The Nature of Code

Introduction

“I am two with nature.” —Woody Allen

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

In Chapter 1, we’re 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 we 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 backwards. 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.

Flip 1	Flip 2	Result
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 us, we begin this book studying a random walk with three goals in mind.

1. We need to review a programming concept central to this book—object-oriented programming. The random walker will serve as a template for how we will use object-oriented design to make things that move around a Processing window.

2. The random walk instigates the two questions that we 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 Processing?”

3. Throughout the book, we’ll periodically need a basic understanding of randomness, probability, and Perlin noise. The random walk will allow us 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 the basics on the Processing website (See page 0) before continuing.

An ***object*** in Processing is an entity that has both data and functionality. We are looking to design 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. Let’s begin by defining this template—what it means to be a Walker object.

The Walker only needs two pieces of data—a number for its x-location and one for its y-location.

```
class Walker {
```

```
int x;                                  Objects have data.
int y;
```

Every class must have a constructor, a special function that is called when the object is first created. You can think of it as the object's **`setup()`**. There, we'll initialize the Walker's starting location (in this case, the center of the window).

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

Finally, in addition to data, classes can be defined with functionality. In this example, a Walker has two functions. We first write a function to display itself (as a white dot).

```
void display() {                        Objects have functions.
  stroke(255);
  point(x,y);
}
```

The second function directs the object to take a step. Now, this is where things get a bit more interesting. Remember that floor on which we were taking random steps? Well, now we can use a Processing window 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 do we pick from these four choices? Earlier we stated that we could flip two coins. In Processing, however, when we want to randomly choose from a list of options, we can pick a random number using **`random()`**.

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

The above line of code picks a random floating point number between 0 and 4 and converts it to an integer, resulting in 0, 1, 2, or 3. Technically speaking, the highest number will never be 4.0, but rather 3.999999999 (with as many 9s as there are decimal places); since the process of converting to an integer lops off the decimal place, the highest int we can get is 3. Next, we take the appropriate step (left, right, up, or down) depending on which random number was picked.

```
    if (choice == 0) {
      x++;
    } else if (choice == 1) {
      x--;
    } else if (choice == 2) {
      y++;
    } else {
      y--;
    }
  }
}
```

The random “choice” determines our step.

Now that we’ve written the class, it’s time to make an actual Walker object in the main part of our sketch—**`setup()`** and **`draw()`**. Assuming we are looking to model a single random walk, we declare one global variable of type Walker.

```
Walker w;
```

A Walker object

Then we create the object in **`setup()`** by calling the constructor with the **`new`** operator.

Example I.1: Traditional Random Walk

```
void setup() {
  size(640,360);
  w = new Walker();  // [bold]
  background(0);
}
```

Create the Walker.

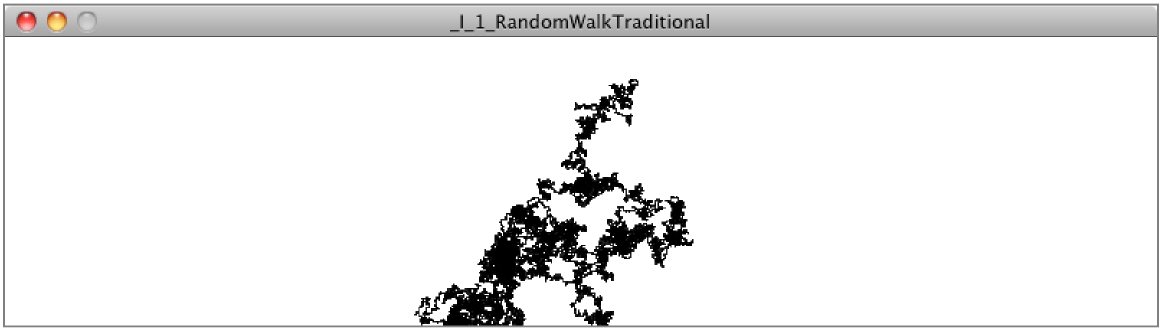
Finally, during each cycle through **`draw()`**, we ask the Walker to take a step and draw a dot.

```
void draw() {
```

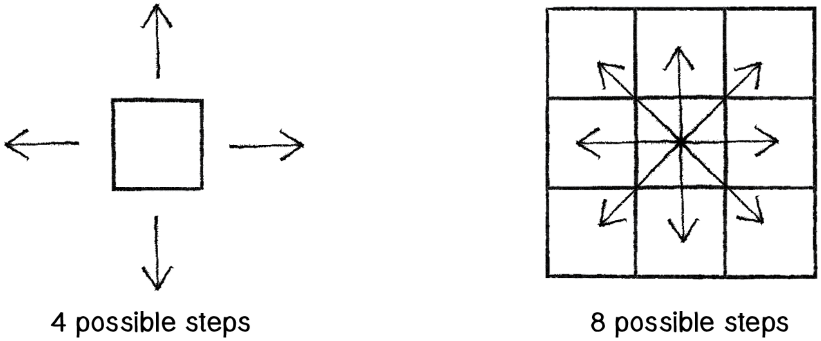
Call functions on the Walker.

```
    w.step(); // [bold]
    w.display(); // [bold]
  }
```

Since we only draw the background once in **`setup()`** (rather than clearing it continually each time through **`draw()`**), we see the trail of the random walk in our Processing window.



There are a couple improvements we could make to the random walker. For one, this walker's step choices are limited to four—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.


4 possible steps

8 possible steps

Figure I.1

To implement a walker that can step to any neighboring pixel (or stay put) we could then pick a number between zero and eight (nine possible choices). However, a more efficient way to write the code would be to simply pick from three possible steps along the x-axis (-1, 0, or 1) and three possible steps along the y-axis.

```
    void step() {
```

Yields -1, 0, or 1

```
    int stepx = int(random(3))-1;
    int stepy = int(random(3))-1;
    x += stepx;
    y += stepy;
  }
```

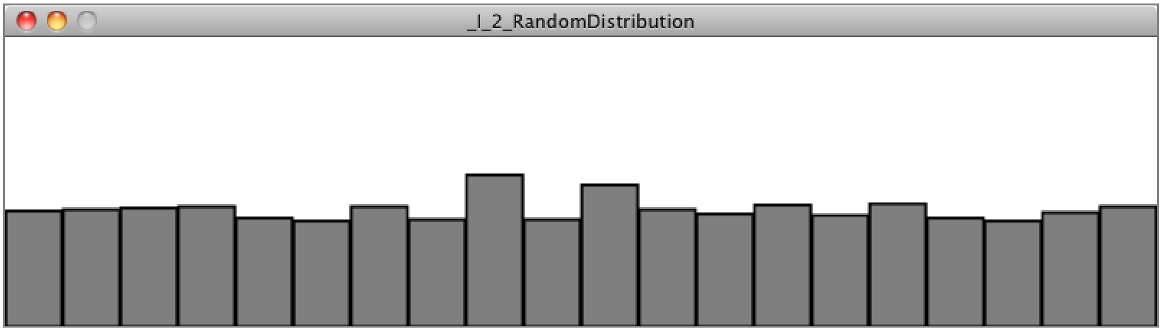
Taking this a step further, we could use floating point numbers (i.e. decimal numbers) for x and y instead and move according to an arbitrary random value between -1 and 1.

```
  void step() {
    float stepx = random(-1, 1);
    float stepy = random(-1, 1);
    x += stepx;
    y += stepy;
  }
```

Yields 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 one in four (or 25%) chance the walker will take any given step. With nine possible steps, it’s a one in nine (or 11.1%) chance.

Conveniently, this is how the **random()** function works. Processing’s random number generator (which operates behind the scenes) produces what is known as a “uniform” distribution of numbers. We can test this distribution with a Processing 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

```
int[] randomCounts;                       An array to keep track of how often random numbers are picked

void setup() {
  size(640,240);
  randomCounts = new int[20];
}

void draw() {
  background(255);
  int index =                             Pick a random number and increase the count.
int(random(randomCounts.length));
  randomCounts[index]++;
  stroke(0);                              Graphing the results
  fill(175);
  int w = width/randomCounts.length;
  for (int x = 0; x < randomCounts.length; x++) {
    rect(x*w,height-randomCounts[x],w-1,randomCounts[x]);
  }
}
```

The above screenshot shows the result of the sketch running for a few minutes. Notice how each bar of the graph differs in height. Our sample size (i.e. the number of random numbers we've picked) is rather small and there are some random 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 we get from the **`random()`** function are not truly random and are therefore 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 us, 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. (We'll see the solution to this in the next section.)

I.3 Probability and Non-Uniform Distributions

Remember when you first started programming in Processing? 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 locations, 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, we’re 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 problems that involve creating an organic or natural-looking simulation.

With a few tricks, we can change the way we use `random()` to produce “non-uniform” distributions of random numbers. This will come in handy throughout the book as we look at a number of different scenarios. When we examine genetic algorithms, for example, we’ll need a methodology for performing “selection”—which members of our population should be selected to pass their DNA down to the next generation. Remember the concept of survival of the fittest? Let’s say we have a population of monkeys evolving. Not every monkey will have a equal chance of reproducing. To simulate Darwinian evolution, we can’t simply pick two random monkeys to be parents. We need the more “fit” ones to be more likely to be chosen. We need to define 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 review the basic principles of probability, first looking at “Single Event Probability,” i.e. the likelihood of something to occur.

Given a system with a certain number of possible outcomes, the probability of any given event occurring is the number of outcomes that qualify as that event divided by the total number of possible outcomes. The simplest example is a coin toss. There are a total of two possible outcomes (heads or tails). There is only one way to flip heads. Therefore, the probability of heads is one divided by two, i.e. 1/2 or 50%.

Consider 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 as the product of the individual probabilities of each event.

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

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

In other words, a coin will land heads three times in a row one out of eight times (with each “time” being three tosses).

> ### Exercise I.2
>
> What is the probability of drawing two aces in a row from the deck of cards?

There are a few different techniques for using the **`random()`** function with probability in code. For example, if we fill an array with a selection of numbers (some repeated), we can randomly pick from that array and generate events based on what we select.

```
int[] stuff = new int[5]
```

```
stuff[0] = 1;
stuff[1] = 1;
stuff[2] = 2;
stuff[3] = 3;
stuff[4] = 3;
```

1 is stored in the array twice to increase its likelihood of being picked.

```
int index = int(random(stuff.length));
```

Picking a random element from an array

If you run this code, there will be a 40% chance of printing the value 1, a 20% chance of printing 2, and a 40% chance of printing 3.

Another strategy is to ask for a random number (for simplicity, we’ll consider random floating point values between 0 and 1) and allow an event to occur only if the random number we pick is within a certain range. For example:

```
float prob = 0.10;
```

A probability of 10%

```
float r = random(1);
```

A random floating point value between 0 and 1

```
if (r < prob) {
```

If our random number is less than 0.1

```
}
```

DO SOMETHING!

This same technique can also be applied to multiple outcomes.

Outcome A — 60% | Outcome B — 10% | Outcome C — 30%

To implement this in code, we pick one random float and check where it falls.

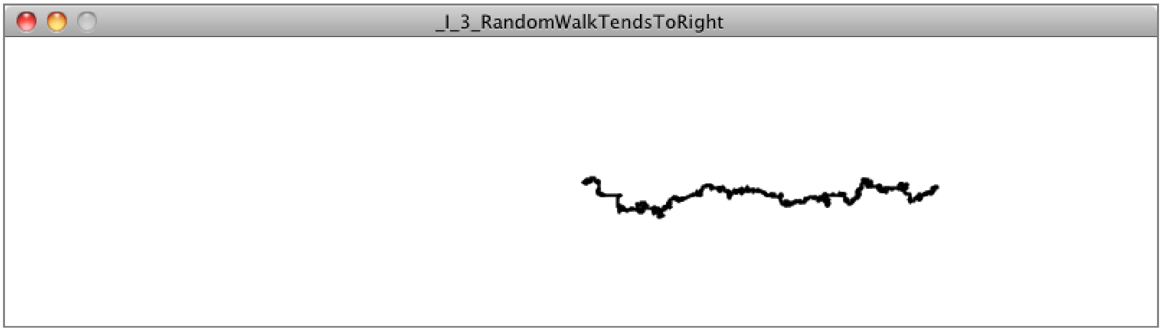
- *between 0.00 and 0.60 (60%) –> outcome A*

- *between 0.60 and 0.70 (10%) –> outcome B*
- *between 0.70 and 1.00 (30%) –> outcome C*

```
float num = random(1);
if (num < 0.6) {                          If random number is less than .6
  println("Outcome A");
} else if (num < 0.7) {                   Between 0.6 or 0.7
  println("Outcome B");
} else {                                  Greater than 0.7
  println("Outcome C");
}
```

We could use the above methodology to create a random walker that tends to move to the right. 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%*



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

```
void step() {

  float r = random(1);
```

```
    if (r < 0.4) {                    A 40% of moving to the right!
      x++;
    } 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 Processing monkeys. Your program generates a thousand “Monkey” objects, each with a “height” value between 200 and 300 (as his is a world of monkeys that have heights between 200 and 300 pixels).

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

Does this accurately depict the heights of real-world beings? 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 the `random()` produces. People’s heights are not uniformly distributed; there are a great deal more people of average height than there are very tall or very short ones. To simulate nature, we may want it to be more likely that our monkeys are of average height (250 pixels), yet allow them to still on occasion be very short or very tall.

A distribution of values that cluster around an average (referred to as the “mean”) is known as a “normal” distribution. It is also called the Gaussian distribution (named for mathematician Carl Friedrich Gauss) or, if you are French, the Laplacian distribution (named for Pierre-Simon Laplace). Both mathematicians were working concurrently in the early nineteenth century on defining such a distribution.

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

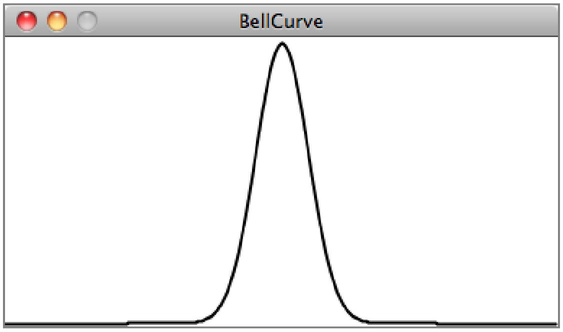


Figure I.2

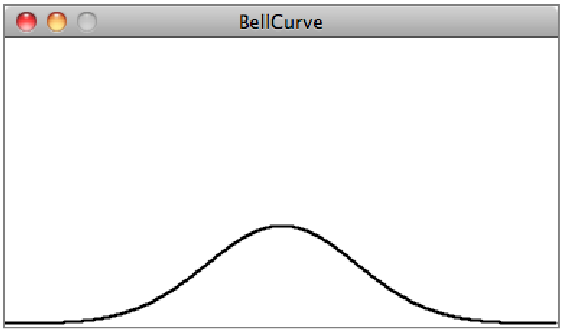


Figure I.3

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 us a hint. The graph on the left shows us the distribution with a very low standard deviation, where the majority of the values cluster closely around the mean. The graph on the right 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, 98% within two standard deviations, and 99.7% within three standard deviations. Given a standard deviation of five 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 (variance). Calculate the average of all these values and take the square root as our standard deviation.

variance table

Score	Difference from Mean	Variance
85	85-81.3 = 3.7	$(3.7)^2$ = 13.69
40	40-81.3 = -41.3	$(-41.3)^2$ = 1705.69
etc.		
	Average Variance:	**254.23**

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

Luckily for us, to use a normal distribution of random numbers in a Processing sketch, we don’t have to do any of these calculations ourselves. Instead, we can make use of a class known as Random, which we get for free as part of the default Java libraries imported into Processing (see http://docs.oracle.com/javase/6/docs/api/java/util/Random.html (See page 0) for more information).

To use the Random class, we must first declare a variable of type Random and create the Random object in **`setup()`**.

```
Random generator;
```

We use the variable name “generator” as what we have here can be thought of as a random number generator.

```
void setup() {
  size(640,360);
```

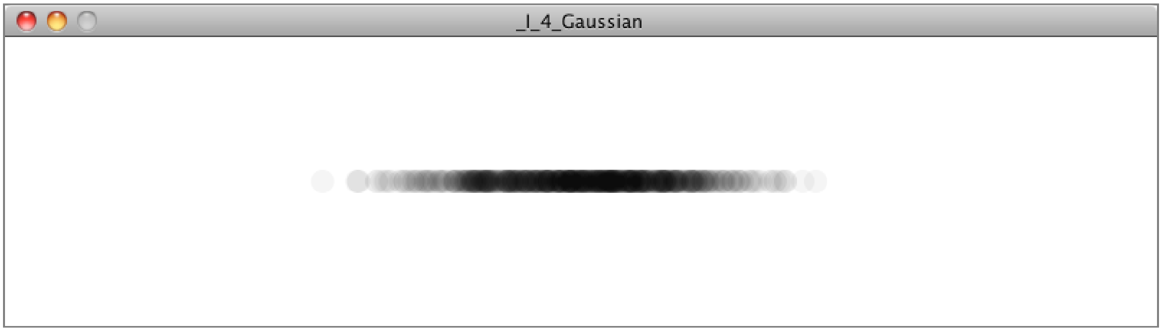
```
    generator = new Random();
  }
```

If we want to produce a random number with a normal (or Gaussian) distribution each time we run through **`draw()`**, it’s as easy as calling the function **`nextGaussian()`**.

```
  void draw() {
  -----------------------------------------------------------------------------------------
    float num = generator.nextGaussian();    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?

The **`nextGaussian()`** 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 we want a mean of 360 (the center horizontal pixel in a window of width 640) and a standard deviation of 60 pixels. We can adjust the value to our parameters by multiplying it by the standard deviation and adding the mean.



Example I.4: Gaussian distribution

```
  void draw() {
  -----------------------------------------------------------------------------------------
    float num = (float)                      Note nextGaussian() returns a double.
  generator.nextGaussian();
    float sd = 60;
    float mean = 360;
  -----------------------------------------------------------------------------------------
    float x = sd * num + mean;               Multiply by standard deviation and add the mean.
    noStroke();
    fill(255,10);
    ellipse(x,180,16,16);
  }
```

By drawing the ellipses on top of each other with some transparency, we can actually see the distribution. The brightest 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 location, but some dots do splatter out towards the edges. Can you use a normal distribution of random numbers to generate the locations of the dots? Can you also use a normal distribution of random numbers to generate a palette of color?

Exercise I.5

A Gaussian random walk is defined as one in which the step size (how far you move in a given direction) is generated with a normal distribution. Implement this variation of our 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 locations 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 location 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, we 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, we saw that we 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()`**. Certainly, we could implement a Levy flight by saying there is a 1% chance of the walker taking a large step.

```
float r = random(1);
if (r < 0.01) {
  xstep = random(-100,100);
  ystep - random(-100,100);
} else {
  xstep = random(-1,1);
  ystep - random(-1,1);
}
```

A 1% chance of taking a large step

However, this reduces the probabilities to a fixed number of options. What if we 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, we could map the likelihood on the y-axis with y = x.

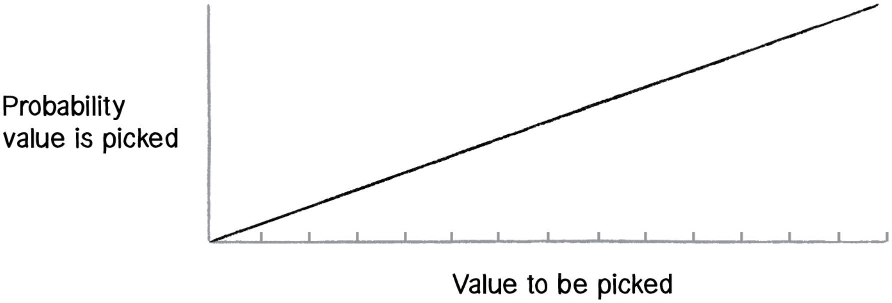


Figure I.4

￼ 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 Monte Carlo method, which was named for the Monte Carlo casino) that implements the above algorithm, returning a random value between zero and one.

```
float montecarlo() {
  while (true) {                   We do this “forever” until we find a qualifying random value.
    float r1 = random(1);          Pick a random value.
    float probability = r1;        Assign a probability.
    float 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 achieved by affecting the range of values picked. Can you map the probability exponentially—i.e. making the likelihood a value is picked equal to the value squared?

```
float stepsize = random(-10,10); A uniform distribution of step sizes. Change this!

float stepx = random(-stepsize,stepsize);
float stepy = random(-stepsize,stepsize);

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

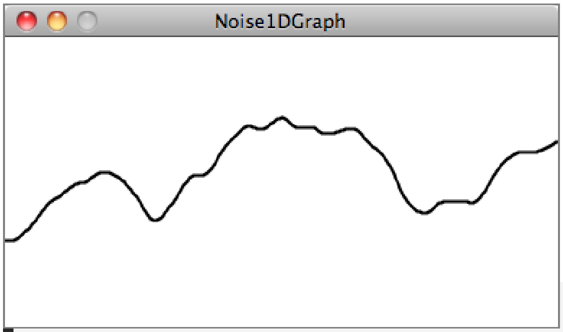
(Later we'll see how to do this more efficiently using vectors.)

I.6 Perlin Noise (A Smoother Approach)

One of the qualities of a good random number generator is that the numbers produced have no relationship. If they exhibit no discernible pattern, they are considered *random*.

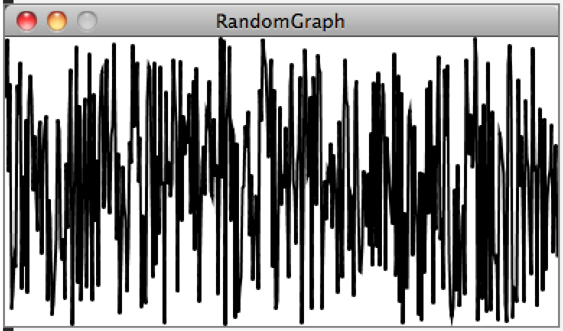
As we are beginning to see, a little bit of randomness can be a good thing when programming organic, lifelike behaviors. However, randomness as the single guiding principle is not necessarily natural. An algorithm known as "Perlin noise," named for its inventor Ken Perlin, takes this concept into account. 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 a variety of interesting effects such as clouds, landscapes, and patterned textures like marble.

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



Noise

Processing has a built-in implementation of the Perlin noise algorithm with the function **noise()**. The **noise()** function takes one, two, or three arguments (referring to the "space" in which noise is computed: one, two, or three dimensions). Let's start by looking at one-dimensional noise.



Random

Noise Detail

If you visit the Processing.org noise reference, you'll find that noise is calculated over several "octaves." You can change the number of octaves and their relative importance by calling the noiseDetail() (See page 0) function. This in turn changes how the noise function behaves.

You can learn more about how noise works from Ken Perlin (See page 0) himself.

Consider for a moment drawing a circle in our Processing window at a random x-location.

```
float x = random(0,width);            A random x-location
ellipse(x,180,16,16);
```

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

```
float x = noise(0,width); //                A noise x-location?
[line-through]
```

While conceptually this is exactly what we want to do—calculate an x-value that ranges between zero 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 zero and one. We'll see in a moment that we can get around this easily with Processing's **`map()`** function, but first we must examine what exactly **`noise()`** expects us to pass in as an argument.

We can think of one-dimensional Perlin noise as a linear sequence of values over time. For example:

noise table

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 in Processing, we have to pass a specific moment in time to the **`noise()`** function. For example:

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

According to the above table, **`noise(3)`** will return 0.364 at time equals three. We could improve this by using a variable for “time” and asking for a noise value continuously in **`draw()`**.

```
float t = 3;

void draw() {
  float n = noise(t);
  println(n);
}
```

We need the noise value for a specific "moment in time."

The above code results in the same value printed over and over. This is because we are asking for the result of the **`noise()`** function at the same point in “time”—3—over and over. If we increment the “time” variable **`t`**, however, we’ll get a different result.

```
float t = 0;

void draw() {
  float n = noise(t);
  println(n);
  t += 0.01;
}
```

Typically we would start at time = 0, though this is arbitrary.

Now, we move forward in time!

How quickly we increment “t” also affects the smoothness of the noise. If we make large jumps in time, then we are skipping ahead and the values will be more random.

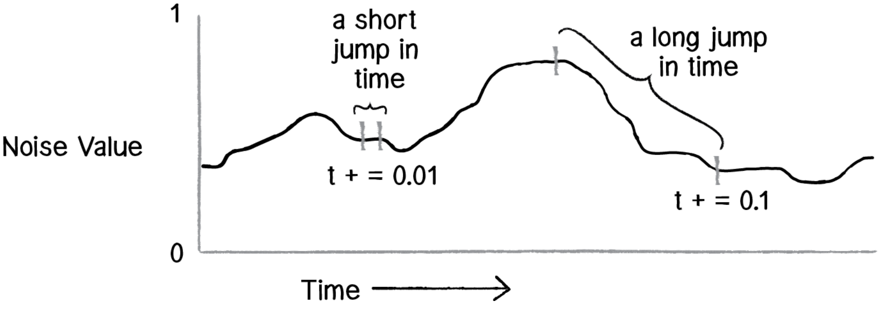
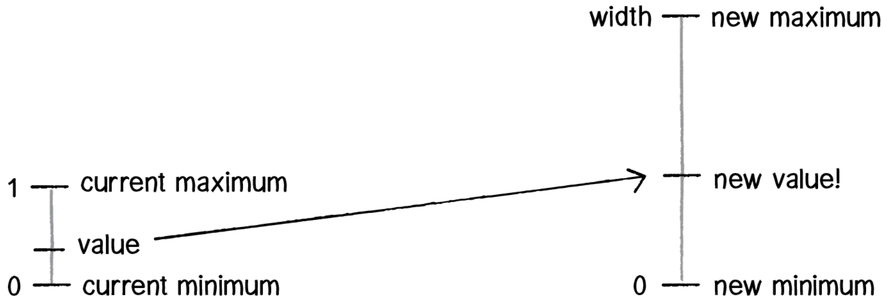


Figure I.7

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 we're ready to answer the question of what to do with the noise value. Once we have the value with a range between zero and one, it's up to us to map that range to what we want. The easiest way to do this is with Processing's **`map()`** function. The **`map()`** function takes five arguments. First up is the value we want to map, in this case **`n`**. Then we have to give it the value's current range (minimum and maximum) followed by our desired range.



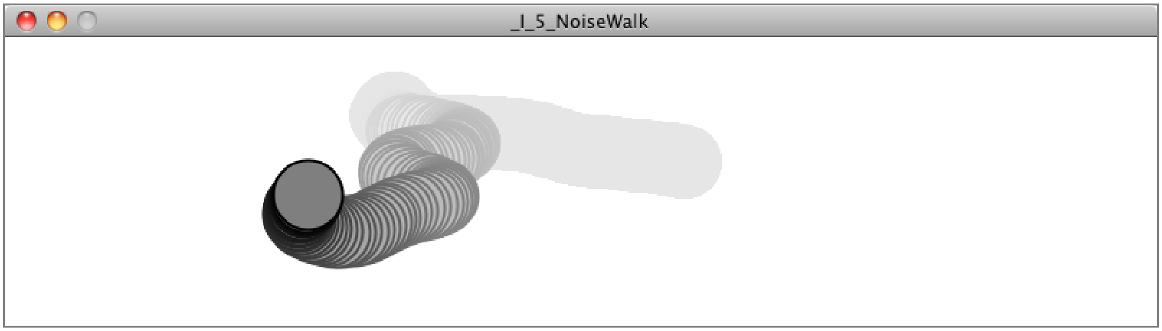
new value= map(value, current minimum, current maximum, new minimum, new maximum

Figure I.8

In this case, we know that noise has a range between zero and one, but we'd like to draw our circle with a range between zero and the window's width.

```
float t = 0;

void draw() {
  float n = noise(t);
-----------------------------------------------------------------------------------------------
  float x = map(n,0,1,0,width);          Using map() to customize the range of Perlin noise
  ellipse(x,180,16,16);

  t += 0.01;
}
```

We can apply the exact same logic to our random walker, and assign both its x- and y-values according to Perlin noise.



Example I.5: Perlin Noise Walker

```
class Walker {
  float x,y;

  float tx,ty;

  Walker() {
    tx = 0;
    ty = 10000;
  }

  void step() {
    x = map(noise(tx), 0, 1, 0, width);
    y = map(noise(ty), 0, 1, 0, height);
    tx += 0.01;
    ty += 0.01;
  }
}
```

x- and y-location mapped from noise

Move forward through “time.”

Notice how the above example requires an additional pair of variables: **tx** and **ty**. This is because we need to keep track of two “time” variables, one for the x-location of the walker and one for the y-location. But there is something a bit odd about these variables. Why does **tx** start at zero and **ty** at 10,000? While these numbers are arbitrary choices, we have very specifically initialized our two time variables with different values. This is because the noise function is deterministic; it gives you the same result for a specific time **t** each and every time. If we asked for the 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 would only move along a diagonal. Instead, we simply use two different parts of the noise space, starting at 0 for **`x`** and 10,000 for **`y`** so that **`x`** and **`y`** can appear to act independently of each other.

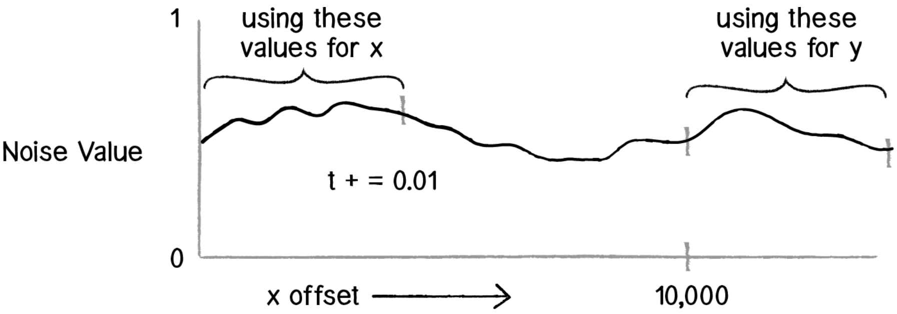


Figure I.9

In truth, there is no true 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 we can ask for a value at a specific x-location whenever we want. 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 location. 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 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 neighbor. Because the value is in one dimension, it only has two neighbors: a value that comes before it (to the left on the graph) and one that comes after it (to the right).

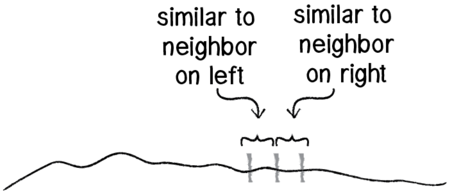


Figure I.10: 1D Noise

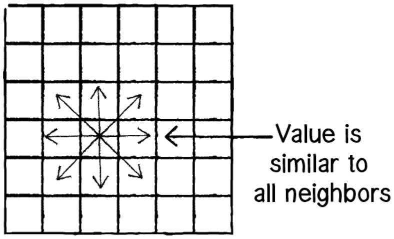
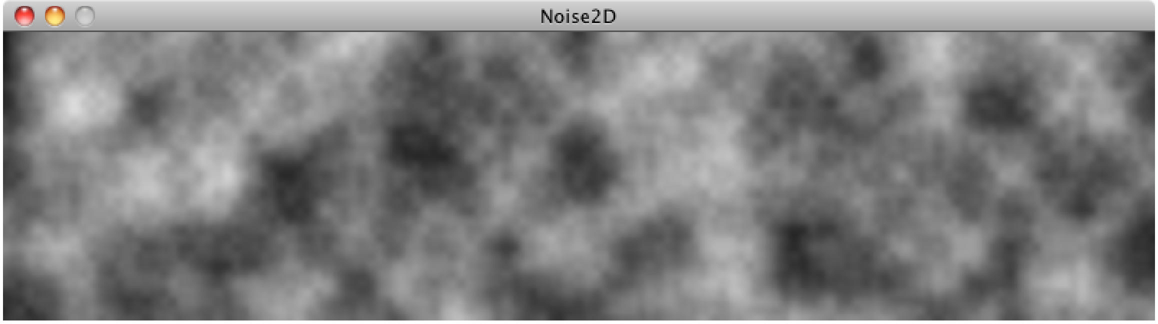


Figure I.11: 2D Noise

Two-dimensional noise works exactly the same way conceptually. The difference of course is that 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 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, we would get something that looks like clouds. White sits next to light gray, which sits next to gray, which sits next to dark grey, which sits next to black, which sits next to dark gray, etc.



This is what noise was originally invented for. Tweak the parameters a bit, play with color, and the resulting image might look more like marble or wood or any other organic texture.

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

```
loadPixels();
for (int x = 0; x < width; x++) {
  for (int y = 0; y < height; y++) {
    float bright = random(255);             A random brightness!
    pixels[x+y*width] = color(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()`**.

```
   float bright =
map(noise(x,y),0,1,0,255); // [bold]
```

A perlin noise brightness!

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.

Example I.6: 2D Perlin Noise

```
float xoff = 0.0; // [bold]

for (int x = 0; x < width; x++) {
```

Start xoff at 0.

```
  float yoff = 0.0; // [bold]

  for (int y = 0; y < height; y++) {
```

For every xoff, start yoff at 0.

```
    float bright =
map(noise(xoff,yoff),0,1,0,255);     // [bold]
```

Use xoff and yoff for noise().

```
    pixels[x+y*width] = color(bright);
```

Use x and y for pixel location.

```
    yoff += 0.01; // [bold]
  }
```

Increment yoff

```
  xoff += 0.01;  // [bold]
}
```

Increment xoff

Exercise I.8

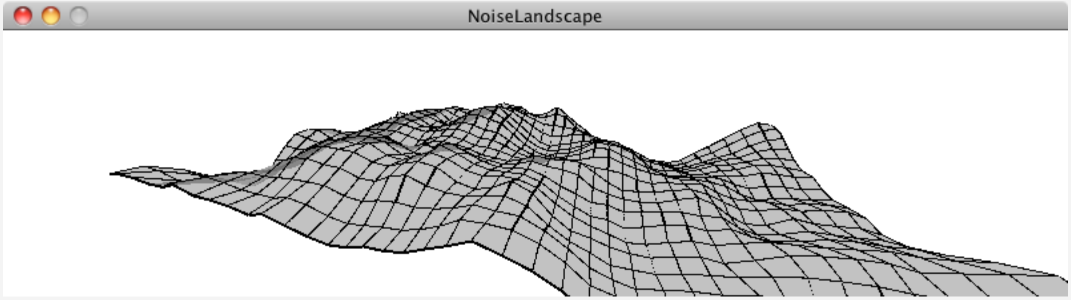
Play with color, **`noiseDetail()`**, and the rate at which **`xoff`** and **`yoff`** are incremented to achieve different visual effects.

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 heights of a landscape. See the screenshot below as a reference.



We've examined several traditional uses of Perlin noise in this section. 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 color. Any example in this book 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 the speed and direction of objects moving along a grid in a flow field simulation.

Figure I.12: Tree with Perlin noise

Figure I.12: Tree with Perlin noise

Figure I.13: Flow field with Perlin noise

Figure I.13: Flow field with Perlin noise

I.7 Onward

We 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, can also be a lazy one.

As we finish off the introduction, 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 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 should not use randomness. Or that you should or should not 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.

Chapter 1. Vectors

> *Roger, Roger. What’s our vector, Victor?*
>
> Captain Oveur (Airplane)

This book is all about looking at the world around us and coming up with clever ways to simulate that world with code. Divided into three parts, the book will start by looking at basic physics—how an apple falls from a tree, a pendulum swings in the air, the earth revolves around the sun, etc. Absolutely everything contained within the first five chapters of this book requires the use of the most basic building block for programming motion—the ***vector***. And so this is where we begin our story.

Now, the word ***vector*** can mean a lot of different things. Vector is the name of a new wave rock band formed in Sacramento, CA in the early 1980s. It’s the name of a breakfast cereal manufactured by Kellogg’s Canada. In the field of epidemiology, a vector is used to describe an organism that transmits infection from one host to another. In the C++ programming language, a Vector (std::vector) is an implementation of a dynamically resizable array data structure. While all these definitions are interesting, they're not what we are looking for. What we want is called a **Euclidean vector** (named for the Greek mathematician Euclid and also known as a geometric vector). When you see the term "vector" in this book, you can assume it refers to a Euclidean vector defined as:

> A vector is an entity that has both magnitude and direction

A vector is typically drawn as a arrow; the direction is indicated by where the arrow is pointing, and the magnitude by the length of the arrow itself.

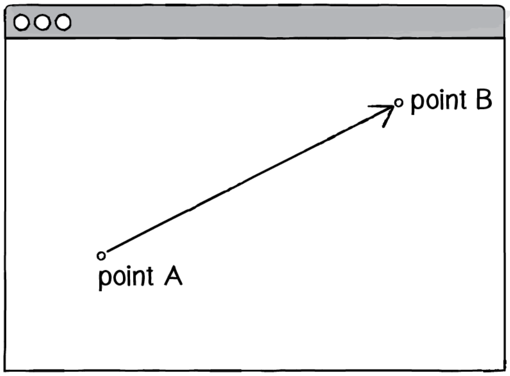
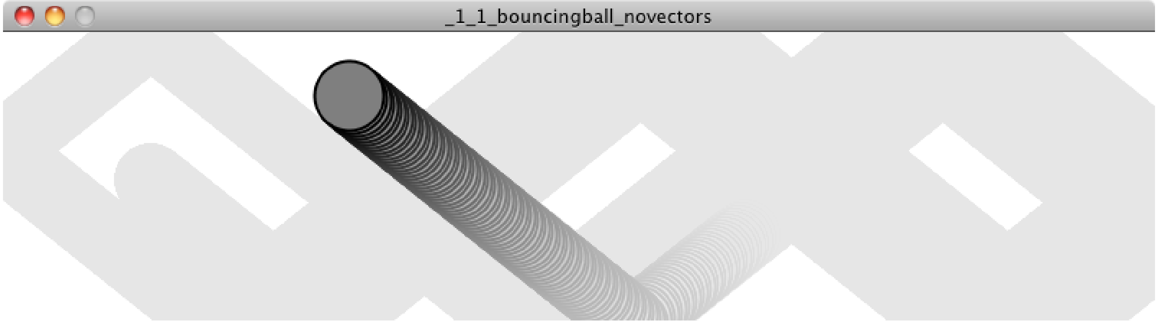


Figure 1.1: A Vector (draw as an arrow) has magnitude (length of arrow) and direction (which way it is pointing)

In the Figure 1.1, the vector is drawn as an arrow from point A to point B and serves as an instruction for how to travel from A to B.

1.1 Vectors, You Complete Me

Before we dive into more of the details about vectors, let's look at a basic Processing example that demonstrates why we should care about vectors in the first place. If you've read any of the introductory Processing textbooks or taken a class on programming with Processing (and hopefully you've done one of these things to help prepare you for this book), you probably, at one point or another, learned to how to write a simple bouncing ball sketch.



If you are reading this book as a PDF or in print, then you will only see screenshots of the code. Motion, of course, is a key element of our discussion, so to the extent possible, the static screenshots will include trails to give a sense of the behavior. For more about how to draw trails, see the code examples available for download.

Example 1.1: Bouncing ball with no vectors

```
float x = 100;
float y = 100;
float xspeed = 1;
float yspeed = 3.3;
```

Variables for location and speed of ball.

```
void setup() {
  size(200,200);
  smooth();
  background(255);
}

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

Remember how Processing works? setup() is executed once when the sketch starts and draw() loops forever and ever (until you quit).

```
  x = x + xspeed;
  y = y + yspeed;
```

Move the ball according to its speed.

```
  if ((x > width) || (x < 0)) {
    xspeed = xspeed * -1;
  }
  if ((y > height) || (y < 0)) {
    yspeed = yspeed * -1;
  }
```

Check for bouncing

```
  stroke(0);
  fill(175);
  ellipse(x,y,16,16);                    Display the ball at the location x,y.
}
```

In the above example, we have a very simple world—a blank canvas with a circular shape (a "ball") traveling around. This ball has some properties, which are represented in the code as variables.

> Location: *x and y*
> Speed: *xspeed and yspeed*

In a more advanced sketch, we could imagine having many more variables:

> Acceleration: *xacceleration and yacceleration*
> Target location: *xtarget and ytarget*
> Wind: *xwind and ywind*
> Friction: *xfriction and yfriction*

It's becoming more and more clear that for every concept in this world (wind, location, acceleration, etc.), we need two variables. And this is only a two-dimensional world. In a 3D world, we'll need **`x`**, **`y`**, **`z`**, **`xspeed`**, **`yspeed`**, **`zspeed`**, and so on.

Wouldn't it be nice if we could simplify our code and use fewer variables?

Instead of:

```
float x;
float y;
float xspeed;
float yspeed;
```

Wouldn't it be nice to have. . .

```
Vector location;
Vector speed;
```

Taking this first step in using vectors won't allow us to do anything new. Just adding vectors won't magically make your Processing sketches simulate physics; however, they will simplify your code and provide a set of functions for common mathematical operations that happen over and over and over again while programming motion.

As an introduction to vectors, we're going to live in two dimensions for quite some time (at least until we get through the first several chapters.) All of these examples can be fairly easily extended to three dimensions (and the class we will use — **`PVector`** — allows for three dimensions.) However, for the time being, it's easier to start with just two.

1.2 Vectors: What are they to us Processing programmers?

One way to think of a vector is the difference between two points. Consider how you might go about providing instructions to walk from one point to another.

Here are some vectors and possible translations:

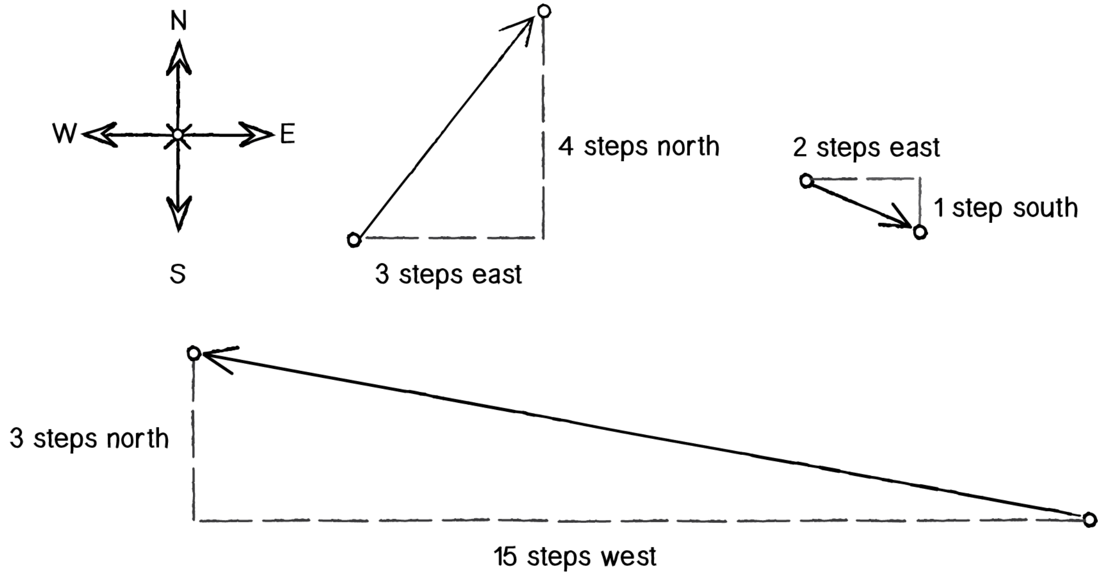


Figure 1.2

> (3, 4) —> Walk three steps east, turn and walk five steps north.
> (2,-1) —> Walk two steps east, turn and walk one step south.
> (-15, 3) —> Walk fifteen steps west, turn and walk three steps north.

You've probably done this before when programming motion. For every frame of animation (i.e. a single cycle through Processing's **`draw()`** loop), you instruct each object on the screen to move a certain number of pixels horizontally and a certain number of pixels vertically.

For every frame:

new location = velocity applied to current location

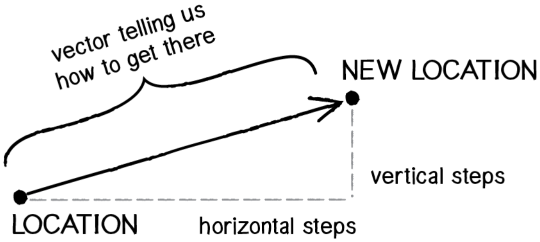


Figure 1.3

If velocity is a vector (the difference between two points), what is location? Is it a vector too? Technically, one might argue that location is not a vector, since it's not describing how to move from one point to another—it's simply describing a singular point in space. And so conceptually, we think of a location as different.

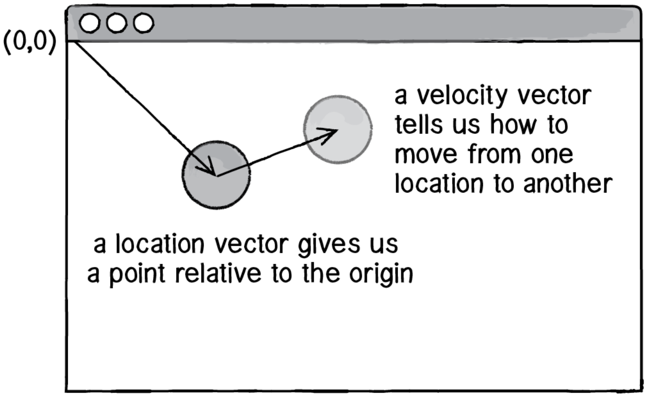

(0,0)
a velocity vector tells us how to move from one location to another
a location vector gives us a point relative to the origin

Figure 1.4

Nevertheless, another way to describe a location is the path taken from the origin to reach that location. Hence, a location can be the vector representing the difference between location and origin.

Let's examine the underlying data for both location and velocity. In the bouncing ball example we had the following:

> location —> **`x,y`**
> velocity —> **`xspeed,yspeed`**

Notice how we are *storing the same data for both* — two floating point numbers, an x and a y. If we were to write a vector class ourselves, we'd start with something rather basic:

```
class PVector {

  float x;
  float y;

  PVector(float x_, float y_) {
    x = x_;
    y = y_;
  }

}
```

At its core, a **PVector** is just a convenient way to store two values (or three, as we'll see in 3D examples.).

And so this ...

```
float x = 100;
float y = 100;
float xspeed = 1;
float yspeed = 3.3;
```

... becomes ...

```
PVector location = new PVector(100,100);
PVector velocity = new PVector(1,3.3);
```

Now that we have two vector objects ("location" and "velocity"), we're ready to implement the algorithm for motion—location = location + velocity. In Example 1.1, without vectors, we had:

```
x = x + xspeed;                      Add each speed to each location.
y = y + yspeed;
```

In an ideal world, we would be able to rewrite the above as:

```
location = location + velocity;          Add the velocity vector to the location vector.
```

However, in Processing, the addition operator ‘+’ is reserved for primitive values (integers, floats, etc.) only. Processing doesn’t know how to add two `PVector` objects together any more than it knows how to add two `PFont` objects or `PImage` objects. Fortunately for us, the `PVector` class includes functions for common mathematical operations.

1.3 Vector Addition

Before we continue looking at the `PVector` class and its `add()` method (purely for the sake of learning since it’s already implemented for us in Processing itself), let’s examine vector addition using the notation found in math and physics textbooks

Vectors are typically written either in boldface type or with an arrow on top. For the purposes of this book, to distinguish a ***vector*** from a **scalar** (scalar refers to a single value, such as an integer or a floating point number), we’ll use the arrow notation:

- Vector: u
- Scalar: **x**

Let’s say I have the following two vectors:

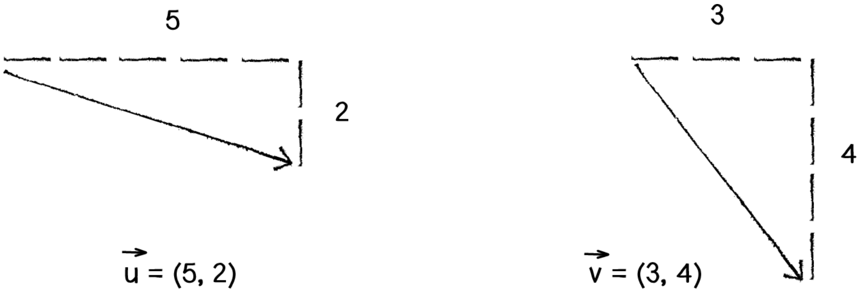


Figure 1.5

Each vector has two components, an x and a y. To add two vectors together we simply add both x’s and both y’s.

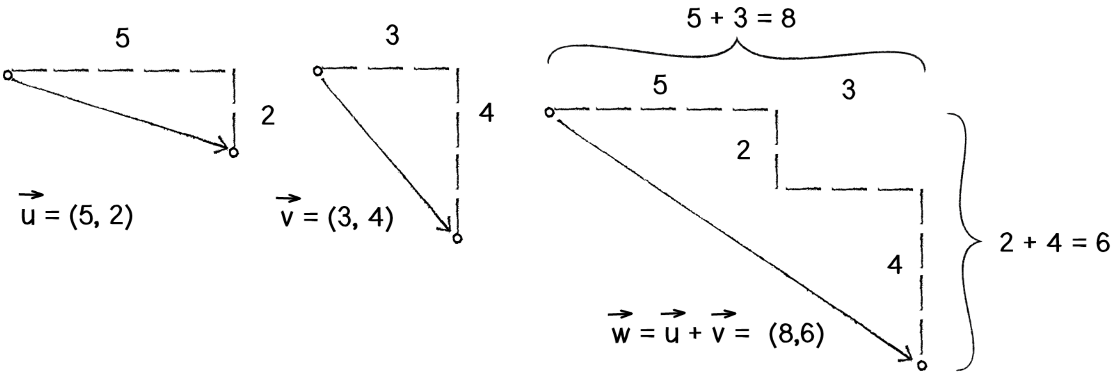


Figure 1.6

In other words:

$w = u + v$

translates to:

$w_x = u_x + v_x$

$w_y = u_y + v_y$

and therefore:

$w_x = 5 + 3$

$w_y = 2 + 4$

and therefore:

$w = (8,6)$

[Math Notation]

Now that we understand how to add two vectors together, we can look at how addition is implemented in the **PVector** class itself. Let's write a function called **add()** that takes as its argument another **PVector** object.

```
class PVector {

  float x;
  float y;
  PVector(float x_, float y_) {
    x = x_;
    y = y_;
  }
```

```
  void add(PVector v) { // [bold]
    y = y + v.y; // [bold]
    x = x + v.x; // [bold]
  } // [bold]
}
```

New! A function to add another PVector to this PVector. Simply add the x components and the y components together.

Now that we see how **`add()`** is written inside of **`PVector`**, we can return to the ***location*** + ***velocity*** algorithm with our bouncing ball example and implement vector addition:

```
location = location + velocity; //
[line-through]
location.add(velocity);
```

Add the current velocity to the location.

And here we are, ready to rewrite the bouncing ball example using **`PVector`**.

Example 1.2: Bouncing ball with PVector!

```
PVector location; // [bold]
PVector velocity; // [bold]

void setup() {
  size(200,200);
  smooth();
  location = new PVector(100,100); // [bold]
  velocity = new PVector(2.5,5); // [bold]
}

void draw() {
  background(255);

  location.add(velocity); // [bold]
```

Instead of a bunch of floats, we now just have two PVector variables.

```
  if ((location.x > width) ||
(location.x < 0)) { // [bold]
    velocity.x = velocity.x * -1; // [bold]
```

We still sometimes need to refer to the individual components of a PVector and can do so using the dot syntax: location.x, velocity.y, etc.

```
    } // [bold]
    if ((location.y > height) || (location.y < 0)) { // [bold]
      velocity.y = velocity.y * -1; // [bold]
    } // [bold]

    stroke(0);
    fill(175);
    ellipse(location.x,location.y,16,16); // [bold]
  }
```

Now, you might feel somewhat disappointed. After all, this may initially appear to have made the code more complicated than the original version. While this is a perfectly reasonable and valid critique, it's important to understand that we haven't fully realized the power of programming with vectors just yet. Looking at a simple bouncing ball and only implementing vector addition is just the first step. As we move forward into a more complex world of multiple objects and multiple ***forces*** (Chapter 2), the benefits of **`PVector`** will become more apparent.

We should, however, make note of an important aspect of the above transition to programming with vectors. Even though we are using **`PVector`** objects to describe two values—the x and y of location and the x and y of velocity—we still often need to refer to the x and y components of each **`PVector`** individually. When we go to draw an object in Processing, there's no means for us to say:

```
  ellipse(location,16,16); // [line-through]
```

The **`ellipse()`** function does not allow for a **`PVector`** as an argument. An ellipse can only be drawn with two scalar values, an x coordinate and a y coordinate. And so we must dig into the **`PVector`** object and pull out the x and y components using object-oriented dot syntax.

```
  ellipse(location.x,location.y,16,16);
```

The same issue arises when testing if the circle has reached the edge of the window, and we need to access the individual components of both vectors: location and velocity.

```
  if ((location.x > width) || (location.x < 0)) {
    velocity.x = velocity.x * -1;
  }
```

Exercise 1.1

Find something you've previously made in Processing using separate **`x`** and **`y`** variables and use PVectors instead.

Exercise 1.2

Take any of the walker examples from the introduction and convert it to use PVectors.

Exercise 1.3

Extend the bouncing ball with vectors example into 3D. Can you get a sphere to bounce around a box?

1.4 More Vector Math

Addition was really just the first step. There are many mathematical operations that are commonly used with vectors. Below is a comprehensive list of the operations available as functions in the **`PVector`** class. We'll go through a few of the key ones now. As our examples get more and more sophisticated in later chapters, we'll continue to reveal the details of more functions.

- **`add()`** — add vectors
- **`sub()`** — subtract vectors
- **`mult()`** — scale the vector with multiplication
- **`div()`** — scale the vector with division
- **`mag()`** — calculate the magnitude of a vector
- **`normalize()`** — normalize the vector to unit length of 1
- **`limit()`** — limit the magnitude of a vector
- **`heading2D()`** — the heading of a vector expressed as an angle
- **`rotate()`** — rotate a 2D vector by an angle

- **`dist()`** — the Euclidean distance between two vectors (considered as points)
- **`angleBetween()`** — find the angle between two vectors
- **`dot()`** — the dot product of two vectors
- **`cross()`** — the cross product of two vectors (only relevant in three dimensions)
- **CHECK WHAT NEW ONES I'M ADDING IN 2.0**

Having already covered addition, let's start with subtraction. This one's not so bad; just take the plus sign and replace it with a minus!

Vector subtraction:

$w=u - v$

translates to:

$\mathbf{w}_x = \mathbf{u}_x - \mathbf{v}_x$

$\mathbf{w}_y = \mathbf{u}_y - \mathbf{v}_y$

[Math Notation]

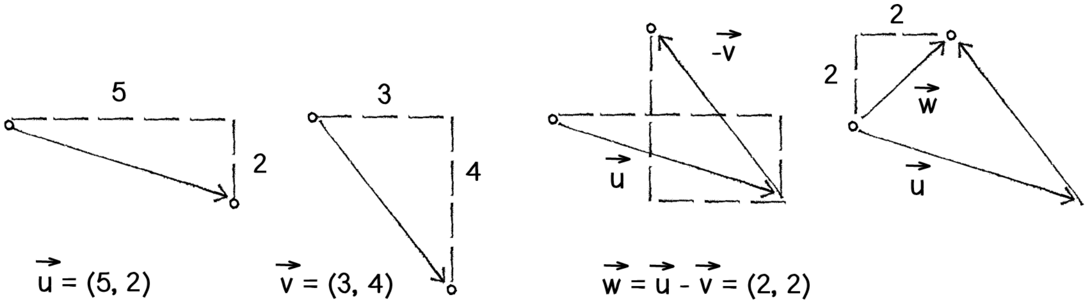
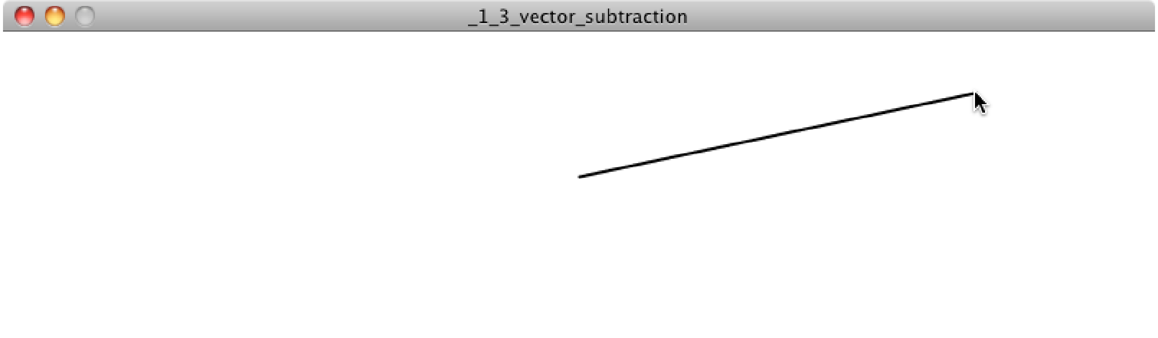


Figure 1.7: Vector Subtraction

and the function inside **`PVector`** therefore looks like:

```
void sub(PVector v) {
  x = x - v.x;
  y = y - v.y;
}
```

The following example demonstrates vector subtraction by taking the difference between two points—the mouse location and the center of the window.



Example 1.3: Vector subtraction

```
void setup() {
  size(200,200);
  smooth();
}

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

```
  PVector mouse  = new
PVector(mouseX,mouseY);
  PVector center = new PVector(width/2,height/2);
```

Two PVectors, one for the mouse location and one for the center of the window.

```
  mouse.sub(center);
```

PVector subtraction!

```
  translate(width/2,height/2);
  line(0,0,mouse.x,mouse.y);

}
```

Draw a line to represent the vector.

Basic Number Properties with Vectors

Both addition and subtraction with vectors follow the same algebraic rules as with real numbers.

The commutative rule:u + v*￼ = [vector]*v + u*￼+ **The associative rule:*
￼[vector]**u** + (v + w) = (￼[vector]**u** + v) + [vector]*w*￼

[Formatting for above is screwy Math notation?]

Fancy terminology and symbols aside, this is really quite a simple concept. We’re just saying that common sense properties of addition apply to vectors as well.

3 + 2 = 2 + 3
(3 + 2) + 1 = 3 + (2 + 1)

Moving on to multiplication, we have to think a little bit differently. When we talk about multiplying a vector, what we typically mean is ***scaling*** a vector. In the case that we want to scale a vector to twice its size or one-third of its size (leaving its direction the same), we would say: “Multiply the vector by 2” or “Multiply the vector by 1/3.” Note we are multiplying a vector by a scalar, a single number, not another vector.

To scale a vector, we multiply each component (x and y) by a scalar.

Vector multiplication:

w = u * n

translates to:

$w_x = u_x * n$
$w_y = u_y * n$

Let’s look at an example with vector notation.

u = (-3,7)
n = 3

w = u * n
$w_x = -3 * 3$ $w_y = 7 * 3$

w = (-9, 21)

[Math Notation]

The function inside the **PVector** class therefore is written as:

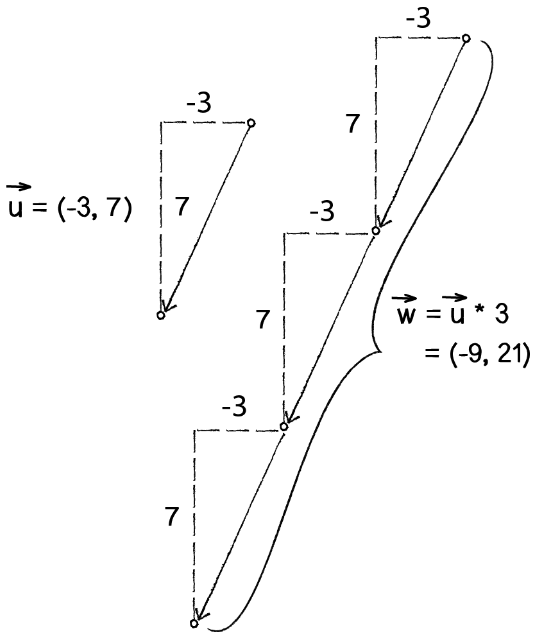


Figure 1.8: Scaling a vector.

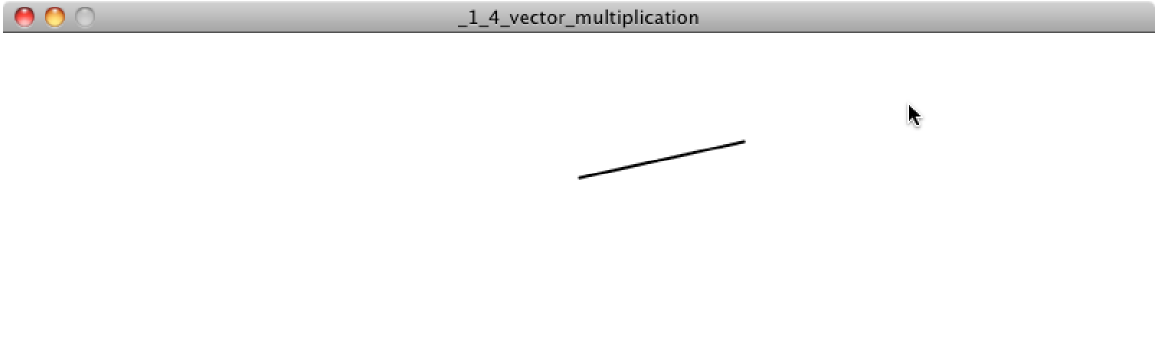
```
void mult(float n) {
   x = x * n;
   y = y * n;
 }
```

With multiplication, the components of the vector are multiplied by a number.

And implementing multiplication in code is as simple as:

```
PVector u = new PVector(-3,7);
u.mult(3);
```

This PVector is now three times the size and is equal to (-9,21).



Example 1.4: Multiplying a vector

```
void setup() {
  size(200,200);
  smooth();
}

void draw() {
  background(255);

  PVector mouse = new PVector(mouseX,mouseY);
  PVector center = new PVector(width/2,height/2);
  mouse.sub(center);
  mouse.mult(0.5);

  translate(width/2,height/2);
  line(0,0,mouse.x,mouse.y);

}
```

Multiplying a vector! The vector is now half its original size (multiplied by 0.5).

Division works just like multiplication—we simply replace the multiplication sign (asterisk) with the division one (forward slash).

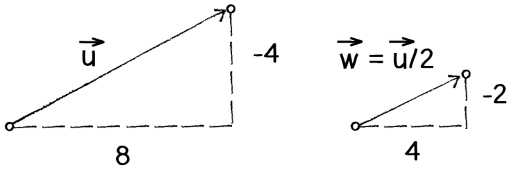


Figure 1.9

```
void div(float n) {
  x = x / n;
  y = y / n;
}

PVector u = new PVector(8,-4);
u.div(2);
```

More Number Properties with Vectors

As with addition, basic algebraic rules of multiplication and division apply to vectors.

The associative rule: (**n** * **m**) ￼**[vector]*v** = **n** * (**m** ￼**[vector]*v**)
The distributive rule with 2 scalars, 1 vector: (**n** * **m**) ￼**[vector]*v** = **n** * v + **m** ￼**[vector]*v**
The distributive rule with 2 vectors, 1 scalar: (u*￼+ [vector]*v) * **n** = **n** * u + **n** * [vector]*v*￼

[make this a table Math notation?]

1.5 Vector Magnitude

Multiplication and division, as we just saw, are means by which the length of the vector can be changed without affecting direction. Perhaps you're wondering: "OK, so how do I know what the length of a vector is? I know the components (**x** and **y**), but how long (in pixels) is the actual arrow?!"

Understanding how to calculate the length (also known as ***magnitude***) of a vector is incredibly useful and important.

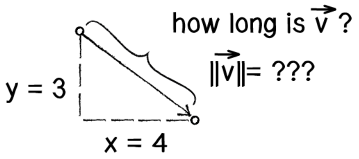


Figure 1.10: The length or "magnitude" of a vector [vector]*v* is often written as: II[vector]*v*II

Notice in the above diagram how the vector, drawn as an arrow and two components (**x** and **y**), creates a right triangle. The sides are the components and the hypotenuse is the arrow itself. We're very lucky to have this right triangle, because once upon a time, a Greek mathematician named Pythagoras developed a lovely formula to describe the relationship between the sides and hypotenuse of a right triangle.

The Pythagorean theorem: **a** squared plus **b** squared equals **c** squared.

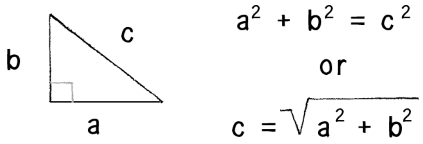


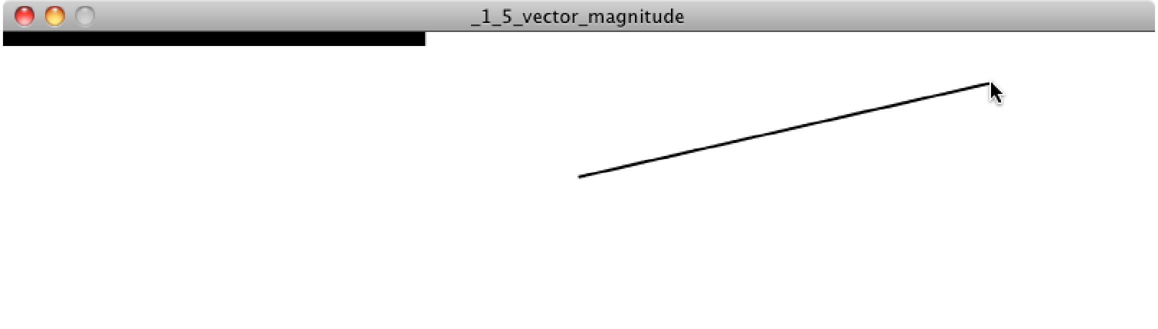
Figure 1.11: The Pythagorean Theorem

Armed with this formula, we can now compute the magnitude of v as follows:

$||\ v\ || = sqrt\ (v_x * v_x + v_y * v_y)$ **[Math notation?]**

or in **PVector**:

```
float mag() {
  return sqrt(x*x + y*y);
}
```



Example 1.5: Vector magnitude

```
void setup() {
  size(200,200);
  smooth();
}

void draw() {
  background(255);

  PVector mouse = new PVector(mouseX,mouseY);
  PVector center = new PVector(width/2,height/2);
  mouse.sub(center);
  float m = mouse.mag();
  fill(0);
  rect(0,0,m,10);

  translate(width/2,height/2);
  line(0,0,mouse.x,mouse.y);

}
```

The magnitude (i.e. length) of a vector can be accessed via the mag() function. Here it is used as the width of a rectangle drawn at the top of the window.

1.6 Normalizing Vectors

Calculating the magnitude of a vector is only the beginning. The magnitude function opens the door to many possibilities, the first of which is ***normalization***. Normalizing refers to the process of making something "standard" or, well, "normal." In the case of vectors, let's assume for the moment that a standard vector has a length of one. To normalize a vector, therefore, is to take a vector of any length and, keeping it pointing in the same direction, change its length to one, turning it into what is called a ***unit vector***.

The ability to quickly access the unit vector is useful since it describes a vector's direction without regard to length, and we'll see this come in handy once we start to work with forces in Chapter 2.

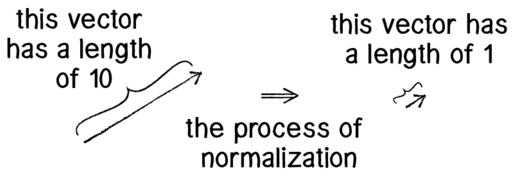

this vector has a length of 10
⟹
the process of normalization
this vector has a length of 1

Figure 1.12

For any given vector u, its unit vector (written as û) is calculated as follows:

û = u / ||u||
[Math notation?]

In other words, to normalize a vector, simply divide each component by its magnitude. This is pretty intuitive. Say a vector is of length 5. Well, 5 divided by 5 is 1. So looking at our right triangle, we then need to scale the hypotenuse down by dividing by 5. In that process the sides shrink, divided by 5 as well.

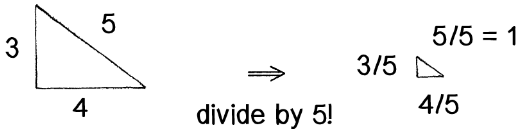


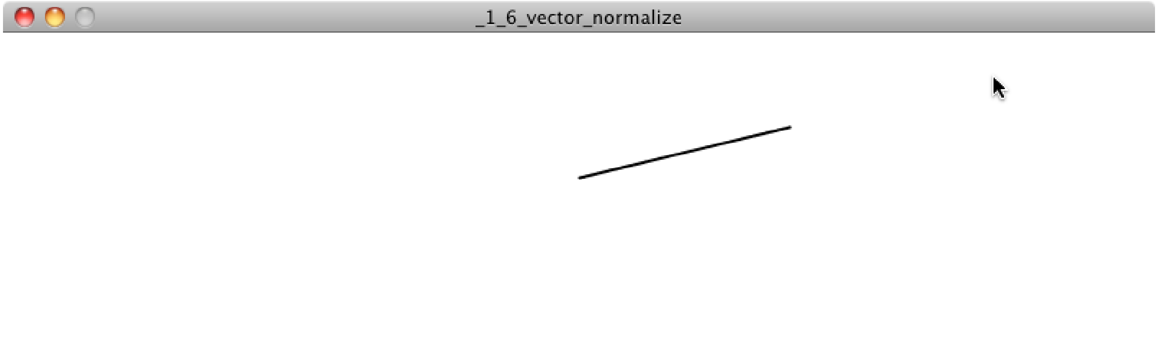
Figure 1.13

In the PVector class, we therefore write our normalization function as follows:

```
void normalize() {
  float m = mag();
  div(m);
}
```

Of course, there's one small issue. What if the magnitude of the vector is zero? We can't divide by zero! Some quick error checking will fix that right up:

```
void normalize() {
 float m = mag();
 if (m != 0) {
   div(m);
 }
}
```



Example 1.6: Normalizing a vector

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

  PVector mouse = new PVector(mouseX,mouseY);
  PVector center = new PVector(width/2,height/2);
  mouse.sub(center);
  mouse.normalize();
  mouse.mult(50);
  translate(width/2,height/2);
  line(0,0,mouse.x,mouse.y);
```

In this example, after the vector is normalized it is multiplied by 50 so that it is viewable onscreen. Note that no matter where the mouse is, the vector will have the same length (50) due to the normalization process.

```
}
```

1.7 Vector Motion: Velocity

Why should we care? Yes, all this vector math stuff sounds like something we should know about, but why exactly? How will it actually help us write code? The truth of the matter is that we need to have some patience. The awesomeness of using the **`PVector`** class will take some time to fully come to light. This is actually quite common when first learning a new data structure. For example, when you first learn about an array, it might seem like much more work to use an array than to just have several variables stand for multiple things. But that plan quickly breaks down when you need a hundred, or a thousand, or ten thousand things. The same can be true for **`PVector`**. What might seem like more work now will pay off later, and pay off quite nicely. And you don't have to wait too long, as your reward will come in the next chapter.

For now, however, we want to focus on simplicity. What does it mean to program motion using vectors? We've seen the beginning of this in Example 1.2 [REF]: the bouncing ball. An object on screen has a location (where it is at any given moment) as well as a velocity (instructions for how it should move from one moment to the next). Velocity is added to location:

```
location.add(velocity);
```

And then we draw the object at that location:

```
ellipse(location.x,location.y,16,16);
```

This is Motion 101.

1. ***Add velocity to location***
2. ***Draw object at location***

In the bouncing ball example, all of this code happened in Processing's main tab, within **`setup()`** and **`draw()`**. What we want to do now is move towards encapsulating all of the logic for motion inside of a ***class***. This way, we can create a foundation for programming moving objects in Processing. In ***[REF]*** Section section I.2 of the introduction, "The Random Walker Class," we briefly reviewed the basics of object-oriented-programming ("OOP"). Beyond that short introduction, this book assumes experience with objects and classes in Processing. If you need a refresher, I encourage you to check out the online OOP Processing tutorial: Processing objects tutorial (See page 0) .

In this case, we’re going to create a generic *Mover* class, a class to describe a thing moving around the screen. And so we must consider the following two questions:

1. ***What data does a Mover have?***
2. ***What functionality does a Mover have?***

Our “Motion 101” algorithm tells us the answers to these questions. A Mover object has two pieces of data: location and velocity, two PVector objects.

```
class Mover {

  PVector location;
  PVector velocity;
```

Its functionality is just about as simple. The Mover needs to move and it needs to be seen. We’ll implement these as functions named **`update()`** and **`display()`**. **`update()`** is where we’ll put all of our motion logic code and **`display()`** is where we will draw the object.

```
  void update() {
    location.add(velocity);
```

The Mover moves.

```
  }

  void display() {
    stroke(0);
    fill(175);
    ellipse(location.x,location.y,16,16);
```

The Mover is displayed.

```
  }

}
```

We’ve forgotten one crucial item, however: the object’s ***constructor***. The constructor is a special function inside of a class that creates the instance of the object itself. It is where you give instructions on how to set up the object. It always has the same name as the class and is called by invoking the ***new*** operator: “[function]*Mover m = new Mover();*”.

In our case, let's arbitrarily decide to initialize our mover object by giving it a random location and a random velocity.

```
Mover() {
  location = new PVector(random(width),random(height));
  velocity = new PVector(random(-2,2),random(-2,2));
}
```

If object-oriented programming is at all new to you, one aspect here may seem a bit confusing. After all, we spent the beginning of this chapter discussing the **`PVector`** class. The **`PVector`** class is the template for making the object "location" and the object "velocity". So what are they doing inside of yet another object, the Mover object? In fact, this is just about the most normal thing ever. An object is simply something that holds data (and functionality). That data can be numbers (integers, floats, etc.) or other objects! We'll see this over and over again in this book. For example, in Chapter 4 ***[REF]***, we'll write a class to describe a system of particles. That "ParticleSystem" object will have as its data a list of Particle objects. . .and each Particle object will have as its data several PVector objects!

Let's finish off the Mover class by incorporating a function to determine what the object should do when it reaches the edge of the window. For now let's do something simple, and just have it wrap around the edges.

```
void checkEdges() {
  if (location.x > width) {
    location.x = 0;
  } else if (location.x < 0) {
    location.x = width;
  }

  if (location.y > height) {
    location.y = 0;
  } else if (location.y < 0) {
    location.y = height;
  }

}
```

When it reaches one edge, set location to the other.

Now that the Mover class is finished, we can look at what we need to do in our main program. We first declare a Mover object:

```
Mover mover;
```

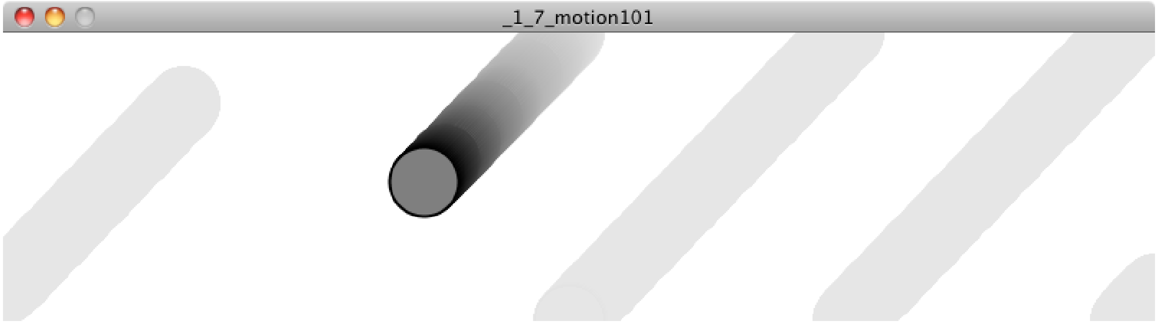
Then initialize the mover in **`setup()`**:

```
mover = new Mover();
```

and call the appropriate functions in **`draw()`**:

```
mover.update();
mover.checkEdges();
mover.display();
```

Here is the entire example for reference:



Example 1.7: Motion 101 (velocity)

```
Mover mover;
```

Declare Mover object.

```
void setup() {
  size(200,200);
  smooth();
```

```
  mover = new Mover();
}
```

Create Mover object.

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

Call functions on Mover object.

```
  mover.update();
  mover.checkEdges();
  mover.display();
}

class Mover {
```

Our object has two PVectors: location and velocity.

```
  PVector location;
  PVector velocity;

  Mover() {
    location = new PVector(random(width),random(height));
    velocity = new PVector(random(-2,2),random(-2,2));
  }

  void update() {
```

Motion 101: Location changes by velocity.

```
    location.add(velocity);
  }

  void display() {
    stroke(0);
    fill(175);
    ellipse(location.x,location.y,16,16);
  }

  void checkEdges() {
    if (location.x > width) {
      location.x = 0;
    } else if (location.x < 0) {
      location.x = width;
    }

    if (location.y > height) {
      location.y = 0;
    } else if (location.y < 0) {
      location.y = height;
    }
  }
}
```

1.8 Vector Motion: Acceleration

OK. At this point, we should feel comfortable with two things: (1) what a **`PVector`** is and (2) how we use ***PVectors*** inside of an object to keep track of its location and movement. This is an excellent first step and deserves a mild round of applause. Before standing ovations and screaming fans, however, we need to make one more, somewhat larger, step forward. After all, watching the Motion 101 example is fairly boring—the circle never speeds up, never slows down, and never turns. For more interesting motion, for motion that appears in the real world around us, we need to add one more **`PVector`** to our class—***acceleration***.

The strict definition of acceleration we're using here is: the rate of *change of velocity*. Let's think about that definition for a moment. Is this a new concept? Not really. Velocity is defined as *the rate of change of location*. In essence, we are developing a "trickle down" effect. Acceleration affects velocity, which in turn affects location (for some brief foreshadowing, this point will become even more crucial in the next chapter when we see how forces affect acceleration, which affects velocity, which affects location.) In code, this reads:

```
velocity.add(acceleration);
location.add(velocity);
```

As an exercise, from this point forward, let's make a rule for ourselves. Let's write every example in the rest of this book without ever touching the value of velocity and location (except to initialize them). In other words, our goal now for programming motion is as follows—come up with an algorithm for how we calculate acceleration and let the trickle-down effect work its magic. (In truth, you'll find reasons to break this rule, but it's important to illustrate the principles behind our motion algorithm.) And so we need to come up with some ways to calculate acceleration:

ACCELERATION ALGORITHMS!

1. A constant acceleration
2. A totally random acceleration
3. Acceleration towards the mouse

Algorithm #1, a constant acceleration, is not particularly interesting, but it is the simplest and will help us begin incorporating acceleration into our code. The first thing we need to do is add another **`PVector`** to the Mover class:

```
class Mover {

  PVector location;
  PVector velocity;
  PVector acceleration;                    A new PVector for acceleration
```

And incorporate acceleration into the **`update()`** function:

```
 void update() {
    velocity.add(acceleration);            Our motion algorithm is now two lines of code!
    location.add(velocity);
  }
```

We’re almost done. The only missing piece is initialization in the constructor.

```
  Mover() {
```

Let’s start the mover object in the middle of the window. . .

```
    location = new PVector(width/2,height/2);
```

with an initial velocity of zero.

```
    velocity = new PVector(0,0);
```

This means that when the sketch starts, the object is at rest. We don’t have to worry about velocity anymore as we are controlling the object’s motion entirely with acceleration. Speaking of which, according to Algorithm #1, our first sketch involves constant acceleration. So let’s pick a value.

```
    acceleration = new PVector(-0.001,0.01);
  }
```

Maybe you’re thinking, “Gosh, those values seem awfully small!” That’s right, they are quite tiny. It’s important to realize that our acceleration values (measured in pixels) accumulate over time in the velocity, about thirty times per second depending on our sketch’s frame rate.

And so to keep the magnitude of the velocity vector within a reasonable range, our acceleration values should remain quite small. We can also help this cause by incorporating the **`PVector`** function **`limit()`**.

```
velocity.limit(10);
```

The limit() function constrains the magnitude of a vector.

This translates to the following:

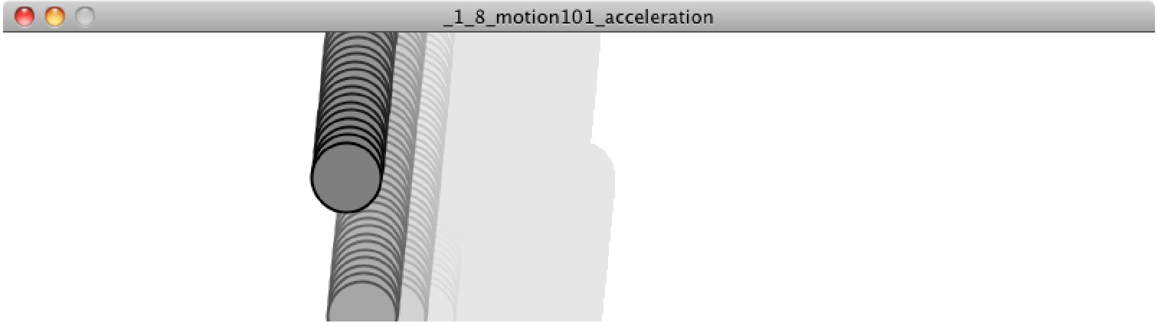
What is the magnitude of velocity? If it's less than 10, no worries; just leave it as is. If it's more than 10, however, reduce it to 10!

Exercise 1.4

Write the **`limit()`** function for the PVector class.

```
void limit(float max) {
  if (_______ > _______) {
    _________();
    ____(max);
  }
}
```

Let's take a look at the changes to the Mover class, complete with **`acceleration`** and **`limit()`**.



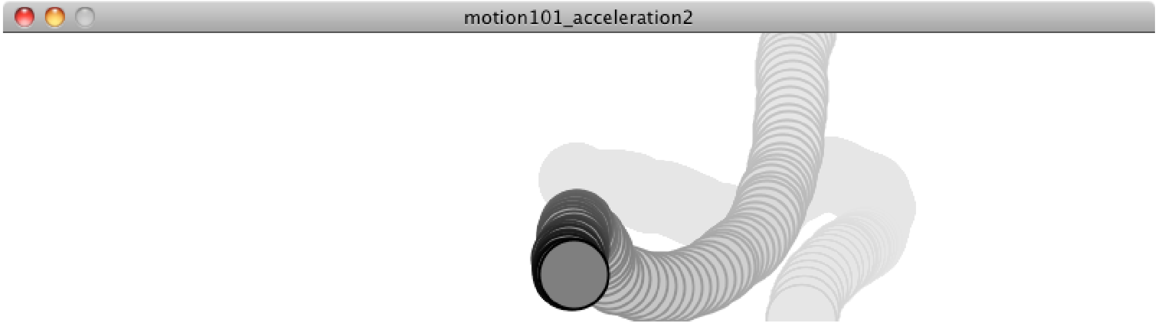
Example 1.8: Motion 101 (velocity and constant acceleration)

```
class Mover {

  PVector location;
  PVector velocity;
  PVector acceleration;                 Acceleration is the key!
  float topspeed;                       The variable, topspeed, will limit the magnitude of velocity.

  Mover() {
    location = new PVector(width/2,height/2);
    velocity = new PVector(0,0);
    acceleration = new PVector(-0.001,0.01);
    topspeed = 10;
  }

  void update() {
    velocity.add(acceleration);         Velocity changes by acceleration and is limited by topspeed.
    velocity.limit(topspeed);
    location.add(velocity);
  }
  void display() {}                     display() is the same
  void checkEdges() {}                  checkEdges() is the same
}
```

Exercise 1.4

Create a simulation of a car (or runner) that accelerates when you press the up key and brakes when you press the down key.

Now to Algorithm #2,“a totally random acceleration.” In this case, instead of initializing acceleration in the object’s constructor, we want to pick a new acceleration each cycle, i.e. each time **`update()`** is called.



Example 1.9: Motion 101 (velocity and random acceleration)

```
void update() {

  acceleration = new PVector(random(-1,1),random(-1,1));
  acceleration.normalize();

  velocity.add(acceleration);
  velocity.limit(topspeed);
  location.add(velocity);
}
```

While normalizing acceleration is not entirely necessary, it does prove useful, as it standardizes the magnitude of the vector, allowing us to try different things. Such as:

(a) scaling the acceleration to a constant value

```
acceleration = new PVector(random(-1,1),random(-1,1));
acceleration.normalize();
acceleration.mult(0.5); // [bold]
```

Constant

(b) scaling the acceleration to a random value

```
acceleration = new PVector(random(-1,1),random(-1,1));
acceleration.normalize();
acceleration.mult(random(2)); // [bold]
```

Random

While this may seem like an obvious point, it's crucial to understand that acceleration does not merely refer to the *speeding up* or *slowing down* of a moving object, but rather *any change* in velocity in either magnitude or direction. Acceleration is used to steer an object, and we'll see this again and again in future chapters as we begin to program objects that make decisions about how to move about the screen.

> **Exercise 1.5**
>
> Referring back to section I.6 of the Introduction **[REF]**, implement acceleration according to Perlin noise.

1.9 Static vs. Non-Static Functions

Before we get to acceleration Algorithm #3 (accelerate towards the mouse), we need to cover one more rather important aspect of working with vectors and the **`PVector`** class: the difference between using **static** methods and **non-static** methods.

Forgetting about vectors for a moment, take a look at the following code:

```
float x = 0;
float y = 5;

x = x + y;
```

Pretty simple, right? **`x`** has the value of 0, we add **`y`** to it, and now **`x`** is equal to 5. We could write the corresponding code pretty easily based on what we've learned about **`PVector`**.

```
PVector v = new PVector(0,0);
PVector u = new PVector(4,5);
v.add(u);
```

The vector **`v`** has the value of (0,0), we add **`u`** to it, and now **`v`** is equal to (4,5). Easy, right?

OK, let's take a look at another example of some simple floating point math:

```
float x = 0;
float y = 5;
float z = x + y;
```

x has the value of 0, we add **y** to it, and store the result in a new variable **z**. The value of **x** does not change in this example (neither does **y**)! This may seem like a trivial point, and one that is quite intuitive when it comes to mathematical operations with floats. However, it's not so obvious with mathematical operations in **PVector**). Let's try to write the code based on what we know so far.

```
PVector v = new PVector(0,0);
PVector u = new PVector(4,5);
PVector w = v.add(u); // [line-through]    Don't be fooled; this is incorrect!!!
```

The above might seem like a good guess, but it's just not the way the **PVector** class works. If we look at the definition of **add()** . . .

```
void add(PVector v) {
    x = x + v.x;
    y = y + v.y;
 }
```

we see that this code does not accomplish our goal. First, it does not return a new **PVector** (the return type is "void") and second, it changes the value of the **PVector** upon which it is called. In order to add two PVector objects together and return the result as a new PVector, we must use the **static add()** function.

Functions that we call from the class name itself (rather than from a specific object instance) are known as **static** functions. Here are two examples of function calls that assume two PVector objects, **v** and **u**:

```
PVector.add(v,u);                         Static: called from the class name.
v.add(u);                                 Not static: called from an object instance.
```

Since you can't write static functions yourself in Processing, you might not have encountered them before. **PVector** 's static functions allow us to perform generic mathematical operations on **PVector** objects without having to adjust the value of one of the input **PVectors**. Let's look at how we might write the static version of **add()**:

```
  static PVector add(PVector v1, PVector
v2) {
    PVector v3 = new PVector(v1.x + v2.x, v1.y + v2.y);
    return v3;
  }
```

The static version of add allows us to add two PVectors together and assign the result to a new PVector while leaving the original PVectors (v and u) intact.

There are several differences here:

- The function is labeled as ***static***.
- The function does not have a ***void*** return type, but rather returns a **`PVector`**.
- The function creates a new PVector (v3) and returns the sum of the components of **`v1`** and **`v2`** in that new PVector.

When you call a static function, instead of referencing an actual object instance, you simply reference the name of the class itself.

```
PVector v = new PVector(0,0);
PVector u = new PVector(4,5);
PVector w = v.add(u); // [line-through]
PVector w = PVector.add(v,u); // [bold]
```

The PVector class has static versions of **`add()`**, **`sub()`**, **`mult()`**, and **`div()`**.

Exercise 1.6

Translate the following pseudocode to code using static or non-static functions where appropriate.

- The PVector [var]*v equals (1,5)
- The PVector **u** equals **v** multiplied by 2.
- The PVector **w** equals **v** minus **u**.
- Divide the PVector w by 3.

```
PVector v = new PVector(1,5);
PVector u = ________._____(__,__);
PVector w = ________._____(__,__);
___________;
```

1.10 Interactivity with acceleration

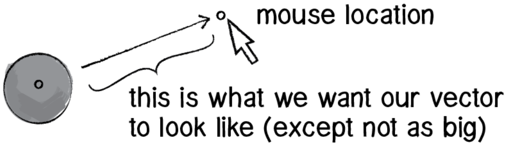

mouse location
this is what we want our vector to look like (except not as big)

Figure 1.14

To finish out this chapter, let's try something a bit more complex and a great deal more useful. We'll dynamically calculate an object's acceleration according to a rule, acceleration Algorithm #3 —"the object accelerates towards the mouse."

Anytime we want to calculate a vector based on a rule or a formula, we need to compute two things: ***magnitude*** and ***direction***. Let's start with direction. We know the acceleration vector should point from the object's location towards the mouse location. Let's say the object is located at the point (**`x`**,**`y`**) and the mouse at (**`mouseX`**,**`mouseY`**).

As illustrated in Figure 1.15, we see that we can get a vector (**dx**,**dy**) by subtracting the object's location from the mouse's location.

- **dx = mouseX - x**
- **dy = mouseY - y**

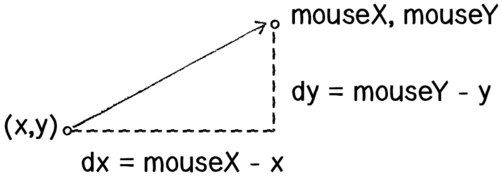


Figure 1.15

Let's rewrite the above using **PVector** syntax. Assuming we are in the Mover class and thus have access to the object's location **PVector**, we then have:

```
PVector mouse = new PVector(mouseX,mouseY);
PVector dir =
PVector.sub(mouse,location);
```

Look! We're using the static reference to sub() because we want a new PVector pointing from one point to another.

We now have a **PVector** that points from the mover's location all the way to the mouse. If the object were to actually accelerate using that vector, it would appear instantaneously at the mouse location. This does not make for good animation, of course, and what we want to do now is decide how quickly that object should accelerate toward the mouse.

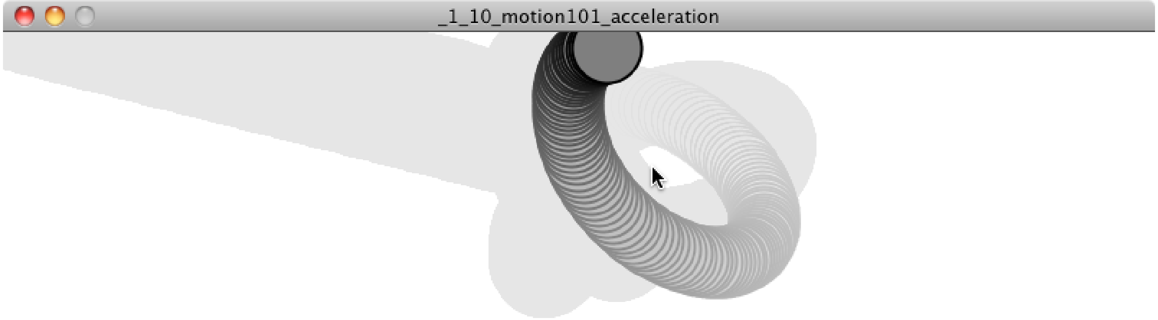
In order to set the magnitude (whatever it may be) of our acceleration PVector, we must first __ that direction vector. That's right, you said it. ***Normalize***. If we can shrink the vector down to its unit vector (of length one) then we have a vector that tells us the direction and can easily be scaled to any value. One multiplied by anything equals anything.

```
float anything = ?????
dir.normalize();
dir.mult(anything);
```

To summarize, we take the following steps:

1. Calculate a vector that points from the object to the target location (mouse).
2. Normalize that vector (reducing its length to 1)
3. Scale that vector to an appropriate value (by multiplying it by some value)
4. Assign that vector to acceleration

And here are those steps in the ***update()*** function itself:



Example 1.10: Accelerating towards mouse

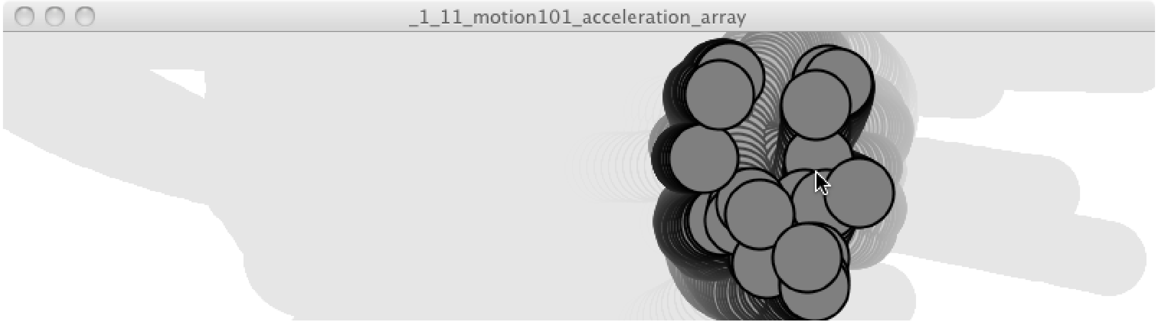
```
 void update() {

    PVector mouse = new PVector(mouseX,mouseY);
    PVector dir =                            Step 1: Direction
PVector.sub(mouse,location);
    dir.normalize();                         Step 2: Normalize
    dir.mult(0.5);                           Step 3: Scale
    acceleration = dir;                      Step 4: Accelerate

    velocity.add(acceleration);
    velocity.limit(topspeed);
    location.add(velocity);

  }
```

You may be wondering why the circle doesn't stop when it reaches the target. It's important to note that the object moving has no knowledge about trying to stop at a destination; it only knows where the destination is and tries to go there as quickly as possible. Going as quickly as possible means it will inevitably overshoot the location and have to turn around, again going as quickly as possible towards the destination, overshooting it again, and so on, and so forth. Stay tuned; in later chapters we'll learn how to program an object to "arrive" at a location (slow down on approach).

Exercise 1.7

This example is remarkably close to the concept of gravitational attraction (in which the object is attracted to the mouse location). Gravitational attraction will be covered in more detail in the next chapter. However, one thing missing here is that the strength of gravity (magnitude of acceleration) is inversely proportional to distance. This means that the closer the object is to the mouse, the faster it accelerates. Try implementing the above example with a variable magnitude of acceleration, stronger when it is either closer or farther away.

Let's see what this example would look like with an array of Mover objects (rather than just one).



Example 1.11: Array of Movers accelerating towards mouse

```
Mover[] movers = new Mover[20];                An array of objects

void setup() {
  size(200,200);
  smooth();
  background(255);
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new Mover();                   Initialize each object in the array.
  }
}

void draw() {
  background(255);
  for (int i = 0; i < movers.length; i++) {
    movers[i].update();
    movers[i].checkEdges();
    movers[i].display();
  }
}
```

Calling functions on all the objects in the array.

```
class Mover {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float topspeed;

  Mover() {
    location = new PVector(random(width),random(height));
    velocity = new PVector(0,0);
    topspeed = 4;
  }

  void update() {

    PVector mouse = new PVector(mouseX,mouseY);
    PVector dir = PVector.sub(mouse,location);
    dir.normalize();
    dir.mult(0.5);
    acceleration = dir;
```

Our algorithm for calculating acceleration: find vector pointing towards mouse normalize scale set to acceleration

```
    velocity.add(acceleration);
    velocity.limit(topspeed);
    location.add(velocity);
  }

  void display() {
    stroke(0);
    fill(175);
    ellipse(location.x,location.y,16,16);
  }

  void checkEdges() {
```

Motion 101! Velocity changes by acceleration. Location changes by velocity.

```
      if (location.x > width) {
        location.x = 0;
      } else if (location.x < 0) {
        location.x = width;
      }

      if (location.y > height) {
        location.y = 0;
      }  else if (location.y < 0) {
        location.y = height;
      }
    }
  }
```

Ecosystem Project:

As mentioned in the preface, one way to use this book is to build a single project over the course of reading it, incorporating elements from each chapter one step at a time. We'll follow the development of an example project throughout this book—an "ecosystem" simulation. Imagine a population of computational creatures swimming around a digital pond, interacting with each other according to various rules.

Step 1 Exercise:

Develop a set of rules for simulating the real-world behavior of a creature, such as a nervous fly, swimming fish, hopping bunny, slithering snake, etc. Can you control the object's motion by only manipulating the acceleration? Try to give the creature a personality through its behavior (rather than through its visual design.)

Chapter 2. Forces

> *"Don't underestimate the Force." —Darth Vader*

In the final example of Chapter 1, we saw how we could calculate a dynamic acceleration based on a vector pointing from a circle on the screen to the mouse location. The resulting motion resembled a magnetic attraction between circle and mouse, as if some *force* were pulling the circle in towards the mouse. In this chapter we will formalize our understanding of the concept of a ***force*** and its relationship to ***acceleration***. Our goal, by the end of this chapter, is to understand how to make multiple objects move around the screen and respond to a variety of environmental forces.

2.1 Forces and Newton's Laws of Motion

Before we begin examining the practical realities of simulating forces in code, let's take a conceptual look at what it means to be a ***force*** in the real world. Just as with the word "vector", "force" is often commonly used to mean a variety of things. It can be used to indicate a powerful intensity, as in "She pushed the boulder with great force" or "He spoke forcefully." The definition of force that we care about is much more formal and comes from Isaac Newton's laws of motion:

Force is a vector that causes an object with mass to accelerate.

The good news here is that we recognize the first part of the definition—"a force is a vector." Thank goodness we just spent a whole chapter learning what a vector is and how to program with PVectors!

Let's look at Newton's three laws in relation to the concept of a force.

Newton's First Law

Newton's first law is commonly stated as:

> *An object at rest stays at rest and an object in motion stays in motion.*

However, this is missing an important element related to forces and so we could expand it by stating:

> *An object at rest stays at rest and an object in motion stays in motion at a constant speed and direction unless acted upon by an unbalanced force.*

By the time Newton came along, the prevailing theory of motion—formulated by Aristotle—was nearly two thousand years old. It stated that if an object is moving, some sort of "force" is required to keep it moving. Unless that moving thing is being pushed or pulled, it will simply slow down or stop. Right?

This, of course, is not true. In the absence of any forces, no force is required to keep an object moving. An object (such as a ball) tossed in the earth's atmosphere slows down because of air resistance (a force). An object's velocity will only remain constant in the absence of any forces or if the forces that act on it *cancel each other out*, i.e. the net force adds up to zero. This is often referred to as ***equilibrium***. The falling ball will reach a terminal velocity (that stays constant) once the force of air resistance equals the force of gravity.

In our Processing world, we could restate Newton’s first law as follows:

> *An object’s PVector **velocity** will remain constant if it is in a state of equilibrium.*

Skipping Newton’s second law (arguably the most important law for our purposes) for a moment, let’s move on to the third law.

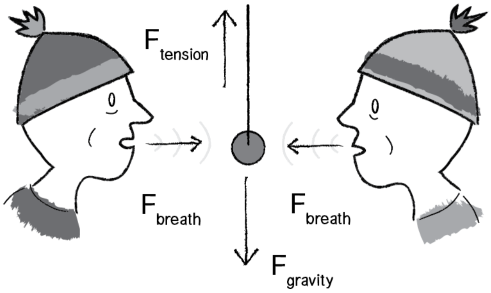


Figure 2.1: The pendulum doesn't move because all the forces cancel each other out (add up to a net force of zero.)

Newton’s Third Law

This law is often stated as:

> ***For every action there is an equal and opposite reaction.***

This law frequently causes some confusion in the way that it is stated. For one, it sounds like one force causes another. Yes, if you push someone, that someone may *actively* decide to push you back. But this is not the action and reaction we are talking about with Newton’s third law.

Let’s say you push against a wall. The wall doesn’t actively decide to push back on you. There is no “origin” force. Your push simply includes both forces, referred to as an “action/reaction pair.”

A better way of stating the law might be:

> *Forces always occur in pairs. The two forces are of equal strength, but in opposite directions.*

Now, this still causes confusion because it sounds like these forces would always cancel each other out. This is not the case. Remember, the forces act on different objects. And just because the two forces are equal, it doesn’t mean that the *movements* are equal (or that the objects will stop moving).

Try pushing on a stationary truck. Although the truck is far more powerful than you, a stationary truck will never overpower you and send you flying backwards. The force you exert on it is equal and opposite to the force exerted on your hand. The outcome depends on a variety of other factors. If the truck is a small truck on an icy downhill, you’ll probably be able to get it to move. On the other hand, if it’s a very large truck on a dirt road and you push hard enough (maybe even take a running start) you could injure your hand.

And if you are wearing roller skates when you push on that truck?

￼ **[REDO DIAGRAM AS TRUCK]**

You’ll accelerate away from the truck sliding along the road while the truck stays put. Why do you slide but not the truck? For one, the truck has a much larger mass (which we’ll get into with Newton’s second law) and there are other forces at work too, namely the friction of the truck’s tires and your roller skates against the road.

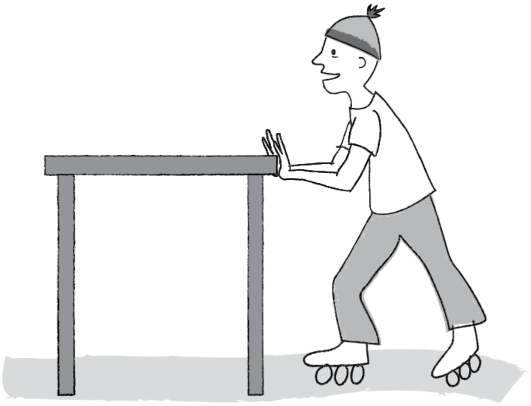


Figure 2.2

Newton’s Third Law as seen through the eyes of Processing

If we calculate a `PVector f` that is a force of object A on object B, we must also apply the force `f` (or `PVector.mult(f,-1);`) that B exerts on object A.

We’ll see that in the world of Processing programming we don’t always have to stay true to the above. Sometimes, such as in the case of gravitational attraction between bodies ([notetoself]see ex. 2.x on p.XX), we’ll want to model equal and opposite forces. Other times, such as when we’re simply saying, “Hey, there’s some wind in the environment,” we’re not going to bother to model the force that a body exerts back on the air. In fact, we’re not modeling the air at all! Remember, we are simply taking inspiration from the physics of the natural world and not simulating everything with perfect precision.

2.2 Forces and Processing—Newton’s Second Law as a Function

And here we are at the most important law for the Processing programmer.

Newton’s Second Law

F = **M** **[var]*A** **need to fix up all these formulas, also vector notation for them**

Force equals mass times acceleration.

Why is this the most important law for us? Well, let’s write this a different way.

A = **F** / **M**

Acceleration is directly proportional to force and inversely proportional to mass. This means that if you get pushed, the harder you are pushed, the faster you’ll move (accelerate). The bigger you are, the slower you’ll move.

> **Weight vs. Mass**
>
> - The **mass** of an object is a measure of the amount of matter in the object (measured in kilograms).
> - **Weight**, though often mistaken for mass, is technically the force of gravity on an object. From Newton’s second law, we can calculate it as mass times the acceleration of gravity (**w** = **m** * **g**). Weight is measured in newtons.
> - **Density** is is defined as the amount of mass per unit of volume (grams per cubic centimeter, for example).
>
> Note that an object that has a mass of one kilogram on earth would have a mass of one kilogram on the moon. However, it would weigh only one-sixth as much.

Now, in the world of Processing, what is mass anyway? Aren’t we dealing with pixels? To start in a simpler place, let’s say that in our pretend pixel world, all of our objects have a mass equal to 1. **F**/ 1 = **F**. And so:

A = **F**

The acceleration of an object is equal to force. This is great news. After all, we saw in Chapter 1 that acceleration was the key to the controlling the movement of our objects on screen. Location is adjusted by velocity, and velocity by acceleration. Acceleration was where it all began. Now we learn that **force** is truly where it all begins.

Let's say we have a class called Mover, with location, velocity, and acceleration.

```
class Mover {
  PVector location;
  PVector velocity;
  PVector acceleration;
}
```

Now our goal is to be able to add forces to this object, perhaps saying:

```
mover.applyForce(wind);
```

or:

```
mover.applyForce(gravity);
```

where wind and gravity are PVectors. According to Newton's second law, we could implement this function as follows.

```
  void applyForce(PVector force) {
    acceleration = force;
  }
```

Newton's second law at its simplest.

2.3 Force Accumulation

This looks pretty good. After all, it's a literal translation of Newton's second law (without mass): ***Acceleration*** = ***Force***. Nevertheless, there's a pretty big problem here. Let's return to what we are trying to accomplish: creating a moving object on the screen that responds to wind and gravity.

```
mover.applyForce(wind);
mover.applyForce(gravity);
mover.update();
mover.display();
```

Ok, let’s *be* the computer for a moment. First, we call **`applyForce()`** with wind. And so the Mover object’s acceleration is now set to the wind PVector. Second, we call **`applyForce()`** with gravity. And so the Mover object’s acceleration is now set to the gravity PVector. Third, we call **`update()`**. What happens in update? Acceleration is added to velocity.

```
velocity.add(acceleration);
```

We’re not going to see any *error* in Processing, but zoinks! We’ve got a major problem. What is the value of acceleration when it is added to velocity? It is equal to the gravity force. Wind has been left out! If we call **`applyForce()`** more than once, it overrides each previous call. How are we going to handle more than one force?

The answer is through a process known as ***force accumulation***. It’s actually very simple; all we need to do is add all of the forces together. At any given moment, there might be one, two, six, twelve, or three hundred and three forces. As long as our object knows how to accumulate them, it doesn’t matter how many forces act on it.

```
void applyForce(PVector force) {
   acceleration.add(force);
}
```

Newton’s second law but with force accumulation. We now add each force to acceleration, one at a time.

Now, we’re not finished just yet. There is one more piece to force accumulation. Since we’re adding all the forces together at any given moment, we have to make sure that we clear acceleration (i.e. set it to zero) before each time **`update()`** is called. Let’s think about wind for a moment. Sometimes the wind is very strong, sometimes it’s weak, and sometimes there’s no wind at all. At any given moment, there might be a huge gust of wind, say, when the user holds down the mouse.

```
if (mousePressed) {
  PVector wind = new PVector(0.5,0);
  mover.applyForce(wind);
}
```

When the user releases the mouse, the wind will stop and according to Newton’s first law, the object will continue to move at a constant velocity. However, if we had forgotten to reset acceleration to zero, the gust of wind would still be in effect. Even worse, it would add onto itself from the previous frame, since we are accumulating forces! Acceleration, in our simulation, has no memory; it is simply calculated based on the environmental forces present at a moment in time. This is different than, say, location, which must remember where the object was the previous frame in order to move properly to the next.

The easiest way to implement clearing the acceleration for each frame is to multiply the **`PVector`** by zero at the end of **`update()`**.

```
void update() {
   velocity.add(acceleration);
   location.add(velocity);
   acceleration.mult(0);
}
```

Exercise 2.1

Using forces, simulate a helium-filled balloon floating upward (and bouncing off the top of a window). Can you add a wind force which changes over time, perhaps according to Perlin noise?

2.4 Dealing with Mass

OK. We've got one tiny little addition to make before we are done with integrating forces into our Mover class and are ready to look at examples. After all, Newton's second law is really **`F`** = **`M`** * **`A`**, not **`F`** = **`A`**. Incorporating mass is as easy as adding an instance variable to our class, but we need to spend a little more time here because a slight complication will emerge.

First we just need to add mass.

```
class Mover {
  PVector location;
  PVector velocity;
  PVector acceleration;
  float mass;
```

Units of Measurement

Now that we are introducing mass it’s important to make a quick note about units of measurement. In the real world, things are measured with specific units. We say that two objects are three meters apart, the baseball is moving at a rate of ninety miles per hour, or this bowling ball has a mass of six kilograms. As we’ll see later in this book, sometimes we will want to take real-world units into consideration. However, in this chapter, we’re going to ignore them for the most part. Our units of measurement are in pixels (“these two circles are one hundred pixels apart”) and frames of animation (“this circle is moving at a rate of two pixels per frame.”) In the case of mass, there isn’t any unit of measurement for us to use. We’re just going to make something up. In this example, we’re arbitrarily picking the number ten. There is no unit of measurement (you might enjoy inventing a unit of your own, like “1 moog” or “1 yurkle.”) It should also be noted that, for demonstration purposes, we’ll tie mass to pixels (drawing, say, a circle with a radius of ten). This will allow us to visualize the mass of an object. In the real world, however, size does not definitely indicate mass. A small metal ball could have a much higher mass than a large balloon due to its higher density.

Mass is a scalar (float), not a vector, as it’s just one number describing the amount of matter in an object. We could be fancy about things and compute the area of a shape as its mass, but it’s simpler to begin by saying, “Hey, the mass of this object is…um, I dunno…how about 10?”

```
Mover() {
  location = new PVector(random(width),random(height));
  velocity = new PVector(0,0);
  acceleration = new PVector(0,0);
  mass = 10.0;
}
```

This isn’t so great since things only become interesting once we have objects with varying mass, but it’ll get us started. Where does mass come in? We use it while applying Newton’s second law to our object.

```
void applyForce(PVector force) {
  force.div(mass);
  acceleration.add(force);
}
```

Newton’s second law (with force accumulation and mass).

Yet again, even though our code looks quite reasonable, we have a fairly major problem here. Consider the following scenario with two *Mover* objects, both being blown away by a wind force.

```
Mover m1 = new Mover();
Mover m2 = new Mover();

PVector wind = new PVector(1,0);

m1.applyForce(wind);
m2.applyForce(wind);
```

Again, let's *be* the computer. Object **`m1`** receives the wind force—(1,0)—divides it by mass (10) and adds it to acceleration.

`m1`:
wind force is equal to: (1,0)
divided by mass = 10: (0.1,0)

OK, moving onto object **`m2`**. It also receives the wind force—(1,0). Wait. Hold on a second. What is the value of wind force? Taking a closer look, the wind force is actually now—(0.1,0)!! Do you remember this little tidbit about working with objects? When you pass an object (in this case a **`PVector`**) into a function, you are passing a reference to that object. It's not a copy! So if a function makes a change to that object (which, in this case, it does by dividing by mass) then that object is permanently changed! But we don't want **`m2`** to receive a force divided by the mass of object **`m1`**. We want it to receive that force in its original state—(1,0). And so we must protect ourselves and make a copy of the PVector f before dividing it by mass. Fortunately, the **`PVector`** class has a convenient method for making a copy—**`get()`**. **`get()`** returns a new **`PVector`** object with the same data. And so we can revise **`applyForce()`** as follows:

```
void applyForce(PVector force) {
  PVector f = force.get();          Making a copy of the PVector before using it!
  f.div(mass);
  acceleration.add(f);
}
```

There's another way we could write the above function, using the static method **`div()`**. For help with this exercise, review static methods in Chapter 1 ([REF] see p. XXX).

Exercise 2.2

Rewrite the applyForce method using the static method div() instead of get().

```
void applyForce(PVector force) {
  PVector f = _______.___(_____,____);
  acceleration.add(f);
}
```

2.5 Creating Forces

Let's take a moment to remind ourselves where we are. We know what a force is (a vector), and we know how to apply a force to an object (divide it by mass, add it to the object's acceleration vector). What are we missing? Well, we have yet to figure out how we get a force in the first place. Where do forces come from?

In this chapter, we'll look at two methods for creating forces in our Processing world.

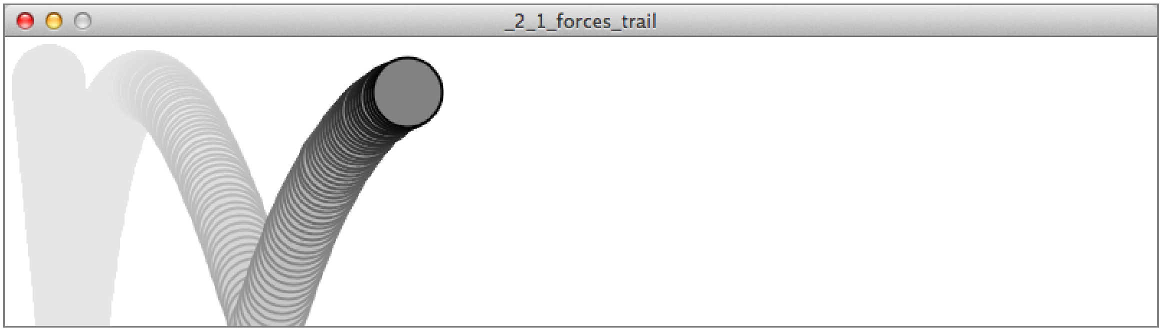
1. **Make up a force!** After all, you are the programmer, the creator of your world. There's no reason why you can't just make up a force and apply it.

2. **Model a force!** Yes, forces exist in the real world. And physics textbooks often contain formulas for these forces. We can take these formulas, translate them into source code, and model real-world forces in Processing.

The easiest way to make up a force is just to just pick a number. Let's start with the idea of simulating wind. How about a wind force that points to the right and is fairly weak? Assuming a Mover object "m", our code would look like:

```
PVector wind = new PVector(0.01,0);
m.applyForce(wind);
```

The result isn't terribly interesting, but it is a good place to start. We create a `PVector` object, initialize it, and pass it into an object (which in turn will apply it to its own acceleration). ￼ If we wanted to have two forces, perhaps wind and gravity (a bit stronger, pointing down), we might say:



Example 2.1

```
PVector wind = new PVector(0.01,0);
PVector gravity = new PVector(0,0.1);
m.applyForce(wind);
m.applyForce(gravity);
```

Now we have two forces, pointing in different directions with different magnitudes, both applied to object "**m**." We're beginning to get somewhere. We've now built a world for our objects in Processing, an environment to which they can actually respond.

Let's look at how we could make this example a bit more exciting with many objects of varying mass. To do this, we'll need to do a quick review of object-oriented programming. Again, we're not covering all the basics of programming here (for that you can check out any of the intro Processing books listed in the introduction). However, since the idea of creating a world filled with objects is pretty fundamental to all the examples in this book, it's worth taking a moment to walk through the steps of going from one object to many.

This is where we are with the Mover class as a whole. Notice how it is identical to the Mover class created in Chapter 1, with two additions—**mass** and a new **applyForce()** function.

```
class Mover {

  PVector location;
  PVector velocity;
  PVector acceleration;
  float mass;

  Mover() {
```

The object now has mass!

```
    mass = 1;
```

And for now, we'll just set the mass equal to 1 for simplicity.

```
    location = new PVector(30,30);
    velocity = new PVector(0,0);
    acceleration = new PVector(0,0);
  }
```

```
  void applyForce(PVector force) {
```

Newton's second law.

```
    PVector f = PVector.div(force,mass);
    acceleration.add(f);
  }
```

Receive a force, divide by mass, and add to acceleration.

```
  void update() {
```

```
    velocity.add(acceleration);
    location.add(velocity);
```

Motion 101 from Chapter 1

```
    acceleration.mult(0);
  }
```

plus clearing the acceleration each time!

```
  void display() {
    stroke(0);
    fill(175);
```

Let's scale the size of the object according to its mass.

```
ellipse(location.x,location.y,mass*16,mass*16);
  }
```

```
  void checkEdges() {
    if (location.x > width) {
      location.x = width;
      velocity.x *= -1;
    } else if (location.x < 0) {
      velocity.x *= -1;
      location.x = 0;
    }

    if (location.y > height) {
      velocity.y *= -1;
      location.y = height;
    }
  }
}
```

Somewhat arbitrarily, we are deciding that an object bounces when it hits the edges of a window.

Now that our class is set, we can choose to create, say, one hundred **`Mover`** objects with an array.

```
Mover[] movers = new Mover[100];
```

And then we can initialize all of those Mover objects in **`setup()`** with a loop.

```
void setup() {
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new Mover();
  }
}
```

But now we have a small issue. If we refer back to the **`Mover`** object's constructor…

```
  Mover() {
    mass = 1;
    location = new PVector(30,30);
    velocity = new PVector(0,0);
    acceleration = new PVector(0,0);
  }
```

…we discover that every Mover object is made exactly the same way. What we want are Mover objects of varying mass that start at varying locations. Here is where we need to increase the sophistication of our constructor by adding arguments.

```
  Mover(float m, float x , float y) {
    mass = m;
    location = new PVector(x,y);
    velocity = new PVector(0,0);
    acceleration = new PVector(0,0);
  }
```

Notice how the mass and location are no longer set to hardcoded numbers, but rather initialized via arguments passed through the constructor. This means we can create a variety of Mover objects: big ones, small ones, ones that start on the left side of the screen, ones that start on the right, etc.

```
Mover m1 = new Mover(10,0,height/2);
```

A big Mover on the left side of the window

```
Mover m1 = new Mover(0.1,width,height/2);
```

A small Mover on the right side of the window

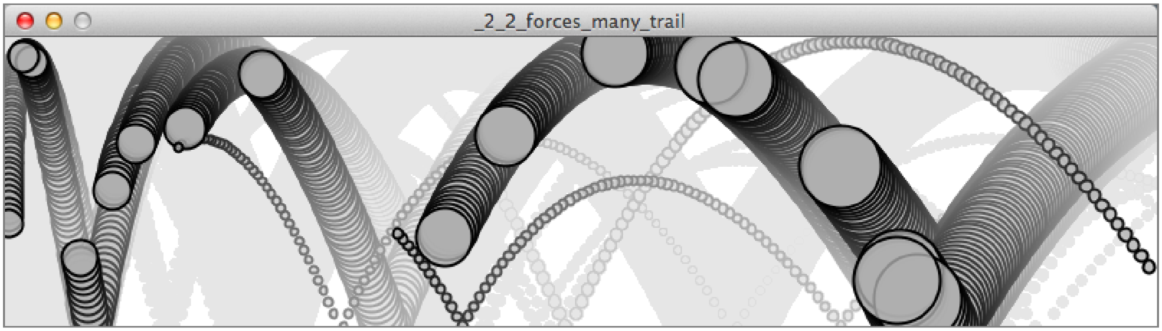
With an array, however, we want to initialize all of the objects with a loop.

```
void setup() {
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new
Mover(random(0.1,5),0,0);
  }
}
```

Initializing many “Mover” objects all with random mass (and all starting at 0,0).

For each “Mover” created, the mass is set to a random value between 0.1 and 5, the starting x-location is set to 0, and the starting y-location is set to 0. Certainly, there are all sorts of ways we might choose to initialize the objects; this is just a demonstration of one possibility.

Once the array of objects is declared, created, and initialized, the rest of the code is simple. We run through every object, hand them each the forces in the environment, and enjoy the show.



Example 2.2

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

  PVector wind = new PVector(0.01,0);
  PVector gravity = new PVector(0,0.1);
```

Make up two forces.

```
  for (int i = 0; i < movers.length; i++) {
    movers[i].applyForce(wind);
    movers[i].applyForce(gravity);
    movers[i].update();
    movers[i].display();
    movers[i].checkEdges();
  }
}
```

Loop through all objects and apply both forces to each object.

Note how in the above image, the smaller circles reach the right of the window faster than the larger ones. This is because of our formula: ***acceleration = force divided by mass***. The larger the mass, the smaller the acceleration.

Exercise 2.3

Create an example where instead of objects bouncing off the edge of the wall, an invisible force pushes back on the objects to keep them in the window. Can you weight the force according to how far the object is from an edge—i.e., the closer it is, the stronger the force?

2.6 Gravity on Earth and Modeling a Force

You may have noticed something woefully inaccurate about this last example. The smaller the circle, the faster it falls. There is a logic to this; after all, we just stated (according to Newton's second law) that the smaller the mass, the higher the acceleration. But this is not what happens in the real world. If you were to climb to the top of the Leaning Tower of Pisa and drop two balls of different masses, which one will hit the ground first? According to legend, Galileo performed this exact test in 1589, discovering that they fell with the same acceleration, hitting the ground at the same exact time. Why is this? As we will see later in this chapter, the force of gravity is calculated relative to an object's mass. The bigger the object, the stronger the force. So if the force is scaled according to mass, it is cancelled out when acceleration is divided by mass. We can implement this in our sketch rather easily, by multiplying our "made up" gravity force by mass.

imgs/chapter02/chp2_ex03.png

Example 2.3

```
for (int i = 0; i < movers.length; i++) {

    PVector wind = new PVector(0.001,0);
    PVector gravity = new                     Scaling gravity by mass to be more accurate
PVector(0,0.1*movers[i].mass);
    movers[i].applyForce(wind);
    movers[i].applyForce(gravity);

    movers[i].update();
    movers[i].display();
    movers[i].checkEdges();
  }
```

While the object's now fall at the same rate, because the strength of the wind force is independent of mass, the smaller objects still accelerate to the right more quickly.

Making up forces will actually get us quite far. The world of Processing is a pretend world of pixels and you are its master. So whatever you deem appropriate to be a force, well by golly, that's the force it should be. Nevertheless, there may come a time where you find yourself wondering: "But how does it really all work?"

Open up any high school physics textbook and you will find some diagrams and formulas describing many different forces—gravity, electromagnetism, friction, tension, elasticity, and more. In this chapter we're going to look at two forces—friction and gravity. The point we're making here is not that friction and gravity are fundamental forces that you always need to have in your Processing sketches. Rather, we want to evaluate these two forces as case studies for the following process:

- Understanding the concept behind a force
- Deconstructing the force's formula into two parts:
    - How do we compute the force's direction?
    - How do we compute the force's magnitude?
- Translating that formula into Processing code that calculates a **`PVector`** to be sent through our Mover's **`applyForce()`** function.

If we can follow the above steps with two forces, then hopefully if you ever find yourself Googling "atomic nuclei weak nuclear force" at 3 a.m., you will have the skills to take what you find and adapt it for Processing.

2.7 Friction

Let’s begin with friction and follow our steps:

What is friction?

Friction is a “dissipative” force. A dissipative force is one in which the total energy of a system decreases when an object is in motion. Let’s say you are driving a car. When you press your foot down on the brake pedal, the car’s brakes use friction to slow down the motion of the tires. Kinetic energy (motion) is converted into thermal energy (heat). Whenever two surfaces come into contact, they experience friction. A complete model of friction would include separate cases for static friction (a body at rest against a surface) and kinetic friction (a body in motion against a surface), but for our purposes, we are going to only look at the kinetic case.

What is the formula for friction?

Dealing with formulae

Ok, in a moment we're going to write out the formula for friction. This isn't the first time we've seen a formula is this book; we just finished up our discussion of Newton's second law, F =MA (or force = mass * acceleration). We didn't spend a lot of time worrying about this formula, because it's a nice and simple one. Nevertheless, it's a scary world out there. Just take a look at the equation for a "normal" distribution which we covered (without looking at the formula) in the introduction (see [REF]).

$$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

formula

[NEED MATH NOTATION]

What we're seeing here is that formulas like to use a lot of symbols (quite often letters from the Greek alphabet). Let's take a look at the formula for friction, which we're about to cover.

$$\overrightarrow{\text{Friction}} = -\mu N \hat{v}$$

formula

[NEED MATH NOTATION] If it’s been a while since you’ve looked at a formula from a math or physics textbook, there are three key points that are important to cover before we move on.

- ***Evaluate the right side, assign to the left side.*** This is just like in code! What we’re doing here is evaluating the right side of the equation and assigning it to the left. In the case above, we want to calculate the force of friction—the left side tells us what we want to calculate and the right side tells us how to do it.
- ***Are we talking about a vector or a scalar?*** It’s important for us to realize that in some cases, we’ll be looking at a vector; in others, a scalar. For example, in this case the force of friction is a vector. It has a magnitude and direction. We can see that by the arrow above the word “friction.” The right side of the equation also has a vector, as indicated by the symbol , which is this case stands for the velocity unit vector.
- ***When symbols are placed next too each other, we mean for them to be multiplied.*** The formula above actually has four elements: -1, -μ, N, and v **[needs to be unit vector v]**. We want to multiply them together and read the formula as:

$$\text{Friction} = -1 * \mu * N * \hat{v}$$

formula

[NEED MATH NOTATION]

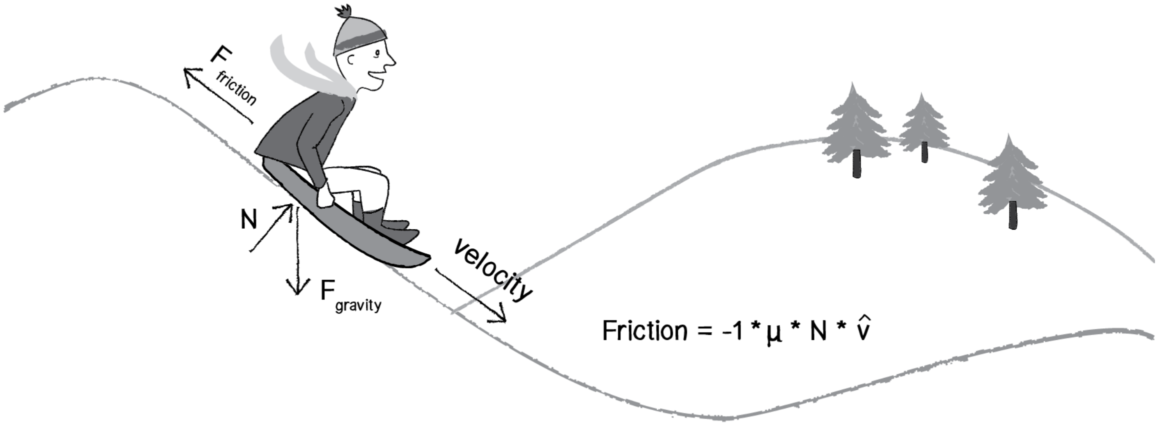


Figure 2.3

It’s now up to us to separate this formula into two components that determine the direction of friction as well as the magnitude. Based on the diagram above, we can see that *friction points in the opposite direction of velocity*. In fact, that’s the part of the formula that says -1 * v **[needs to be unit vector v]** or negative one times the velocity unit vector. In Processing, this would mean taking the velocity vector, normalizing it, and multiplying by -1.

```
PVector friction = velocity.get();
friction.normalize();
friction.mult(-1);
```

Let’s figure out the direction of the friction force (a unit vector in the opposite direction of velocity).

Notice two additional steps here. First, it’s important to make a copy of the velocity vector first as we don’t want to reverse the object’s direction by accident. Second, we normalize the vector. This is because the magnitude of friction is not associated with how fast it is moving, and we want to start with a friction vector of magnitude 1 so that it can easily be scaled.

According to the formula, the magnitude is **μ** * **N**. **μ** is the Greek letter Mu (pronounced “mew”), which is used here to describe the “coefficient of friction.” The coefficient of friction establishes the strength of a friction force for a particular surface. The higher it is, the stronger the friction; the lower, the weaker. A block of ice, for example, will have a much lower coefficient of friction than, say, sandpaper. Since we’re in a pretend Processing world, we can arbitrarily set the coefficient based on how much friction we want to simulate.

```
float c = 0.01;
```

Now for the second part: **N**. **N** refers to the “normal” force, the force perpendicular to the object’s motion along a surface. Think of a vehicle driving along a road. The vehicle pushes down against the road with gravity, and Newton’s third law tells us that the road in turn pushes back against the vehicle. That’s the normal force. The greater the gravitational force, the greater the normal force. As we’ll see in the next section, gravity is associated with mass and so a lightweight sports car would experience less friction than a massive tractor trailer truck. With the diagram above, however, where the object is moving along a surface at an angle, computing the normal force is a bit more complicated because it doesn’t point in the same direction as gravity. We’ll need to know something about angles and trigonometry.

All of these specifics are important; however, in Processing, a “good enough” simulation can be achieved without them. We can, for example, make friction work with the assumption that the normal force will always have a magnitude of 1. When we get into trigonometry in the next chapter, we’ll remember to return to this question and make our friction example a bit more sophisticated. Therefore:

```
float normal = 1;
```

Now that we have both the magnitude and direction for friction, we can put it all together:

```
float c = 0.01;
float normal = 1;
float frictionMag = c*normal;
```

Let’s figure out the magnitude of friction (really just an arbitrary constant).

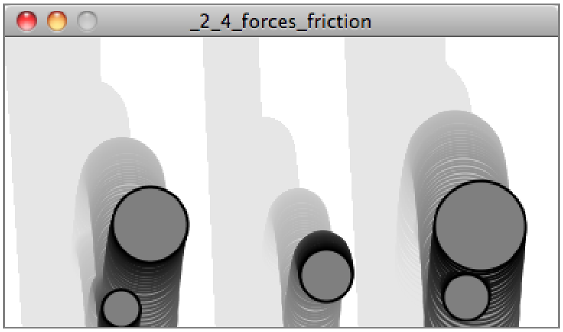
```
PVector friction = velocity.get();
friction.mult(-1);
friction.normalize();
friction.mult(frictionMag);
```

Take the unit vector and multiply it by magnitude and we have our force vector!

...and add it to our “forces” example, where many objects experience wind, gravity, and now friction:

No friction

No friction



With friction

Example 2.4: Including friction

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

  PVector wind = new PVector(0.001,0);
  PVector gravity = new PVector(0,0.1);    We could scale by mass to be more accurate

  for (int i = 0; i < movers.length; i++) {

    float c = 0.01; // [bold]
    PVector friction = movers[i].velocity.get(); // [bold]
    friction.mult(-1); // [bold]
    friction.normalize(); // [bold]
    friction.mult(c); // [bold]
    movers[i].applyForce(friction); //      Apply friction force vector to object
[bold]
    movers[i].applyForce(wind);
    movers[i].applyForce(gravity);

    movers[i].update();
    movers[i].display();
    movers[i].checkEdges();
  }

}
```

Running this example, you'll notice that the circles don't even make it to the right side of the window. Since friction continuously pushes against the object in the opposite direction of its movement, the object continuously slows down. This can be a useful technique or a problem depending on the goals of your visualization.

> **Exercise 2.4**
>
> Create pockets of friction in a Processing sketch so that objects only experience the friction when crossing over that area. What if you vary the strength (friction coefficient) of each area? What if you make some of them the opposite of friction—i.e., when you enter a given pocket you actually speed up instead of slowing down?

2.8 Air and Fluid Resistance

Friction also occurs when a body passes through a liquid or gas. This force has many different names, all really meaning the same thing: viscous force, drag force, fluid resistance. While the result is ultimately the same as our previous friction examples (the object slows down), the way in which we calculate a drag force will be slightly different. Let's look at the formula:

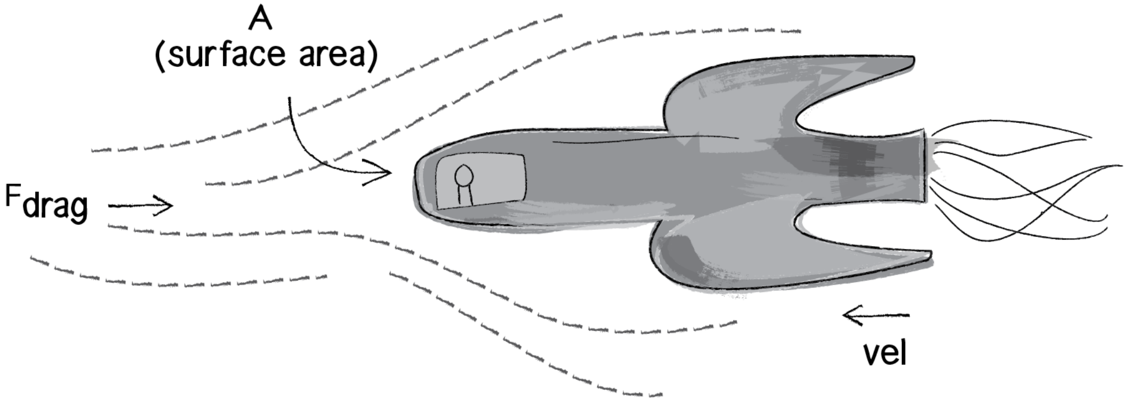


Figure 2.4

$$\mathbf{F}_d = -\frac{1}{2}\rho v^2 A C_d \hat{\mathbf{v}}$$

formula

[Need Math Notation]

Now let’s break this down and see what we really need for an effective simulation in Processing, making ourselves a much simpler formula in the process.

- $\mathbf{F_d}$ refers to “Drag Force”, the vector we ultimately want to compute and pass into our **`applyForce()`** function.
- - 1/2 is a constant: -0.5! This is fairly irrelevant in terms of our Processing world, as we will be making up values for other constants anyway. However, the fact that it is negative is important, as it tells us that the force is in the opposite direction of velocity (just as with friction). -￼ρ is the Greek letter rho, and refers to the density of the liquid, something we don’t need to worry about. We can simplify the problem and consider this to have a constant value of 1.
- **v** refers to the speed of the object moving. OK, we’ve got this one! The object’s speed is the magnitude of the velocity vector: velocity.magnitude(). And $\mathbf{v}^2$ just means **v** squared or **v** * **v**.
- A refers to the frontal area of the object that is pushing through the liquid (or gas). An aerodynamic Lamborghini, for example, will experience less air resistance than a boxy Volvo. Nevertheless, for a basic simulation, we can consider our object to be spherical and ignore this element.

- C_d is the coefficient of drag, exactly the same as the coefficient of friction (μ). This is a constant we'll determine based on whether we want the drag force to be strong or weak.

- v **[need math notation, velocity unit vector]** Look familiar? It should. This refers to the velocity unit vector, i.e. velocity.normalize(). Just like with ￼friction, drag is a force that points in the opposite direction of velocity.

Now that we've analyzed each of these components and determined what we need for a simple simulation, we can reduce our formula to:

[var]*F_d = -1 * C_d * v^2 * v **need math notation, velocity unit vector**

or:

```
float c = 0.1;
float speed = v.mag();
---------------------------------------------------------------------------
float dragMagnitude = c * speed * speed;   Part 1 of our formula (magnitude): Cd * v2
PVector drag = velocity.get();
---------------------------------------------------------------------------
drag.mult(-1);                             Part 2 of our formula (direction): -1 * velocity￼
drag.normalize();
---------------------------------------------------------------------------
drag.mult(dragMagnitude);                  Magnitude and direction together!
```

Let's implement this force in our Mover example with one addition. When we wrote our friction example, the force of friction was always present. Whenever an object was moving, friction would slow it down. Here, let's introduce an element to the environment—a "liquid" that the Mover objects pass through. The liquid object will be a rectangle and will know about its location, width, height, and "coefficient of drag"—i.e., is it easy for objects to move through it (like air) or difficult (like molasses)? In addition, it should include a function to draw itself on the screen (and two more functions, which we'll see in a moment.)

```
class Liquid {
---------------------------------------------------------------------------
  float x,y,w,h;                           The liquid object includes a variable defining its coefficient of drag.
  float c;

  Liquid(float x_, float y_, float w_, float h_, float c_) {
    x = x_;
    y = y_;
    w = w_;
    h = h_;
    c = c_;
  }

  void display() {
    noStroke();
    fill(175);
    rect(x,y,w,h);
  }

}
```

The main program will now include a Liquid object reference as well as a line of code that initializes that object.

```
Liquid liquid;

void setup() {
  liquid = new Liquid(0, height/2,
width, height/2, 0.1);
}
```

Initialize a Liquid object. Note how the coefficient value is low (0.1). Otherwise, the object would come to a halt fairly quickly (which may someday be the effect you want).

Now comes an interesting question: how do we get the Mover object to talk to the Liquid object? In other words, we want to execute the following:

When a Mover passes through a Liquid it experiences a Drag force.

or in object-oriented speak (assuming we are looping through an array of Mover objects with index i):

```
if (movers[i].isInside(liquid)) {
  movers[i].drag(liquid);
}
```

If a Mover is inside a Liquid, apply the drag force.

The above code tells us that we need to add two functions to the Mover class: (1) a function that determines if a Mover object is inside the liquid, and (2) a function that computes and applies a drag force on the Mover object.

The first is easy; we can simply use a conditional statement to determine if the location vector rests inside the rectangle defined by the liquid.

```
boolean isInside(Liquid l) {
  if (location.x > l.x && location.x < l.x + l.w && location.y > l.y && location.y < l.y +
l.h) {
    return true;
  } else {
    return false;
  }
}
```

This conditional statement determines if the PVector location is inside the rectangle defined by the Liquid class.

The **drag()** function is a bit more complicated; however, we've written the code for it already. This is simply an implementation of our formula. The drag force is equal to *the coefficient of drag multiplied by the speed of the Mover squared in the opposite direction of velocity!*

[var]*F_d = -1 * C_d * v^2 * v **[need math notation, velocity unit vector]**

```
 void drag(Liquid l) {

    float speed = velocity.mag();
    float dragMagnitude = l.c * speed *
speed;

    PVector drag = velocity.get();
    drag.mult(-1);
    drag.normalize();
    drag.mult(dragMagnitude);
    applyForce(drag);
  }
```

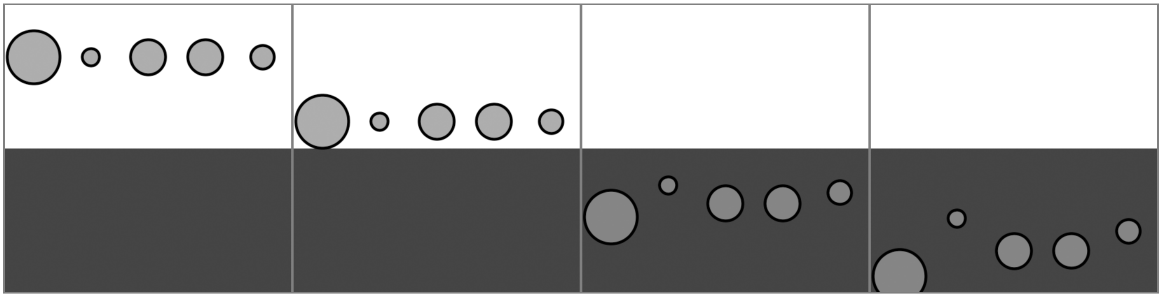
The force's magnitude: Cd * v2

The force's direction: -1 * velocity

Finalize force: magnitude and direction -Cd * speed * speed * velocity unit vector

Apply the force

And with these two functions added to the Mover class, we're ready to put it all together in the main tab:



Example 2.5: Fluid Resistance

```
Mover[] movers = new Mover[100];

Liquid liquid;

void setup() {
  size(360, 640);
  smooth();
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new Mover(random(0.1,5),0,0);
  }
  liquid = new Liquid(0, height/2, width, height/2, 0.1);
}

void draw() {
  background(255);

  liquid.display();

  for (int i = 0; i < movers.length; i++) {

    if (movers[i].isInside(liquid)) {
      movers[i].drag(liquid);
    }
    PVector gravity = new PVector(0,
0.1*movers[i].mass);
    movers[i].applyForce(gravity);

    movers[i].update();
```

Note we are scaling gravity according to mass.

```
      movers[i].display();
      movers[i].checkEdges();
    }
  }
```

Running the example, you should notice that we are simulating balls falling into water. The objects only slow down when crossing in the gray area at the bottom of the window (representing the liquid). You'll also notice that the smaller objects slow down a great deal more than the larger objects. Remember Newton's second law? **`A`** = **`F`** / **`M`**. Acceleration equals Force ***divided*** by mass. A massive object will accelerate less. A smaller object will accelerate more. In this case, the acceleration we're talking about is the "slowing down" due to drag. The smaller objects will slow down at a greater rate than the larger ones.

Exercise 2.6

Take a look at our formula for drag again. ***DRAG FORCE = COEFFICIENT * SPEED * SPEED***. The faster an object moves, the greater the drag force against it. In fact, an object not moving in water experiences no drag at all. Expand the example to drop the balls from different heights. How does this affect the drag as they hit the water?

Exercise 2.7

The formula for drag also included surface area. Can you create a simulation of boxes falling into water with a drag force dependent on the length of the side hitting the water?

Exercise 2.8

Fluid resistance does not work only opposite to the velocity vector, but also perpendicular to it. This is known as "lift-induced drag" and will cause an airplane with an angled wing to rise in altitude. Try creating a simulation of lift.

2.9 Gravitational Attraction

Probably the most famous force of all is gravity. We humans on earth think of gravity as an apple hitting Isaac Newton on the head. Gravity means that stuff falls down. But this is only our experience of gravity. In truth, just as the earth pulls the apple towards it due to a gravitational force, the apple pulls the earth as well. The thing is, the earth is just so freaking big that it overwhelms all the other gravity interactions. Every object with mass exerts a gravitational force on every other object. And there is a formula for calculating the strengths of these forces:

Let's examine this formula a bit more closely:

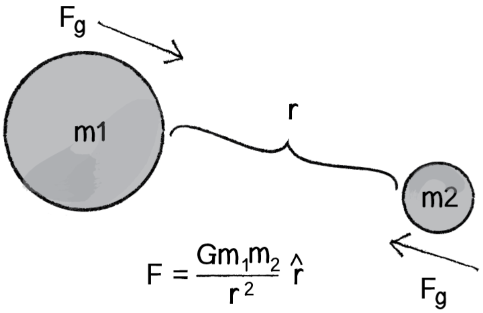


Figure 2.5

- **F** refers to the gravitational force, the vector we ultimately want to compute and pass into our applyForce() function.

- **G** is the "Universal Gravitational Constant" and in our world equals 6.67428 x 10-11 meters cubed per kilogram per second squared. This is a pretty important number if your name is Isaac Newton or Albert Einstein. It's not an important number if you are a Processing programmer. Again, it's a constant that we can use to make the forces in our world weaker or stronger. Just making it equal to one and ignoring it isn't such a terrible choice either.

- m_1 and m_2 are the masses of objects 1 and 2. As we saw with Newton's second law (F = M*A), mass is also something we could choose to ignore. After all, shapes drawn on the screen don't actually have a physical mass. However, if we keep these values, we can create more interesting simulations where "bigger" objects exert a stronger gravitational force than smaller ones.

- r **[math notation]** refers to the unit vector pointing from object 1 to object 2. As we'll see in a moment, we can compute this direction vector by subtracting the location of one object from the other.

- r^2 refers to the distance between the two objects squared. Let's take a moment to think about this a bit more. With everything on the top of the formula—**G**, m_1, m_2*—the bigger its value, the stronger the force. Big mass, big force. Big [var]*G, big force. Now, when we divide by something we have the opposite. The strength of the force is inversely proportional to the distance squared. The *further* away an object is, the *weaker* the force; the *closer*, the *stronger*.

Hopefully by now the formula makes some sense to us. We've looked at a diagram and dissected the individual components of the formula. Now it's time to figure out how we translate the math into Processing code. Let's make the following assumptions.

We have two objects and:

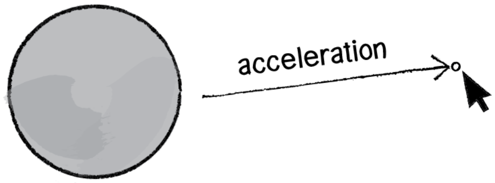
1. Each object has a location: **`PVector location1`** and **`PVector location2`**.
2. Each object has a mass: **`float mass1`** and **`float mass2`**.
3. There is a variable **`float G`** for the universal gravitational constant.

Given these assumptions, we want to compute **`PVector force`**, the force of gravity. We'll do it in two parts. First, we'll compute the direction of the force r **[math notation]** in the formula above). Second, we'll calculate the strength of the force according to the masses and distance.

Remember in Chapter 1, when we figured out how to have an object accelerate towards the mouse?

A vector is the difference between two points. To make a vector that points from the circle to the mouse, we simply subtract one point from another:

```
PVector dir =
PVector.sub(mouse,location);
```



See Chapter 1, Exercise 1.10 (?)

Figure 2.6

In our case, the direction of the attraction force that object 1 exerts on object 2 is equal to:

```
PVector dir = PVector.sub(location1,location2);
dir.normalize();
```

Don't forget that since we want a unit vector, a vector that tells us about direction only, we'll need to ***normalize*** the vector after subtracting the locations.

OK, we've got the direction of the force. Now we just need to compute the magnitude and scale the vector accordingly.

```
float m = (G * mass1 * mass2) / (distance * distance);
dir.mult(m);
```

The only problem is that we don't know the distance. **`G`**, **`mass1`**, and **`mass2`** were all givens, but we'll need to actually compute distance before the above code will work. Didn't we just make a vector that points all the way from one location to another? Wouldn't the length of that vector be the distance between two objects?

Well, if we add just one line of code and grab the magnitude of that vector before normalizing it, then we'll have the distance.

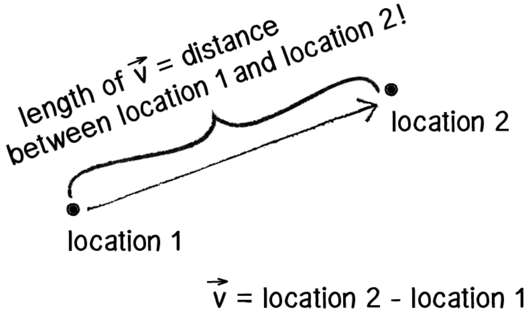

length of $\vec{v}$ = distance between location 1 and location 2!
location 2
location 1
$\vec{v}$ = location 2 - location 1

Figure 2.7

```
PVector force =
PVector.sub(location1,location2);
```

The vector that points from one object to another

```
float distance = force.magnitude();
```

The length (magnitude) of that vector is the distance between the two objects.

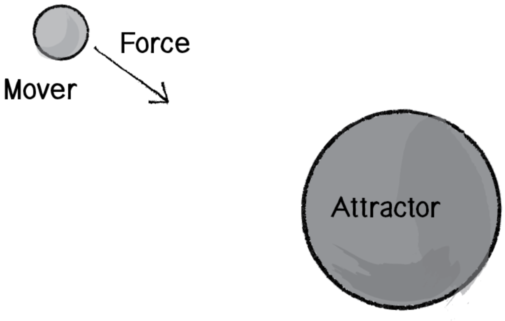
```
float m = (G * mass1 * mass2) / (distance * distance);
```

```
force.normalize();
force.mult(m);
```

Use the formula for gravity to compute the strength of the force. Normalize and scale the force vector to the appropriate magnitude.

Note that I also renamed the PVector "dir" as "force." After all, when we're finished with the calculations, the PVector we started with ends up being the actual force vector we wanted all along.

Now that we've worked out the math and the code for calculating an attractive force (emulating gravity), we need to turn our attention to applying this technique in the context of an actual Processing sketch. In Example 2.x, you may recall how we created a simple Mover object—a class with PVector's location, velocity, and acceleration as well as an **`applyForce()`**. Let's take this exact class and put it in a sketch with:



- A single Mover object.
- A single Attractor object (a new class that will have a fixed location).

The Mover object will experience a gravitational pull towards the Attractor object, as illustrated in Figure X.X.

We can start by making the new Attractor class very simple—a location and a mass, along with a function to display itself (tying mass to size).

```
class Attractor {
  float mass;
  PVector location;

  Attractor() {
    location = new PVector(width/2,height/2);
    mass = 20;
  }

  void display() {
    stroke(0);
    fill(175,200);
    ellipse(location.x,location.y,mass*2,mass*2);
  }
}
```

Our Attractor is a simple object that doesn’t move. We just need a mass and a location.

And in our main program, we can add an instance of the Attractor class.

```
Mover m;
Attractor a;

void setup() {
  size(200,200);
  m = new Mover();
  a = new Attractor();
}
```

Initialize Attractor object.

```
void draw() {
  background(255);
  a.display();                          Display Attractor object.

  m.update();
  m.display();
}
```

This is a good structure: a main program with a Mover and Attractor object, and a class to handle the variables and behaviors of Movers and Attractors. The last piece of the puzzle is how to get one object to attract the other. How do we get these two objects to talk to each other?

There are a number of ways we could do this. Here are just a few possibilities:

Table

1. A function that receives both an **Attractor** and a **Mover**:	attraction(a,m);
2. A function in the **Attractor** class that receives a **Mover**:	a.attract(m);
3. A function in the **Mover** class that receives an **Attractor**:	m.attractedTo(a);
4. A function in the **Attractor** class that receives a **Mover** and returns a **PVector**, which is the attraction force. That attraction force is then passed into the **Mover*'s [function]*applyForce()** function:	PVector f = a.attract(m); m.applyForce(f);

and so on. . .

It's good to look at a range of options for making objects talk to each other, and you could probably make arguments for each of the above possibilities. I'd like to at least discard the first one, since an object-oriented approach is really a much better choice over an arbitrary function not tied to either the Mover or Attractor class. Whether you pick (2) or (3) is the difference between saying "The attractor attracts the mover" or "The mover is attracted to the attractor." Number 4 is really my favorite, at least in terms of where we are in this book. After all, we spent a lot of time working out the **applyForce()** function and I think our examples will be clearer if we continue with the same methodology.

In other words, where we once had:

```
PVector f = new PVector(0.1,0);          Made-up force
m.applyForce(f);
```

We now have:

```
PVector f = a.attract(m);                Attraction force between two objects
m.applyForce(f);
```

And so our draw() function can now be written as:

```
void draw() {
  background(255);
  *PVector f = a.attract(m);             Calculate attraction force and apply it.
  m.applyForce(f);*

  m.update();

  a.display();
  m.display();

}
```

We're almost there. Since we decided to put the attract() function inside of the Attractor class, we'll need to actually write that function. The function needs to receive a Mover object and return a PVector, i.e.:

```
PVector attract(Mover m) {

}
```

And what goes inside that function? All of that nice math we worked out for gravitational attraction!

```
PVector attract(Mover m) {
  PVector force =                        What's the force's direction?
PVector.sub(location,m.location);
  float distance = force.mag();
  force.normalize();
  float strength = (G * mass * m.mass) / What's the force's magnitude?
(distance * distance);
  force.mult(strength);
  return force;                          Return the force so that it can be applied!
}
```

And we're done. Sort of. Almost. There's one small kink we need to work out. Let's look at the above code again. See that symbol for divide, the slash? Whenever we have one of these, we need to ask ourselves the question: What would happen if the distance happened to be a really, really small number or (even worse!) zero??! Well, we know we can't divide a number by zero, and if we were to divide a number by something like 0.0001, that is the equivalent of multiplying that number by 10,000! Yes, this is the real-world formula for the strength of gravity, but we don't live in the real world. We live in the *Processing* world. And in the Processing world, the Mover could end up being very, very close to the Attractor and the force could become so strong the Mover would just fly way off the screen. And so with this formula, it's good for us to be practical and constrain the range of what distance can actually be. Maybe, no matter where the Mover actually is, we should never consider it less than 5 pixels or more than 25 pixels away from the Attractor.

```
  distance = constrain(distance,5,25);
```

For the same reason we need to constrain the minimum distance, it's useful for us to do the same with the maximum. After all, if the Mover were to be, say, 500 pixels from the Attractor (not unreasonable), we'd be dividing the force by 250,000. That force might end up being so weak that it's almost as if we're not applying it at all.

Now, it's really up to you to decide what behaviors you want. But in the case of, "I want reasonable-looking attraction that is never absurdly weak or strong," then constraining the distance is a good technique.

Our Mover class hasn't changed at all, so let's just look at the main program and Attractor class as a whole, adding a variable "g" for the universal gravitational constant. (On the web site, you'll find that this example also has code that allows you to move the Attractor object with the mouse):

imgs/chapter02/chapter02_33-01.png

Example 2.6: Attraction

```
Mover m;
Attractor a;

void setup() {
  size(200,200);
  m = new Mover();
  a = new Attractor();
}

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

A Mover and an Attractor

```
  PVector force = a.attract(m);
  m.applyForce(force);
  m.update();

  a.display();
  m.display();
}

class Attractor {
  float mass;
  PVector location;
  float G;

  Attractor() {
    location = new PVector(width/2,height/2);
    mass = 20;
    G = 0.4;
  }

  PVector attract(Mover m) {
    PVector force = PVector.sub(location,m.location);
    float distance = force.mag();
```

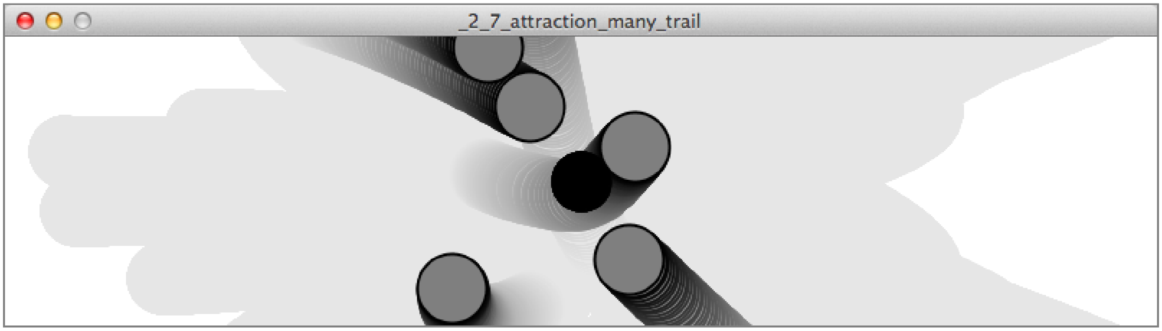
Apply the attraction force from the Attractor on the Mover.

```
    distance =
constrain(distance,5.0,25.0);
```

Remember, weneed to constrain the distance so that our circle doesn’t spin out of control.

```
    force.normalize();
    float strength = (G * mass * m.mass) / (distance * distance);
    force.mult(strength);
    return force;
  }

  void display() {
    stroke(0);
    fill(175,200);
    ellipse(location.x,location.y,mass*2,mass*2);
  }
}
```

And we could, of course, expand this example using an array to include many Mover objects, just as we did with friction and drag:



Example 2.7: Attraction with many Movers

```
Mover[] movers = new Mover[10];          Now we have 10 Movers!

Attractor a;

void setup() {
  size(400,400);
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new                      Each Mover is initialized randomly
Mover(random(0.1,2),random(width),random(height));
  }
  a = new Attractor();
}

void draw() {
  background(255);

  a.display();

  for (int i = 0; i < movers.length; i++) {
    PVector force = a.attract(movers[i]); We calculate an attraction force for each Mover object.
    movers[i].applyForce(force);

    movers[i].update();
    movers[i].display();
  }

}
```

Exercise 2.9

In the example above, we have a system (i.e. array) of Mover objects and one Attractor object. Build an example that has both systems of Movers and Attractors. What if you make the Attractors invisible? Can you create a pattern / design from the trails of objects moving around attractors? (See the Metropop Denim project by Clayton Cubitt and Tom Carden: http://processing.org/exhibition/works/metropop/ (See page 0)) for an example.)

Exercise 2.10

It’s worth noting that gravitational attraction is a model we can follow to develop our own forces. This chapter isn’t suggesting that you should exclusively create sketches that use gravitational attraction. Rather, you should be thinking creatively about how to design your own rules to drive the behavior of objects. For example, what happens if you design a force that is weaker the closer it gets and stronger the farther it gets? Or what if you design your attractor to attract far away objects, but repel close ones?

2.10 Everything Attracts (or Repels) Everything

Hopefully, you found it helpful that we started with a simple scenario: *one object attracts another object*, moving on to *one object attracts many objects*. However, it's likely that you are going to find yourself in a slightly more complex situation: *many objects attract each other*. In other words, every object in a given system attracts every other object in that system (except for itself).

We've really done almost all of the work for this already. Let's consider a Processing sketch with an array of Mover objects:

```
Mover[] movers = new Mover[10];

void setup() {
  size(400,400);
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new Mover(random(0.1,2),random(width),random(height));
  }
}

void draw() {
  background(255);
  for (int i = 0; i < movers.length; i++) {
    movers[i].update();
    movers[i].display();
  }
}
```

The **`draw()`** function is where we need to work some magic. Currently, we're saying: "for every Mover i, update and display yourself." Now what we need to say is: "for every Mover i, be attracted to every other Mover j, and update and display yourself."

To do this, we need to nest a second loop.

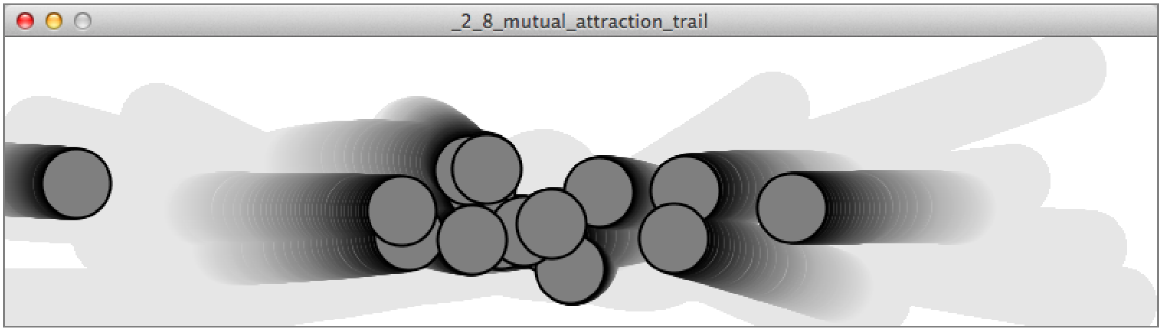
```
 for (int i = 0; i < movers.length; i++) {
    for (int j = 0; j < movers.length;    For every Mover, check every Mover!
j++) {
      PVector force = movers[j].attract(movers[i]);
      movers[i].applyForce(force);
    }
```

```
      movers[i].update();
      movers[i].display();
    }
```

In the previous example, we had an **Attractor** object with a function named **attract()**. Now, since we have **Movers** attracting **Movers**, all we need to do is copy the **attract()** function into the **Mover** class.

```
  class Mover {
  ==============================================================================
    PVector attract(Mover m) {        all the other stuff we had before plus. . .
                                      The Mover now knows how to attract another Mover.
      PVector force =
  PVector.sub(location,m.location);
      float distance = force.mag();
      distance = constrain(distance,5.0,25.0);
      force.normalize();

      float strength = (G * mass * m.mass) / (distance * distance);
      force.mult(strength);
      return force;
    }
  }
```

Of course, there's one small problem. When we are looking at every **Mover i** and every **Mover j** , are we OK with the times that **i** equals **j**? For example, should **Mover** #3 attract **Mover** #3? The answer, of course, is no. If there are five objects, we only want **Mover** #3 to attract 0, 1, 2, and 4, skipping itself. And so, we finish this example by adding a simple conditional statement to skip applying the force when i equals j.



Example 2.8: Mutual Attraction

```
Mover[] movers = new Mover[20];

float g = 0.4;

void setup() {
  size(400,400);
  for (int i = 0; i < movers.length; i++) {
    movers[i] = new Mover(random(0.1,2),random(width),random(height));
  }
}

void draw() {
  background(255);

  for (int i = 0; i < movers.length; i++) {
    for (int j = 0; j < movers.length; j++) {
      if (i != j) {
        PVector force = movers[j].attract(movers[i]);
        movers[i].applyForce(force);
      }
    }
    movers[i].update();
    movers[i].display();
  }
}
```

Don’t attract yourself!

Exercise 2.11

Change the attraction force in Example 2.x to a repulsion force. Can you create an example where all of the Mover objects are attracted to the mouse, but repel each other? Think about how you need to balance the relative strength of the forces and how to most effectively use distance in your force calculations.

The Ecosystem Project:

Step 2 Exercise:

Incorporate the concept of forces into your ecosystem. Try introducing other elements into the environment (food, a predator) that the creature interacts with. Does the creature experience attraction or repulsion to things in its world? Can you think more abstractly and design forces based on the creature’s desires or goals?