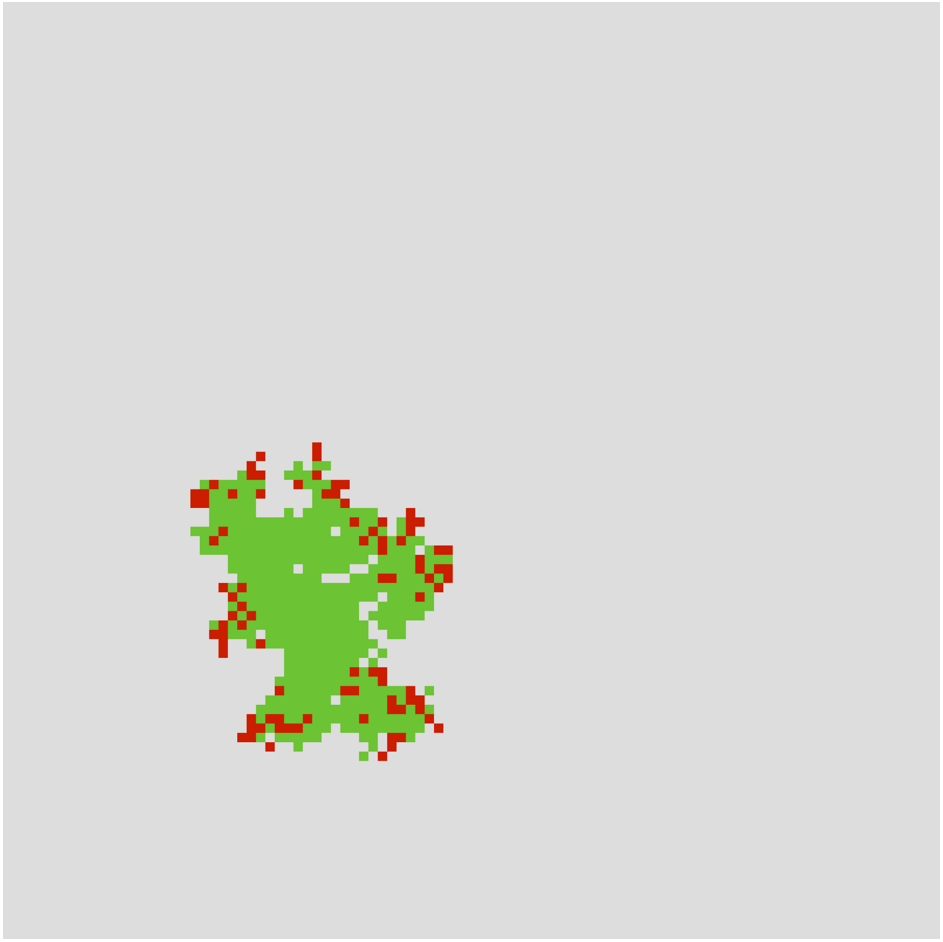
```
…
expose_neighbors(i,ii)
try_recovery(i,ii);
…
```

and then implement the `try_recovery(i,ii)` function:

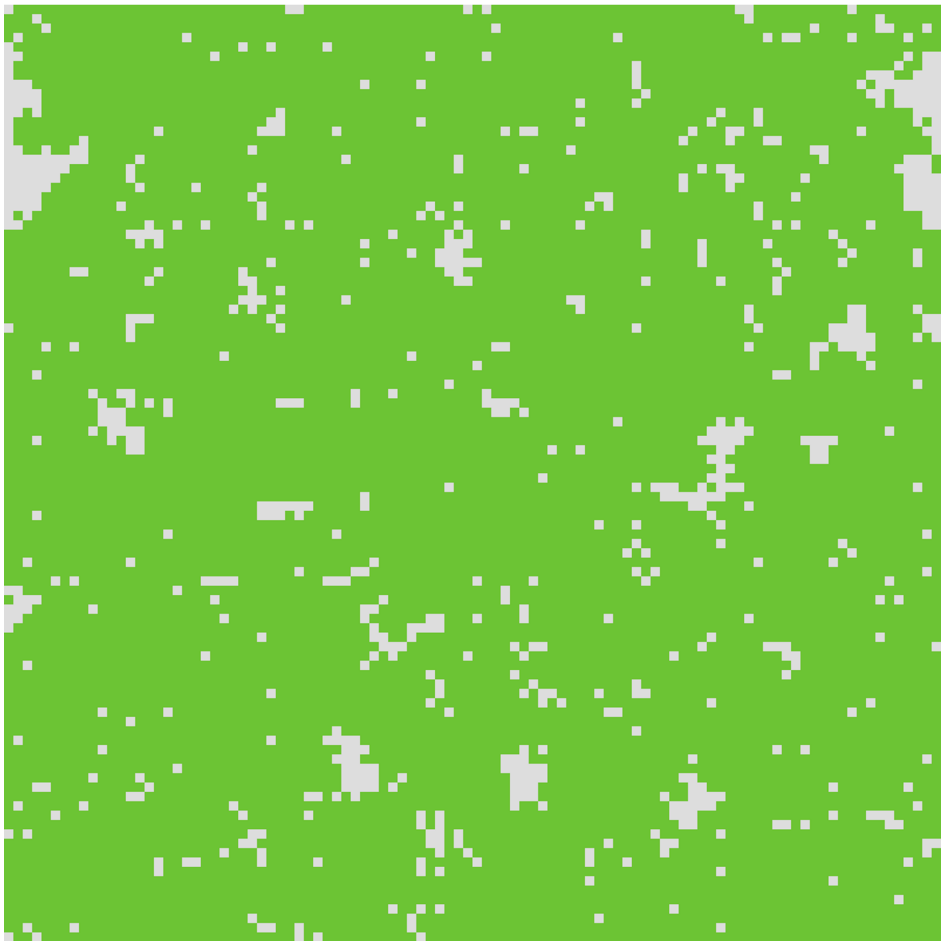
```
function try_recovery(i,ii) {
    if (grid[i][ii] == "I") {
        if (Math.random() < gamma) {
           temp_grid[i][ii] = "R";
        }
    }
}
```

As you can see, this function is pretty straightforward: it checks if the individual at coordinates `i`, `ii` is infected. If that's the case, the individual recovers with probability `gamma`.

Let's go ahead and rerun the simulation. Keep in mind that recovered individuals are colored green:



As you should be able to observe, the outbreak expands at the red / gray edges, where susceptible individuals (in gray) are exposed to infected individuals (in red). However, infected individuals quickly recover and turn green. At that stage, they can't get reinfected, nor can they infect others. Eventually, the outbreak will come to a stop, and you will see something like this:



As you can see, all infected individuals have recovered. However, there are still some susceptible individuals left. How is that possible?

Before we introduced recovery, all individuals got infected. Because infected individuals never recovered, susceptible individuals that were exposed to infected individuals would eventually get infected. It may have taken a long time in some cases, but if would eventually happen.

Once we introduce recovery, however, the dynamics change. Now, infected individuals may recover before they infect neighboring cells. Overall, this might still be a rare event, depending on the values of `gamma` and `beta`. However, it may happen - and when it happens, other neighboring cells are also more likely to share the same fate. Imagine you are the susceptible neighbor of both an individual that just recovered, and an individual that did not get infected because the first neighbor recovered quickly. Your own risk of getting infected is now much lower, because two of your own neighbors are not infected. This is why the remaining susceptible individuals are often clustered in small groups, as you can see in the figure above.

Indeed, the idea that individuals can recover before they infect someone else can have profound consequences right at the beginning of an outbreak. If you are the 1,436th individual to become infected, the consequences of you recovering before passing on the infection are probably small.

However, if you are the very first person with the infection (patient 0), the question of whether you will recover before infecting others is essential: it is the difference between a potentially massive outbreak, and no outbreak at all.

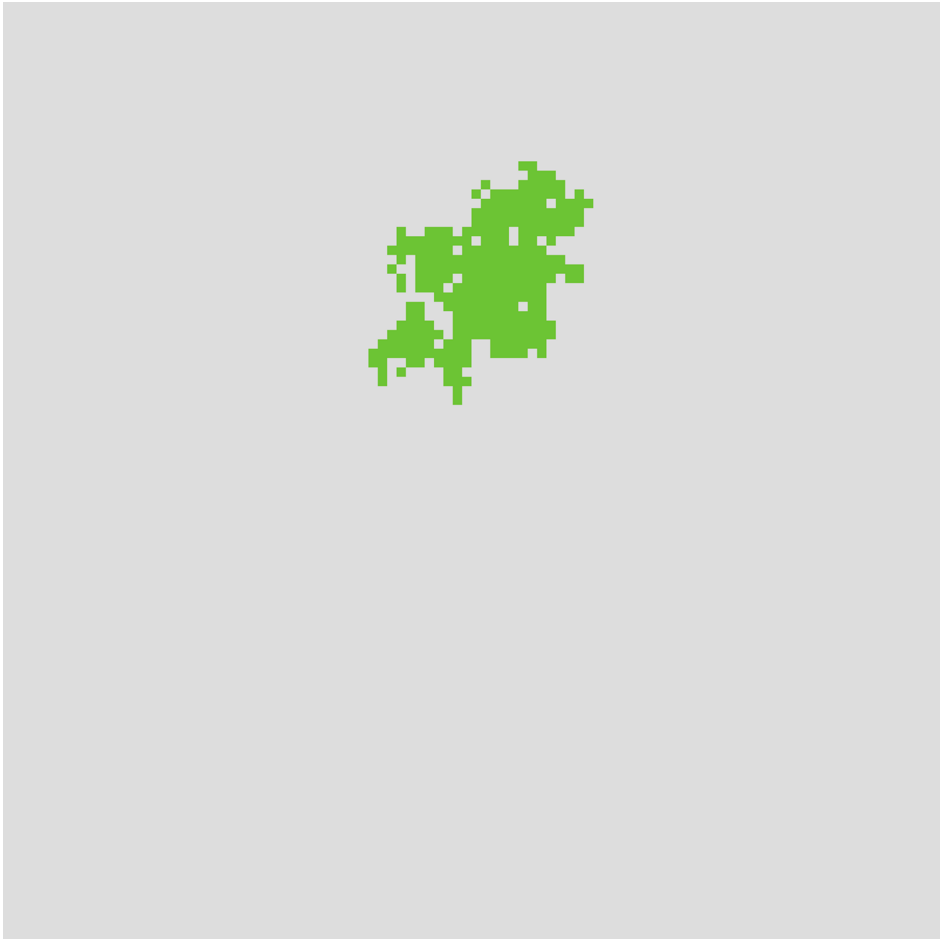
If you rerun your simulation multiple times, it will sometimes end like this:

The first individual has recovered before passing on the infection, and no outbreak occurred. The epidemic hasn't managed to take off. But you'll also notice that in practically all cases where patient 0 does indeed infect other individuals, the epidemic will take off and infect almost the entire grid. Thus, with the current parameters for `beta` (the infection probability per contact and unit time) and `gamma` (the recovery probability per unit time), we find ourselves in an "either or" type of situation: either there is no epidemic, or the epidemic is all-consuming.

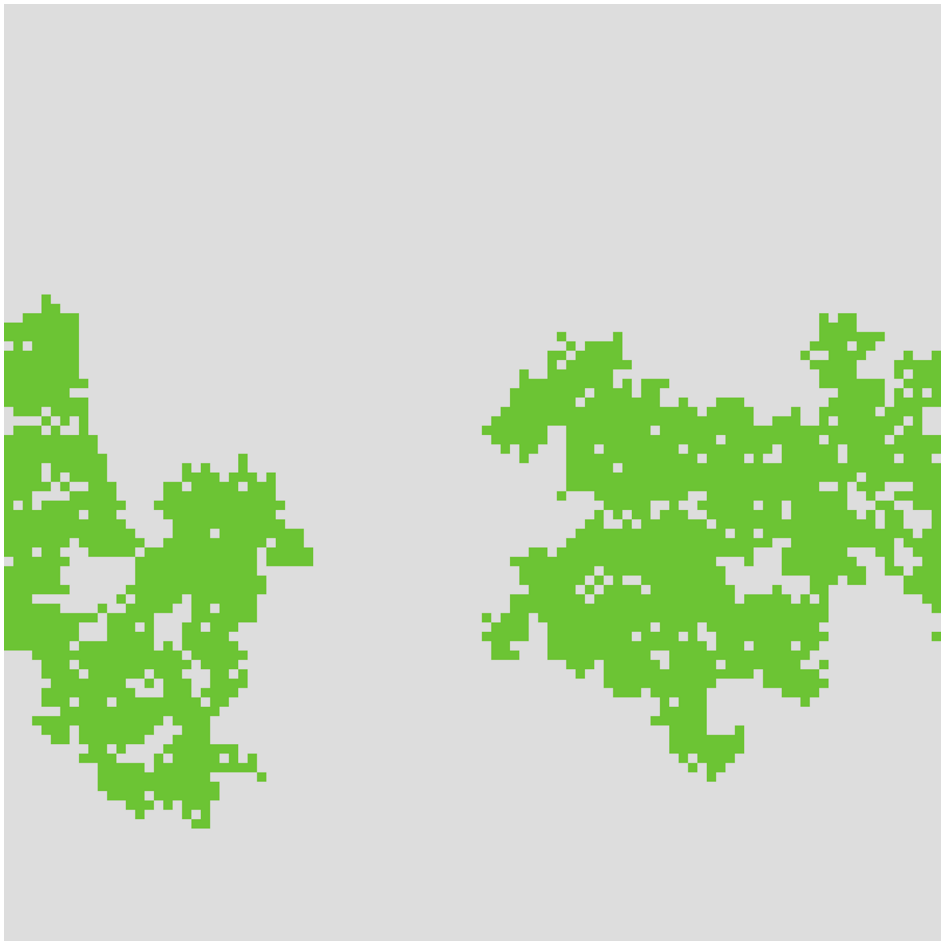
Let's go ahead an increase our recovery probability from `0.1` to `0.15`, by setting `gamma` as follows:

```
var gamma = 0.15;
```

Now go ahead and run a few simulations. You'll find that the size of the epidemic can vary considerably. An outbreak can stop quickly, and be rather small:



or it can go on for quite some time, but eventually still fizzle out:



It's a Small World

Let's now focus on something that is a very active area in contemporary epidemiological research: the role of host contact structure. The host contact structure is the answer to the question "Who is in contact with whom?", i.e. "who could pass the infection on to whom?". Think of the host contact structure as the traffic system on which a disease can spread. If you think about this analogy, it becomes obvious that the structure will affect the dynamics of disease spread. Indeed, if you are in the unfortunate situation of having to commute in a badly designed traffic system, you intuitively grasp the concept.

We have so far focused our exploration on epidemics that occur on a very special host contact structure: a regular grid. The regular grid is a good approximate structure for populations where individuals don't move around a lot, for example plants, or medieval human populations. Clearly, the way we implemented the contact structure is still highly stylized, but it captures the essential feature of spatial spread (again keeping in mind that a model should be as simple as possible but no simpler). On our grid, each individual is in contact with exactly eight other individuals, and these individuals are the spatial neighbors.

We are now going to introduce a very small change to our contact structure, but one that will have dramatic consequences. We are going to assume that 1% of exposed contacts - i.e. contacts between an infected individual and another individual - will not be with a spatial neighbor, but with a *random* individual on the grid. This simulates a situation where an infected individual briefly travels to another location on the grid, and exposes an individual there, rather than a local neighbor.

In order to implement this change, find the following line of code:

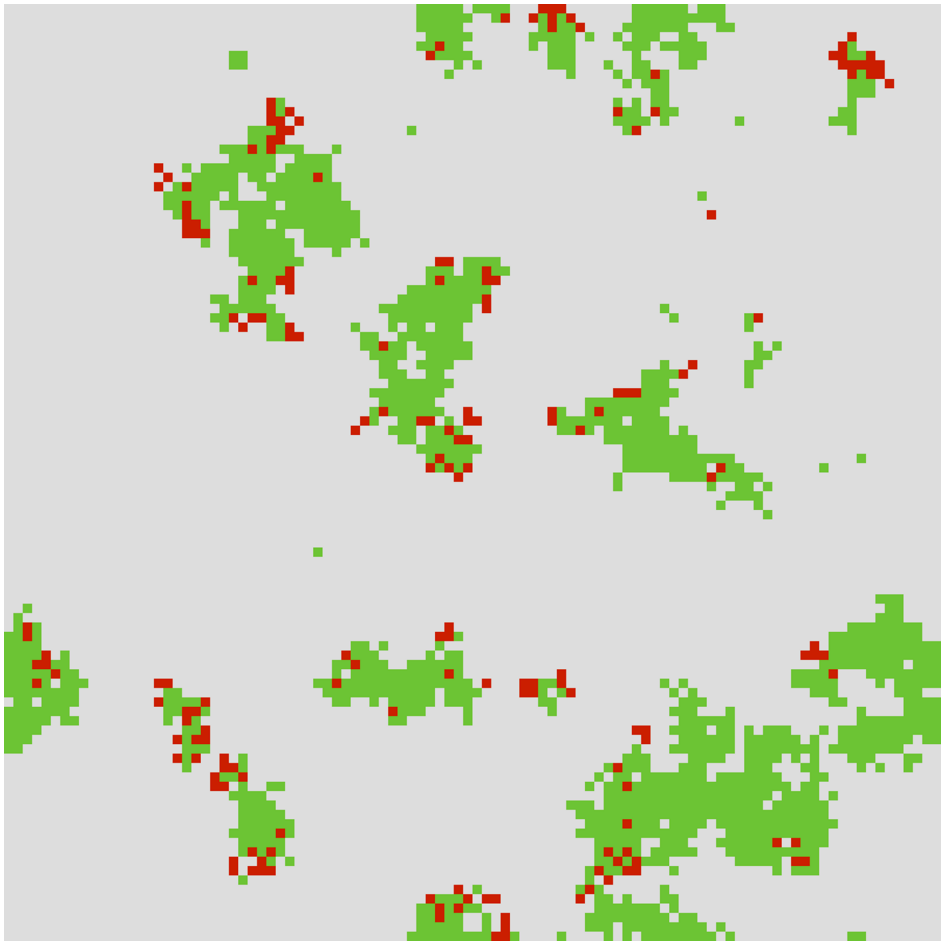
```
try_infection(get_bounded_index(n_i),get_bounded_index(n_ii));
```

and replace it with this:

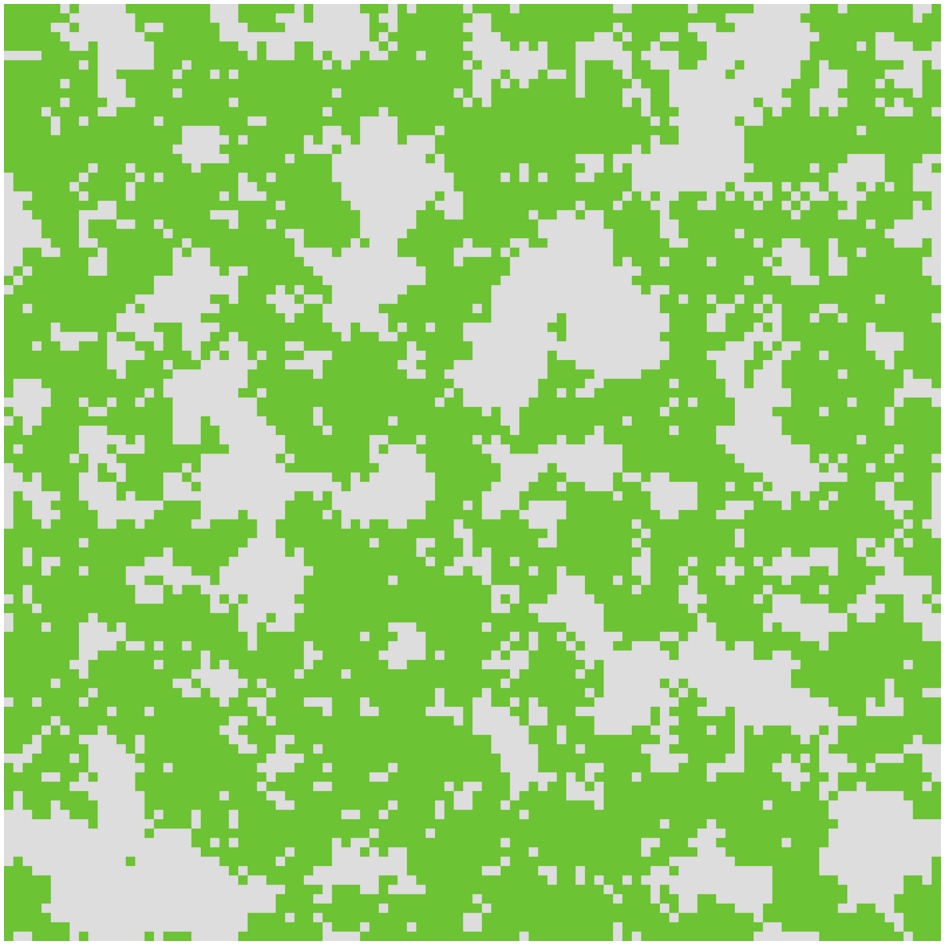
```
if (Math.random() < 0.01) {
    var random_i = get_bounded_index(get_random_int(0,grid_length-1));
    var random_ii = get_bounded_index(get_random_int(0,grid_length-1));
    try_infection(random_i, random_ii);
}
else {
    try_infection(get_bounded_index(n_i),get_bounded_index(n_ii));
}
```

What we are doing here is running the original code in 99% of the time, but in 1% of the time, we expose a random individual, rather than the local neighbor. Notice that we use the same function and parameters to get random coordinates as we used to define patient 0 in the function `init_grid()`.

Run the simulation, and notice that the dynamics are now strikingly different. Before, we would get a local outbreak that would eventually fizzle out on its own. But now, we generally get very large epidemics, and they unfold much faster than before:

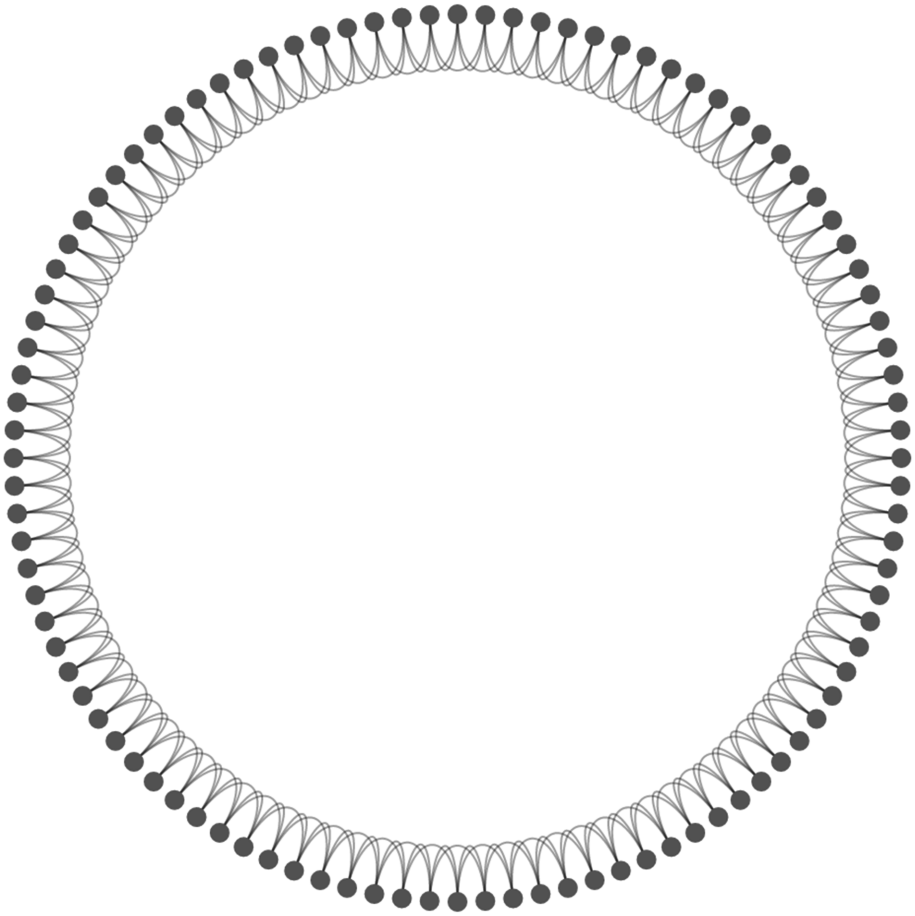


Notice how the epidemic starts in one location, but is quickly spreading in other locations as well, due to these occasional, rare jumps of infected individuals across the grid. Each of the small clusters of recovered and infected individuals that you can make out in the figure above was started by a "traveling" infected individual. The clusters are now growing in parallel, making the overall epidemic progress much faster than before, and usually leaving a pattern like this:

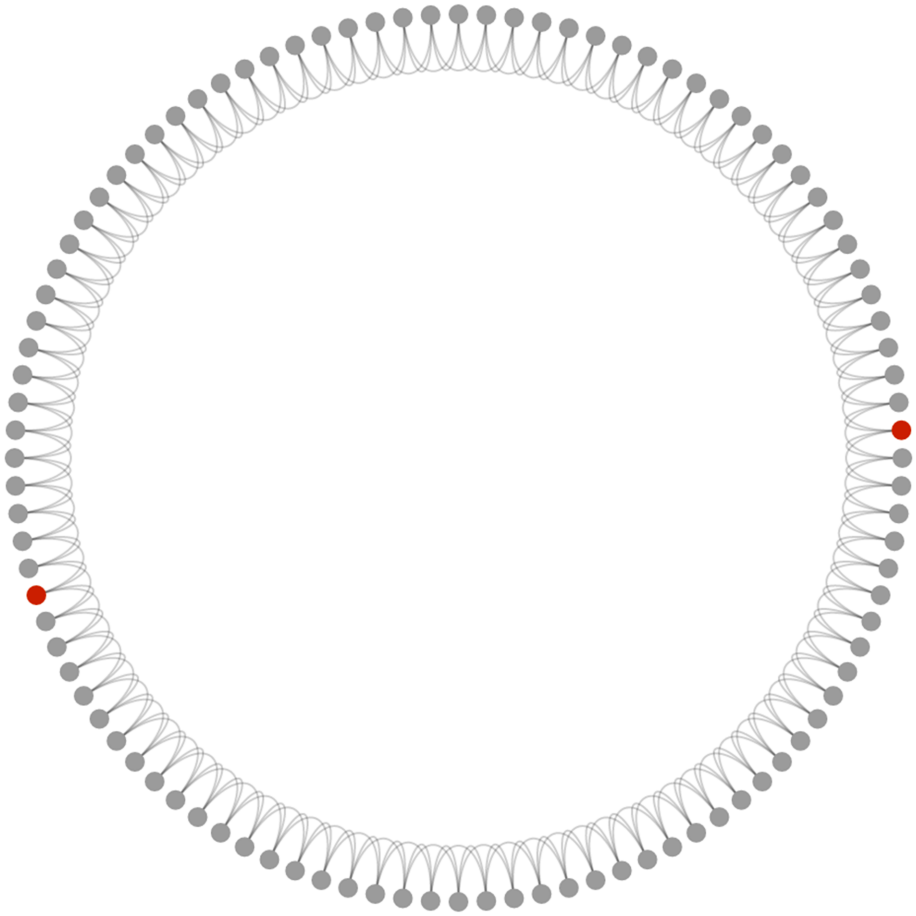


This fascinating example is an example of a so-called *small world* phenomenon. To understand this phenomenon, let's pick two random individuals, and ask ourselves how far they are apart, in terms of the number of contacts you would have to go through to get from one individual to the other.

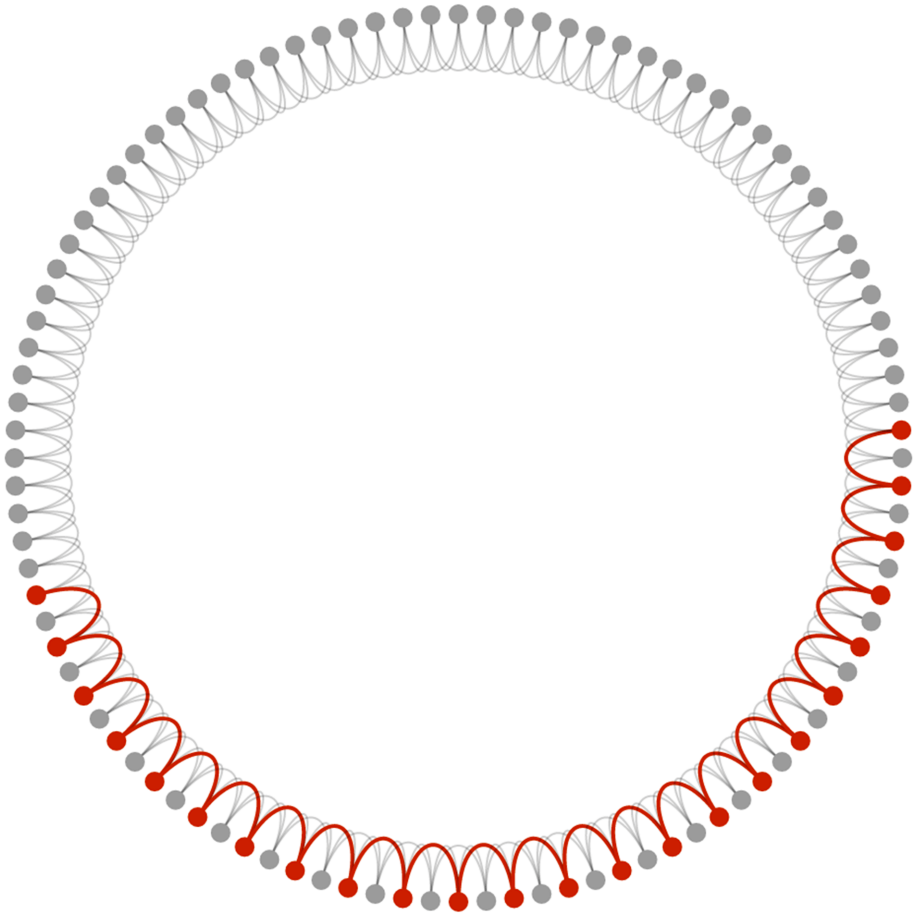
This problem is much easier to visualize if we put individual on the circumference of a circle. Imagine that every individual is connected to its immediate two neighbors on either side (i.e. each individual has a total of four contacts). We could visualize this as follows:



Let's pick two nodes, and think about how long it would take, at minimum, to go from one node to the other. I've color-coded the nodes in red in the figure below:

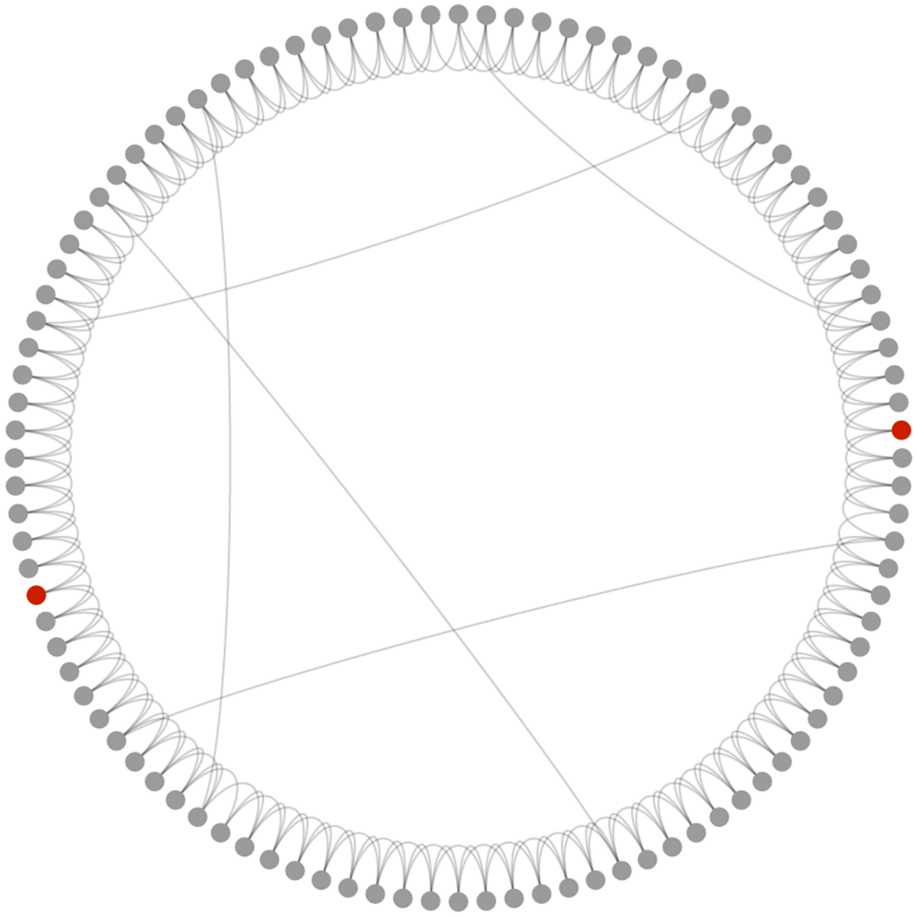


The shortest path between two nodes is the following, color-coded in red also:

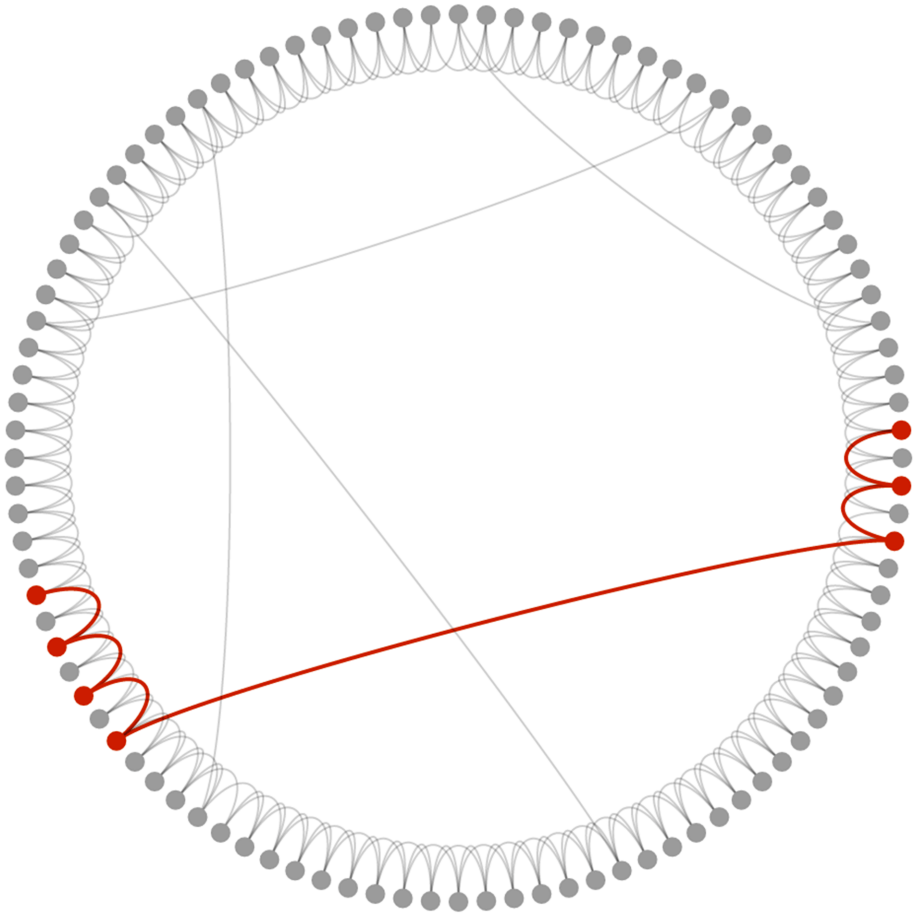


There are 22 other nodes we have to go through to get from one node to the other.

Now, let's go ahead and reconnect a handful of connections at random. Just as in our grid example, imagine that occasionally, rather than being connected to local neighbors only, a node is connected to a random node anywhere on the ring. So let's go ahead and randomly reconnect some of the links:



We only reshuffled 5 links, out of 200. But if we now visualize the shortest path between our two nodes, we can see a dramatic change:



As you can see, there are now only 5 nodes to go through to get from one node to the other. Compared to the previous 22 nodes, that is a more than a four-fold reduction!

Now, before you accuse me of reconnecting the links so that it suits me making a great example: the small world phenomenon is a well-understood phenomenon, even though its broad applicability was only discovered in 1998. In fact, it is now well understood that reconnecting 1% of the links can reduce the average path length more than ten-fold, dependent on the settings.

For our discussion of infectious disease dynamics, this insight has enormous implications, as our last simulation has demonstrated. In a small world - i.e. a world that is mostly dominated by local interactions, but that also has some fraction of "global" interactions - diseases can spread much more rapidly. In addition, as we have seen in our example, diseases are also more likely to cause large outbreaks, because they can start new local outbreaks even when earlier local outbreaks are about to fizzle out.

But rather than taking my word for it, let's use our coding skills to investigate if this is really true. It is sometimes easy to convince yourself of something based on a few visual examples. It's always

better to verify your intuition with hard data. Let's go ahead and do just that.

The first thing we'll do in our simulation is to turn off the visualization, since we don't need the visual clues anymore. Instead, we want to run multiple simulations, and count, in each simulation, how many individuals got infected. In order to do that, let's replace this code:

```
draw_grid(grid,["S","#dcdcdc","I","#c82605","R","#6fc041"]);

function simulate_and_visualize() {
    run_time_step();
    update_grid(grid,["S","#dcdcdc","I","#c82605","R","#6fc041"]);
}

setInterval(simulate_and_visualize, 50);
```

with this:

```
function run_simulation() {
    init_grid();
    while (get_number_of_infected() > 0) {
        run_time_step();
    }
}

for (var i = 0; i < number_of_simulations; i++) {
    run_simulation();
    data.push(get_number_of_recovered());
}
```

As you read through the new code, you'll find that it makes use of a few variables and functions we haven't defined yet. We'll get to that in a minute, but let's first go through the code and try to understand the logic.

The function `run_simulation()` simply (re)sets the grid, then calls the function `run_time_step()` for as long as there are infectious individuals around. Then, we set up a loop that calls the `run_simulation()` function a certain number of times (defined by the variable `number_of_simulations`), and stores the number of recovered individuals in a `data` array. The number of recovered individuals corresponds to the size of the outbreak, since each recovered individual was infected at one point.

You may have noticed something curious. Take a look at the first line of the new for loop:

```
for (var i = 0; i < number_of_simulations; i++) {
```

Notice something unusual?

Until now, we would have written this line as follows:

```
for (var i = 0; i < number_of_simulations; i = i + 1) {
```

Here, I've introduced a common shorthand notation that is available in JavaScript (and many other languages). Instead of writing

```
something = something + 1
```

if you want to increase the value of `something` by 1, you can use the shorthand notation and write

```
something++
```

There isn't any particular advantage to this shorthand notation - it's simply shorter. However, it's very commonly used, and I wanted to make sure you knew about it.

There is also the equivalent of decreasing a value by 1, so that instead of writing

```
something = something - 1
```

you can write

```
something--
```

There are a few other shorthand notations in the same spirit. If you want to modify a variable by adding to it, subtracting from it, multiplying or dividing it, you can use the following shorthand notations:

	Full version	Shorthand
Addition	`something = something + value`	`something += value`
Subtraction	`something = something - value`	`something -= value`
Multiplication	`something = something * value`	`something *= value`
Division	`something = something / value`	`something /= value`

With that out of the way, let's modify the code to make sure we have everything set up. First, we are calling `init_grid()` for every single generation run in the `run_simulation()` function - so be sure to remove the one `init_grid()` call that we used right after we defined the function (it wouldn't be a drama if you'd forget that - it would simply be an unnecessary function call in the beginning).

Next, we need to define the methods `get_number_of_infected()` and `get_number_of_recovered()`. Here's how we can do that:

```
function get_number_of_infected() {
    return get_number_in_state("I");
}

function get_number_of_recovered() {
    return get_number_in_state("R");
}

function get_number_in_state(state) {
    var number_in_state = 0;
    for (var i = 0; i < grid_length; i++) {
        for (var ii = 0; ii < grid_length; ii++) {
            if (grid[i][ii] == state) {
                number_in_state++;
            }
        }
    }
    return number_in_state;
}
```

What I'm doing here is to first define a generic version of the function called `get_number_in_state()`, where I'm going through all the cells and count the number of cells that are in a given state, passed to the function as an argument. That would in principle be sufficient, and I could just call

```
get_number_in_state("I")
```

if I wanted to get the number of individuals that are infected. However, I'm adding the two convenience functions `get_number_of_infected()` and `get_number_of_recovered()` just to make the code a little cleaner and more readable. These functions are called *convenience* functions because they don't do any heavy lifting by themselves, they just call the appropriate functions for us.

We're almost done - all that is left to do is to define two new variables, `number_of_simulations` and `data`. Let's add them to our list of variables at the beginning of the code:

```
var data = [];
var number_of_simulations = 100;
```

We're all set! Now that we're storing the size of each epidemic (measured in number of individuals infected) in the data array, we can start to look at the data. The first obvious thing to do is to look at the epidemic sizes that we collect. So at the end of our code, let's simply add

```
console.log(data);
```

Before you run it, one more thing - remember we want to compare the full spatial grid with the small-world grid that has occasional random contacts? Recall that the random contacts are introduced in the function `expose_neighbors(i,ii)`, like so:

```
if (Math.random() < 0.01) {
    var random_i = get_bounded_index(get_random_int(0,grid_length-1));
    var random_ii = get_bounded_index(get_random_int(0,grid_length-1));
    try_infection(random_i, random_ii);
}
else {
    try_infection(get_bounded_index(n_i),get_bounded_index(n_ii));
}
```

If you want no random contacts, that probability would have to be `0`. Let's do ourselves a favor and capture this probability in a variable, rather than having this value hardcoded in this function. So go ahead and replace

```
if (Math.random() < 0.01)
```

with

```
if (Math.random() < rewiring_probability)
```

And, of course, be sure to define the variable `rewiring_probability` at the top of our code where we set up all the other variables:

```
var rewiring_probability = 0;
```

Starting with a fully local grid (i.e. no random contacts), my output of epidemic sizes for 100 simulations is as follows:

```
                                                    example.html:203
[1020, 2, 109, 1360, 194, 27, 356, 308, 19, 579, 2, 453, 1, 1268,
202, 1, 70, 73, 1, 127, 7, 472, 3, 119, 381, 429, 6, 221, 660, 59,
1028, 143, 89, 4, 826, 2, 252, 2, 240, 38, 13, 1, 2, 2, 1, 1, 1, 3,
1, 803, 1, 3, 1, 1510, 5, 1539, 4, 5, 1, 9, 1, 156, 23, 90, 1, 201,
22, 34, 894, 101, 1, 3589, 1, 541, 1, 3, 923, 5, 1, 2, 836, 180,
376, 1, 638, 2491, 1, 2406, 3, 11, 75, 1813, 1370, 8, 3022, 1, 1,
1202, 39, 107]
>
```

As you can see, many epidemics don't even get started - the outbreak is limited to one infected individual, patient 0. The others fizzle out very quickly, while some infect hundreds of individuals; and a few even over 1,000 individuals (the maximum is 3,589 individuals, which is about 37% of the population).

Now let's change a few contacts to be random, rather than local, as we have done before. Go ahead and set the rewiring probability to 1%:

```
var rewiring_probability = 0.01;
```

and run the code again. This time, my output is the following:

```
                                                          example.html:203
[6475, 6476, 6848, 6240, 1, 2, 1, 5956, 6541, 2, 1, 1, 2, 4, 5918,
1, 1, 6720, 6960, 33, 6290, 1, 2, 5770, 1, 8, 6568, 6533, 18, 5449,
5749, 6863, 1, 2, 69, 8, 8, 6157, 15, 1, 6544, 6464, 6655, 6155,
6707, 6008, 7, 6448, 2, 17, 2, 6721, 6485, 61, 6329, 7124, 6477,
25, 1, 7003, 6, 4819, 1, 6836, 6261, 1, 6524, 75, 6561, 115, 5640,
5099, 7053, 5419, 6626, 27, 6181, 6432, 6759, 6868, 16, 5976, 6451,
3, 6474, 5689, 5874, 3, 6232, 7037, 6978, 6347, 6801, 6582, 2, 13,
4, 409, 6706, 6470]
>
```

We can instantly see that while some epidemics still don't get started, those that do end up being very large, very often more than 6,000 individuals. The largest epidemic in this run infected 7,124 individuals - more than 71% of the population.

The extreme epidemic size seems to be about twice as large - but what about the average? Let's go ahead and find out. We can define a function that calculates the average of the values in the data like so:

```
function calculate_average_size(data) {
    var sum = 0;
    for (var i = 0; i < data.length; i++) {
        sum += data[i];
    }
    return sum / data.length;
}
```

(note the use of +=) and then output the return value of that function, rather than outputting the individual values:

```
//console.log(data);
console.log(calculate_average_size(data));
```

Before running this code, let's be sure to first reset the rewiring probability to `0` in order to run completely local simulations with no random contacts. After doing that, I get the following values - recall that these are averages from 100 independent simulations - when I run the code a few times:

```
364.07
523.61
417.86
414.74
382.64
```

Thus, the average epidemic size seems to be around 400 individuals, or about 4-5% of the population.

When I set the rewiring probability to `0.01` in order to run small world simulations with a few random contacts, I get the following values:

```
3332.8
2987.95
2511.06
2790.62
3418
```

which corresponds to about 25-30% of the population. Thus, our visual intuition is confirmed - turning just a few local contacts into random contacts has a dramatic effect, resulting in outbreaks that are on average about six times larger.

We could go one step further and ask, what is the average outbreak size, given that an outbreak occurs in the first place? Or in other words, if we ignore all those cases where patient 0 does not infect at least one other individual, what is the average epidemic size? In order to answer this question, we need to adapt the function `calculate_average_size(data)` slightly:

```
console.log(calculate_average_size(data, 2));

function calculate_average_size(data, min_size) {
    var count = 0;
    var sum = 0;
    for (var i = 0; i < data.length; i++) {
        if (data[i] >= min_size) {
            sum += data[i];
            count++;
```

```
        }
    }
    return sum / count;
}
```

Here, we've added a second argument to the function, `min_size`, which defines the minimum epidemic size required to be included in the calculation. Thus, in order to include only epidemics that are larger than just one individual, the parameter to be passed as `min_size` needs to be `2`. Also note that we need to define a new local variable (`count`) that keeps track of the number of data points that fulfill our minimum size requirement.

Resetting `rewiring_probability` to `0` again, the new code yields the following values (I'm showing only two significant places):

```
583.29
543.41
526.24
269.40
510.23
```

whereas simulations with a rewiring probability of `0.01` yield

```
3500.33
3941.92
3790.97
4284.65
3891.30
```

In short, while the specific numbers change, the broad picture remains the same. Even if we would require at least `10` individuals to be infected in order to be included in the calculations, we would get an average of about 600 individual infected on the fully regular grid, and about 5,000 individuals infected on the small world grid.

Thus, a few non-local interactions can have quite dramatic effects. This is the main reason why travel, especially long-distance travel - has made our world so much more connected, and why infectious diseases in the most remote places in the world are a threat to public health anywhere in the world.

And that's where we wrap it up. One could write volumes of books on this fascinating topic, and we have once again barely scratched the surface. However, I hope you've learned a few key insights about how we can create virtual epidemics in a computer. You can imagine that we could use such computational simulations to investigate how certain interventions (like vaccination programs, or quarantines) would affect the dynamics of epidemics. Indeed, modeling of infectious

disease dynamics has become a major field of interest. Obvisouly, one cannot run experiments about epidemics on human populations, and real-world epidemics can't be replayed again and again (thankfully!). Recreating epidemics with computer models is a great way to run millions of epidemics under varying circumstances, and the insights gained from these models help to inform real-world public health.

Let's briefly summarize what we've learned in this chapter.

- Infectious diseases can be modeled with SIR models, where individuals are either susceptible, infected, or have recovered.
- The dynamics of epidemics are shaped by how well the infection can be transmitted, and by how fast infected individuals recover.
- Because of the stochastic nature of transmission, not every patient 0 will trigger an outbreak. An epidemic may fizzle out on its own in the early stages.
- Even when an epidemic takes off, not everyone will get infected: the epidemic will eventually run out of susceptible individuals to infect.
- Small changes in the host contact structure can have dramatic effects on epidemics. Compared to a host structure with only local contacts, a small world host structure will generate dramatically larger epidemics
- We've also introduced a few widely used shortcuts for arithmetic manipulations in JavaScript.

Despite the deep fascination of parasites, and the infectious disease they cause, I would like to continue the book on a more cheery note. In the next chapter, we'll dig into game theory, and use our coding skills to show how cooperation among animals (including humans!) can evolve.

8. Cooperation: Good Guys Can Finish First

In this chapter, we are going to explore a simple question: Why is there cooperation? The narrative of evolution - “struggle for life”, “survival of the fittest”, “the selfish gene”, etc. - seems to suggest, superficially, that cooperation is a mystery. Evolution, this line of reasoning goes, is about cutthroat competitiveness, and raw self-interest: eat or be eaten. In such a world, why would individuals be cooperative or altruistic? This type of behavior should be wiped out by natural selection. But then why are these behaviors widespread in nature?

It turns out that natural selection and cooperation can be perfectly compatible. This has been one of the most exciting findings in evolutionary biology in the past few decades. The evolution of cooperation is still a very active research area, and we will once again only be able to scratch the surface of this exciting field. However, my goal is to convince you that we should not be puzzled by - and indeed often expect - cooperative behavior among individuals under many circumstances. And as before, I don’t want you to take my word for it: I want you to create a virtual world in code, where individuals are subject to ruthless natural selection. And we will find that in such a world, cooperation can readily evolve, given certain conditions.

First, we will introduce a formal model than can be used to study the evolution of cooperation. Then, we’ll see that in a very simple implementation of that model, cooperation doesn’t stand a chance, confirming our original intuition that cooperation and natural selection are not necessarily natural bedfellows. Following that, we’ll show that a few simple but realistic modifications to the model will give rise to cooperative behavior.

In addition, this chapter introduces a brand new programming concept: *objects*. Objects are the heart and soul of JavaScript, but as I hope to have demonstrated throughout the book, you can do an awful lot in JavaScript without ever knowing about objects. Nevertheless, understanding what objects are, and how to use them, enables you to write both cleaner and more complex code.

Let’s get started!

Game Theory

The most common framework for studying cooperation is that of *game theory*. Game theory is the study of strategic decision making. It was originally developed to study economic behavior, but was later adapted by evolutionary biologists to study behavior in non-human species as well.

Game theory gets its name from games, which are played by “players” who can act in certain ways, given some information. The outcome of the game is a payoff. When applied to economic

theory, the payoff is usually monetary. However, when applied to evolutionary biology, the payoff is evolutionary fitness.

The game that we would like to play here is as follows. Since we are interested in cooperation, we are going to assume that an individual can either *cooperate*, or *defect* (i.e. not cooperate). These are the only two possible actions. Given these two actions, or strategies, there are four possible combinations of two players playing a game:

	Individual 2 Defecting	**Individual 2 Cooperating**
Individual 1 Defecting	Payoff Individual 1: ? Payoff Individual 2: ?	Payoff Individual 1: ? Payoff Individual 2: ?
Individual 1 Cooperating	Payoff Individual 1: ? Payoff Individual 2: ?	Payoff Individual 1: ? Payoff Individual 2: ?

In order to define the game, we now have to come up with the fitness values for each individual in each of the four possible combinations. By doing so, we are generating a payoff matrix that defines a game.

If we broadly think of cooperation as paying a cost, *c*, to help another player receive a benefit, *b*, we can fill in the payoff matrix generically as follows:

	Individual 2 Defecting	**Individual 2 Cooperating**
Individual 1 Defecting	Payoff Individual 1: 0 Payoff Individual 2: 0	Payoff Individual 1: b Payoff Individual 2: -c
Individual 1 Cooperating	Payoff Individual 1: -c Payoff Individual 2: b	Payoff Individual 1: b-c Payoff Individual 2: b-c

If the benefit is larger than the cost (i.e. $b > c$), this payoff matrix represents the so-called "Prisoner's dilemma" game, which is a classic game in evolutionary game theory. The game was originally developed in a non-biological context, but its logic can easily be applied to many biological situations. The gist of the game - and the reason why it is called a dilemma - is as follows. In essence, it would be best for a pair of players to cooperate - they would get a positive payoff of *b-c*, which is larger than 0. On the other hand, if they both defected, they would not receive any payoff (0). However, if one of them defects, and the other cooperates, the defector receives the benefit *b*, without having to pay the cost *c*, while the cooperator has to pay the cost *c*, without receiving the benefit *b*.

The consequence of this payoff matrix is that *it is always in the selfish interest of individuals to defect*. Why is that? Imagine you are playing this game, and you are contemplating your move - should you cooperate, or defect? If you expect the other player to cooperate, it will be better for you to defect, because your payoff will be *b* instead of the smaller *b-c*. Equally, if you expect the other player to defect, it will also be better for you to defect, because your payoff will be 0, which is still better than the negative payoff of *-c*. In short, no matter what the other player plays, your best move, selfishly, is to defect. Sadly, the outcome of this is that all individuals will defect, leading

to the worst possible outcome for all interactions. It would be much better for everyone if everyone cooperated (they would all receive a positive payoff of *b*-*c*), but because it is individually better to defect, everyone will end up defecting, receiving no payoff at all.

It's important to recall that in a biological situation, the payoff is measured in evolutionary fitness (and as always, we care about the relative fitness values, rather than the absolute values). If we assume that playing a strategy is genetically encoded, a higher payoff simply means that an allele causing the individual to play a certain strategy will on average leave more of its copies in the next generation - and we'll get evolution by natural selection, as discussed in chapter 6. Of course, behavior is generally a very complex mechanism, influenced by many genes as well as by the environment. Thus, while there is certainly no one gene or allele "for" a complex behavior, we will continue to use this language metaphorically. As long as there is inheritance of whatever it is that causes individuals to play certain strategies, the logic of natural selection applies.

Let's go ahead and implement this game in JavaScript.

A Game in Code

First, as always, we'll set up some key variables:

```
var c = 1;
var b = 4;

var population = [];
var population_size = 100;
var number_of_time_steps = 1000;
var mutation_rate = 0.001;

var strategies = ["ALL_C","ALL_D"];
var data = [];
```

The variables `c` and `b` define the costs and benefits in our model. The array `population` is where we will store all the individuals in the population, and `population_size` defines how many individuals we will keep track of in the population. We'll be running a simulation for `number_of_time_steps` time steps, and we'll mutate strategies at the rate given by `mutation_rate` (more about that below). There are two possible strategies for an individual to take: `ALL_C` (meaning the individual will always cooperate) or `ALL_D` (meaning the individual will always defect). We store these two strategies as strings in the array `strategies`. Finally, we will store the frequency of the strategies in the `data` array for easy plotting.

You can think about time steps as generations, if you want to. But you could also imagine that time steps are simply arbitrary time intervals in which everyone in a population plays a game. I'll stick with the term time steps, rather than generations, because it is a more generic term.

To run a simulation, we will define, and immediately call, the function `run_simulation()`. Here's what it'll look like:

```
function run_simulation() {
    init_simulation();
    for (var i = 0; i < number_of_time_steps; i++) {
        run_time_step();
    }
}

run_simulation();
```

The function `run_simulation()` first calls another function, `init_simulation()`, to set up the initial population and the data collection. Then, using a `for` loop, we'll call the function `run_time_step()` as many times as defined by the variable `number_of_time_steps`.

Let's go ahead and implement these two functions. The function `init_simulation()` looks as follows:

```
function init_simulation() {
    for (var i = 0; i < population_size; i++) {
        population.push(new Individual("ALL_C",0));
    }
    for (i = 0; i < strategies.length; i++) {
        data.push([]);
    }
}
```

As you can see, there are two `for` loops used here: first, to initialize the population, and second, to set up the data collection using the `data` array. The second `for` loop should be self-explanatory at this stage: it simply adds an empty array to `data` for each strategy. So once again, `data` will be a two-dimensional array, where the inner arrays will contain the frequencies for the corresponding strategy at each time step.

The first `for` loop needs a more extensive discussion. The loop itself is rather simple: it simply pushes elements into the `population` array. But what kind of elements? Well, our `population` array is supposed to keep track of all the individuals. In the previous examples in this book, we kept track of individuals simply as genotypes, or alleles, or as states, usually by using a number or a string (such as `"A1"`, `"A2"`, `"S"`, `"I"`, `"R"`, etc.). However, many times, the elements that we want to store are more complex than that. For example, for the current model, we would like to store both a strategy - such as "always cooperate" or "always defect" - and a numerical payoff value for each individual. We could go ahead and store these values in an array:

```
population.push(["ALL_C",0]);
```

But then, we would always have to remember that the strategy is stored as the first element of the array, and the payoff as the second element. And what if we added more values to an individual? We would have to extend the array. And let's hope we would never come up with the idea to change the order in which we store these individual properties in the array - we would have to change the code everywhere we access the properties! This sounds like a recipe for disaster.

Enter *objects*. Objects are a very powerful concept, and yet they are very easy to understand. In JavaScript, an object is simply a container of properties. Properties are named values, i.e. they have a name, and a value. You can define an object using the shorthand notation `{}`.

Recall how we introduced arrays in chapter 3 - we saw that you can initialize an empty array using the shorthand notation `[]`, like so:

```
var array = [];
```

or you can initialize the array with some values right away:

```
var array = [1,4,7];
```

Similarly, you can initialize an empty object like this:

```
var object = {};
```

and alternatively, you can initialize the object with a few properties right away:

```
var object = {"first_name":"marcel", "last_name":"salathe"};
```

As you can see, this object contains two properties. One has the name `"first_name"`, and the value `"marcel"`; the other has the name `"last_name"`, and the value `"salathe"`. Property names can be strings, but you can omit the quotes if the names are legal JavaScript variable names (which I highly recommend they are). In other words, you could initialize the same object as follows:

```
var object = {first_name:"marcel", last_name:"salathe"};
```

So that's how you set the property values - but how do you retrieve them when you need them later on? JavaScript provides two notations to access property values: the *dot notation*, and the shorthand notation `[]`. For example, in order to retrieve the value of the property `first_name`, you could use either

```
object.first_name
```

or

```
object["first_name"]
```

independently of how you defined it. Throughout the rest of this book, I will use the dot notation, which I prefer because it is shorter and cleaner.

Now, for the purpose of our model, we would like to store individuals as objects with two properties. Those properties are (i) the strategy of the individual, and (ii) the payoff. The strategy can be of value `"ALL_C"` or `"ALL_D"` (for cooperator and defector, respectively), and the payoff is a numerical value initially set to `0`. Thus, we could create individuals like so:

```
var individual_0 = {strategy:"ALL_C", payoff:0};
var individual_1 = {strategy:"ALL_D", payoff:0};
var individual_2 = {strategy:"ALL_C", payoff:0};
var individual_3 = {strategy:"ALL_D", payoff:0};
...
```

But this is a bit cumbersome, especially if we repeatedly want to generate new objects of the same kind. It turns out that JavaScript provides a very handy object generator. It is called *constructor function*, which is simply a function that by convention starts with a capital letter, and is invoked with the `new` keyword. Let's take a look at this step by step.

Like any function, a constructor function can take any number of arguments. For example, a constructor function for an individual could look like this:

```
function Individual(strategy, payoff) {

}
```

This function doesn't do anything - yet. What we would like to do, though, is for the function to create a new object, assign values to the properties, and return the object. It turns out that when we invoke the function with the `new` keyword, like so:

```
var example_individual = new Individual("C",0);
```

then the constructor implicitly creates a new empty object, assigns it to a local variable called `this`, and returns `this` at the end of the function. Thus, if we want to assign the parameter values to the corresponding properties, our constructor function should look like this:

```
function Individual(strategy, payoff) {
    this.strategy = strategy;
    this.payoff = payoff;
}
```

Note that we neither have to write `var this = {};` at the beginning, nor `return this;` at the end - both of these things happen automatically if the function is invoked with the keyword `new`. Which is of course exactly what we are doing in the first `for` loop in the function `init_simulation()`:

```
for (var i = 0; i < population_size; i++) {
    population.push(new Individual("ALL_C",0));
}
```

Thus, in every iteration of this loop, we create a new object using the `Individual` constructor function, and each of these created objects has two properties, `strategy` and `payoff`, which are set to `"ALL_C"` and `0`, respectively.

After this brief detour to introduce JavaScript objects, let's return to our function `run_simulation()`, which repeatedly calls the function `run_time_step()` which is currently undefined - let's fix that and define `run_time_step()` like this:

```
function run_time_step() {
    games();
    selection();
    mutation();
}
```

As you can see, this function does not do much by itself, other than calling three functions in sequence: `games()`, which is the function where all individuals play their games; `selection()`, which is the function that replaces the strategies of individuals that have low payoffs with strategies of individuals that have high payoffs; and `mutation()`, which introduces the occasional random strategy change. Let's take a look at each of these functions in detail.

The `games()` function is where we make all individuals play a game against another, randomly chosen individual. We can implement this as follows:

```
function games() {
    for (var i = 0; i < population_size; i++) {
        var current_individual = population[i];
        var other_individual = get_other_individual(current_individual);
        play_game(current_individual, other_individual);
    }
}
```

This function loops through all individuals, picks another, random individual, and then makes the two individuals play the game. Let's first implement the function `get_other_individual()`:

```
function get_other_individual(individual) {
    do {
        var random_index = get_random_int(0,population_size-1);
        var other_individual = population[random_index];
    }
    while (other_individual == individual);
    return other_individual;
}

// get_random_int helper function that we developed in chapter 5
function get_random_int(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

As you can see, the reason why we pass an `individual` to this function is so that we can make sure that the other, random individual truly is another individual: otherwise, we would occasionally play a game against ourselves. Playing the game against yourself would give you an unfair advantage because you would potentially receive twice the payoff. We can easily ensure that the other individual is not equal to the individual we passed in as an argument by comparing whether they are equal, as we do in the line

```
while (other_individual == individual);
```

What the function does is to pick a random individual for as long as it is necessary (i.e. for as long as the two individuals are in fact the same individual).

There is a JavaScript gotcha here that I want you to be aware of. Until now, whenever we compared values and tested for equality, we were comparing so-called *primitive* types such as numbers (e.g. `1`, `2`), booleans (`true`, `false`), and strings (e.g. `"A1"`, `"A2"`). These values are called primitive types because they are not objects. When you are comparing primitive types, you are comparing their

values. However, when you are comparing objects, you are not actually comparing their values - you are simply comparing their *references*. What does this mean?

A reference is essentially an address in memory where your object lives. When you compare two objects with the equality operator ==, you are not comparing the values of their properties; you are simply comparing the references.

Consider the following code:

```
var obj1 = {foo:1, bar:2};
var obj2 = {foo:1, bar:2};
console.log(obj1 == obj2);
```

This code would write `false` into the console. The reason is that even though the two objects have the exact same properties (identical names and values), they are different in that they live at two different addresses in memory. They literally are two different objects.

You may ask yourself, can objects ever live at the same address? Yes, they can - consider the following code which is only slightly changed from the code above:

```
var obj1 = {foo:1, bar:2};
var obj2 = obj1;
console.log(obj1 == obj2);
```

This code would write `true` into the console. The main reason lies in the second line, when `obj2` is assigned the value of `obj1`. **Oops, I misspoke!** In fact - and that is the main message here - it is the *reference*, not the *value* of `obj1`, that is assigned to `obj2`.

The official wording for this in computer science is that in JavaScript, objects are *passed by reference*, rather than by value. Primitive types, on the other hand, are passed by value.

This has a number of very important implications. Consider this code:

```
var a = 5;
var b = a;
console.log(a,b);
a = 3;
console.log(a,b);
```

Can you guess what this code will do? At first, you would see

```
5 5
```

in the console, because both variables have the value `5`. When `a` is overwritten to be of value `3`, `b` is not changed, and the output is

```
3 5
```

as we would intuitively expect.

However, things change when it comes to objects. Let's look at the following code with objects, which is the same code in spirit as the one above:

```
var a = {foo:1, bar:2};
var b = a;
console.log(a,b);
a.foo = 3;
console.log(a,b);
```

At first, you will see the output

```
Object {foo: 1, bar: 2} Object {foo: 1, bar: 2}
```

(Your output might look slightly different depending on your browser.) But then, you might be surprised to see the second output, which reads:

```
Object {foo: 3, bar: 2} Object {foo: 3, bar: 2}
```

You may have thought that you had only changed the value of the `foo` property in object `a`. *But you don't really have two distinct objects*: `b` is simply assigned a reference to `a`, and because you changed `a.foo`, you will see the change reflected in both `a` and `b`.

We have never talked about this issue before because when you work with primitive types, things seem intuitively logical, as seen in the code example above. But with objects, we have to be more careful when we assign objects to variables, since we are really only assigning the reference to the variable. If that seems a bit confusing a first, don't worry - you're in good company. We'll revisit this issue multiple times until it sinks in.

After this short detour about objects, let's bring our attention back to the cooperation model code. We were just talking about this function:

```
function get_other_individual(individual) {
    do {
        var random_index = get_random_int(0,population_size-1);
        var other_individual = population[random_index];
    }
    while (other_individual == individual);
    return other_individual;
}
```

which returns an `other_individual` that is different from the `individual`. And we now understand that it is different in the sense that it is indeed a different object, even though it might have the exact same properties.

Now that we have a second, distinct individual, we pass both individuals as arguments to the `games()` function, where they play a game together. Keep in mind that a game is defined by the payoff matrix that we developed above, which looks as follows:

	Individual 2 Defecting	**Individual 2 Cooperating**
Individual 1 Defecting	Payoff Individual 1: 0 Payoff Individual 2: 0	Payoff Individual 1: b Payoff Individual 2: -c
Individual 1 Cooperating	Payoff Individual 1: -c Payoff Individual 2: b	Payoff Individual 1: b-c Payoff Individual 2: b-c

How can we incorporate this in code? Here's how:

```
function play_game(individual1, individual2) {
    var move_individual1 = individual1.compute_move();
    var move_individual2 = individual2.compute_move();
    if (move_individual1 == "C") {
        if (move_individual2 == "C") {
            individual1.add_to_payoff(b - c);
            individual2.add_to_payoff(b - c);
        }
        else {
            individual1.add_to_payoff(-c);
            individual2.add_to_payoff(b);
        }
    }
    else {
        if (move_individual2 == "C") {
            individual1.add_to_payoff(b);
            individual2.add_to_payoff(-c);
```

```
        }
    }
}
```

This code is in principle quite easy to understand. You first compute which of the two possible moves - cooperate (`"C"`) or defect (`"D"`) each individual is going to make. Then, depending on the combination of the two moves, you assign payoff values as outlined in the payoff matrix above.

What is perhaps a little perplexing is that the functions that we are calling to do these things - `compute_move()` for computing an individual's move, `add_to_payoff()` for adding a value to the payoff of an individual - are not just stand-alone functions, as we have used them throughout the book. Instead, they are functions called directly on the objects, using the dot notation like so:

```
individual1.compute_move();
```

When a function is used on an object, like we are doing here, it is usually called a *method*. While we have not defined our own methods until now, we have used methods before - remember our good old friend `Math.random()`? Yep, `random()` is a method, defined in the `Math` object, and called using the dot notation, exactly as we do in `individual1.compute_move()`!

Thus, all that is left to do is to define these two methods. Since they are part of the `Individual` object, they will need to be defined in the corresponding constructor function. After adding the two functions, our `Individual` constructor function will read like this:

```
function Individual(strategy, payoff) {
    this.strategy = strategy;
    this.payoff = payoff;
    this.compute_move = function() {
        return this.strategy == "ALL_C" ? "C" : "D";
    };
    this.add_to_payoff = function(game_payoff) {
        this.payoff += game_payoff;
    };
}
```

Let's take a detailed look at this, because this looks nothing like anything we have seen before. Let's first take a look at the following three lines:

```
this.compute_move = function() {
    return this.strategy == "ALL_C" ? "C" : "D";
};
```

This defines the `compute_move()` method. It returns `"C"` if the individual's strategy is `"ALL_C"`, and `"D"` otherwise, using the ternary operator. Defining a method in an object has the following notation:

```
this.method_name = function() {};
```

where you are then free to make the method do whatever it needs to do inside the curly brackets `{}`.

This looks a little strange at first, because we have so far defined regular functions like so:

```
function function_name() {
}
```

Indeed, we could always have written this instead as

```
var function_name = function() {};
```

which would have had the same effect. In JavaScript, functions are in fact special types of objects, and just like you can assign a regular object to a variable, you can also assign a function to a variable. However, for stand-alone functions in my code, I generally prefer the slightly shorter notation

```
function function_name() {
}
```

You can in principle use this latter version even within an object constructor function. However, you would then have to assign the function to a local variable in a separate step. I find it much more convenient to combine all this steps by using the former version, and simply write:

```
this.compute_move = function() {
    return this.strategy == "ALL_C" ? "C" : "D";
};
```

This defines the function AND assigns it to `this.compute_move` at the same time, for latter usage in our code.

After defining a method, you can then use it on any object as part of which it was defined - in our case, we can use it on any `Individual` object, as we do in the `games()` function:

```
individual1.compute_move();
```

The second method we have defined is the following:

```
this.add_to_payoff = function(game_payoff) {
    this.payoff += game_payoff;
};
```

This should now make more sense. The method `add_to_payoff()` has one parameter, `game_payoff` (which is the payoff from a single game), which it adds to the `payoff` property of the object like so:

```
this.payoff += game_payoff;
```

As this line of code demonstrates, you still have access to the object in the method body, through the variable `this`. Because of that, methods allow you to read and write the values of all the properties of the object, which is to a large extent the purpose of object methods.

Now that we are finished with the `games()` function, let's move on and define the `selection()` function.

As we mentioned at the beginning of the chapter, in evolutionary game theory, the payoff is not monetary, but rather evolutionary fitness: the higher your payoff, the higher your fitness. In chapter 6, where we discussed natural selection, we implemented selection by increasing the frequency of a genotype in a population in proportion to its fitness. In the model here, however, we are using a slightly modified version that is better suited for individual-based models. On average, it will roughly produce the same outcome: individuals with higher fitness will increase in frequency, and individuals with lower fitness will decrease in frequency.

Concretely, we are going to implement a very simple idea: for each individual, we'll pick another individual randomly, and then compare the fitnesses (i.e. payoffs) of the two individuals. If the randomly picked individual has a higher fitness than the focal individual, the focal individual will copy the random individual's strategy, otherwise it will keep its current strategy.

Here is how we will implement this idea:

```
function selection() {
    var temp_population = [];
    for (var i = 0; i < population_size; i++) {
        var current_individual = population[i];
        var other_individual = get_other_individual(current_individual);
        if (other_individual.payoff > current_individual.payoff) {
            temp_population[i] = other_individual;
        }
        else {
            temp_population[i] = current_individual;
        }
    }
    for (i = 0; i < population_size; i++) {
```

```
        current_individual = population[i];
        current_individual.strategy = temp_population[i].strategy;
        current_individual.payoff = 0;
    }
}
```

As we have done a few times before, we'll make use of a temporary data structure - an array called `temp_population` in this case - to store the new values in order to ensure that all individuals get updated simultaneously (if you don't know why we're doing this, be sure to re-read chapter 5).

The function is divided into two `for` loops. The first `for` loop iterates over all individuals in the population, and for each individual, we pick a another, random individual, and compare the two fitness values. Then, depending on the values, we copy the properties of one of the two individuals into the temporary array.

Let's take a look at this line:

```
temp_population[i] = other_individual;
```

As we have discussed just a few pages earlier, because we're dealing with objects, `temp_population[i]` will now simply contain a reference to the `other_individual` object, rather than a copy of the object. This is important - let's now look at the second `for` loop and see why. It reads:

```
for (i = 0; i < population_size; i++) {
    current_individual = population[i];
    current_individual.strategy = temp_population[i].strategy;
    current_individual.payoff = 0;
}
```

As you can see, we are copying the `strategy` property from the individual that's referenced by `temp_population[i]`, and we are resetting the `payoff` property of the individual to `0` (because at each new time step, individuals should start with a payoff of `0` before they participate in a new round of games).

Let's take a look at what would be the *wrong* way to do this. The wrong way would be to implement this loop as follows:

```
// the WRONG way!!
for (i = 0; i < population_size; i++) {
    current_individual = temp_population[i];
    current_individual.payoff = 0;
}
```

This is very tempting: simply copy over the individual from the `temp_population` array, and then set its payoff back to `0` - simple, right?

The problem with this is that the line

```
current_individual = temp_population[i];
```

doesn't copy an object - it simply assigns a reference. Thus, `current_individual`, rather than being the same object as before (but simply with new property values) would now actually reference another object somewhere in memory, which will lead to very confusing bugs and completely wrong dynamics.

Because this concept can be a little trick to wrap your head around at first, I'm going to divert from our cooperation code, and will briefly introduce a completely independent code example that explains the concept in a simpler context. We'll get back to our cooperation code right after this.

Revisiting Object Assignments

Take a look at the following code:

```
function Individual(fitness, trait) {
    this.fitness = fitness;
    this.trait = trait;
}

var individual1 = new Individual(1, "A");
var individual2 = new Individual(1.1, "B");
var individual3 = new Individual(1.2, "C");

console.log("initial population:")
console.log(individual1); // shows trait "A"
console.log(individual2); // shows trait "B"
console.log(individual3); // shows trait "C"

// selection:
individual1 = individual3;
individual2 = individual3;
// individuals 2 and 3 should now have trait "C"

console.log("population after selection:")
console.log(individual1); // shows trait "C"
console.log(individual2); // shows trait "C"
console.log(individual3); // shows trait "C"

// mutate only individual3
individual3.trait = "D";
```

```
console.log("population after mutation:")
console.log(individual1); // oops - shows trait "D"!!
console.log(individual2); // oops - shows trait "D"!!
console.log(individual3); // shows trait "D"
```

Let's walk through this code slowly. This is a very simplified version of a selection-mutation process, with a population of three individuals.

On the first few lines, we defined the object `Individual`, which has two properties: a `fitness` property, and a `trait` property. We then create three individuals with different values. The object `individual1` has a `fitness` value of `1`, and `trait` value of `"A"`. The object `individual2` has a `fitness` value of `1.1`, and `trait` value of `"B"`. The object `individual3` has a `fitness` value of `1.2`, and `trait` value of `"C"`. We then print those objects to the console so that we can check that everything is working fine.

Then, selection takes place, and we want both `individual1` and `individual2` to copy over the values of the fittest individual in the population, `individual3`:

```
// selection:
individual1 = individual3;
individual2 = individual3;
```

Spoiler alert: this is where things go wrong- but more about that in a second. In any case, if we now print the three `Individual` objects into the console, we see that they all have trait `"C"`, which is what we wanted.

In the final step of this example, we assume that `individual3` mutates, and changes its trait to `"D"`. Note that we only change `individual3` - we touch neither `individual1` nor `individual2`. However, if we now print the objects one more time into the console, we see that they all have trait `"D"`! How did that happen? We did not touch the traits of `individual1` or `individual2` anywhere in the code!

As I mentioned above, the problem is in these lines:

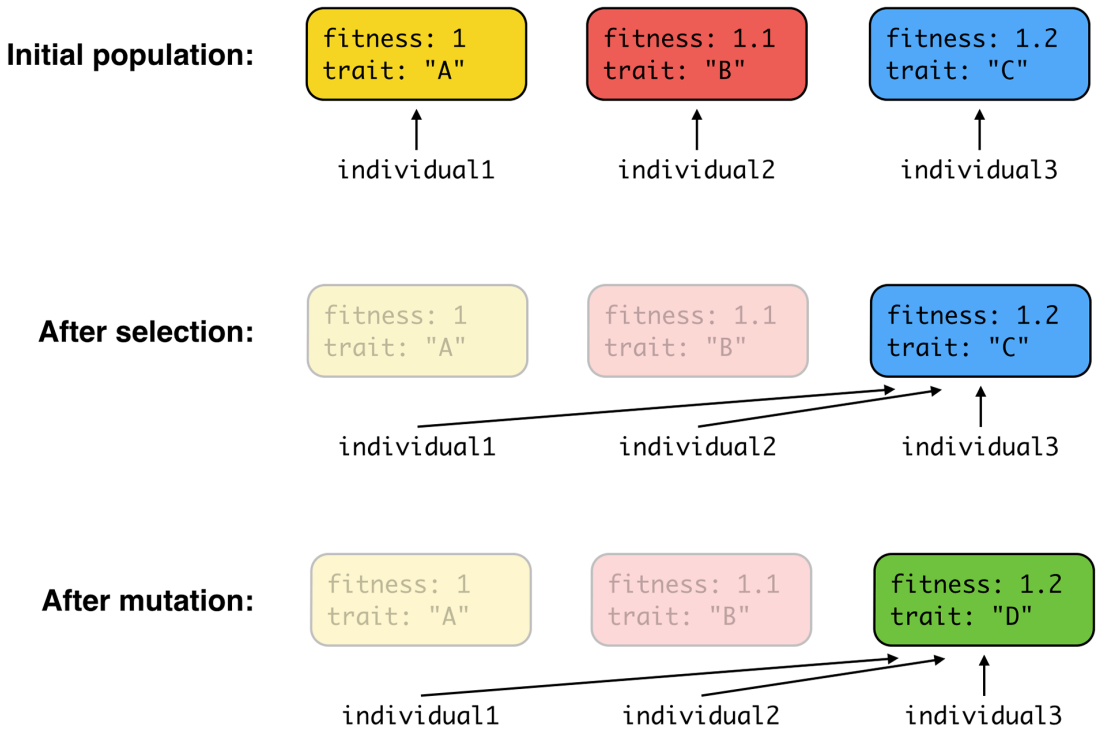
```
// selection:
individual1 = individual3;
individual2 = individual3;
```

We might think we're copying the object `individual3`, with all its properties, into `individual1` and `individual2` - *but we're not.* Instead, we are copying the *reference* to `individual3`. After these two lines, both `individual1` and `individual2` are simply references to `individual3`. In a way, with these two lines, we have "lost" two objects - all we have left is object `individual3`, and two other variables (`individual1` and `individual2`) referencing `individual3`.

When we print the three objects into the console after the selection step, it's very easy to think everything went fine, because all three objects now have trait `"C"` - but this is an illusion! It's simply

that they are all referencing the same object (`individual3`), which happens to have trait `"C"`. And this illusion is exposed when we assign a new value to the trait of `individual3`: suddenly all individuals have trait `"D"`, because they are all pointing to `individual3`.

Here is a visual representation of what's going on (I've color-coded the objects to reflect distinct trait values):



I hope this clarifies the issue, and ensures that you'll be on the lookout whenever you assign objects to variables. But having identified the problem, what is the correct way of doing it? In this particular example, the correct way would be to do the following - rather than copying the objects (or the references, as you now know), copy the properties instead. In other words, instead of these two lines:

```
individual1 = individual3;
individual2 = individual3;
```

you would have these four lines (assuming you want to copy both properties):

```
individual1.trait = individual3.trait;
individual1.fitness = individual3.fitness;
individual2.trait = individual3.trait;
individual2.fitness = individual3.fitness;
```

If you replace the two wrong lines with three correct lines and run this code again in your browser, you will see that after the mutation step, only `individual3` will have the trait value `"D"` - both `individual1` and `individual2` will still have trait value `"C"`, as they should have.

If you have very complex objects with many properties, it might be worthwhile writing a custom method for your object that copies the object's values. In our example, we could have decided to do this for our `Individual` object, which would look like this:

```
function Individual(fitness, trait) {
    this.fitness = fitness;
    this.trait = trait;
    this.copy = function() {
        return new Individual(this.fitness, this.trait);
    }
}
```

With that in place, we can replace the lines

```
individual1.trait = individual3.trait;
individual1.fitness = individual3.fitness;
individual2.trait = individual3.trait;
individual2.fitness = individual3.fitness;
```

with

```
individual1 = individual3.copy();
individual2 = individual3.copy();
```

The benefit of this method is that you can adapt it easily when your object definition changes.

One final caveat before we return to the main thread of cooperation: if your objects get very complex, some of the object's properties may themselves be objects! Indeed, their properties could also be objects, etc. - as many hierarchical levels as you want. If that's the case, be extra careful about what we just talked about; all objects behave the same, whether they are standalone, or deeply nested within other objects.

When Defection Outcompetes Cooperation

Let's now return to our cooperation code. We have just finished implementing our `selection()` method. The final major method we need to implement is `mutation()`, which we will do as follows:

```
function mutation() {
    for (var i = 0; i < population_size; i++) {
        if (Math.random() < mutation_rate) {
            var current_individual = population[i];
            current_individual.mutate();
        }
    }
}
```

This function simply goes through all individuals in the population, and calls the `mutate()` method - which we'll need to define - on some of the individuals.

The `mutate()` method can be implemented as follows in the `Individual` constructor function:

```
this.mutate = function() {
    if (this.strategy == "ALL_C") {
        this.strategy = "ALL_D";
    }
    else {
        this.strategy = "ALL_C";
    }
};
```

Thus, your complete `Individual` constructor function will look like this:

```
function Individual(strategy, payoff) {
    this.strategy = strategy;
    this.payoff = payoff;
    this.compute_move = function() {
        return this.strategy == "ALL_C" ? "C" : "D";
    };
    this.add_to_payoff = function(game_payoff) {
        this.payoff += game_payoff;
    };
    this.mutate = function() {
        if (this.strategy == "ALL_C") {
            this.strategy = "ALL_D";
        }
        else {
            this.strategy = "ALL_C";
        }
    };
}
```

And that's it - almost! The simulation would be working right now, but there is no way for us to see what's going on. First, let's go ahead and add some code so that we can print the results of the simulation into the console.

Of course, we'd like to see the frequencies of the strategies at each time step. So let's go ahead and modify our `run_time_step()` function which currently reads:

```
function run_time_step() {
    games();
    selection();
    mutation();
}
```

to

```
function run_time_step() {
    games();
    selection();
    mutation();
    for (var i = 0; i < strategies.length; i++) {
        data[i].push(get_number_with_strategy(strategies[i]) / population_size);
    }
}
```

For each strategy, we need to know how many of the individuals have that specific strategy. We'll then divide that number by the population size, which gives us the frequency of the strategy in the population. The number will then be pushed in the corresponding array within `data`, for later usage. In order to get the number of individuals in the population with a given strategy, we'll define a helper function `get_number_with_strategy()`:

```
function get_number_with_strategy(strategy) {
    var count = 0;
    for (var i = 0; i < population_size; i++) {
        if (population[i].strategy == strategy) {
            count++;
        }
    }
    return count;
}
```

This function simply iterates over all individuals and increases a counter variable (called `count`) whenever the strategy of an individual is equal to a given strategy that was passed to the function as an argument.

Excellent! Now all that is left to do is print the results from the `data` array into the console. To do that, add the following code after your call to `run_simulation()`:

```
for (var i = 0; i < number_of_time_steps; i++) {
    console.log(i,"ALL_C",data[0][i],"ALL_D",data[1][i]);
}
```

Let's go ahead and run the entire code now! If you run it by loading the document in the browser, and look at the JavaScript console, you'll see something like this:

```
0 "ALL_C" 1 "ALL_D" 0
1 "ALL_C" 1 "ALL_D" 0
2 "ALL_C" 1 "ALL_D" 0
3 "ALL_C" 1 "ALL_D" 0
4 "ALL_C" 1 "ALL_D" 0
5 "ALL_C" 1 "ALL_D" 0
6 "ALL_C" 0.99 "ALL_D" 0.01
7 "ALL_C" 0.98 "ALL_D" 0.02
8 "ALL_C" 0.94 "ALL_D" 0.06
9 "ALL_C" 0.92 "ALL_D" 0.08
10 "ALL_C" 0.92 "ALL_D" 0.08
11 "ALL_C" 0.95 "ALL_D" 0.05
12 "ALL_C" 0.9 "ALL_D" 0.1
13 "ALL_C" 0.88 "ALL_D" 0.12
14 "ALL_C" 0.93 "ALL_D" 0.07
15 "ALL_C" 0.94 "ALL_D" 0.06
16 "ALL_C" 0.95 "ALL_D" 0.05
17 "ALL_C" 0.94 "ALL_D" 0.06
18 "ALL_C" 0.95 "ALL_D" 0.05
19 "ALL_C" 0.96 "ALL_D" 0.04
20 "ALL_C" 0.97 "ALL_D" 0.03
21 "ALL_C" 0.96 "ALL_D" 0.04
22 "ALL_C" 0.98 "ALL_D" 0.02
23 "ALL_C" 0.96 "ALL_D" 0.04
24 "ALL_C" 0.95 "ALL_D" 0.05
25 "ALL_C" 0.92 "ALL_D" 0.08
```

What this shows is that the population starts with 100% `ALL_C` individuals, i.e. the population is composed entirely of cooperators; then, after a few generation, the first `ALL_D` (defector) mutants show up. From that point on, you can follow the fate of the two strategies by looking at the numbers.

If everything went ok with your code, you shouldn't have to scroll too far down in your JavaScript console to see something like this:

```
43 "ALL_C" 0.99 "ALL_D" 0.01
44 "ALL_C" 0.98 "ALL_D" 0.02
45 "ALL_C" 0.95 "ALL_D" 0.05
46 "ALL_C" 0.95 "ALL_D" 0.05
47 "ALL_C" 0.88 "ALL_D" 0.12
48 "ALL_C" 0.89 "ALL_D" 0.11
49 "ALL_C" 0.9 "ALL_D" 0.1
50 "ALL_C" 0.85 "ALL_D" 0.15
51 "ALL_C" 0.82 "ALL_D" 0.18
52 "ALL_C" 0.76 "ALL_D" 0.24
53 "ALL_C" 0.74 "ALL_D" 0.26
54 "ALL_C" 0.63 "ALL_D" 0.37
55 "ALL_C" 0.58 "ALL_D" 0.42
56 "ALL_C" 0.62 "ALL_D" 0.38
57 "ALL_C" 0.49 "ALL_D" 0.51
58 "ALL_C" 0.45 "ALL_D" 0.55
59 "ALL_C" 0.35 "ALL_D" 0.65
60 "ALL_C" 0.19 "ALL_D" 0.81
61 "ALL_C" 0.15 "ALL_D" 0.85
62 "ALL_C" 0.05 "ALL_D" 0.95
63 "ALL_C" 0 "ALL_D" 1
64 "ALL_C" 0 "ALL_D" 1
65 "ALL_C" 0 "ALL_D" 1
66 "ALL_C" 0 "ALL_D" 1
```

(your specific numbers will vary of course - remember that this is a stochastic simulation). What you can see here is that the cooperators are driven to extinction by the defectors in just a few generations. Go ahead and reload the page - this happens every single simulation run, without an exception.

And of course, we shouldn't be surprised - indeed, we have just implemented a stochastic simulation of the prisoner's dilemma, and as we've seen in the beginning of the chapter, the outcome of the prisoner's dilemma is that defectors will always win, because it is always better for an individual (in terms of its payoff) to defect, rather than to cooperate. That is, after all, why it is called a dilemma: even though the total population payoff would be much higher if everyone cooperated, individuals will be better of if they defect. Thus, if everyone acts in their own self-interest, cooperation will collapse and be outcompeted by defection.

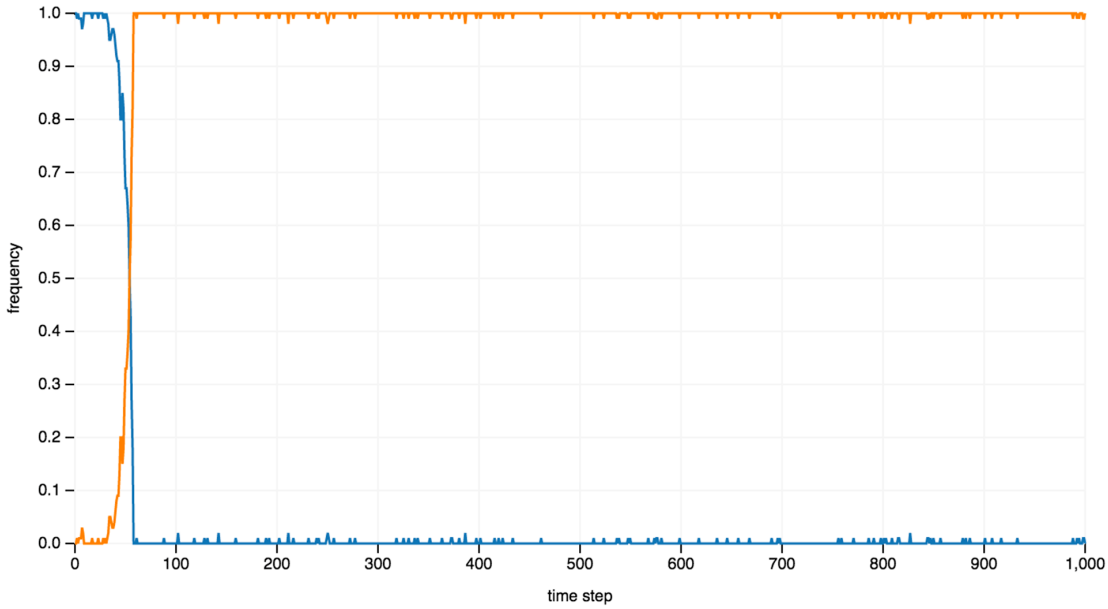
But we've shown repeatedly throughout this book that we can do better than print numbers into the JavaScript console! In fact, the `draw_line_chart()` function that we've used so many times comes in quite handy here. Go ahead and replace this code:

```
for (var i = 0; i < number_of_time_steps; i++) {
    console.log(i,"ALL_C",data[0][i],"ALL_D",data[1][i]);
}
```

by this line

```
draw_line_chart(data,"time step", "frequency",[]);
```

and you will see the following dynamic in your browser upon reload:



The blue line is the frequency of cooperators (recall that we start each simulation with all cooperators), and the orange line is the frequency of defectors. Go ahead and reload the page - the dynamics, with slight stochastic variations, will always be identical: the cooperators are outcompeted by the defectors, generally within the first 100 time steps. The small bumps you can see in both lines are mutants showing up in the population, but none of them seem to be able to revert the dynamics. Again, this is as we expect it to be: in the regular prisoner's dilemma, it is always better for individuals to defect.

But hang on a second, I hear you say - didn't we start this chapter with the idea the cooperation would readily evolve? Glad you asked! Indeed, that's the promise with which I started the chapter. What we are going to do next is to introduce a small but realistic modification that will change the dynamics quite strongly. But first, I needed to convince you that pure cooperation, all by itself, does not stand a chance against pure defectors. And along the way, we developed an individual-based JavaScript simulation of a game-theoretical model (which is quite cool if you ask me - but of course I'm biased). This simulation will now form the basis of our exploration about the factors than can prevent the defectors from always winning.

Repeated games

So far, we have assumed that all interactions are a one-off. Once you interact with an individual, you may never interact with that individual again (and if you do, there will be no memory of your

past interaction). In reality, however, when we interact with others, we often do so repeatedly with the same people, fully aware that they will remember how we interacted with them before.

It turns out that repeated interactions change the dynamics of the prisoner's dilemma quite a bit. In particular, when you play games repeatedly with the same players, you can come up with a rich set of possible strategies, such as having your next move depend on the previous move. The literature in this field was kicked off by a paper in 1981 by Robert Axelrod and Bill D. Hamilton, who used computer simulations to run a "tournament". They asked their colleagues to submit strategies - in code - and then they let those strategies compete against each other in a simulation of an iterated prisoner's dilemma. The winning strategy was submitted by Anatol Rapoport. It was a strategy he called "Tit-for-Tat".

"Tit-for-Tat", when played against the same player repeatedly, is very simple. It starts by cooperating in the first round, and then adopts whichever strategy the other player played in the *previous* round. Let's go ahead and implement this strategy in our code.

The first thing to do is to expand our list of strategies by `"TFT"`:

```
var strategies = ["ALL_C","ALL_D","TFT"];
```

Next, since we're implementing repeated interactions, we need to define how many repeated interactions there should be per round. We can define this in a new global variable called `number_of_game_repeats`:

```
var number_of_game_repeats = 100;
```

Now let's go ahead and make sure that our code actually performs multiple repeated interactions among players. Currently, the game is played in the function `play_game()` which looks as follows:

```
function play_game(individual1, individual2) {
    var move_individual1 = individual1.compute_move();
    var move_individual2 = individual2.compute_move();
    if (move_individual1 == "C") {
        if (move_individual2 == "C") {
            individual1.add_to_payoff(b - c);
            individual2.add_to_payoff(b - c);
        }
        else {
            individual1.add_to_payoff(-c);
            individual2.add_to_payoff(b);
        }
    }
    else {
```

```
        if (move_individual2 == "C") {
            individual1.add_to_payoff(b);
            individual2.add_to_payoff(-c);
        }
    }
}
```

Since this code performs exactly one interaction, we can simply wrap it with a `for` loop that iterates this code as many times as necessary:

```
function play_game(individual1, individual2) {
    for (var i = 0; i < number_of_game_repeats; i++) {
        // previous code here
    }
}
```

There is something else we should take care of in this function. If an individual has the strategy `TFT`, then the individual's moves will depend on the *previous* moves of the opponent. Thus, we need to constantly keep track of the previous move of the opponent, so that our `compute_move()` method can tell us which move to perform during a given interaction. In order to keep track of the previous move of the opponent, we'll need to add a new property to our `Individual` object:

```
this.last_move_by_opponent = "";
```

We're going to initialize this property with an empty string, because initially, an individual won't have played any games, and thus there is no last move by an opponent.

Then, in the `play_game()` function, we can update this property once we know what the opponent will play:

```
individual1.last_move_by_opponent = move_individual2;
individual2.last_move_by_opponent = move_individual1;
```

The full function `play_games()` is now:

```
function play_game(individual1, individual2) {
    for (var i = 0; i < number_of_game_repeats; i++) {
        var move_individual1 = individual1.compute_move();
        var move_individual2 = individual2.compute_move();
        individual1.last_move_by_opponent = move_individual2;
        individual2.last_move_by_opponent = move_individual1;
        if (move_individual1 == "C") {
            if (move_individual2 == "C") {
                individual1.add_to_payoff(b - c);
                individual2.add_to_payoff(b - c);
            }
            else {
                individual1.add_to_payoff(-c);
                individual2.add_to_payoff(b);
            }
        }
        else {
            if (move_individual2 == "C") {
                individual1.add_to_payoff(b);
                individual2.add_to_payoff(-c);
            }
        }
    }
}
```

Since we now have three possible strategies, we also need to update the `compute_move()` method in the `Individual` object. At the moment, it reads:

```
this.compute_move = function() {
    return this.strategy == "ALL_C" ? "C" : "D";
};
```

With the new `TFT` strategy, we will have to modify this method to the following:

```
this.compute_move = function() {
    if (this.strategy == "ALL_D") {
        return "D";
    }
    else if (this.strategy == "ALL_C") {
        return "C";
    }
    else if (this.strategy == "TFT") {
        if (this.last_move_by_opponent == "") {
            return "C";
        }
        else {
            return this.last_move_by_opponent;
        }
    }
};
```

The first couple of lines in the method are equivalent to what we had before - return `"C"` if the strategy is `ALL_C`, and `"D"` if the strategy is `ALL_D`. In addition, we now take care of the case where the strategy is `TFT`, in which case we return `"C"` if these two players haven't interacted previously (i.e. if we're in the first round), and the last move of the opponent otherwise.

Now, we need to remind ourselves that `TFT` should cooperate at *every* first interaction with each player, not just at the first interaction of the entire simulation. Thus, we need to reset the `last_move_by_opponent` property to the empty string after all interactions with the same individual. We'll do this in the `selection()` function (right after resetting another property, `payoff`):

```
current_individual.last_move_by_opponent = "";
```

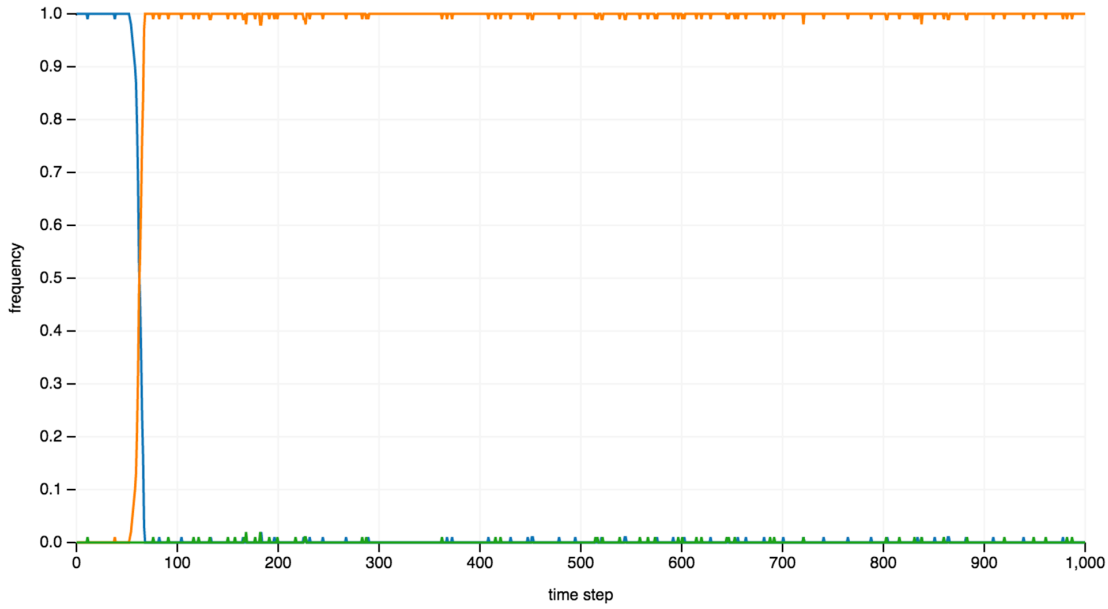
And finally, we will need to adapt our `mutate()` method in the `Individual` object. At the moment, mutations will only generate `ALL_D` or `ALL_C` strategies. We'll now change this method into a more generic version:

```
this.mutate = function() {
    do {
        var new_s = strategies[get_random_int(0, strategies.length - 1)];
    } while (new_s == this.strategy);
    this.strategy = new_s;
};
```

Your complete `Individual` object now reads as follows:

```
function Individual(strategy, payoff) {
    this.strategy = strategy;
    this.payoff = payoff;
    this.last_move_by_opponent = "";
    this.compute_move = function() {
        if (this.strategy == "ALL_D") {
            return "D";
        }
        else if (this.strategy == "ALL_C") {
            return "C";
        }
        else if (this.strategy == "TFT") {
            if (this.last_move_by_opponent == "") {
                return "C";
            }
            else {
                return this.last_move_by_opponent;
            }
        }
    };
    this.add_to_payoff = function(game_payoff) {
        this.payoff += game_payoff;
    };
    this.mutate = function() {
        do {
            var new_s = strategies[get_random_int(0, strategies.length - 1)];
        } while (new_s == this.strategy);
        this.strategy = new_s;
    };
}
```

And that's it! Save your code, and reload the document in the browser. You'll most likely see something like this:



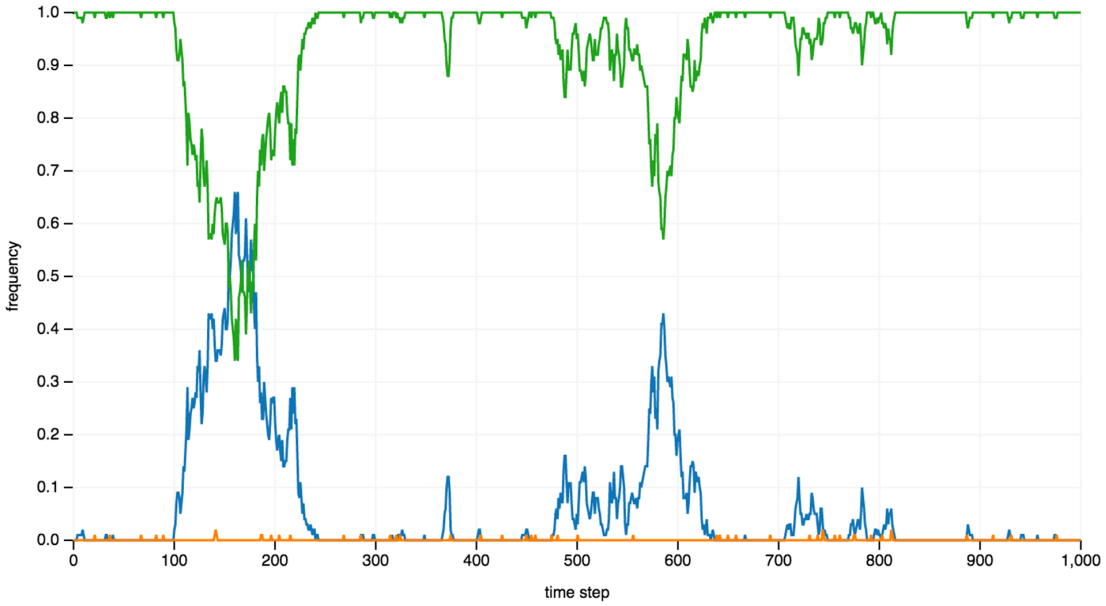
The new green line represents the frequency of the `TFT` strategy - but what's that? It doesn't seem to stand a chance against defectors (in orange) either! That's true, but remember that we're still initiating the population with `ALL_C`s (the blue line), which will very quickly be invaded by `ALL_D`, as we've shown before. Let's go ahead and initialize the population with all `TFT`s instead. In your code, find the function `init_simulation()` and change this line:

```
population.push(new Individual("ALL_C",0));
```

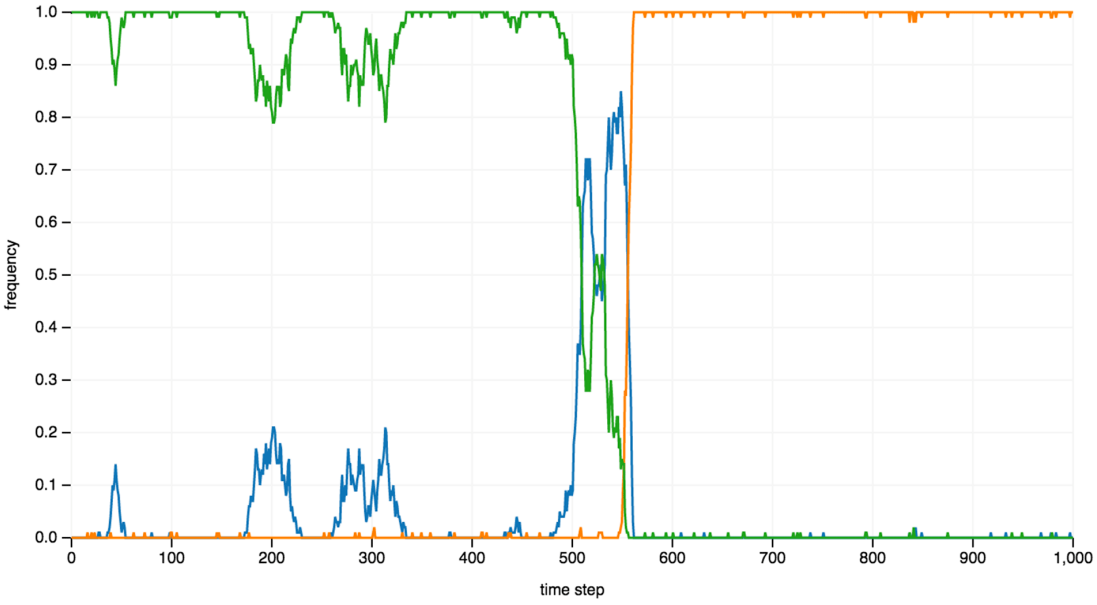
to

```
population.push(new Individual("TFT",0));
```

Save and reload. You will now see one of two typical outcomes, either



or



In the first picture, `TFT` remains the dominant strategy throughout almost the entire simulation. There are moments when `ALL_C` becomes more frequent, and occasionally even the most frequent strategy. The reason is that when `TFT` and `ALL_C` play together, they are performing the exact same moves - they always cooperate. Thus, `TFT` and `ALL_C` are *selectively neutral* with respect to each other.

The second graph is more interesting. Here, `ALL_C` starts to invade `TFT` by simply drifting to

dominance. However, ALL_C, as we've seen before, will almost immediately be invaded by ALL_D! And once ALL_D is the dominant strategy, it can't be invaded by either ALL_C or TFT (recall that TFT always starts out by cooperating, and so will always have a lower payoff when playing against ALL_D).

And this is ultimately the big weakness of TFT - it can't prevent the naive ALL_C from drifting to dominance, which will then be immediately exploited by ALL_D. In fact, all simulations will eventually have the same fate - sometimes this happens within 1,000 time steps, and sometimes in happens much later.

Is this the end of the story? Does this mean that cooperation is only ever a transitory state of affairs? Not so fast! There is another strategy that is able to prevent the invasion of both ALL_D *and* the naive ALL_C. This is important, because as we have just seen, an invasion by ALL_C, even though seemingly a positive development, immediately makes the population vulnerable to invasion by ALL_D.

Win-stay, lose-shift

Win-stay, lose-shift (WSLS) is based on a simple idea: if a player does well, it will keep playing the previous move; it doesn't, it will switch. By playing this strategy, a player simply monitors its own payoff, given its current move.

Let's remind ourselves briefly of the payoff matrix of the prisoner's dilemma:

	Individual 2 Defecting	**Individual 2 Cooperating**
Individual 1 Defecting	Payoff Individual 1: 0 Payoff Individual 2: 0	Payoff Individual 1: b Payoff Individual 2: -c
Individual 1 Cooperating	Payoff Individual 1: -c Payoff Individual 2: b	Payoff Individual 1: b-c Payoff Individual 2: b-c

Let's take the perspective of Individual 1 in this table. If the individual is defecting, it "wins" whenever the other individual is cooperating (payoff b), and it "loses" whenever the other individual is defecting (payoff 0). If the individual is cooperating, it "wins" whenever the other individual is cooperating (payoff b-c), and it "loses" whenever the other individual is defecting (payoff -c). *Note that winning and losing here are simply assessed by comparing the payoff earned to the one that would have been earned if the opponent had made another move.*

If you re-read the preceding paragraph, you'll note that the strategy can be simplified to the following simple rule: If my opponent cooperated in the previous interaction, I will continue doing what I was doing, otherwise I'll switch. In some sense, this is quite a strange strategy - the player is evaluating what would have happened if the *other* player had played differently, but then goes on to change *its own* strategy! For example, if a WSLS player defected against another defector, it will in the next round switch to cooperating, which is not how most of us would react.

Let's go ahead an implement this strategy in our code and see how well it does.

First, let's add the strategy to our `strategies` array:

```
var strategies = ["ALL_C","ALL_D","TFT","WSLS"];
```

We've just learned that the `WSLS` strategy depends on the previous move of the opponent - a player will switch if the opponent cooperated in the previous round. Ever since our implementation of `TFT`, we're keeping track of the opponent's previous move (via the property `last_move_by_opponent`) - what we're not currently keeping track of, however, is the previous move of the player itself (as opposed to that of the opponent). Let's fix that.

First, add the property `last_move` to the `Individual` object:

```
this.last_move = "";
```

As with `last_move_by_opponent`, we initialize this property with an empty string.

Then, in the function `play_game()`, let's update the last move for both players and add the following two lines (add the lines right after or before the lines that keep track of the `last_move_by_opponent`):

```
individual1.last_move = move_individual1;
individual2.last_move = move_individual2;
```

The final modification with respect to this property is to make sure that we reset it to the empty string in the `selection()` function (again right after or before resetting `last_move_by_opponent`):

```
current_individual.last_move = "";
```

Great - now let's go ahead and implement the actual strategy. For this, we need to update the `compute_move()` method of the `Individual` object, by adding this bit at the end:

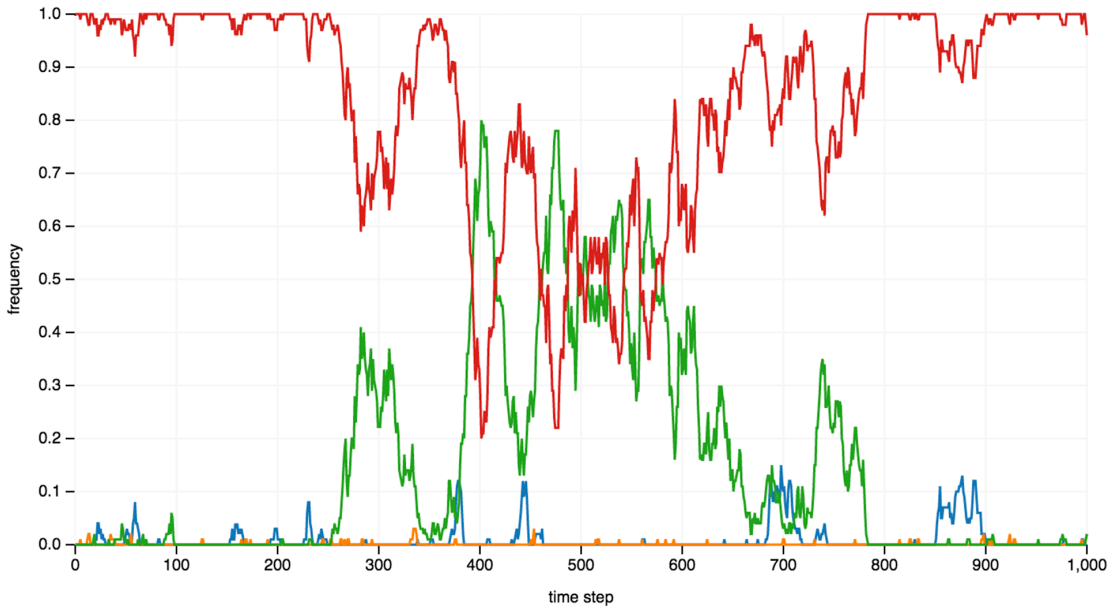
```
else if (this.strategy == "WSLS") {
    if (this.last_move == "") {
        return "C";
    }
    else {
        if (this.last_move_by_opponent == "D") {
            return (this.last_move == "C" ? "D" : "C");
        }
        else {
            return this.last_move;
        }
    }
}
```

WSLS, just like TFT, starts out being nice - by cooperating in the first round. Then, as outlined above, it will compute the next move based on whether the opponent defected or cooperated in the previous round. If the opponent defected, the player will switch (from cooperate to defect or vice versa). Otherwise, it will continue to play the way it played in the previous round.

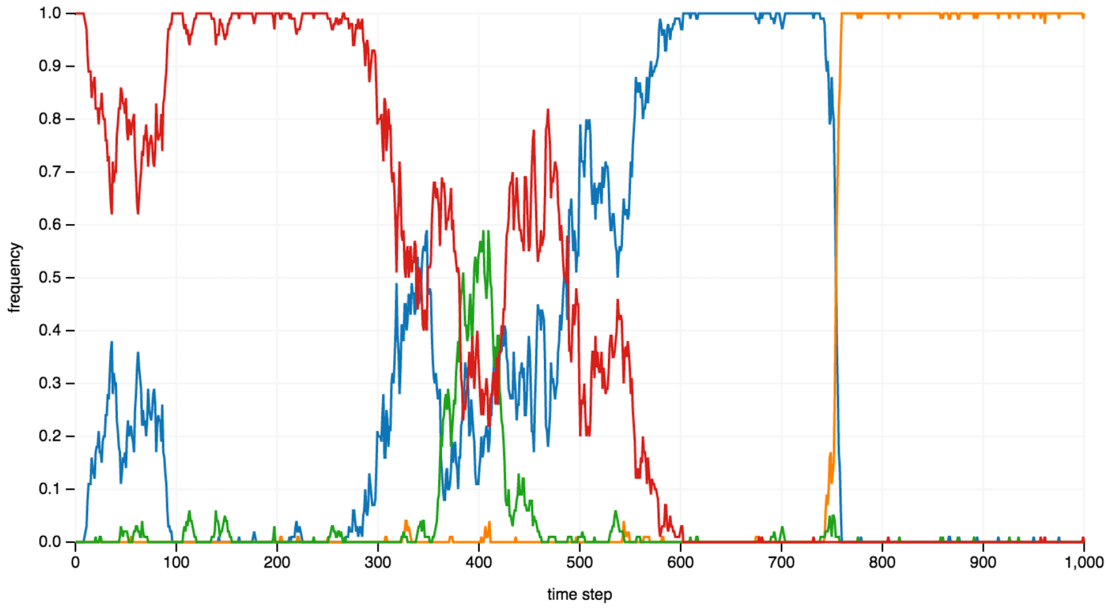
Now, as before, let's give the newcomer a fighting chance by initiating the entire population with this new strategy. Modify the function `init_simulation()` to say:

```
population.push(new Individual("WSLS",0));
```

Then, save the document and reload the browser. The WSLS strategy will appear as the forth line, in red:



Looking good! But it seems as if WSWL and TFT are selectively neutral. And that wouldn't bode well for cooperation, since we know that TFT and ALL_C are also selectively neutral against each other, which would mean that WSWL would also be selectively neutral against ALL_C. And indeed, a common outcome you will find when you reload is this:



Oh no! That was exactly what we wanted to avoid with `WSLS`! What is going on?

Well, as you've seen above when we implemented the `WSLS` code, a `WSLS` individual will change its move only when the opponent previously defected. In other words, in a happy world where `TFT` and `ALL_C` cooperate, `WSLS` will cooperate as well, and the three strategies are effectively identical with respect to their behavior. As a consequence, `ALL_C` will be able to drift to dominance, which is never a good thing, as `ALL_D` won't wait for long to seize that opportunity and overtake the population.

So, what's so special about `WSLS` then?

`WSLS` starts to shine when we introduce a small but realistic modification to our model. We have so far assumed that in all repeated interactions, the strategies stick to their program without making a single mistake, ever. But of course, mistakes will sometimes occur even with the best of intentions. That is, a player who would normally cooperate will occasionally defect, and a player who would normally defect will occasionally cooperate, even if just by accident.

Taking this issue into consideration is important. Consider what would happen if two `TFT` players played together. Normally, the sequence of moves would look like this:

TFT: C C C C C C C C C C C C C C C C C ...
TFT: C C C C C C C C C C C C C C C C C ...

However, imagine that one of the players makes a mistake and accidentally defects (the mistake is marked in bold):

TFT: C C C C C C **D** C D C D C D C D C D C ...
TFT: C C C C C C C D C D C D C D C D C D ...

As you can see, one original mistake will lead to a series of alternating defections for the rest of the game.

Thankfully, when `TFT` plays against `ALL_C` who makes a mistake, it is rather forgiving:

```
ALL_C:    C C C C C C D C C C C C C C C C C ...
TFT:      C C C C C C C D C C C C C C C C C ...
```

Now, what happens when we add `WSLS` to the mix? Here are the three combinations:

```
ALL_C:    C C C C C C D C C C C C C C C C C ...
WSLS:     C C C C C C C D D D D D D D D D D ...

TFT:      C C C C C C D C D D C D D C D D C D ...
WSLS:     C C C C C C C D D C D D C D D C D D ...

WSLS:     C C C C C C D D C C C C C C C C C ...
WSLS:     C C C C C C C D C C C C C C C C C ...
```

The first of these combinations is the crucial combination. As you can see, a simple mistake by `ALL_C` will trigger a change in the strategy of `WSLS`, switching into defector mode. As a consequence, `WSLS` will have a much higher payoff than `ALL_C`. In the second combination, both players will have a mixed payoff. However, in the last pattern, we can see that `WSLS` can correct mistakes quickly when it plays against itself. After a single round of mutual defection, both players switch back to cooperating, ending up with a high overall payoff. Thus, `WSLS` is doing very well in a population of strategies that make occasional mistakes, and that consists mostly of `WSLS`. In other words, it can resist invasion.

Let's demonstrate that by implementing occasional mistakes in the `compute_move()` method. Let's add a new global variable `error_rate` that defines the probability at which errors will occur when individuals calculate their moves. Note that this rate is different from the `mutation_rate`, which defines the probability at which strategies mutate to other strategies. Go ahead and define it right after the `mutation_rate`:

```
var error_rate = 0.002;
```

Then, right at the beginning of the `compute_move()` method, add these lines:

```
if (Math.random() < error_rate) {
    return (Math.random() < 0.5) ? "C" : "D";
}
```

Thus, with a small probability of `0.002`, we'll just flip a coin and return either `"C"` or `"D"`, without even checking what strategy the individual has.

If you now save your document and reload the browser, you'll see something like this:

As you can see, `TFT` and `ALL_C` cannot invade the `WSLS` population by neutral drift anymore.

We now have a population of individuals that cooperate at all times (with the exception of the occasional defection errors), and that cannot be invaded by unconditional defectors or unconditional cooperators (which would pave the way for the invasion of defectors).

We've come a long way! Let's briefly summarize what we've learned in this chapter.

- We learned about JavaScript objects, the fundamental building blocks of JavaScript programs (indeed, both arrays and functions are objects too).
- We built an individual-based simulation using a custom object, and we defined our own object methods.
- We learned that in JavaScript, objects are passed by reference, not by value, and discussed potential pitfalls.
- The prisoner's dilemma is a famous game-theoretical model to study the evolution of cooperation.
- Pure cooperators are immediately invaded by pure defectors.
- Tit-for-tat can resist invasion of pure defectors, but not of pure cooperators. Once the population is dominated by pure cooperators, it will immediately be invaded by defectors.
- The strategy win-stay, lose-shift can resist invasion by other strategies - notably by pure defectors and cooperators - if strategies make occasional mistakes.

And with that, you have come to the end of the book. Congratulations!

Epilogue

"A journey of a thousand miles begins with a single step." What was true for the Chinese philosopher Laozi 2,500 years ago is true for anyone learning programming today. Whether you were an absolute beginner or someone with some previous experience in programming, expressing your ideas in code is a never ending journey - so keep on writing code, keep on making mistakes, keep on asking, keep on learning.

If you're like most people, then the thrill of programming doesn't come from those rare times when everything goes smoothly. Rather, it comes from those times when you just can't seem to find that bug in your code that prevents your program from running - and then, after hours of dissecting the code, finding it and fixing it. "IT FINALLY WORKS!!!" It's a feeling that is hard to describe to someone who has never felt it.

Programming, after all, is like solving a puzzle, like constructing a building - a building of thoughts, of ideas. And as you are building, never forget that the world is increasingly made of millions and millions of such buildings, and there are millions of builders out there just like you, who get hung up on the same problems as you occasionally will. Consult websites such as www.stackoverflow.com[14], where you can join hundreds of thousands of people at various stages on their learning journey, asking and answering hundreds of thousands of questions. Build something that matters to you, and make it public so that others can learn from you, learn with you, improve your code, or build on top of what you have built.

Good luck!

[14] http://www.stackoverflow.com