Introduction

"Reading about nature is fine, but if a person walks in the woods and listens carefully, they can learn more than what is in books."

— George Washington Carver

Here we are: the beginning. Well, almost the beginning. If it's been a while since you've done any programming in JavaScript (or any math, for that matter), this introduction will get your mind back into computational thinking before I approach some of the more difficult and complex material.

In Chapter 1, I'm going to talk about the concept of a vector and how it will serve as the building block for simulating motion throughout this book. But before I take that step, let's think about what it means for something to simply move around the screen. Let's begin with one of the best-known and simplest simulations of motion—the random walk.

I.1 Random Walks

Imagine you are standing in the middle of a balance beam. Every ten seconds, you flip a coin. Heads, take a step forward. Tails, take a step backward. This is a random walk—a path defined as a series of random steps. Stepping off that balance beam and onto the floor, you could perform a random walk in two dimensions by flipping that same coin twice with the following results:

Flip 1	Flip 2	Result
Heads	Heads	Step forward.
Heads	Tails	Step right.
Tails	Heads	Step left.
Tails	Tails	Step backward.

Yes, this may seem like a particularly unsophisticated algorithm. Nevertheless, random walks can be used to model phenomena that occur in the real world, from the movements of molecules in a gas to the behavior of a gambler spending a day at the casino. For this book, the random walk is the perfect place to start with the following three goals in mind.

1. I need to review a programming concept central to this book—object-oriented programming. The random walker will serve as a template for how I will use object-oriented design to make things that move around a p5.js canvas.
2. The random walk instigates the two questions that I will ask over and over again throughout this book: "How do we define the rules that govern the behavior of our objects?" and then, "How do we implement these rules in JavaScript?"
3. Throughout the book, you'll periodically need a basic understanding of randomness, probability, and Perlin noise. The random walk will allow me to demonstrate a few key points that will come in handy later.

I.2 The Random Walker Class

Let's review a bit of object-oriented programming (OOP) first by building a `Walker` object. This will be only a cursory review. If you have never worked with OOP before, you may want something more comprehensive; I'd suggest stopping here and reviewing this video tutorial on the basics of ES6 classes (see page 0) with p5.js before continuing.

An ***object*** in JavaScript is an entity that has both data and functionality. I am looking designing a `Walker` object that both keeps track of its data (where it exists on the screen) and has the capability to perform certain actions (such as draw itself or take a step).

A ***class*** is the template for building actual instances of objects. Think of a class as the cookie cutter; the objects are the cookies themselves.

I'll begin by defining the `Walker` class—what it means to be a `Walker` object. The `Walker` only

needs two pieces of data—a number for its x-position and one for its y-position. These are initialized in the "constructor" function, appropriately named `constructor`. You can think of it as the object's `setup()`. There, we'll initialize the `Walker` object's starting position (in this case, the center of the window). Also note the use of the keyword `this` to attach the properties to the newly created object itself.

```
class Walker {
  constructor() {                          Objects have a constructor where they are initialized.
    this.x = width / 2;                    Objects have data.
    this.y = height / 2;
  }
```

Code formatting

Since the above lines of code above are the first to appear in this book, I'd like to take a moment to highlight a few important conventions I'm using in code.

- Code will always appear in a `monospaced font`.
- Comments that address what is happening in the code float to the right of the code.
- The light grey highlighting matches the comments with their corresponding lines of code.
- Code that appears broken up by paragraphs of text (like) often appears as unfinished snippets. For example, the closing bracket — `}` — for the Walker class does not appear until later below.

Finally, in addition to data, classes can be defined with functionality. In this example, a `Walker` object has two functions. The first one `show()` the object to display itself (as a black dot). Once again, never forget the `this.` when references the properties of the object.

```
show() {                                   Objects have functions.
  stroke(0);
  point(this.x, this.y);
}
```

The second function directs the `Walker` object to take a step. Now, this is where things get a bit more interesting. Remember that floor on which you were taking random steps? Well, now I can use a p5.js canvas in that same capacity. There are four possible steps. A step to the right can be simulated by incrementing `x` (`x++`); to the left by decrementing `x` (`x--`); forward by going down a pixel (`y++`); and backward by going up a pixel (`y--`). How does one pick from these four choices? Earlier I stated that you could flip two coins. In p5.js, however, when you want to randomly choose from a list of options, you can pick a random number using `random()`.

```
void step() {
  let choice = floor(random(4));                    0, 1, 2, or 3
```

The above line of code picks a random floating point number between 0 and 4 and coverts it to an integer (aka whole number) with a result of 0, 1, 2, or 3. Technically speaking, the random number picked can never be 4.0, but rather the highest possibility is 3.999999999 (with as many 9s as there are decimal places); since the `floor()` function lops off the decimal place, the highest possible `integer` is 3. Next, the walker takes the appropriate step (left, right, up, or down) depending on which random number was picked.

```
if (choice == 0) {                                  The random "choice" determines our step.
  x++;
} else if (choice == 1) {
  x--;
} else if (choice == 2) {
  y++;
} else {
  y--;
}
    }
  }
```

Now that I've written the class, it's time to make an actual `Walker` object in the main sketch—`setup()` and `draw()`. Assuming you are looking to model a single random walk, a single global variable.

```
let walker;                                         A Walker object
```

Then the object is created in `setup()` by referencing the class name with the `new` operator.

Example I.1: Traditional random walk

Each time you see the above Example heading in this book, it means there is a corresponding code example available on GitHub (see page 0).

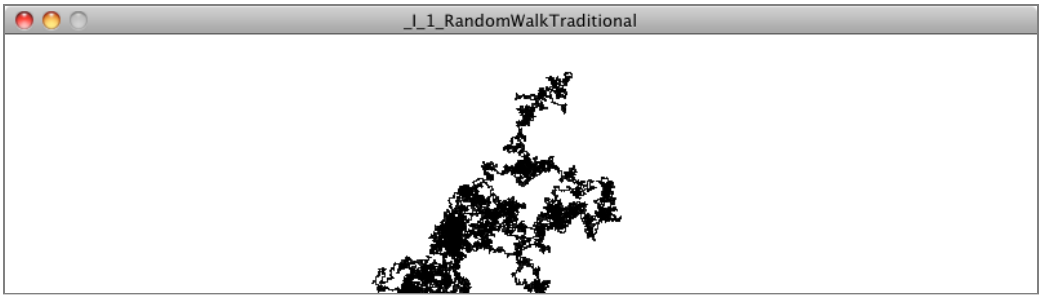
```
function setup() {
  createCanvas(640, 360);
  walker = new Walker();                            Create the Walker.
  background(255);
}
```

Finally, during each cycle through `draw()`, the `Walker` takes a step and draws a dot.

```
function draw() {
```

```
  walker.step();                                    Call functions on the Walker.
  walker.display();
}
```

Since the background is drawn once in `setup()`, rather than clearing it continually each time through `draw()`, the trail of the random walk is visible in the canvas.



There are a couple improvements that could be made to the random walker. For one, this `Walker` object's steps are limited to four options—up, down, left, and right. But any given pixel in the window has eight possible neighbors, and a ninth possibility is to stay in the same place.

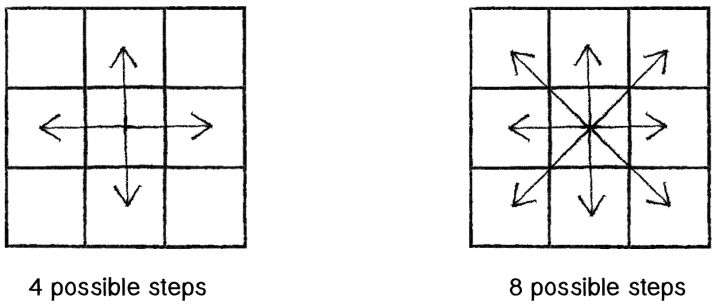


Figure I.1

To implement a `Walker` object that can step to any neighboring pixel (or stay put), I could pick a number between 0 and 8 (nine possible choices). However, another way to write the code would be to pick from three possible steps along the x-axis (-1, 0, or 1) and three possible steps along the y-axis.

```
function step() {
  let xstep = floor(random(3)) - 1;                 Yields -1, 0, or 1
  let ystep = floor(random(3)) - 1;

  this.x += xstep;
  this.y += ystep;
}
```

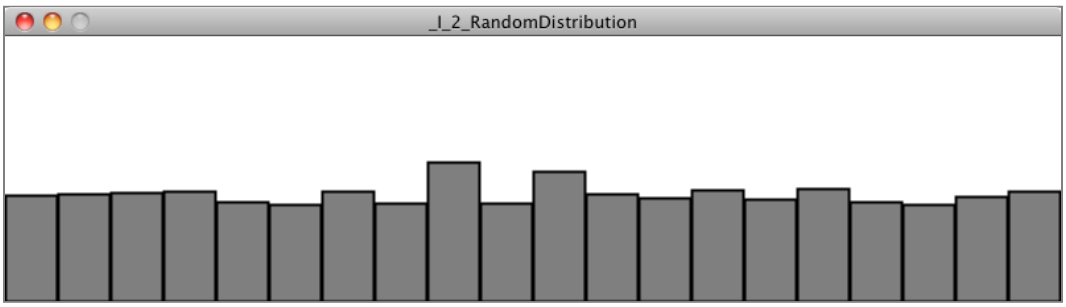
Taking this further, floating point numbers (i.e. decimal numbers) can be used to take a step with a continuous random value between -1 and 1.

```
function step() {
  let xstep = random(-1, 1);
  let ystep = random(-1, 1);
  this.x += xstep;
  this.y += ystep;
}
```

Any floating point number between -1.0 and 1.0

All of these variations on the "traditional" random walk have one thing in common: at any moment in time, the probability that the `Walker` will take a step in a given direction is equal to the probability that the `Walker` will take a step in any direction. In other words, if there are four possible steps, there is a 1 in 4 (or 25%) chance the `Walker` will take any given step. With nine possible steps, it's a 1 in 9 (or ~11.1%) chance.

Conveniently, this is how the `random()` function works. p5's random number generator (which operates behind the scenes) produces what is known as a "uniform" distribution of numbers. You can test this distribution with a sketch that counts each time a random number is picked and graphs it as the height of a rectangle.



Example I.2: Random number distribution

```
let randomCounts = [];
```

An array to keep track of how often random numbers are picked

```
function setup() {
  createCanvas(640, 240);
  for (let i = 0; i < 20; i++) {
    randomCounts[i] = 0;
  }
}

function draw() {
  background(255);
```

Pick a random number and increase the count.

```
  let index =
floor(random(randomCounts.length));
  randomCounts[index]++;
```

```
  stroke(0);
  fill(175);
  let w = width / randomCounts.length;
```

Graphing the results

```
  for (let x = 0; x < randomCounts.length; x++) {
    rect(x * w, height - randomCounts[x], w - 1, randomCounts[x]);
  }
```

Notice how each bar of the graph above differs slightly in height. The sample size (i.e. the number of random numbers picked) is small and there are some occasional discrepancies where certain numbers are picked more often. Over time, with a good random number generator, this would even out.

Pseudo-Random Numbers

The random numbers from the `random()` function are not truly random; they are known as "pseudo-random." They are the result of a mathematical function that simulates randomness. This function would yield a pattern over time, but that time period is so long that for the examples in this book, it's just as good as pure randomness!

Exercise I.1

Create a random walker that has a tendency to move down and to the right. (The solution follows in the next section.)

I.3 Probability and Non-Uniform Distributions

Remember when you first started programming with p5.js? Perhaps you wanted to draw a lot of circles on the screen. So you said to yourself: “Oh, I know. I’ll draw all these circles at random positions, with random sizes and random colors.” In a computer graphics system, it’s often easiest to seed a system with randomness. In this book, however, I am looking to build systems modeled on what we see in nature. Defaulting to randomness is not a particularly thoughtful solution to a design problem—in particular, the kind of problem that involves creating an organic or natural-looking simulation.

With a few tricks, I can use the `random()` function to produce “non-uniform” distributions of random numbers. This will come in handy throughout the book for a variety of scenarios. In chapter 9's genetic algorithms, for example, I’ll need a methodology for performing “selection”—which members of the population should be selected to pass their DNA to the next generation? This is akin to the Darwinian concept of “survival of the fittest”? Say you have a population of monkeys evolving. Not every monkey will have an equal chance of reproducing. To simulate Darwinian natural selection, you can’t simply pick two random monkeys to be parents. The more “fit” ones should be more likely to be chosen. This could vbe considered the “probability of the fittest.” For example, a particularly fast and strong monkey might have a 90% chance of procreating, while a weaker one has only a 10% chance.

Let’s pause here and take a look at probability’s basic principles. First I’ll examine single event probability, i.e. the likelihood that a given event will occur.

If you have a system with a certain number of possible outcomes, the probability of the occurrence of a given event equals the number of outcomes that qualify as that event divided by the total number of all possible outcomes. A coin toss is a simple example—it has only two possible outcomes, heads or tails. There is only one way to flip heads. The probability that the coin will turn up heads, therefore, is one divided by two: 1/2 or 50%.

Take a deck of fifty-two cards. The probability of drawing an ace from that deck is:

```
number of aces / number of cards = 4 / 52 = 0.077 = ~ 8%
```

The probability of drawing a diamond is:

```
number of diamonds / number of cards = 13 / 52 = 0.25 = 25%
```

We can also calculate the probability of multiple events occurring in sequence. To do this, multiply the individual probabilities of each event.

The probability of a coin turning up heads three times in a row is:

```
(1/2) * (1/2) * (1/2) = 1/8 (or 0.125)
```

...meaning that a coin will turn up heads three times in a row one out of eight times on average. For example if you flipped a coin three times in a row 500 times, you would expect to see an outcome of three consecutive heads an average of one eighth of the time or ~63 times.

Exercise I.2

What is the probability of drawing two aces in a row from a deck of fifty-two cards?

There are a couple of ways in which you can use the `random()` function with probability in code. One technique is to fill an array with numbers—some of which are repeated—then choose random elements from that array and generate events based on those choices.

`let stuff = {1, 1, 2, 3, 3};`	1 and 3 are stored in the array twice, making them more likely to be picked than 2.
`let value = random(stuff);` `print(value);`	Picking a random element from an array

Running this code will produce a 40% chance of printing the value 1, a 20% chance of printing 2, and a 40% chance of printing 3.

You can also ask for a random number (let's make it simple and just consider random floating point values between 0 and 1) and allow an event to occur only if our random number is within a certain range. For example:

`let probability = 0.1;`	A probability of 10%
`let r = random(1);`	A random floating point between 0 and 1

`if (r < prob) {`
`  print("Sing!");`
`}`
	If the random number is less than 0.1, sing!

This method can also be applied to multiple outcomes. Let's say that singing has a 60% chance of happening, dancing, a 10% chance, and sleeping, a 30% chance. This can be implemented by picking a random number and seeing into what range it falls.

- *between 0.00 and 0.60 (60%) ➜ Singing*
- *between 0.60 and 0.70 (10%) ➜ Dancing*
- *between 0.70 and 1.00 (30%) ➜ Sleeping*

```
let num = random(1);
```

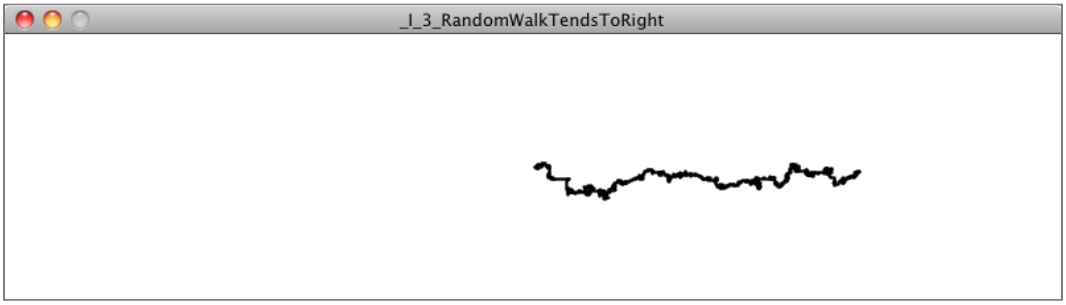
`if (num < 0.6) {` `  print("Sing!");`	If random number is less than 0.6

```
} else if (num < 0.7) {
  print("Dance!");
```
Between 0.6 and 0.7

```
} else {
  print("Sleep!");
}
```
Greater than 0.7

The above methodology can be used to create a random walker that tends to move in a particular direction. Here is an example of a `Walker` with the following probabilities:

- *chance of moving up: 20%*
- *chance of moving down: 20%*
- *chance of moving left: 20%*
- *chance of moving right: 40%*



intro ex03

Example I.3: Walker that tends to move to the right

```
function step() {

  let r = random(1);

  if (r < 0.4) {
    x++;
```
A 40% chance of moving to the right!

```
  } else if (r < 0.6) {
    x--;
  } else if (r < 0.8) {
    y++;
  } else {
    y--;
  }
}
```

Exercise I.3

Create a random walker with dynamic probabilities. For example, can you give it a 50% chance of moving in the direction of the mouse?

I.4 A Normal Distribution of Random Numbers

Let's go back to that population of simulated monkeys. Your program generates a thousand `Monkey` objects, each with a height value between 200 and 300 (as this is a world of monkeys that have heights between 200 and 300 pixels).

```
let h = random(200, 300);
```

Is this an accurate algorithm for creating a population of monkey heights? Think of a crowded sidewalk in New York City. Pick any person off the street and it may appear that their height is random. Nevertheless, it's not the kind of random that `random()` produces by default. People's heights are not uniformly distributed; there are many more people of average height than there are very tall or very short ones. To simulate a more "natural" distribution, the values picked for the monkeys' heights should mostly be around average (250 pixels), yet still be, on occasion, very short or very tall.

One of the most famous examples of a distribution of values that cluster around an average (referred to as the "mean") is called a "normal" distribution. It is also sometimes referred to as the Gaussian distribution (named for mathematician Carl Friedrich Gauss).

When you graph this distribution, you get something that looks like the following, informally known as a bell curve:

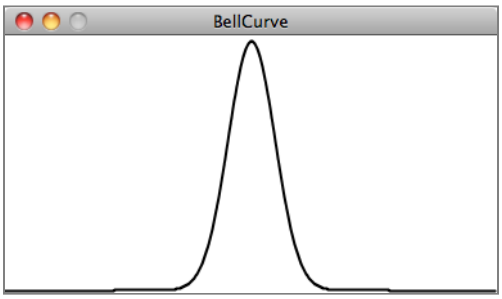


Figure I.2

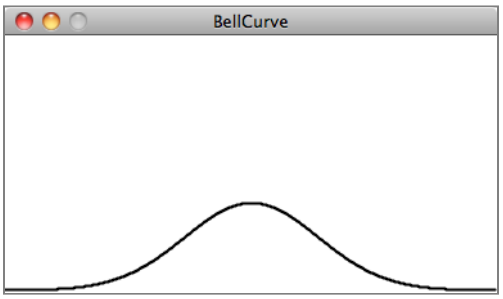


Figure I.3

The curve is generated by a mathematical function that defines the probability of any given value occurring as a function of the mean (often written as μ, the Greek letter *mu*) and standard deviation (σ, the Greek letter *sigma*).

The mean is pretty easy to understand. In the case of our height values between 200 and 300, you probably have an intuitive sense of the mean (i.e. average) as 250. However, what if I were to say that the standard deviation is 3? Or 15? What does this mean for the numbers? The graphs above should give you a hint. Figure I.2 shows the distribution with a very low standard deviation, where the majority of the values pile up around the mean. Figure I.3 shows us a higher standard deviation, where the values are more evenly spread out from the average.

The numbers work out as follows: Given a population, 68% of the members of that population will have values in the range of one standard deviation from the mean, 95% within two standard deviations, and 99.7% within three standard deviations. Given a standard deviation of 5 pixels, only 0.3% of the monkey heights will be less than 235 pixels (three standard deviations below the mean of 250) or greater than 265 pixels (three standard deviations above the mean of 250).

Calculating Mean and Standard Deviation

Consider a class of ten students who receive the following scores (out of 100) on a test:

85, 82, 88, 86, 85, 93, 98, 40, 73, 83

The mean is the average: 81.3

The standard deviation is calculated as the square root of the average of the squares of deviations around the mean. In other words, take the difference from the mean for each person and square it (aka "variance"). Calculate the average of all these values, take the square root, and you have the standard deviation.

Score	Difference from Mean	Variance
85	85 - 81.3 = 3.7	$(3.7)^2$ = 13.69
40	40 - 81.3 = -41.3	$(-41.3)^2$ = 1,705.69
etc.		
	Average Variance:	228.81

The standard deviation is the square root of the average variance: 15.13

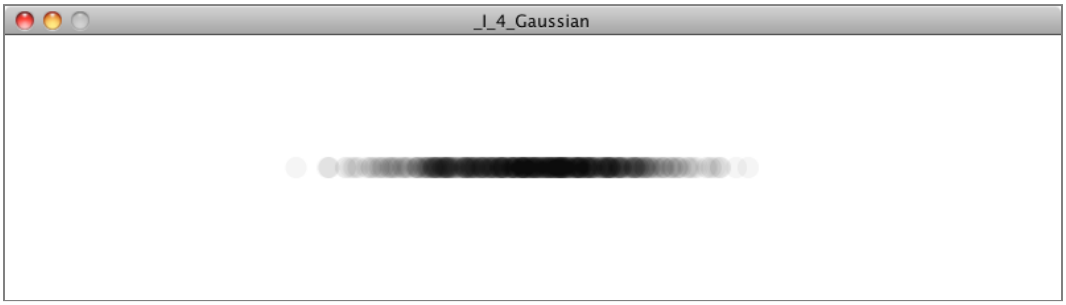
Luckily for you, to use a normal distribution of random numbers in a p5.js sketch, you don't have to do any of these calculations. Instead, the `randomGaussian()` function returns random numbers with a normal distribution.

```
function draw() {
  let num = randomGaussian();          Asking for a Gaussian random number.
}
```

Here's the thing. What are we supposed to do with this value? What if we wanted to use it, for example, to assign the x-position of a shape we draw on screen?

By default, the `randomGaussian()` function returns a normal distribution of random numbers with the following parameters: *a mean of zero* and *a standard deviation of one*. Let's say you want a mean of 320 (the center horizontal pixel in a window of width 640) and

a standard deviation of 60 pixels. The parameters can be adjusted by passing in two arguments, the mean followed by the standard deviation.



Example I.4: Gaussian distribution

```
void draw() {
  float x = randomGaussian(60, 320);
  noStroke();
  fill(0,10);
  ellipse(x, 180, 16, 16);
}
```

Note that randomGaussian() without arguments returns a value between 0 and 1.

I'll note that while it's convenient to pass in the arguments, the math for this is quite simple and just involves multiplying a normalized value between 0 and 1 by the standard deviation and adding it to the mean.

```
let x = 60 * randomGaussian() + 320;
```

By drawing the ellipses on top of each other with some transparency, you can begin to see see the distribution. The darkest spot is near the center, where most of the values cluster, but every so often circles are drawn farther to the right or left of the center.

Exercise I.4

Consider a simulation of paint splatter drawn as a collection of colored dots. Most of the paint clusters around a central position, but some dots do splatter out towards the edges. Can you use a normal distribution of random numbers to generate the positions of the dots? Can you also use a normal distribution of random numbers to generate a color palette? Try attaching a slider to standard deviation.

Exercise I.5

A Gaussian random walk is defined as one in which the step size (how far the object moves in a given direction) is generated with a normal distribution. Implement this variation of the random walk.

I.5 A Custom Distribution of Random Numbers

There will come a time in your life when you do not want a uniform distribution of random values, or a Gaussian one. Let's imagine for a moment that you are a random walker in search of food. Moving randomly around a space seems like a reasonable strategy for finding something to eat. After all, you don't know where the food is, so you might as well search randomly until you find it. The problem, as you may have noticed, is that random walkers return to previously visited positions many times (this is known as "oversampling"). One strategy to avoid such a problem is to, every so often, take a very large step. This allows the walker to forage randomly around a specific position while periodically jumping very far away to reduce the amount of oversampling. This variation on the random walk (known as a Lévy flight) requires a custom set of probabilities. Though not an exact implementation of a Lévy flight, one could state the probability distribution as follows: the longer the step, the less likely it is to be picked; the shorter the step, the more likely.

Earlier in this prologue, I wrote that you could generate custom probability distributions by filling an array with values (some duplicated so that they would be picked more frequently) or by testing the result of `random()`. A Lévy flight could be implemented with a 1% chance of the walker taking a large step.

```
let r = random(1);

if (r < 0.01) {                                    A 1% chance of taking a large step
  xstep = random(-100, 100);
  ystep = random(-100, 100);
} else {
  xstep = random(-1, 1);
  ystep = random(-1, 1);
}
```

However, this reduces the probabilities to a fixed number of options. What if you wanted to make a more general rule—the higher a number, the more likely it is to be picked? 3.145 would be more likely to be picked than 3.144, even if that likelihood is just a tiny bit greater. In other words, if `x` is the random number, the likelihood of it being picked could be mapped to the y-axis with the function `y = x`.

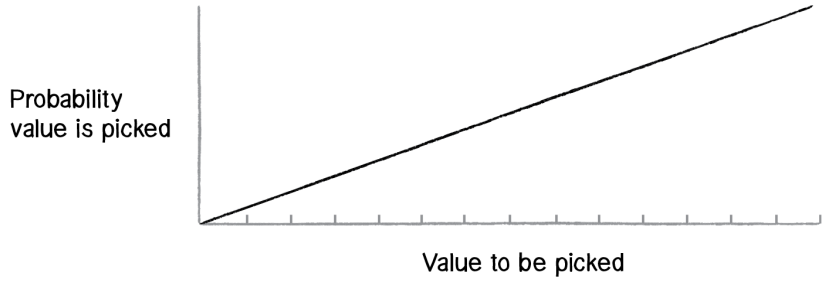


Figure I.4

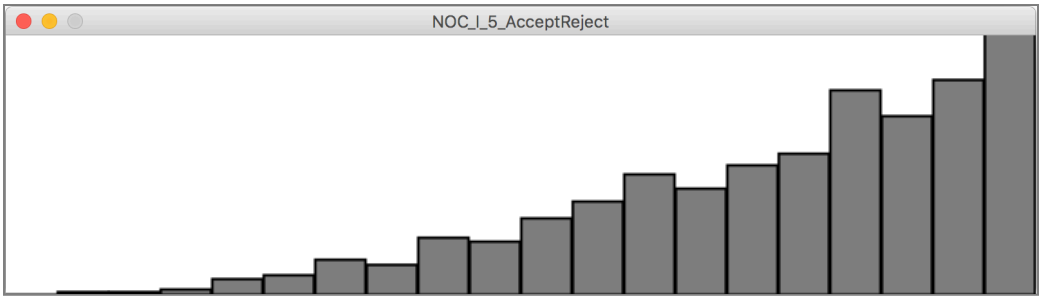
If we can figure out how to generate a distribution of random numbers according to the above graph, then we will be able to apply the same methodology to any curve for which we have a formula.

One solution is to pick two random numbers instead of one. The first random number is just that, a random number. The second one, however, is what we'll call a "qualifying random value." It will tell us whether to use the first one or throw it away and pick another one. Numbers that have an easier time qualifying will be picked more often, and numbers that rarely qualify will be picked infrequently. Here are the steps (for now, let's consider only random values between 0 and 1):

1. Pick a random number: R1
2. Compute a probability P that R1 should qualify. Let's try: P = R1.
3. Pick another random number: R2
4. If R2 is less than P, then we have found our number—R1!
5. If R2 is not less than P, go back to step 1 and start over.

Here we are saying that the likelihood that a random value will qualify is equal to the random number itself. Let's say we pick 0.1 for R1. This means that R1 will have a 10% chance of qualifying. If we pick 0.83 for R1 then it will have a 83% chance of qualifying. The higher the number, the greater the likelihood that we will actually use it.

Here is a function (named for the accept-reject algorithm, a type of Monte Carlo method, which was named for the Monte Carlo casino) that implements the above algorithm, returning a random value between 0 and 1.



Example I.5: Accept-Reject distribution

```
function acceptreject() {
  while (true) {                          // We do this "forever" until we find a qualifying random value.
    let r1 = random(1);                   // Pick a random value.
    let probability = r1;                 // Assign a probability.
    let r2 = random(1);                   // Pick a second random value.
    if (r2 < probability) {               // Does it qualify? If so, we're done!
      return r1;
    }
  }
}
```

Exercise I.6

Use a custom probability distribution to vary the size of a step taken by the random walker. The step size can be determined by influencing the range of values picked. Can you map the probability to a quadratic function—i.e. making the likelihood that a value is picked equal to the value squared?

```
let step = 10;                            // A uniform distribution of random step sizes. Change this!
let stepx = random(-step, step);
let stepy = random(-step, step);

x += stepx;
y += stepy;
```

(Later we'll see how to do this more efficiently using vectors.)

I.6 Perlin Noise (A Smoother Approach)

A good random number generator produces numbers that have no relationship and show no discernible pattern. As we are beginning to see, a little bit of randomness can be a good thing when programming organic, lifelike behaviors. However, randomness as the single guiding principle is not necessarily natural. An algorithm known as "Perlin noise," named for its inventor Ken Perlin, takes this concept into account. Perlin developed the noise function while working on the original *Tron* movie in the early 1980s; it was designed to create procedural textures for computer-generated effects. In 1997 Perlin won an Academy Award in technical achievement for this work. Perlin noise can be used to generate various effects with natural qualities, such as clouds, landscapes, and patterned textures like marble.

Perlin noise has a more organic appearance because it produces a naturally ordered ("smooth") sequence of pseudo-random numbers. The graph on the left below shows Perlin noise over time, with the x-axis representing time; note the smoothness of the curve. The graph on the right shows pure random numbers over time. (The code for generating these graphs is available in the accompanying book downloads.)

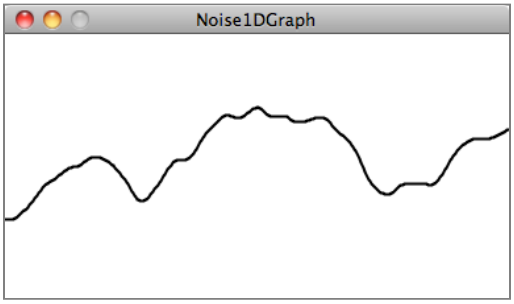


Figure I.5: Noise

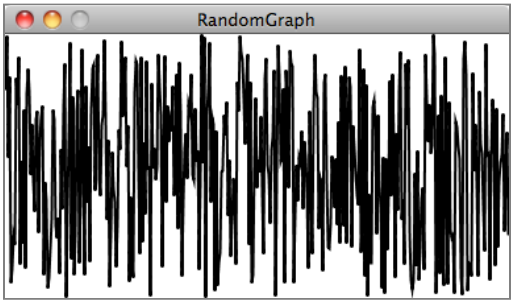


Figure I.6: Random

p5.js has a built-in implementation of the Perlin noise algorithm: the function `noise()`. The `noise()` function takes one, two, or three arguments, as noise is computed in one, two, or three dimensions. Let's start by looking at one-dimensional noise.

Noise Detail

The p5.js noise reference (see page 0) explains that noise is calculated over several "octaves." Calling the `noiseDetail()` (see page 0) function changes both the number of octaves and their importance relative to one another. This in turn changes quality of the noise values produced.

You can learn more about the history of Perlin noise Ken Perlin's website (see page 0).

Let's begin exploring noise by drawing a circle on a canvas at a random x-position.

```
let x = random(0, width);
ellipse(x, 180, 16, 16);
```

A random x-position

Now, instead of a random x-position, I want a Perlin noise x-position that is "smoother." You might think that all you need to do is replace `random()` with `noise()`, i.e.

```
let x = noise(0, width);
```

A noise x-position?

While conceptually this is exactly what we want to do—calculate an x-value that ranges between 0 and the width according to Perlin noise—this is not the correct implementation. While the arguments to the `random()` function specify a range of values between a minimum and a maximum, `noise()` does not work this way. Instead, the output range is fixed—it always returns a value between 0 and 1. You'll see in a moment that you can get around this easily with p5's `map()` function, but first let's examine what exactly `noise()` expects us to pass in as an argument.

One-dimensional Perlin noise can be thought of as a linear sequence of values over time. For example:

Time	Noise Value
0	0.365
1	0.363
2	0.363
3	0.364
4	0.366

Now, in order to access a particular noise value, a "moment in time" must be specified and passed to the `noise()` function. For example:

```
let n = noise(3);
```

According to the above table, `noise(3)` returns 0.364. The next step in exploring noise is to use a variable for time and ask for a noise value continuously in `draw()`.

```
let t = 3;

function draw() {
  let n = noise(t);                    We need the noise value for a specific moment in time.

  print(n);
}
```

The above code results in the same value printed over and over. This happens because I am asking for the result of the `noise()` function at the same point in time—3—over and over. If the time variable `t` increments, however, I'll get a different result.

```
let t = 0;                             Typically we would start with an offset of time = 0, though this is arbitrary.

function draw() {
  let n = noise(t);
  print(n);

  t += 0.01;                           Now, we move forward in time!

}
```

How quickly `t` increments also affects the smoothness of the noise. Large jumps in time that skip ahead through the noise space produce values that are less smooth, and more random.

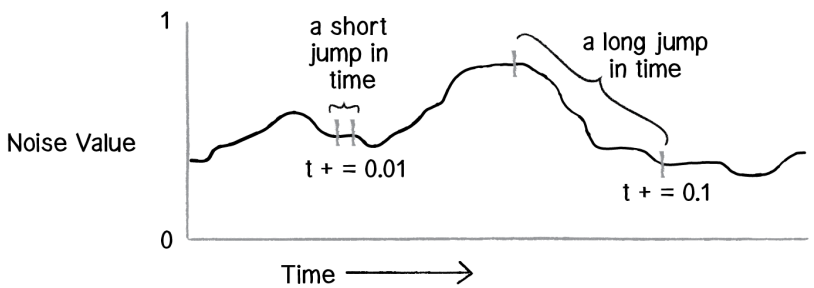


Figure I.7

Try running the code several times, incrementing `t` by 0.01, 0.02, 0.05, 0.1, 0.0001, and you will see different results.

Mapping Noise

Now it's time to answer the question of what to do with the noise value. Once you have the value with a range between 0 and 1, it's up to you to map that range accordingly. The easiest way to do this is with p5's `map()` function. The `map()` function takes five arguments. First is the value you want to map, in this case `n`. This is followed by the value's current range (minimum and maximum), followed by the desired range.

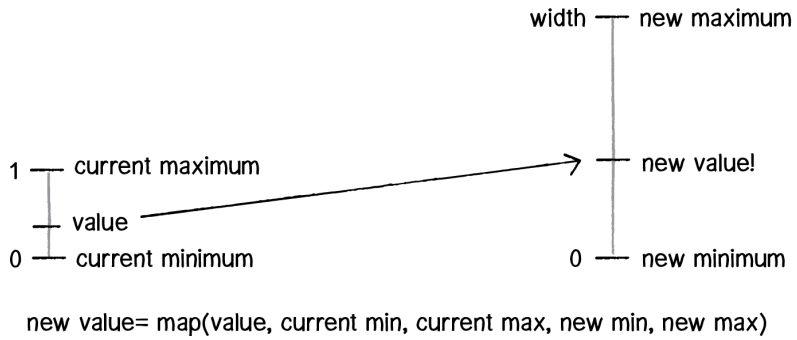


Figure I.8

In this case, while noise has a range between 0 and 1, I'd like to draw a circle with a range between 0 and the canvas's width.

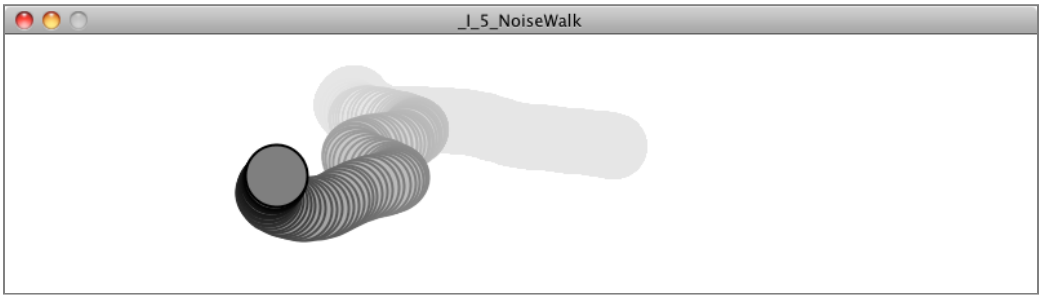
```
let t = 0;

function draw() {
  let n = noise(t);
  let x = map(n, 0, 1, 0, width);
  ellipse(x, 180, 16, 16);

  t += 0.01;
}
```

Using map() to customize the range of Perlin noise

The exact same logic can be applied to the random walker, assigning both its x- and y-values according to Perlin noise.



Example I.6: Perlin noise walker

```
class Walker {
  Walker() {
    this.tx = 0;
    this.ty = 10000;
  }

  void step() {
    this.x = map(noise(tx), 0, 1, 0, width);          x- and y-position mapped from noise
    this.y = map(noise(ty), 0, 1, 0, height);

    this.tx += 0.01;                                  Move forward through "time."
    this.ty += 0.01;

  }
}
```

Notice how the above example requires a new pair of variables: `tx` and `ty`. This is because we need to keep track of two time variables, one for the x-position of the `Walker` object and one for the y-position. But there is something a bit odd about these variables. Why does `tx` start at 0 and `ty` at 10,000? While these numbers are arbitrary choices, I have intentionally initialized the two time variables this way. This is because the noise function is deterministic: it gives you the same result for a specific time `t` each and every time. If I asked for the noise value at the same time `t` for both `x` and `y`, then `x` and `y` would always be equal, meaning that the `Walker` object would only move along a diagonal. Instead, I use two different parts of the noise space, starting at 0 for `x` and 10,000 for `y` so that `x` and `y` appear to act independently of each other.

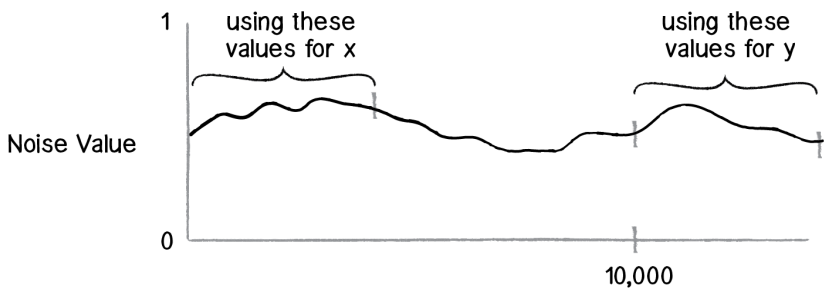


Figure I.9

In truth, there is no actual concept of time at play here. It's a useful metaphor to help us understand how the noise function works, but really what we have is space, rather than time. The graph above depicts a linear sequence of noise values in a one-dimensional space, and values are retrieved at a specific x-position. In examples, you will often see a variable named `xoff` to indicate the x-offset along the noise graph, rather than `t` for time (as noted in the

diagram).

Exercise I.7

In the above random walker, the result of the noise function is mapped directly to the walker's position. Create a random walker where you instead map the result of the `noise()` function to a walker's step size.

Two-Dimensional Noise

This idea of noise values living in a one-dimensional space is important because it leads right into a discussion of two-dimensional space. Think about this for a moment. With one-dimensional noise, there is a sequence of values in which any given value is similar to its neighbor. Because the values live in one dimension, each has only two neighbors: a value that comes before it (to the left on the graph) and one that comes after it (to the right).

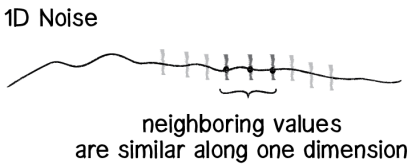


Figure I.10: 1D Noise

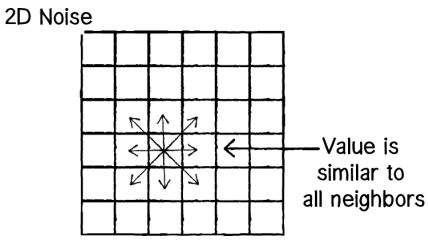
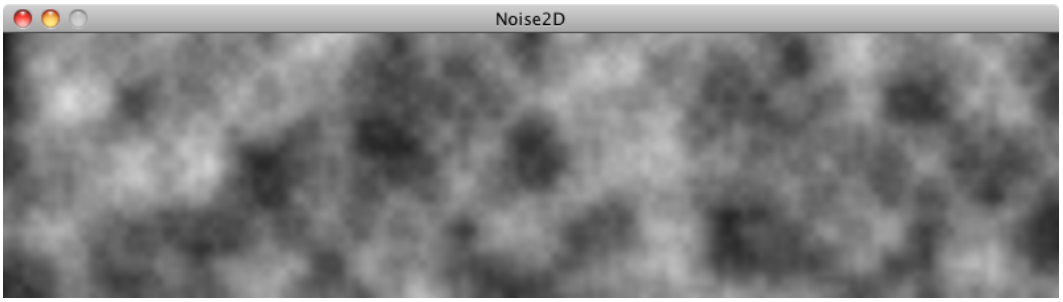


Figure I.11: 2D Noise

Two-dimensional noise works exactly the same way conceptually. The difference of course is that the values don't live along a linear path, but rather sit on a grid. Think of a piece of graph paper with numbers written into each cell. A given value will be similar to all of its neighbors: above, below, to the right, to the left, and along any diagonal.

If you were to visualize this graph paper with each value mapped to the brightness of a color, you would get something that looks like clouds. White sits next to light gray, which sits next to gray, which sits next to dark gray, which sits next to black, which sits next to dark gray, etc.



This is why noise was originally invented. If you tweak the parameters and play with color, the resulting images look more like marble or wood or any other organic texture.

Let's take a quick look at how to implement two-dimensional noise. If you wanted to color every pixel of a canvas randomly, you would need a nested loop, one that accessed each pixel and picked a random brightness. (Note that is p5, the pixels are arranged in an array with 4 spots for each: red, green, blue, and alpha. For details, see this video tutorial on the pixel array (see page 0).

```
loadPixels();
for (let x = 0; x < width; x++) {
  for (let y = 0; y < height; y++) {
    let index = (x + y * width) * 4;
    let bright = random(255);                    A random brightness!
    pixels[index    ] = bright;                  Setting the red, green, and blue values
    pixels[index + 1] = bright;
    pixels[index + 2] = bright;
  }
}
updatePixels();
```

To color each pixel according to the `noise()` function, we'll do exactly the same thing, only instead of calling `random()` we'll call `noise()`.

```
let bright = map(noise(x, y), 0, 1, 0, 255);      A Perlin noise brightness!
```

This is a nice start conceptually—it gives you a noise value for every (x,y) position in a two-dimensional space. The problem is that this won't have the cloudy quality we want. Jumping from pixel 200 to pixel 201 is too large of a jump through noise. Remember, with one-dimensional noise, I incremented the time variable by 0.01 each frame, not by 1! A pretty good solution to this problem is to just use different variables for the noise arguments. For example, a variable called `xoff` can be incremented each time x increases horizontally, and a `yoff` variable each time `y` moves vertically through the nested loops.

Example I.7: 2D Perlin noise

```
let xoff = 0.0;                                    Start xoff at 0.
for (let x = 0; x < width; x++) {
  let yoff = 0.0;                                  For every xoff, start yoff at 0.
  for (let y = 0; y < height; y++) {
    let bright = map(noise(xoff, yoff), 0, 1, 0,   Use xoff and yoff for noise().
255);
    let index = (x + y * width) * 4;               Use x and y for pixel position.
    pixels[index    ] = bright;                    Setting the red, green, and blue values
    pixels[index + 1] = bright;
    pixels[index + 2] = bright;
    yoff += 0.01;                                  Increment yoff.
  }
  xoff += 0.01;                                    Increment xoff.
}
```

Exercise I.8

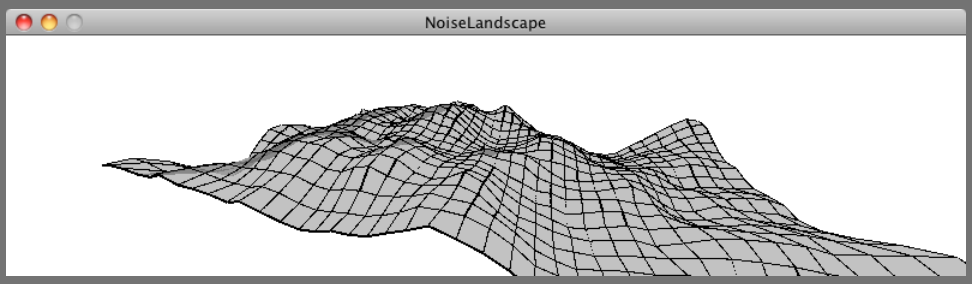
Play with color, `noiseDetail()`, and the rate at which `xoff` and `yoff` are incremented to achieve different visual effects.

Exercise I.9

Add a third argument to noise that increments once per cycle through `draw()` to animate the two-dimensional noise.

Exercise I.10

Use the noise values as the elevations of a landscape. See the screenshot below as a reference.



intro exc10

I've examined several traditional uses of Perlin noise in this section. With one-dimensional noise, smooth values were assigned to the position of an object to give the appearance of wandering. With two-dimensional noise, a cloudy pattern was generated with smoothed values on a plane of pixels. It's important to remember, however, that Perlin noise values are just that—values. They aren't inherently tied to pixel positions or color. Any example in this book that has a variable could be controlled via Perlin noise. When I model a wind force, its strength could be controlled by Perlin noise. Same goes for the angles between the branches in a fractal tree pattern, or the speed and direction of objects moving along a grid in a flow field simulation.

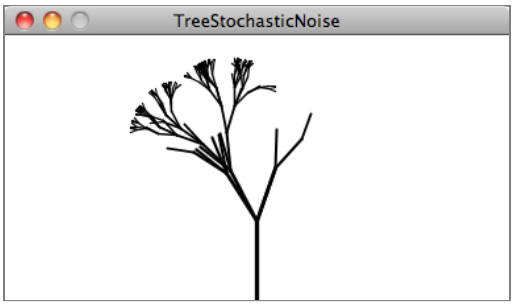


Figure I.12: Tree with Perlin noise

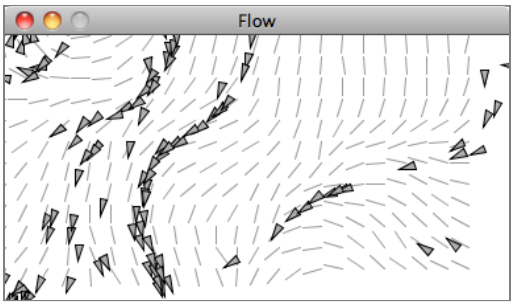


Figure I.13: Flow field with Perlin noise

I.7 Onward

I began this chapter by talking about how randomness can be a crutch. In many ways, it's the most obvious answer to the kinds of questions we ask continuously—how should this object move? What color should it be? This obvious answer, however, is sometimes a lazy one.

As I finish the introduction, it's also worth noting that you could just as easily fall into the trap of using Perlin noise as a crutch. How should this object move? Perlin noise! What color should it be? Perlin noise! How fast should it grow? Perlin noise!

The point of all of this is not to say that you should or shouldn't use randomness. Or that you should or shouldn't use Perlin noise. The point is that the rules of your system are defined by you, and the larger your toolbox, the more choices you'll have as you implement those rules. The goal of this book is to fill your toolbox. If all you know is random, then your design thinking is limited. Sure, Perlin noise helps, but you'll need more. A lot more.

I think we're ready to begin.