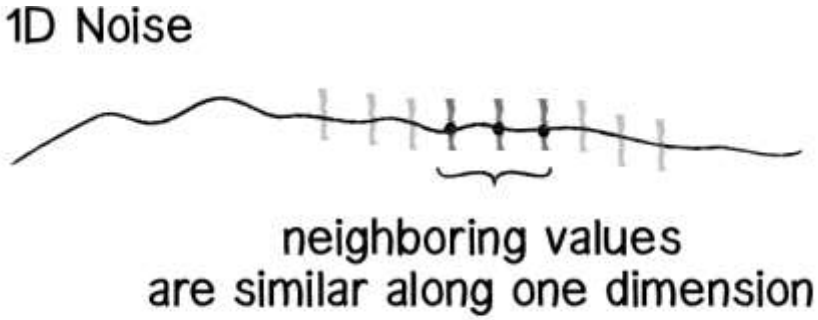
Two dimensional noise

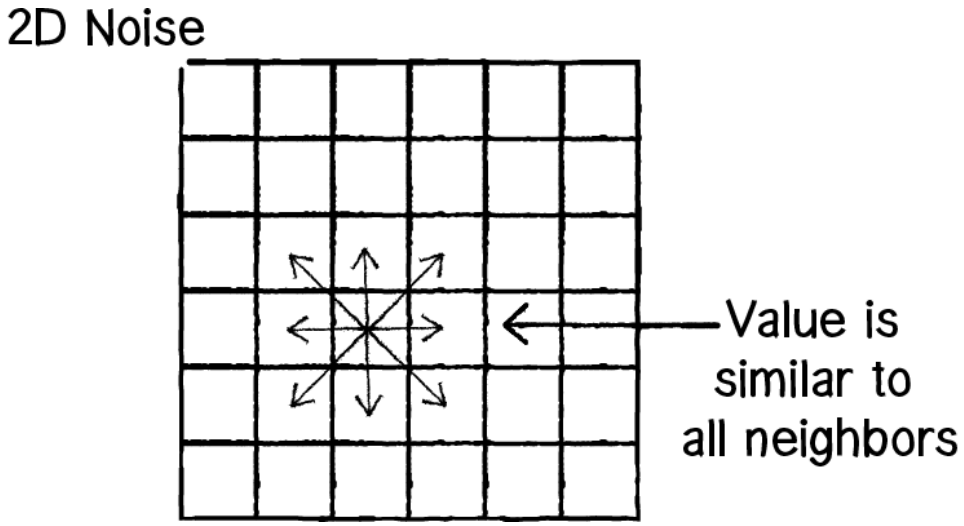
This idea of noise values living in a one-dimensional space is important because it leads us right into a discussion of two-dimensional space. Let's think about this for a moment. With one-dimensional noise, we have a sequence of values in which any given value is similar to its neighbour. Because the value is in one dimension, it only has two neighbours: a value that comes before it (to the left on the graph) and one that comes after it (to the right).



Nature of Code Image

Figure I.10: 1D Noise

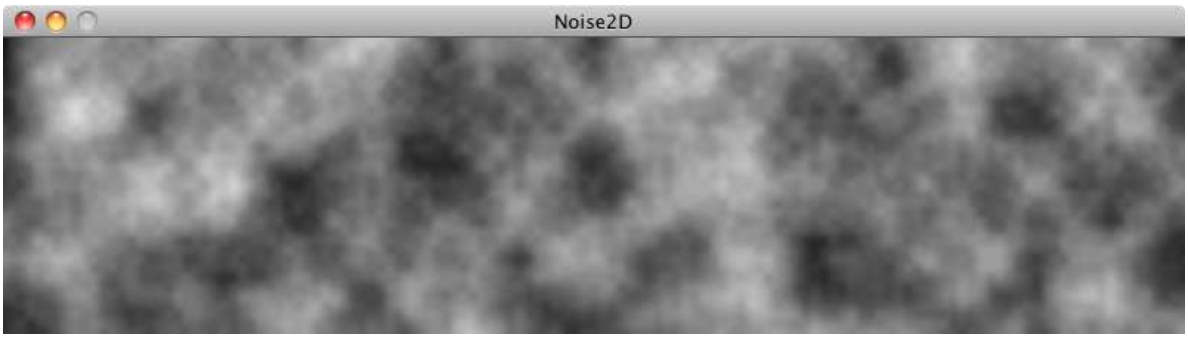
Two-dimensional noise works exactly the same way conceptually. The difference of course is that we aren't looking at values along a linear path, but values that are sitting on a grid. Think of a piece of graph paper with numbers written into each cell. A given value will be similar to all of its neighbours: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



Nature of Code Image

Figure I.11: 2D Noise

If you were to visualize this graph paper with each value mapped to the brightness of a colour, 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.



Nature of Code Image

This is why noise was originally invented. You tweak the parameters a bit or play with colour to make the resulting image look more like marble or wood or any other organic texture.

Let’s take a quick look at how to implement two-dimensional noise in ProcessingJS. If you wanted to colour every pixel of a window randomly, you would need a nested loop, one that accessed each pixel and picked a random brightness.

```
for (var x = 0; x < 100; x++) {
```

```
  for (var y = 0; y < 100; y++) {
    // A random brightness!
    var bright = random(255);
    stroke(bright, bright, bright);
    point(x, y);
  }
}
```

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

```
var bright = map(noise(x,y), 0, 1, 0, 255);
```

This is a nice start conceptually—it gives you a noise value for every (`x`,`y`) location in our 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, when we worked with one-dimensional noise, we incremented our 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, we could increment a variable called `xoff` each time we move horizontally, and a `yoff` variable each time we move vertically through the nested loops.

```
var xoff = 0.0;
for (var x = 0; x < 100; x++) {
  var yoff = 0.0;
  for (var y = 0; y < 100; y++) {
    var bright = map(noise(xoff, yoff), 0, 1, 0, 255);
    stroke(bright, bright, bright);
    point(x, y);
    yoff += 0.01;
```

```
    }
    xoff += 0.01;
}
```

We've examined several traditional uses of Perlin noise in this tutorial. With one-dimensional noise, we used smooth values to assign the location of an object to give the appearance of wandering. With two-dimensional noise, we created a cloudy pattern 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 locations or colour. Any example in these tutorials that has a variable could be controlled via Perlin noise. When we 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, like in the program below.

We began the last tutorial 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 colour should it be? This obvious answer, however, can also be a lazy one.

As we finish off this tutorial, it's also worth noting that we could just as easily fall into the trap of using Perlin noise as a crutch. How should this object move? Perlin noise! What colour 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 these tutorials is to help you fill your toolbox. If all you know is one type of randomness, then all of your designs will include only one

type of randomness. Perlin noise is another tool for randomness you can use in your programs. After a little practice with Perlin noise we will move on to another type of tool- vectors!