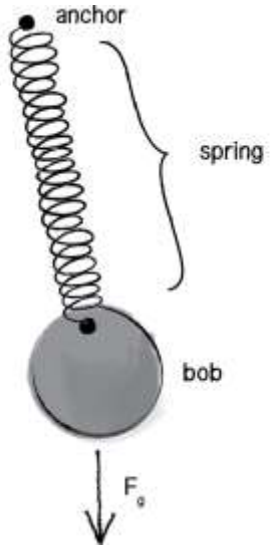
Spring forces

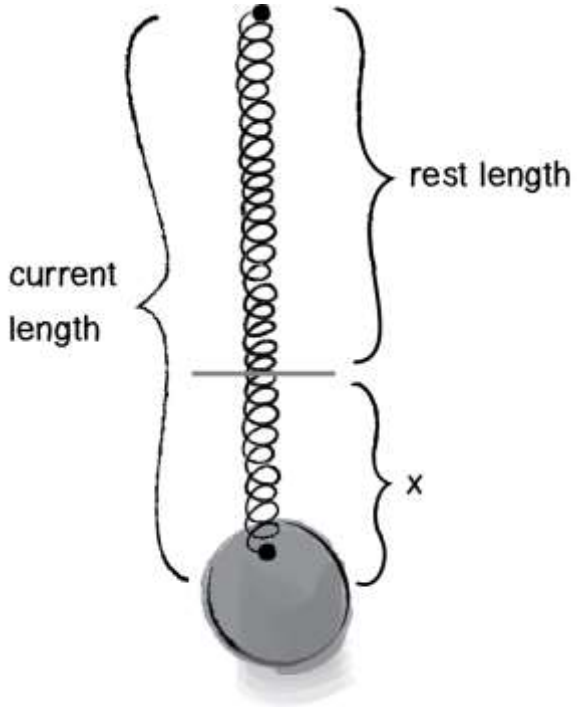
In the beginning of this section, we looked at modelling simple harmonic motion by mapping the sine wave to a pixel range, and had you model a bob on a spring using that sine wave. While using the `sin()` function is a quick-and-dirty, one-line-of-code way of getting something up and running, it won't do if what we really want is to have a bob hanging from a spring in a two-dimensional space that responds to other forces in the environment (wind, gravity, etc.) To accomplish a simulation like this (one that is identical to the pendulum example, only now the arm is a springy connection), we need to model the forces of a spring using `PVector`.



The force of a spring is calculated according to Hooke's law, named for Robert Hooke, a British physicist who developed the formula in 1660. Hooke originally stated the law in Latin: "Ut tensio, sic vis," or "As the extension, so the force." Let's think of it this way:

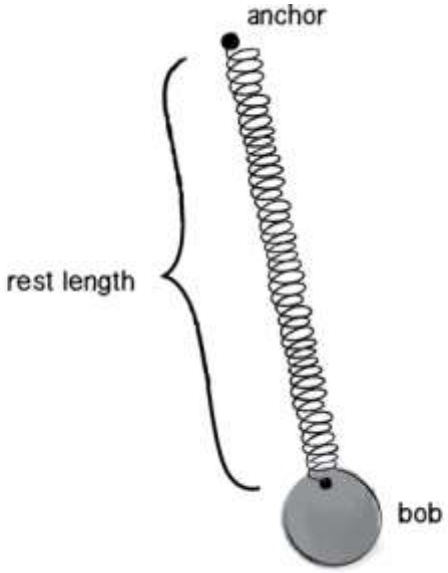
The force of the spring is directly proportional to the extension of the spring.

In other words, if you pull on the bob a lot, the force will be strong; if you pull on the bob a little, the force will be weak. Mathematically, the law is stated as follows:



F_spring = -k \times x$F_{spring} = -k \times x$F, start subscript, s, end subscript, p, r, i, n, g, equals, minus, k, times, x

- `k` is constant and its value will ultimately scale the force. Is the spring highly elastic or quite rigid?
- `x` refers to the displacement of the spring, i.e. the difference between the current length and the rest length. The rest length is defined as the length of the spring in a state of equilibrium.
  Now remember, force is a vector, so we need to calculate both magnitude and direction. Let’s look at one more diagram of the spring and label all the givens we might have in a program.



Let's establish the following three starting variables as shown in the diagram above, with some reasonable values.

```
var anchor = new PVector(100, 10);
var bob = new PVector(110, 100);
var restLength = 20;
```

First, let's use Hooke's law to calculate the magnitude of the force. We need to know `k` and `x`. `k` is easy; it's just a constant, so let's make something up.

```
var k = 0.1;
```

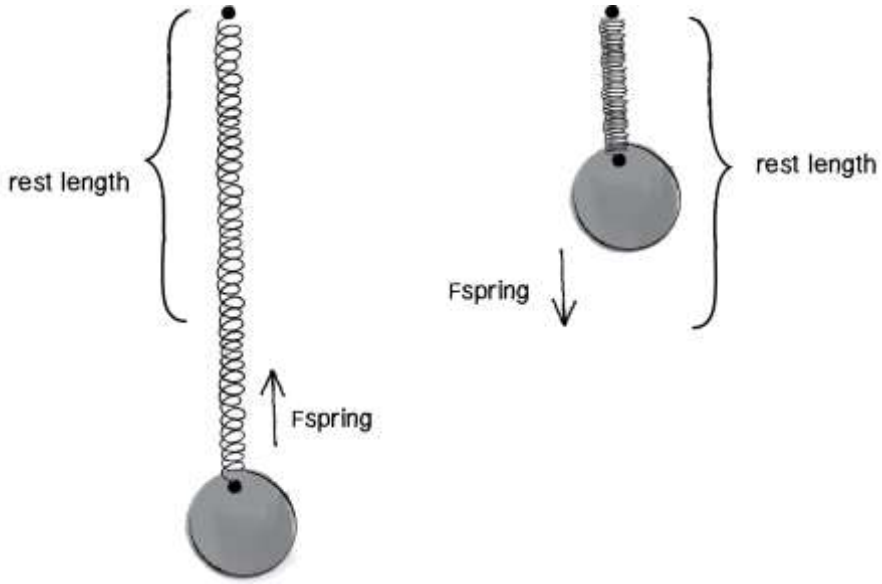
`x` is perhaps a bit more difficult. We need to know the "difference between the current length and the rest length." The rest length is defined as the variable `restLength`. What's the current length? The distance between the anchor and the bob. And how can we calculate that distance? How about the magnitude of a vector that points from the anchor to the bob? (Note that this is exactly the same process we employed when calculating distance in the Gravitational Attraction section.)

```
var dir = PVector.sub(bob, anchor);
```

```
var currentLength = dir.mag();

var x = restLength - currentLength;
```

Now that we've sorted out the elements necessary for the magnitude of the force `(-1 * k * x)`, we need to figure out the direction, a unit vector pointing in the direction of the force. The good news is that we already have this vector. Right? Just a moment ago we thought to ourselves: "How we can calculate that distance? How about the magnitude of a vector that points from the anchor to the bob?" Well, that same vector is the direction of the force!



In the diagram above, we can see that if we stretch the spring beyond its rest length, there should be a force pulling it back towards the anchor. And if it shrinks below its rest length, the force should push it away from the anchor. This reversal of direction is accounted for in the formula with the -1. And so all we need to do is normalize the `PVector` we used for the distance calculation! Let's take a look at the code and rename that `PVector` variable as "force."

```
var k = 0.01;

var force = PVector.sub(bob, anchor);

var currentLength = force.mag();
```

```
var x = restLength - currentLength;
// Direction of spring force, a unit vector
force.normalize();
// Putting it together: direction and magnitude!
force.mult(-1 * k * x);
```

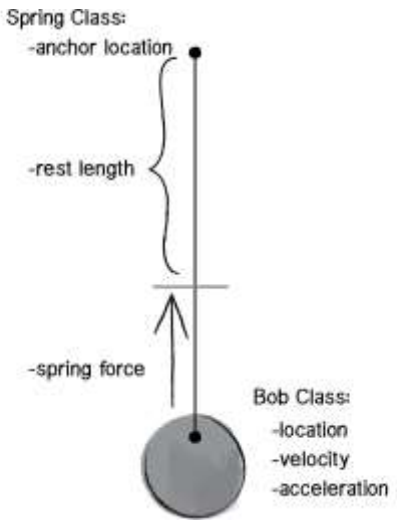
Now that we have the algorithm worked out for computing the spring force vector, the question remains: what object-oriented programming structure should we use? This, again, is one of those situations in which there is no “correct” answer. There are several possibilities; which one we choose depends on the program’s goals and one’s own personal coding style. Still, since we’ve been working all along with a `Mover` object, let’s keep going with this same framework. Let’s think of our `Mover` object as the spring’s “bob.” The bob needs location, velocity, and acceleration vectors to move about the screen. Perfect—we’ve got that already! And perhaps the bob experiences a gravity force via the `applyForce()` method. Just one more step—we need to apply the spring force:

```
var bob = new Bob();

draw = function()  {
  // Our “make-up-a-gravity force”
  var gravity = new PVector(0, 1);
  bob.applyForce(gravity);
  // We need to also calculate and apply a spring force!
  var springForce = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_????
  bob.applyForce(spring);

  // Our standard update() and display() methods
  bob.update();
  bob.display();
```

```
};
```



One option would be to write out all of the spring force code in the main `draw()` loop. But thinking ahead to when you might have multiple bobs and multiple spring connections, it makes a good deal of sense to write an additional object, a `Spring` object. As shown in the diagram above, the `Bob` object keeps track of the movements of the bob; the `Spring` object keeps track of the spring's anchor and its rest length and calculates the spring force on the bob.

This allows us to write this lovely code to tie them together:

```
var bob = new Bob();
var spring = new Spring();


draw = function()  {
  // Our "make-up-a-gravity force"
  var gravity = new PVector(0, 1);
  bob.applyForce(gravity);
  // Spring.connect will take care of computing and applying the spring
force
  spring.connect(bob);
```

```
  // Our standard update() and display() methods
  bob.update();
  bob.display();
};
```

You may notice here that this is quite similar to what we did in the Gravity section with an attractor. There, we said something like:

```
var force = attractor.calculateAttraction(mover);
mover.applyForce(force);
```

The analogous situation here with a spring would be:

```
var force = spring.calculateForce(bob);
bob.applyForce(force);
```

Nevertheless, in this example, all we did was:

```
spring.connect(bob);
```

What gives? Why don't we need to call `applyForce()` on the bob? The answer is, of course, that we do need to call `applyForce()` on the bob. Only instead of doing it in `draw()`, we're just demonstrating that a perfectly reasonable (and sometimes preferable) alternative is to ask the `connect()` method to internally handle calling `applyForce()` on the bob.

```
Spring.prototype.connect(bob) {
  var force = /* some fancy calculations */;
  bob.applyForce(force);
};
```

Why do it one way with the `Attractor` object and another way with the `Spring` object? When we were first learning about forces, it was a bit

clearer to show all the forces being applied in the main `draw()` loop, and hopefully this helped you learn about force accumulation. Now that we're more comfortable with that, perhaps it's simpler to embed some of the details inside the objects themselves.

Let's put it all together, in the program embedded below. We've added a few things: (1) the `Bob` object includes functions for mouse interactivity so that the bob can be dragged around the window, and (2) the `Spring` object includes a function to constrain the connection's length between a minimum and a maximum.

```
var Spring = function(x, y, l) {
  this.anchor = new PVector(x, y);
  this.restLength = l;
  this.k = 0.2;
};


// Calculate and apply spring force
Spring.prototype.connect = function(b) {
  // Vector pointing from anchor to bob location
  var force = PVector.sub(b.position, this.anchor);
  // What is distance
  var d = force.mag();
  // Stretch is difference between current distance and rest length
  var stretch = d - this.restLength;


  // Calculate force according to Hooke's Law
  // F = k * stretch
  force.normalize();
  force.mult(-1 * this.k * stretch);
```

```
  b.applyForce(force);
};

// Constrain the distance between bob and anchor between min and max
Spring.prototype.constrainLength = function(b, minLength, maxLength) {
  var dir = PVector.sub(b.position, this.anchor);
  var d = dir.mag();
  // Is it too short?
  if (d < minLength) {
    dir.normalize();
    dir.mult(minLength);
    // Reset location and stop from moving (not realistic physics)
    b.position = PVector.add(this.anchor, dir);
    b.velocity.mult(0);
    // Is it too long?
  }
  else if (d > maxLength) {
    dir.normalize();
    dir.mult(maxLength);
    // Reset location and stop from moving (not realistic physics)
    b.position = PVector.add(this.anchor, dir);
    b.velocity.mult(0);
  }
};

Spring.prototype.display = function() {
  stroke(0);
```

```
  fill(175);
  strokeWeight(2);
  rectMode(CENTER);
  rect(this.anchor.x, this.anchor.y, 10, 10);
};

Spring.prototype.displayLine = function(b) {
  strokeWeight(2);
  stroke(0);
  line(b.position.x, b.position.y, this.anchor.x, this.anchor.y);
};

// Bob object, just like our regular Mover (location, velocity, acceleration, mass)

var Bob = function(x, y) {
  this.position = new PVector(x,y);
  this.velocity = new PVector();
  this.acceleration = new PVector();
  this.mass = 24;
  // Arbitrary damping to simulate friction / drag
  this.damping = 0.98;
  // For user interaction
  this.dragOffset = new PVector();
  this.dragging = false;
};

// Standard Euler integration
```

```
Bob.prototype.update = function() {
   this.velocity.add(this.acceleration);
   this.velocity.mult(this.damping);
   this.position.add(this.velocity);
   this.acceleration.mult(0);
};

// Newton's law: F = M * A
Bob.prototype.applyForce = function(force) {
   var f = force.get();
   f.div(this.mass);
   this.acceleration.add(f);
};

// Draw the bob
Bob.prototype.display = function() {
   stroke(0);
   strokeWeight(2);
   fill(175);
   if (this.dragging) {
     fill(50);
   }
   ellipse(this.position.x, this.position.y, this.mass*2, this.mass*2);
};

Bob.prototype.handleClick = function(mx, my) {
   var d = dist(mx, my, this.position.x, this.position.y);
```

```
  if (d < this.mass) {
    this.dragging = true;
    this.dragOffset.x = this.position.x-mx;
    this.dragOffset.y = this.position.y-my;
  }
};

Bob.prototype.stopDragging = function() {
  this.dragging = false;
};

Bob.prototype.handleDrag = function(mx, my) {
  if (this.dragging) {
    this.position.x = mx + this.dragOffset.x;
    this.position.y = my + this.dragOffset.y;
  }
};

// Create objects at starting location
// Note third argument in Spring constructor is "rest length"
var bob = new Bob(width/2, 100);
var spring = new Spring(width/2, 10, 100);

var draw = function()  {
 background(255);
 // Apply a gravity force to the bob
 var gravity = new PVector(0,2);
```

```
  bob.applyForce(gravity);

  // Connect the bob to the spring (this calculates the force)
  spring.connect(bob);
  // Constrain spring distance between min and max
  spring.constrainLength(bob, 30, 200);

  // Update bob
  bob.update();

  // Draw everything
  spring.displayLine(bob); // Draw a line between spring and bob
  bob.display();
  spring.display();

  fill(0);
  text("click on bob to drag", 10, height-5);
};

mousePressed = function() {
  bob.handleClick(mouseX, mouseY);
};

mouseDragged = function() {
  bob.handleDrag(mouseX, mouseY);
};
```

```
mouseReleased = function() {
  bob.stopDragging();
};
```