Chapter 2. Forces

"Don't underestimate the Force."

— Darth Vader

In the final example of Chapter 1, I demonstrated how to calculate a dynamic acceleration based on a vector pointing from a circle on the canvas to the mouse position. The resulting motion resembles a magnetic attraction between shape and mouse, as if some *force* was pulling the circle in towards the mouse. In this chapter I will detail the concept of a force and its relationship to acceleration. The goal, by the end of this chapter, is to build a simple physics engine and understand how objects move around a canvas by responding to a variety of environmental forces.

2.1 Forces and Newton's Laws of Motion

Before I examine the practical realities of simulating forces and building a basic physics engine in code, let's take a conceptual look at what it means to be a force in the real world. Just like the word "vector," the term "force" can be used to mean a variety of things. It can indicate a powerful intensity, as in "They pushed the boulder with great force" or "They spoke forcefully." The definition of ***force*** that I care about is more formal and comes from Sir Isaac Newton's laws of motion:

> ***A force is a vector that causes an object with mass to accelerate.***

The good news is that you hopefully recognize the first part of the definition: *a force is a vector*. Thank goodness you just spent a whole chapter learning what a vector is and how to program with vectors!

Let's define Newton's three laws of motion 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. I could expand the definition 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.

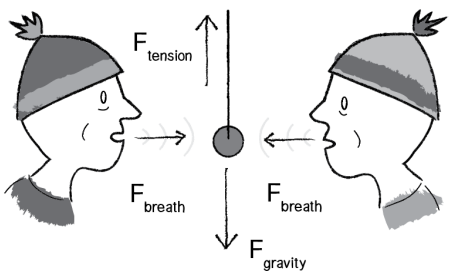


Figure 2.1: The pendulum doesn't move because all the forces cancel each other out (add up to a net force of zero).

Considering a p5.js canvas, I could restate Newton's first law as follows:

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

Skipping Newton's second law (arguably the most important law for the purposes of this book) for a moment, let's move on to the third law.

Newton's Third Law

This law is often stated as:

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

This law frequently causes 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 referred to in 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).

Consider pushing on a stationary truck. Although the truck is far more powerful than you, unlike a moving one, 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 hands. 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?

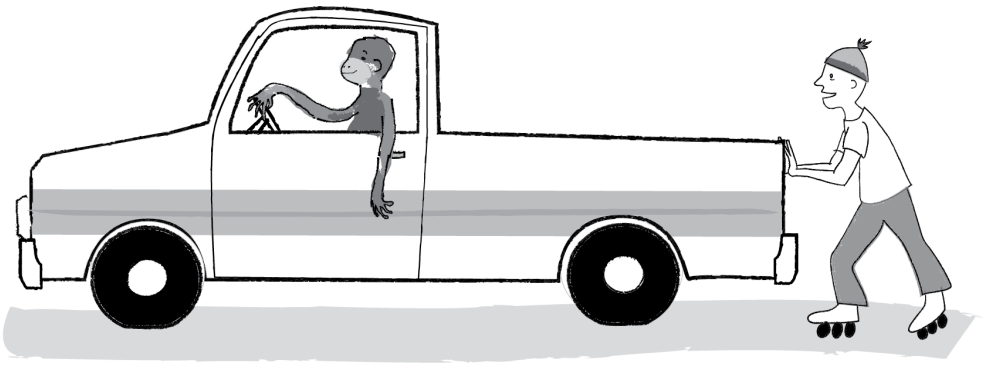


Figure 2.2

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 I'll get into with Newton's second law). There are other forces at work too, namely the friction of the truck's tires and your roller skates against the road.

Newton's Third Law (as seen through the eyes of p5.js)

> ***If you calculate a p5.Vector*** `f` ***that is a force of object A on object B, you must also apply the force—***`p5.Vector.mult(f, -1)`***;—that B exerts on object A.***

You'll soon see that in the world of coding simulation, it's often not necessary to stay true to the above. Sometimes, such as in the case of gravitational attraction between bodies (see page 0), I'll want to model equal and opposite forces in my example code. Other times, such as a scenario where I'll say, "Hey, there's some wind in the environment," I'm not going to bother to model the force that a body exerts back on the air. In fact, I'm not going to bother modeling the air at all! Remember, this examples in this book are taking inspiration from the physics of the natural world for the purposes of creativity and interactivity and do not require perfect precision.

2.2 Forces and p5.js—Newton's Second Law as a Function

Now it's time for most important law for you, the p5.js coder.

Newton's Second Law

This law is stated as:

> *Force equals mass times acceleration.*

Or:

$$\vec{\mathrm{F}} = \mathrm{M} \times \vec{\mathrm{A}}$$

Why is this the most important law for this book? Well, let's write it a different way.

$$\vec{\mathrm{A}} = \vec{\mathrm{F}} / \mathrm{M}$$

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 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 p5.js, what is mass anyway? Aren't we dealing with pixels? To start in a simpler place, let's say that in a pretend pixel world, all objects have a mass equal to 1. `F / 1 = F`. And so:

$$\vec{A} = \vec{F}$$

The acceleration of an object is equal to force. This is great news. After all, in Chapter 1 I described acceleration as the key to controlling the movement of objects on a canvas. `position` changes according to `velocity`, and `velocity` according to `acceleration`. Acceleration was where it all began. Now you can see that *force* is truly where it all begins.

Let's take the `Mover` class, with position, velocity, and acceleration.

```
class Mover {
  constructor(){
    this.position = createVector();
    this.velocity = createVector();
    this.acceleration = createVector();
  }
}
```

Now the goal is to be able to add forces to this object, with code something like:

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

or:

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

where wind and gravity are `p5.Vector` objects. According to Newton's second law, I could implement this function as follows.

```
applyForce(force) {
  this.acceleration = force;                    Newton's second law at its simplest.
}
```

2.3 Force Accumulation

This looks pretty good. After all, *acceleration = force* is a literal translation of Newton's second law (in a world without mass). Nevertheless, there's a pretty big problem here which I'll quickly encounter when I return to my original goal: creating object that responds to wind and gravity forces.

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

OK, I'll *be* the computer for a moment. First, I call `applyForce()` with wind. And so the `Mover` object's acceleration is now assigned the vector `wind`. Second, I call `applyForce()` with gravity. Now the `Mover` object's acceleration is set to the gravity vector. Finaally, I call `update()`. What happens in `update()`? Acceleration is added to velocity.

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

If you run this code, you will not see error in the console, but zoinks! There's a major problem. What is the value of acceleration when it is added to velocity? It is equal to the gravity vector. Wind has been left out! Anytime `applyForce()` is called, acceleration is overwritten. How can I handle more than one force?

The answer lies in the full definition of Newton's second law itself which I now to confess to having simplified. Here's a more accurate way to put it:

> ***Net Force equals mass times acceleration.***

Or, acceleration is equal to the *sum of all forces* divided by mass. This makes perfect sense. After all, as you saw in Newton's first law, if all the forces add up to zero, an object experiences an equilibrium state (i.e. no acceleration). This can be implemented through a process known as ***force accumulation***— adding all of the forces together. At any given moment, there might be 1, 2, 6, 12, or 303 forces. As long as the object knows how to accumulate them, it doesn't matter how many forces act on it.

```
applyForce(force) {
  this.acceleration.add(force);          Newton's second law, but with force
                                         accumulation. I now add each force to
                                         acceleration, one at a time.
}
```

Now, I'm not finished just yet. Force accumulation has one more piece. Since I'm adding all the forces together at any given moment, I have to make sure that I clear acceleration (i.e. set it to zero) before each time `update()` is called. Consider a wind force for a moment. Sometimes wind is very strong, sometimes it's weak, and sometimes there's no wind at all. For example, you might write code that creates a gust of wind, say, when the user holds down the mouse.

```
if (mouseIsPressed) {
  let wind = createVector(0.5, 0);
  mover.applyForce(wind);
}
```

When the user releases the mouse, the wind should stop, and according to Newton's first law, the object continues moving at a constant velocity. However, if I forgot 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! Acceleration, in a time-based physics simulation, has no memory; it is calculated based on the environmental forces present at any given moment (frame) in time. This is different than, say, position, which must remember its previous location in order to move properly to the next.

One way to implement clearing the acceleration for each frame is to multiply the vector by 0 at the end of `update()`.

```
update() {
  this.velocity.add(this.acceleration);
  this.position.add(this.velocity);

  this.acceleration.mult(0);                    Clearing acceleration after it's been applied
}
```

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 that changes over time, perhaps according to Perlin noise?

2.4 Dealing with Mass

OK. I've got one small (but fundamental) addition to make before integrating forces into the `Mover` class. After all, Newton's second law is really $\vec{\mathrm{F}} = \mathrm{M} \times \vec{\mathrm{A}}$, not $\vec{\mathrm{F}} = \vec{\mathrm{A}}$. Incorporating mass is as easy as adding an instance variable to the class, but I need to spend a little more time here because of another impending complication.

First I'll add mass.

```
class Mover {
  constructor(){
    this.position = createVector();
    this.velocity = createVector();
    this.acceleration = createVector();
    this.mass = ????;                          Adding mass as a number
  }
}
```

Units of Measurement

Now that I am introducing mass, it's important to make a quick note about units of measurement. In the real world, things are measured in specific units. Two objects are 3 meters apart, the baseball is moving at a rate of 90 miles per hour, or this bowling ball has a mass of 6 kilograms. As you'll see later in this book, sometimes you will want to take real-world units into consideration. However, in this chapter, I'm going to ignore them for the most part. The units of measurement are in pixels ("These two circles are 100 pixels apart") and frames of animation ("This circle is moving at a rate of 2 pixels per frame"). In the case of mass, there isn't any unit of measurement to use. I'm just going to make something up. In this example, I will arbitrarily picking the number 10. There is no unit of measurement, though you might enjoy inventing a unit of your own, like "1 moog" or "1 yurkle." It should also be noted that, for demonstration purposes, I'll tie mass to pixels (drawing, say, a circle with a radius of 10). This will allow me to visualize the mass of an object albeit inaccurately. In the real world size does not indicate mass. A small metal ball could have a much higher mass than a large balloon due to its higher density. And for two circular objects with equal density, the mass should be tied to the formula for area which is not as simple as the radius alone.

Mass is a scalar, not a vector, as it's just one number describing the amount of matter in an object. I could get fancy 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?"

```
constructor() {
  this.position = createVector(random(width), random(height));
  this.velocity = createVector(0, 0);
  this.acceleration = createVector(0, 0);
  this.mass = 10;
}
```

This isn't so great since things only become interesting once I have objects with varying mass, but it's enough to get us started. Where does mass come in? It's needed for applying Newton's second law to the object.

```
applyForce(force) {
  force.div(mass);                          Newton's second law (with force accumulation
  this.acceleration.add(force);             and mass)
}
```

Yet again, even though the code looks quite reasonable, there is a major problem here. Consider the following scenario with two `Mover` objects, both being blown away by a wind force.

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

let wind = createVector(1, 0);

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

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

`m1 equals wind force:     (1,0)`
`Divided by mass of 10:    (0.1,0)`

OK. Moving on to object `m2`. It also receives the wind force—(1,0). Wait. Hold on a second. What is the value of the 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 `p5.Vector`) 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 I don't want `m2` to receive a force divided by the mass of object `m1`. I want it to receive that force in its original state—(1,0). And so I must protect the original vector and make a copy of the it before dividing it by mass. Fortunately, the `p5.Vector` class has a convenient method for making a copy—`copy()`. `copy()` returns a new `p5.Vector` object with the same data. And so I can revise `applyForce()` as follows:

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

There's another way I could write the above function, using the static method `div()`. For help with this exercise, review static methods in Chapter 1 (see page 0).

Exercise 2.2

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

```
applyForce(force) {
  const f = _______._______(_______,_______);
  this.acceleration.add(f);
}
```

2.5 Creating Forces

Let's take a moment to recap what I've covered so far. I've defined what a force is (a vector), and how to apply a force to an object (divide it by mass and add it to the object's acceleration vector). What is missing? Well, I have yet to figure out how to calculate a force in the first place. Where do forces come from?

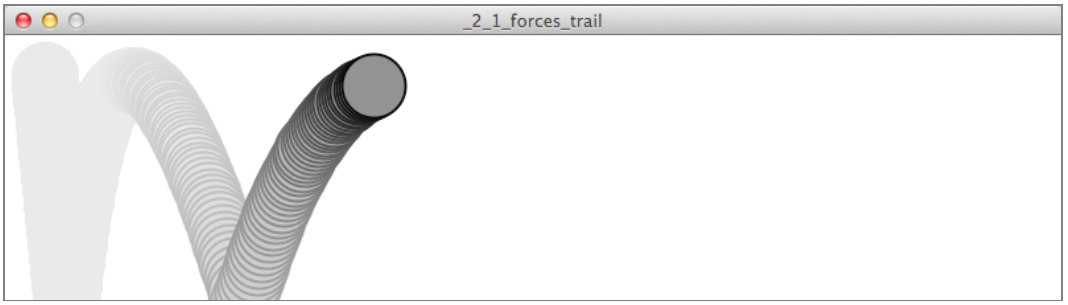
In this chapter, I'll look at two methods for creating forces in a p5.js world.

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. You can take these formulas, translate them into source code, and model real-world forces in JavaScript.

The easiest way to make up a force is to just pick a number (or two numbers really). 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 `mover`, the code would look like:

```
let wind = createVector(0.01, 0);
mover.applyForce(wind);
```

The result isn't terribly interesting, but it is a good place to start. I create a `p5.Vector` object, initialize it, and pass it into an object (which in turn will apply it to its own acceleration). To finish off this example, I'll add one more force–gravity (pointing down)–and only engage the wind force when the mouse is pressed.



Example 2.1: Forces

```
let gravity = createVector(0, 0.1);
mover.applyForce(gravity);

if (mouseIsPressed) {
  let wind = createVector(0.1, 0);
  mover.applyForce(wind);
}
```

Now I have two forces, pointing in different directions with different magnitudes, both applied to object `mover`. I'm beginning to get somewhere. I've built a world, an environment with forces that act on objects.

Let's look at what happens now when I add a second object with a variable mass. To do this, we'll need a quick review of object-oriented programming. Again, I'm not covering all the basics of programming here (for that you can check out any of the intro p5.js books or video tutorials 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 I left the `Mover` class. Notice how it is identical to the `Mover` class created in Chapter 1, with two additions—`mass` and a new `applyForce()` function.

```
class Mover {

  constructor() {
    this.mass = 1;                                    // And for now, we'll just set the mass equal to 1 for simplicity.
    this.position = createVector(width / 2, 30);
    this.velocity = createVector(0, 0);
    this.acceleration = createVector(0, 0);
  }
```

```
  applyForce(force) {
```
Newton's second law.

```
    let f = p5.Vector.div(force, this.mass);
    this.acceleration.add(f);
```
Receive a force, divide by mass, and add to acceleration.

```
  }

  update() {
```

```
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
```
Motion 101 from Chapter 1

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

  display() {
    stroke(0);
    fill(175);
```
Now add clearing the acceleration each time!

```
    ellipse(this.position.x, this.position.y,
this.mass * 16, this.mass * 16);
```
Scaling the size according to mass.

```
  }

  checkEdges() {
    if (this.position.x > width) {
      this.position.x = width;
      this.velocity.x *= -1;
    } else if (this.position.x < 0) {
      this.velocity.x *= -1;
      this.position.x = 0;
    }

    if (this.position.y > height) {
```
Somewhat arbitrarily, I have decided that an object bounces when it hits the edges of a window.

```
      this.velocity.y *= -1;
      this.position.y = height;
```
Even though I said not to touch position and velocity directly, there are some exceptions. Here I are doing so as a quick and easy way to reverse the direction of the object when it reaches the edge.

```
    }
  }
}
```

Now that the class is set, we can choose to create more than one `Mover` objects.

```
let moverA = new Mover();
let moverB = new Mover();
```

But there is an issue. Referring back to the `Mover` object's constructor...

```
constructor() {
```

```
  this.mass = 1;
  this.position = createVector(width / 2, 30);
```
Every object has a mass of 1 and a position of (width / 2, 30).

```
  this.velocity = createVector(0, 0);
  this.acceleration = createVector(0, 0);
}
```

...you will notice that every `Mover` object is made exactly the same way. What I want are `Mover` objects of variable mass that start at variable positions. A nice way to accomplish this is to add constructor arguments.

```
Mover(x, y, m) {
  this.mass = m;
  this.position = createVector(x, y);
```
Now setting these variables with arguments

```
  this.velocity = createVector(0, 0);
  this.acceleration = createVector(0, 0);
}
```

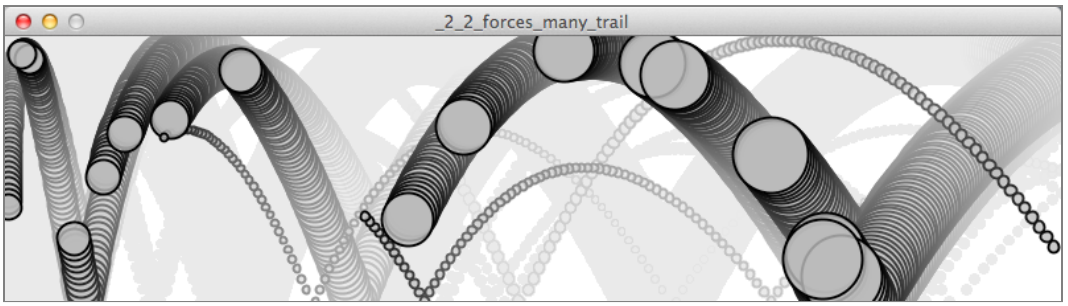
Notice how the mass and position are no longer set to hardcoded numbers, but rather initialized via arguments passed through the constructor. This means I can create a variety of `Mover` objects: big ones, small ones, ones that start on the left side of the screen, ones that start on the right, etc.

```
let moverA = new Mover(100, 30, 10);
```
A large Mover on the left side of the window

```
let moverB = new Mover(400, 30, 2);
```
A smaller Mover on the right side of the window

There are all sorts of ways I could choose to initialize the values (random, perlin noise, in a grid, etc.) and an array most likely makes more sense to manage large numbers of Mover objects. This is just a demonstration of the basic idea to get started and I'll introduce other techniques throughout this chapter.

Once the objects are declared and initialized, the rest of the code follows as before. For each object, pass the forces in the environment to `applyForce`, and enjoy the show.



Example 2.2: Forces acting on two objects

```
function draw() {
  background(51);

  let gravity = createVector(0, 0.1);          // Make up a gravity force and apply it.
  moverA.applyForce(gravity);
  moverB.applyForce(gravity);

  if (mouseIsPressed) {                        // Make up a wind force and apply when mouse is pressed.
    let wind = createVector(0.1, 0);
    moverA.applyForce(wind);
    moverB.applyForce(wind);
  }

  moverA.update();
  moverA.display();
  moverA.checkEdges();

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

Note how in the above image, the small circle reaches the bottom of the window faster than the larger one. This is because of our formula: *acceleration = force divided by mass*. The larger the mass, the smaller the acceleration. While such a difference makes sense for the wind force this is not physically accurate for a simulation of the earth's gravitational pull! I'll get into why and how to correct this in the next section.

Exercise 2.3

Instead of objects bouncing off the edge of the wall, create an example in which 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?

Exercise 2.x

Create many objects with array.

Exercise 2.x

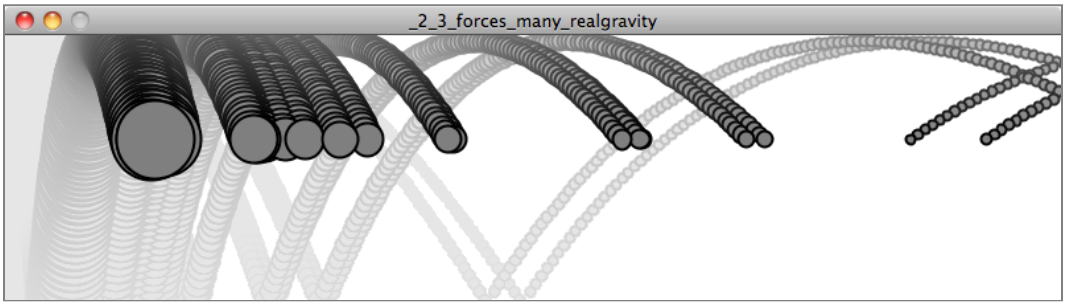
Fix the bouncing off the sides of the canvas so that the circle changes direction when it's edge hits the side, rather than its center.

Exercise 2.x

Create a wind force that is variable. Can you make it interactive, for example, think of a fan located where the mouse is and pointed towards the circles?

2.6 Gravity on Earth and Modeling a Force

I left the last section with a reference to the inaccuract of this example 2.3. The smaller the circle, the faster it falls. There is a logic to this; after all, according to Newton's second law 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 time. Why is this? I'll dive even deeper into this shortly, but the quick answer is that 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 canceled out when acceleration is divided by mass. A quick fix is to implement this in the sketch by multiplying the made-up gravity force by mass.



Example 2.3: Gravity scaled by mass

```
let gravity = createVector(0, 0.1);          Made-up gravity force
```

```
let gravityA = p5.Vector.mult(gravity,
moverA.mass);
moverA.applyForce(gravityA);
```

Scaled by mover A's mass

```
let gravityB = p5.Vector.mult(gravity,
moverB.mass);
moverB.applyForce(gravityB);
```

Scaled by mover B's mass

While the objects now fall at the same rate, because the strength of the wind force is independent of mass, when the mouse is pressed the smaller circle still accelerates to the right more quickly. ()‘ve also included a solultion to exercise 2.x in this example with the addition of a `radius` variable in the `Mover` class.)

Making up forces will actually get you quite far. The world of p5.js is an orchestra of pixels and you are its conductor. 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 I’m going to look at two forces—friction and gravity. The point I’d like to make here is not that friction and gravity are fundamental forces that you always need to have in your simulations. Rather, I want to demonstrate 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 you compute the force’s direction?
    - How do you compute the force’s magnitude?
- Translating that formula into p5.js code that calculates a vector to be sent through the `Mover's applyForce()` function.

If you can follow the above steps with two example forces I‘ll provide here, then hopefully when you 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 p5.js.

Dealing with formulas

OK, in a moment I'm going to write out the formula for friction. This won't be the first time you've seen a formula in this book; I just finished up the discussion of Newton's second law, $\vec{\mathrm{F}} = \mathrm{M} \times \vec{\mathrm{A}}$ (or force = mass * acceleration). You hopefully 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 I covered (without looking at the formula) in the Introduction (see page 0).

$$\sigma\sqrt{}$$

Formulas are regularly written with many symbols (often with letters from the Greek alphabet). Here's the formula for friction.

$$\vec{\mathrm{f}}\mathrm{riction} = -\mu * \mathrm{N} * \hat{\mathrm{v}}$$

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 I move on.

- ***Evaluate the right side, assign to the left side.*** This is just like in code! In the case above, I want to calculate the force of friction—the left side represents what I want to calculate and the right side elaborates on how to do it.
- ***Am I talking about a vector or a scalar?*** It's important to realize that in some cases, you'll be calculating a vector; in others, a scalar. For example, in this case the force of friction is a vector. That is indicated by the arrow above the word "friction." It has a magnitude and direction. The right side of the equation also has a vector, as indicated by the symbol {unitv}, which in this case stands for the velocity unit vector.
- ***When symbols are placed next to each other, this typically means multiply them.*** The formula above has four elements: -1, μ, *N*, and $\hat{\mathrm{v}}$. They should be multiplied together, reading the formula as: $\vec{\mathrm{f}}\mathrm{riction} = -1 * \mu * \mathrm{N} * \hat{\mathrm{v}}$

2.7 Friction

Let's begin with friction and follow the above steps.

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 simplification here, I am only going to look at the kinetic case.

Here's the formula for friction:

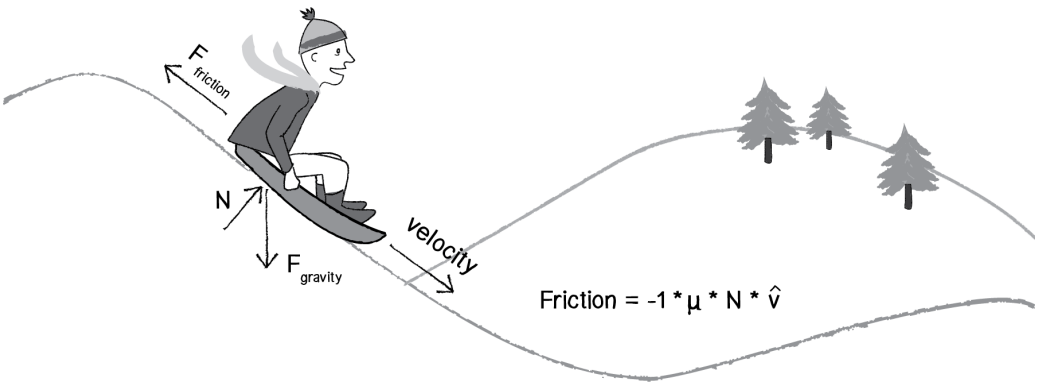


Figure 2.3

It's now time to separate this formula into two components that determine the direction of friction as well as the magnitude. The diagram above indicates that *friction points in the opposite direction of velocity*. In fact, that's the part of the formula that says -1 * $\hat{v}$, or -1 times the velocity unit vector. In p5.js, this would mean taking the velocity vector, normalizing it, and multiplying by -1.

```
let friction = this.velocity.copy();
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, as I don't want to reverse the object's direction by accident. Second, the vector is normalized. This is because the magnitude of friction is not associated with the speed of the

object, and I want to start with a vector of length 1 so that it can easily be scaled.

According to the formula, the magnitude is μ * N. μ, the Greek letter *mu* (pronounced "mew"), 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 this is a pretend p5.js world, I can arbitrarily set the coefficient based on how much friction I want to simulate.

```
let 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 you'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 magnitude and direction of the normal force is a bit more complicated because it doesn't point in the opposite direction of gravity. You'd need to know something about angles and trigonometry.

All of these specifics are important; however, a "good enough" simulation can be achieved without them. I can, for example, make friction work with the assumption that the normal force will always have a magnitude of 1. When I get into trigonometry in the next chapter, you could return to this question and make the friction example more sophisticated. Therefore:

```
let normal = 1;
```

Now that I have the magnitude and direction for friction, I can put it all together in code.

```
let c = 0.1;
let normal = 1;

let frictionMag = c * normal;                    // Calculate the magnitude of friction (really just an arbitrary constant).

let friction = mover.velocity.copy();
friction.mult(-1);
friction.normalize();

friction.mult(frictionMag);                      // Take the unit vector and multiply it by magnitude and this is the force vector!
```

The above code calculates a friction force, but doesn't answer the question of *when* to apply it? There is no answer to this question, of course, given this is all a made-up world visualized in a two dimensional p5.js canvas. I'll make the aribitrary, but logical, decision to apply friction

when the circle comes into contact with the bottom of the canvas by adding a function to the `Mover` class called `contactEdge()`

```
contactEdge() {
  return (this.position.y > height -
this.radius - 1);
}
```

The mover is touching the edge when it's within one pixel

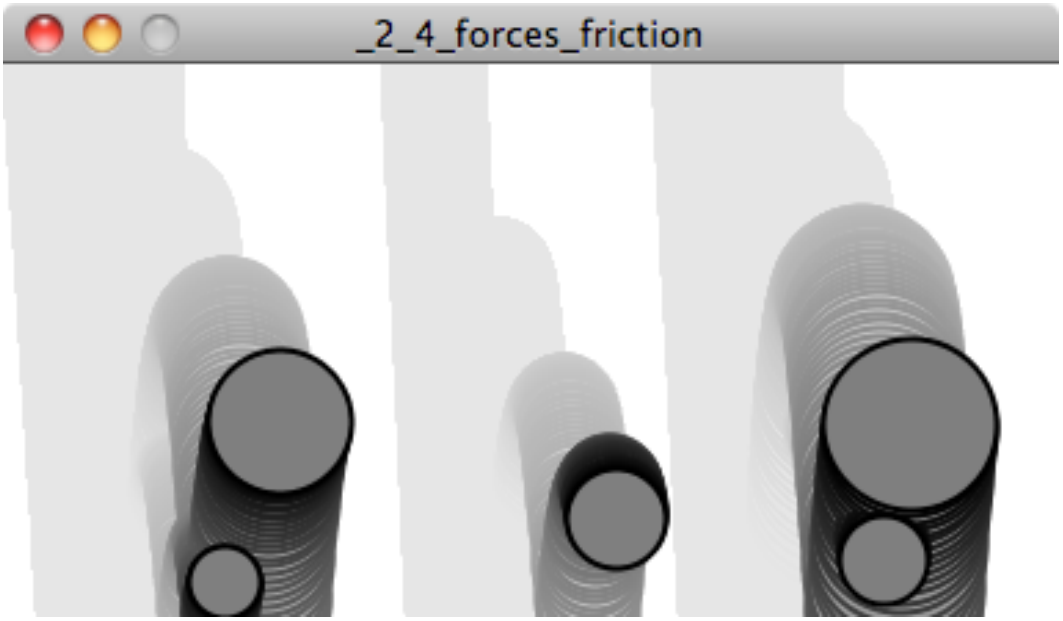
This is a good time for me to also mention that the actual bouncing off the edge here simulates an “idealized elsatic collision”, meaning no kinetic energy is lost when the circle and the edge collide. This is rarely true in the real world, pick up a tennis ball and drop it against any surface and the height at which it bounces will slowly lower until it rests against the ground. There are many factors at play here (including air resistence which I will cover in the next section), but a quick way to simulate an inelastic collision is by reducing the magnitude of velocity by a percentage with each bounce.

```
bounceEdges() {
  let bounce = -0.9;

  if (this.position.y > height - this.radius) {
    this.position.y = height - this.radius;
    this.velocity.y *= bounce;
  }
}
```

A new variable to simulate an inelastic collision

10% of the velocity's x or y component is lost

Finally, I can add all these pieces to the “forces” example, and simulate the object experiencing three forces: wind (when mouse is clicked), gravity (always), and now friction (when in contact with the bottom of the canvas):



The circle eventually rolls to a stop with friction.

Example 2.4: Including friction

```
function draw() {
  background(51);

  let gravity = createVector(0, 1);
  mover.applyForce(gravity);                    // I should scale by mass to be more accurate, but this example only has one circle

  if (mouseIsPressed) {
    let wind = createVector(0.5, 0);
    mover.applyForce(wind);
  }

  if (mover.contactEdge()) {
    let c = 0.1;
    let friction = mover.velocity.copy();
    friction.mult(-1);
    friction.setMag(c);

    mover.applyForce(friction);                 // Apply the friction force vector to the object.
```

```
    }

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

Running this example, you'll notice that the circle eventually comes to rest. You can make this happen more or less quickly by varying the coefficient of friction as well as the percentage speed loss in the `bounceEdges()` function.

Exercise 2.x

Add a second object to example 2.x. How do you handle having two objects of different masses? What if each object has its own coefficient of friction relative to the bottom surface? Does it make sense to encapsulate the friction force calculation into a Mover method? Try an array of mover objects.

Exercise 2.x

Instead of wind, can you add functionality to this example that allows you to toss the circle with mouse interaction?

2.8 Air and Fluid Resistance

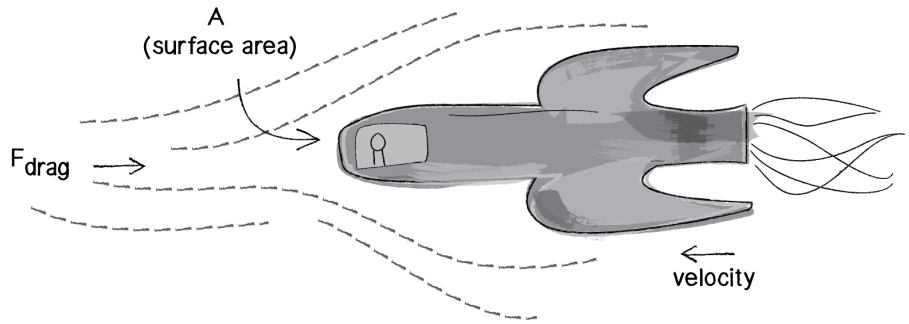


Figure 2.4

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 calculation of a drag force is slightly different. Let's look at the formula:

$$F_d = -\frac{1}{2}\rho v^2 A C_d \hat{v}$$

Now let's break this down and see what we really need for an effective simulation in p5, making a simpler formula in the process.

- F_d refers to *drag force*, the vector you ultimately want to compute and pass into the `applyForce()` method.
- - 1/2 is a constant: -0.5. This is irrelevant in terms of a p5.js world, as I will be making up values for other constants anyway. However, the fact that it is negative is important, as it indicates that the force points 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 you don't need to worry about at the moment. For my example I will consider this to have a constant value of 1.
- v refers to the speed of the object moving. OK, you've got this one! The object's speed is the magnitude of the velocity vector: `velocity.mag()`. And v^2 just means v squared or $v \times v$.
- 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, I will consider the 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 constant will determine the relative strength of the drag force.
- $\hat{v}$ 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 I've analyzed each of these components and determined what is needed for a simple simulation, I can reduce the formula to:

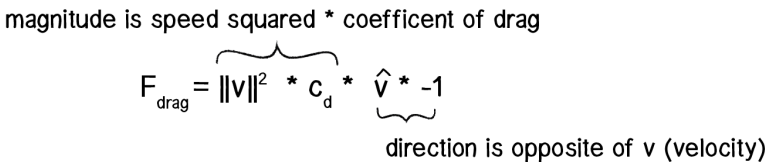


Figure 2.5: My simplified drag force formula

or:

```
let c = 0.1;
let speed = this.velocity.mag();
let dragMagnitude = c * speed * speed;    // Part 1 of the formula (magnitude): Cd * v2
let drag = this.velocity.copy();
drag.mult(-1);                            // Part 2 of the formula (direction): -1 * velocity
drag.normalize();
drag.mult(dragMagnitude);                 // Magnitude and direction together!
```

Let's implement this force in the `Mover` example with an added feature. For the friction example, I enabled the force whenever the object was in contact with the bottom edge of the canvas. Whenever the object was in contact, friction would slow it down. Here, I will introduce an element to the environment—a "liquid" that the mover passes through. The `Liquid` object will be drawn as a rectangle with position, 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 will a `display()` method and more.

```
class Liquid {

  constructor(x, y, w, h, c) {
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.c = c;                           // The liquid object includes a variable defining its coefficient of drag.
  }

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

}
```

The sketch also includes a variable `liquid` that is initialized in `setup()`.

```
let liquid;

function setup() {
  liquid = new Liquid(0, height/2, width, height/2, 0.1);   // Initialize a Liquid object. Note the coefficient 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 does the `Mover` object talk to the `Liquid` object? In other words, I want to implement the following:

When a mover passes through a liquid it experiences a drag force.

...or in object-oriented speak:

```
if (liquid.contains(mover) {
  liquid.drag(mover);
}
```

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

The above code serves as instructions for what I need to add to the `Liquid` class: (1) a function that determines if a `Mover` object is inside the `Liquid` object's area, and (2) a function that computes and applies a drag force on that mover.

The first is easy; I can use a Boolean expression to determine if the position vector rests inside the rectangle defined by the liquid.

```
contains(mover) {
  let pos = mover.position;
  return (pos.x > this.x && pos.x < this.x + this.w && pos.y > this.y && pos.y < this.y +
this.h);
}
```

This Boolean expression determines if the position vector is contained within the rectangle defined by the Liquid class.

The `drag()` function requires a bit more complexity; however, I've written the code for it already. This is an implementation of the drag 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*.

```
drag(mover) {
 let speed = mover.velocity.mag();
 let dragMagnitude = this.c * speed * speed;
 let drag = mover.velocity.copy();
 drag.setMag(dragMagnitude);
 mover.applyForce(drag);
}
```

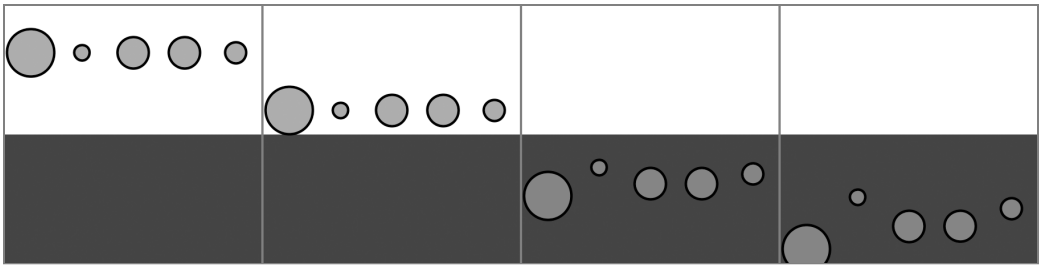
The force's magnitude: Cd * v~2~

The force's direction: -1 * velocity

Finalize the force: magnitude and direction together.

Apply the force.

And with these two functions added to the `Liquid` class, I'm ready to put it all together! In the following example, I'll expand the code to use an array (with mover objects spaced out evenly) to demonstrate how the drag force behaves with objects of variable mass.



Example 2.5: Fluid Resistance

```
let movers = [];

let liquid;

function setup() {
  createCanvas(360, 640);
  for (let i = 0; i < 10; i++) {
    let mass = random(0.1, 5);
    movers[i] = new Mover(i * 20, 0, mass);
  }
  liquid = new Liquid(0, height/2, width,
height/2, 0.1);
}

function draw() {
  background(255);

  liquid.display();

  for (let i = 0; i < movers.length; i++) {
    if (liquid.contains(movers[i])) {
      liquid.drag(movers[i]);
    }
    let m = 0.1 * movers[i].mass;
    let gravity = createVector(0, m);
    movers[i].applyForce(gravity);

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

Initializee an array of Mover objects

Random mass

x value is spaced out evenly according to i

Note that gravity is scaled according to mass.

Running the example, you may notice that it appears to simulate objects falling into water.

The objects only slow down when crossing through 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 is the "slowing down" due to drag. The smaller objects slow down at a greater rate than the larger ones.

Exercise 2.5

Take a look at the 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 (velocity of zero) experiences no drag at all. Expand the example to drop the objects from variable height. How does this affect the drag as they hit the liquid?

Exercise 2.6

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 2.7

In addition to drag being a force in opposition to the velocity vector, a drag force can be also be perpendicular. 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 (this is Newton's third law). And there is a formula for calculating the strengths of these forces, as depicted in Figure 2.6.

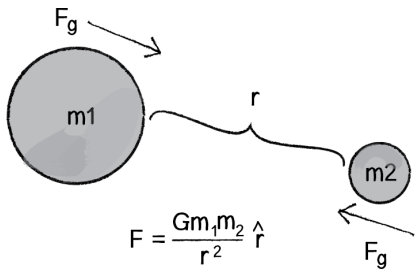


Figure 2.6

Let's examine this formula a bit more closely.

- `F` refers to the gravitational force, the vector to compute and pass into the `applyForce()` function.
- `G` is the *universal gravitational constant*, which in our world equals 6.67428 x 10^{-11} meters cubed per kilogram per second squared. This is a pretty important number if you are a human being. It's not an important number if you are a shape wandering around a p5.js canvas. Again, it's a constant can be used to to make the forces in the world stronger or weaker. 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 I initially did with Newton's second law ($\vec{F} = M \times \vec{A}$), mass is also something I could choose to ignore. After all,

  shapes drawn on the screen don't have a physical mass. However, if you keep track of this value, you can create more interesting simulations in which "bigger" objects exert a stronger gravitational force than smaller ones.
- $\hat{r}$ refers to the unit vector pointing from object 1 to object 2. As you'll see in a moment, this direction vector can be computed by subtracting the position 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 `G`, big force. Now, when you divide by something, the the opposite occurs. The strength of the force is inversely proportional to the distance squared. The *farther away* an object is, the *weaker* the force; the *closer*, the *stronger*.

Hopefully by now the formula makes some sense. I've shown you a diagram and dissected the individual components of the formula. Now it's time to figure out how to translate the math into p5.js code. Let's make the following assumptions.

We have two objects, and:

1. Each object has a position: `position1` and `position2`.
2. Each object has a mass: `mass1` and `mass2`.
3. There is a variable `G` for the universal gravitational constant.

Given these assumptions, I want to compute a vector, the force of gravity. I'll do it in two parts. First, I'll compute the direction of the force $\hat{r}$ in the formula above. Second, I'll calculate the strength of the force according to the masses and distance.

Remember in Chapter 1 (see page 0), when I created an example of an object accelerating towards the mouse? (See Figure 2.7.)

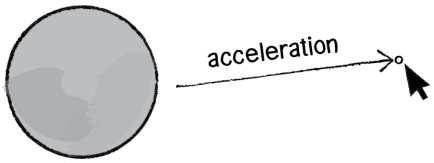


Figure 2.7

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

```
let dir = p5.Vector.sub(mouse, position);
```

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

```
let dir = p5.Vector.sub(position1, position2);
dir.normalize();
```

Don't forget that since you want a unit vector, a vector that indicates direction only, it's important to *normalize* the vector after subtracting the positions. (Later, I might skip this step and use `setMag()` instead.)

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

```
let magnitude = (G * mass1 * mass2) / (distance * distance);
dir.mult(magnitude);
```

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

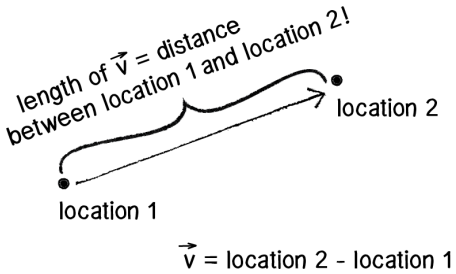


Figure 2.8

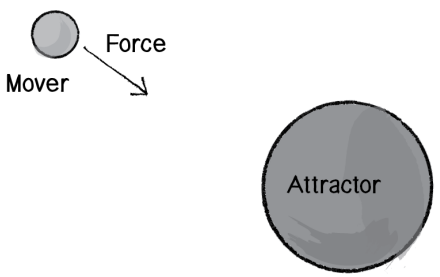
Indeed, if I add one more line of code and grab the magnitude of that vector before normalizing it, then I'll have the distance. And this time, I'll skip the `normalize()` step and use `setMag()`.

`let force = p5.Vector.sub(position1, position2);`	The vector that points from one object to another
`const distance = force.mag();`	The length (magnitude) of that vector is the distance between the two objects.
`let magnitude = (G * mass1 * mass2) / (distance * distance);`	Use the formula for gravity to compute the strength of the force.
`force.setMag(magnitude);`	Normalize and scale the force vector to the appropriate magnitude.

Note that I also renamed the vector `dir` to `force`. After all, when the calculations are finished, the vector I started with ends up being the actual force vector I wanted all along.

Now that I've worked out the math and code for calculating an attractive force (emulating gravity), let's turn our attention to applying this technique in the context of an actual p5.js sketch. In Example 2.1, I set up the foundation for all of these examples, a `Mover` class—a template for making objects with `p5.Vector` objects for position, velocity, and acceleration as well as an `applyForce()` method. 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 position).



The `Mover` object will experience a gravitational pull towards the `Attractor` object, as illustrated in Figure 2.9.

Figure 2.9

I'll start with the most basic `Attractor` class—giving it a position and a mass,

along with a function to display itself (tying mass to size).

```
class Attractor {
  constructor() {                                    The Attractor is an object that doesn't move. I just
    this.position = createVector(width/2, height/    need a mass and a position.
2);
    this.mass = 20;
  }

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

And in the sketch, I can add an instance of the `Attractor` class.

```
let mover;
let attractor;

function setup() {
  createCanvas(640, 360);
  mover = new Mover(300, 100, 5);
  attractor = new Attractor();                       Initialize Attractor object.
}

function draw() {
  background(255);

  attractor.display();                               Display Attractor object.
  mover.update();
  mover.display();
}
```

This is a good start: a sketch with a `Mover` object and an `Attractor` object, made from classes that 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 these two objects communicate with each other?

There are a number of ways this could be done. Here are just a few possibilities.

Task	Function
1. A function that receives both an Attractor and a Mover:	`attraction(attractor, mover);`
2. A function in the Attractor class that receives a Mover:	`attractor.attract(mover);`
3. A function in the Mover class that receives an Attractor:	`mover.attractedTo(attractor);`
4. A function in the Attractor class that receives a Mover and returns a p5.Vector, which is the attraction force. That attraction force is then passed into the Mover's applyForce() function:	`let force = attractor.attract(mover);` `mover.applyForce(force);`

and so on. . .

It's good to look at a range of options, and you could probably make arguments for each of the above possibilities. I'd like to at least discard the first one, since I lean towards an object-oriented approach which in my view is a better choice over an arbitrary function not tied to either the `Mover` or `Attractor` class. Whether you pick option 2 or option 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 the exmaples here. After all, I spent a lot of time working out the `applyForce()` function, and I think the examples are clearer continuing with the same methodology.

In other words, where I once wrote:

```
let force = createVector(0, 0.1);                Made-up force
mover.applyForce(force);
```

I now have:

```
let force = attractor.attract(mover);            Attraction force between two objects
mover.applyForce(force);
```

And so the `draw()` function can be written as:

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

  let force = attractor.attract(mover);          Calculate attraction force and apply it.
  mover.applyForce(force);
```

```
    mover.update();

    attractor.display();
    mover.display();
  }
```

I'm almost there. Since I decided to put the `attract()` function inside of the `Attractor` class, I still need to actually write that function. The function should receive a `Mover` object and return a Vector, i.e.:

```
  attract(m) {
    return _____________;                                  all the math
  }
```

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

```
  attract(mover) {
    let force = p5.Vector.sub(this.position,               What's the force's direction?
  mover.position);
    let distance = force.mag();
    float strength = (this.G * this.mass * m.mass)         What's the force's magnitude?
  / (distance * distance);
    force.setMag(strength);

    return force;                                          Return the force so that it can be applied!
  }
```

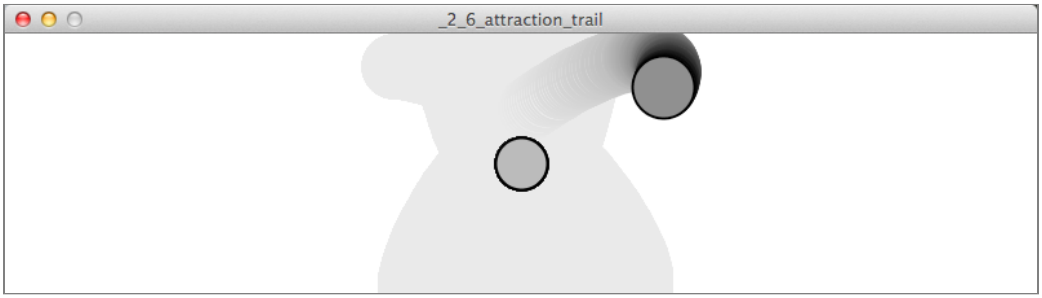
And I'm done. Sort of. Almost. There's one small kink I need to work out. Let's look at the above code again. See that symbol for divide, the slash? Whenever you have one of these, you should ask yourself the question: What would happen if the distance happened to be a really, really small number or (even worse!) zero??! Well, you can't divide a number by 0, and if you 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 p5.js is not the real world. And in the p5.js world, the mover could end up being very, very close to the attractor and the force could become so strong the mover would fly way off the canvas. And so with this formula, it's can be useful to be practical and constrain the range of what `distance` can actually be. Maybe, no matter where the `Mover` actually is, you 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 that you need to constrain the minimum distance, it's useful to do the same with the maximum. After all, if the mover were to be, say, 500 pixels from the attractor (not unreasonable), that is the equivalent of dividing the force by 250,000. That force might end up being so weak that it's almost as if it's not applied 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.

The `Mover` class hasn't changed at all, so let's just look at the main sketch and the `Attractor` class as a whole, adding a variable `G` for the universal gravitational constant. (On the website, you'll find that this example also has code that allows you to move the `Attractor` object with the mouse.)



Example 2.6: Attraction

```
let mover;
let attractor;
```

A Mover and an Attractor

```
function setup() {
  size(640, 360);
  mover = new Mover(300, 100, 5);
  attractor = new Attractor();
}

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

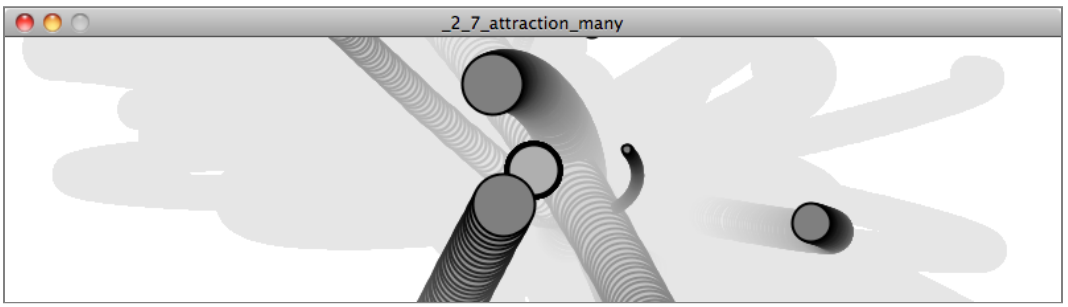
```
  const force = a.attract(m);
  mover.applyForce(force);
```

Apply the attraction force from the Attractor on the Mover.

```
    mover.update();

    attractor.display();
    mover.display();
  }

  class Attractor {

    constructor() {
      this.position = createVector(width/2, height/2);
      this.mass = 20;
      this.G = 0.4;
    }

    attract(mover) {
      let force = p5.Vector.sub(position, mover.position);
      let distance = force.mag();
      distance = constrain(distance, 5, 25);
      let strength = (this.G * this.mass * m.mass) / (distance * distance);
      force.setMag(strength);
      return force;
    }

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

Remember, we need to constrain the distance so that our circle doesn't spin out of control.

And you could, of course, expand this example to use an array for many `Mover` objects, just as I did with drag:



Example 2.7: Attraction with many Movers

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

let attractor;

function setup() {
  size(640, 360);
  for (let i = 0; i < 10; i++) {                          Each Mover is initialized randomly.
    movers[i] = new Mover(random(width), random(height), random(0.1, 2));
  }
  attractor = new Attractor();
}

function draw() {
  background(255);

  aattractor.display();

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

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

}
```

Exercise 2.8

In the example above, there is a system (i.e. array) of `Mover` objects and one `Attractor` object. Build an example that has systems of both 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/) for an example.

Exercise 2.9

It's worth noting that gravitational attraction is a model you can follow to invent your 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 faraway objects, but repel close ones?

2.10 Everything Attracts (or Repels) Everything

Hopefully, you found it helpful that I started with a simple scenario—*one object attracts another object*–and moved 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).

You've really done almost all of the work for this already. Let's consider a p5.js sketch with an array of `Mover` objects:

```
let movers = [];

function setup() {
  createCanvas(640, 360);
  for (let i = 0; i < 10; i++) {
    movers[i] = new Mover(random(width), random(height), random(0.1, 2));
  }
}

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

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

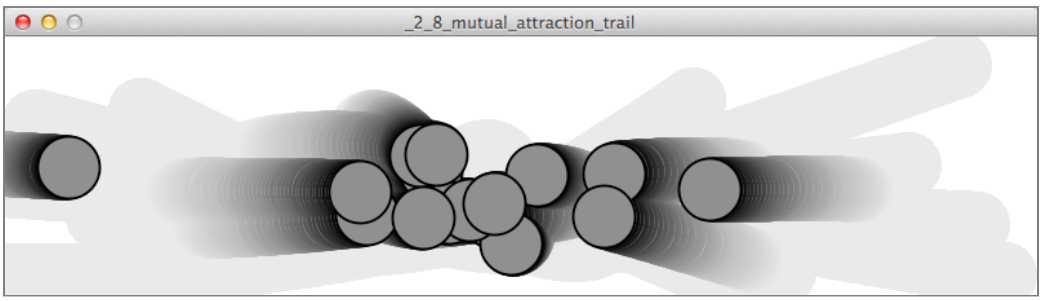
To do this, I need to nest a second loop.

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

In the previous example, I had an `Attractor` object with a function named `attract()`. Now, since there are movers attracting movers, all I need to do is copy the `attract()` function into the `Mover` class itself.

```
class Mover {

                                                    [inline] All the other stuff from before plus. . .

  attract(other) {                                  The Mover now knows how to attract another
                                                    Mover.
    const force = p5.Vector.sub(this.position, other.position);
    let distance = force.mag();
    distance = constrain(distance, 5, 25);
    force.normalize();

    let strength = (this.G * this.mass *            Note to add a value for gravity in the Mover
m.mass) / (distance * distance);                    class: this.G
    force.setMag(strength);
    return force;
  }
}
```

Of course, there's one small problem. When every mover `i` attracts every mover `j`, what about when `i` equals `j`? Should mover #3 attract mover #3? The answer, of course, is no. If there are five objects, you only want mover #3 to attract 0, 1, 2, and 4, skipping itself. And so, I finish this example by adding a conditional statement to skip applying the force when i equals j.



Example 2.8: Mutual attraction

```
let movers = [];

function setup() {
  createCanvas(640, 360);
  for (let i = 0; i < 20; i++) {
    movers[i] = new Mover(random(width), random(height), random(0.1, 2));
  }
}

function draw() {
  background(255);

  for (let i = 0; i < movers.length; i++) {
    for (let j = 0; j < movers.length; j++) {
      if (i !== j) {                                    Don't attract yourself!
        let force = movers[j].attract(movers[i]);
        movers[i].applyForce(force);
      }
    }
    movers[i].update();
    movers[i].display();
  }
}
```

Exercise 2.10

Change the attraction force in Example 2.8 to a repulsion force. Can you create an example in which 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) for the creature to interact 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?