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 six 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’re looking for. What I want to focus on is 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 *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.

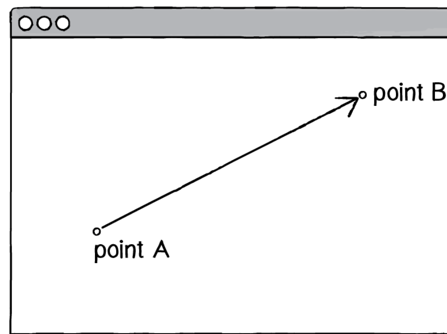
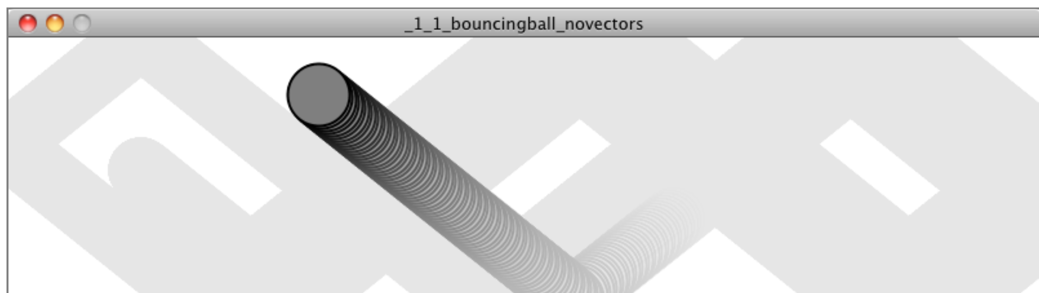


Figure 1.1 A vector (drawn as an arrow) has magnitude (length of arrow) and direction (which way it is pointing).

In the above illustration, 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, I'd like to create a basic p5.js example that demonstrates why you should care about vectors in the first place. If you've read any of the introductory p5.js textbooks or taken an introduction to creative coding course (and hopefully you've done one of these things to help prepare you for this book), you probably, at one point or another, learned 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 canvas. Motion, of course, is a key element of the 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 linked from the website.

Example 1.1: Bouncing ball with no vectors

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

Variables for position and speed of ball.

```
function setup() {
  createCanvas(640, 360);
  background(255);
}
```

Remember how p5 works? setup() is executed once when the sketch starts and draw() loops forever and ever (until you quit).

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

  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 position (x,y).

```
}
```

In the above example, there is 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.

position	*x and y*
Speed	*xspeed and yspeed*

In a more sophisticated sketch, you might have many more variables:

Acceleration	*xacceleration and yacceleration*
Target position	*xtarget and ytarget*
Wind	*xwind and ywind*
Friction	*xfriction and yfriction*

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

Wouldn't it be nice if I could simplify the code and use fewer variables?

Instead of:

```
let x;
let y;
let xspeed;
let yspeed;
```

I would love to have…

```
let position;
let speed;
```

Taking this first step in using vectors won't allow me 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, I'm going to stick to two dimensions for quite some time (at least the first several chapters). All of these examples can be fairly easily extended to three dimensions (and the class I will use—`p5.Vector`—allows for three dimensions.) However, it's easier to start with just two.

1.2 Vectors for p5.js 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:

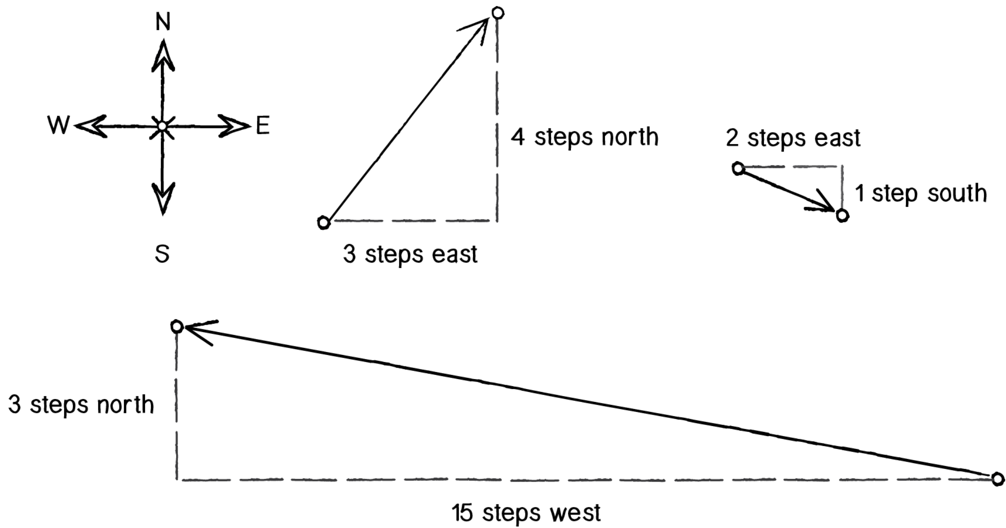


Figure 1.2

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

You've probably done this before when programming motion. For every frame of animation (i.e. a single cycle through p5'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.

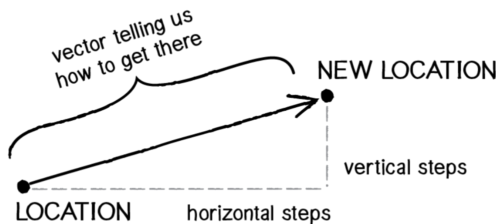


Figure 1.3

For every frame:

new position = velocity applied to current position

If velocity is a vector (the difference between two points), what is position? Is it a vector too? Technically, you could argue that position is not a vector, since it's not describing how to move from one point to another—it's describing a singular point in space.

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

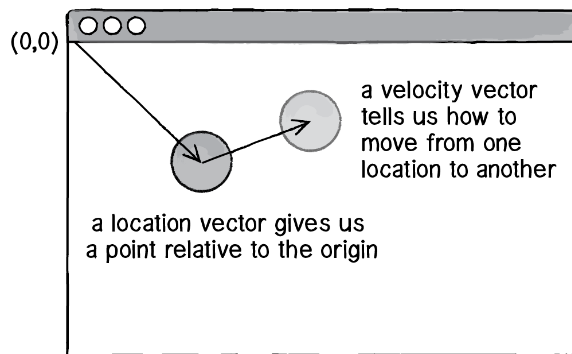


Figure 1.4

Let's examine the underlying data for both position and velocity. In the bouncing ball example, I had the following:

position	*x,y*
velocity	*xspeed,yspeed*

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

```
class Vector {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

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

And so this …

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

becomes …

```
let position = createVector(100, 100);
let velocity = createVector(1, 3.3);
```

I'll note that in the above code the vector objects are not created, as you might expect, by invoking a constructor function. Instead of `new Vector(x, y)` (or more accurately in p5 `new p5.Vector(x, y)`), `createVector(x, y)` is called. The createVector() function is included in a p5.js as a helper function to take care of some details behind the scenes as well as simplify the code. Except in special circumstances, `p5.Vector objects` should always be created with `createVector()`.

Now that I have two vector objects (position and velocity), I'm ready to implement the algorithm for motion—***position = position + velocity***. In Example 1.1, without vectors, I had:

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

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

```
position = position + velocity;     // Add the velocity vector to the position vector.
```

However, in JavaScript, the addition operator + is reserved for primitive values (integers, floats, etc.) only. JavaScript doesn't know how to add two `p5.Vector` objects together any more than it knows how to add two `p5.Font` objects or `p5.Image` objects. Fortunately, the `p5.Vector` class includes functions for common mathematical operations.

1.3 Vector Addition

Before I continue looking at the `p5.Vector` class and its `add()` method (purely for the sake of learning since it's already implemented for us in p5.js 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: $\vec{v}$
- Scalar: x

Let's say I have the following two vectors:

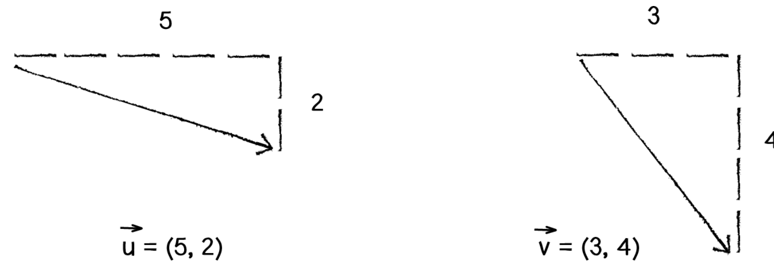


Figure 1.5

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

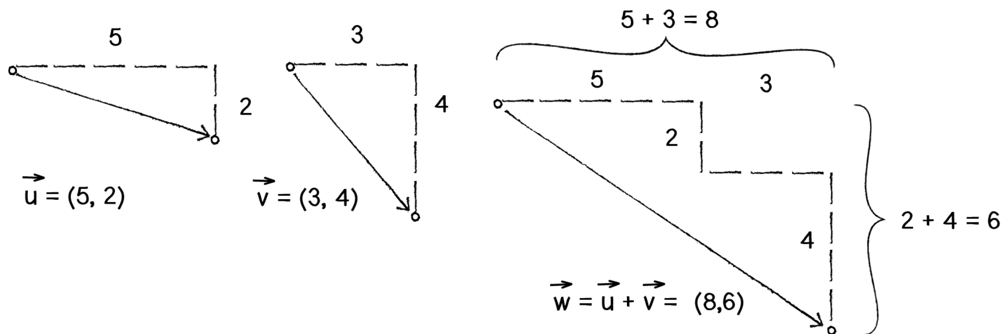


Figure 1.6

In other words:

$$\vec{w} = \vec{u} + \vec{v}$$

can be written as:

$$w_x = u_x - v_x w_y = u_y - v_y$$

Then, replacing u and v with their values from Figure 1.6, you get:

$$w_x = 5 + 3 w_y = 2 + 4$$

which means that:

$$w_x = 8 w_y = 6$$

Finally, writing that as a vector:

$\vec{w} = (8, 6)$

Now that I've covered how to add two vectors together, you can look at how addition is implemented in the `p5.Vector` class itself. Let's write a function called `add()` that takes another `Vector` object as its argument.

```
class Vector {

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

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

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

Basic Number Properties with Vectors

Addition with vectors follow the same algebraic rules as with real numbers.

The commutative rule: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$

The associative rule: $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$

Fancy terminology and symbols aside, this is really quite a simple concept. The common sense properties of addition apply to vectors as well.

$3 + 2 = 2 + 3$

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

Now that I've covered how `add()` is written inside of `p5.Vector`, I can return to the bouncing ball example with its ***position + velocity*** algorithm and implement vector addition:

```
~~position = position + velocity;~~
position.add(velocity);
```

Add the current velocity to the position.

And here I am, ready to rewrite the bouncing ball example using vectors.

Example 1.2: Bouncing ball with vectors!

```
let position;
let velocity;
```

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

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

  position = createVector(100, 100);
  velocity = createVector(2.5, 5);

}

function draw() {
  background(255);

  position.add(velocity);

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

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

```
  stroke(0);
  fill(175);
  ellipse(position.x, position.y, 16, 16);
}
```

Now, you might feel somewhat disappointed. After all, this may 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 I 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 I move forward into a more complex world of multiple objects and multiple ***forces*** (which I'll introduce in Chapter 2), the benefits of vectors will become more apparent.

I should, however, note an important aspect of the above transition to programming with vectors. Even though I am using `p5.Vector` objects to describe two values—the `x` and `y` of position and the `x` and `y` of velocity—I will still often need to refer to the *x* and *y* components of each vector individually. When I go to draw an object in p5.js, there's no means to say:

```
~~ellipse(position, 16, 16);~~
```

The `ellipse()` function does not allow for a `p5.Vector` as an argument. An ellipse can only be drawn with two scalar values, an `x`-coordinate and a `y`-coordinate. And so you must dig into the `p5.Vector` object and pull out the *x* and *y* components using object-oriented dot syntax.

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

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

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

Exercise 1.1

Find something you've previously made in p5.js using separate `x` and `y` variables and use vectors instead.

Exercise 1.2

Take one of the walker examples from the introduction and convert it to use vectors.

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 commonly used with vectors. Below is a comprehensive list of the operations available as functions in the `p5.Vector` class. I'll go through a few of the key ones now. As the examples get more sophisticated in later chapters, I'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
- `setMag()` — set the magnitude of a vector

- `normalize()` — normalize the vector to a unit length of 1
- `limit()` — limit the magnitude of a vector
- `heading()` — the 2D heading of a vector expressed as an angle
- `rotate()` — rotate a 2D vector by an angle
- `lerp()` — linear interpolate to another vector
- `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)
- `random2D()` — make a random 2D vector
- `random3D()` — make a random 3D vector

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

$$\vec{w} = \vec{u} - \vec{v}$$

can be written as:

$$w_x = u_x - v_x$$
$$w_y = u_y - v_y$$

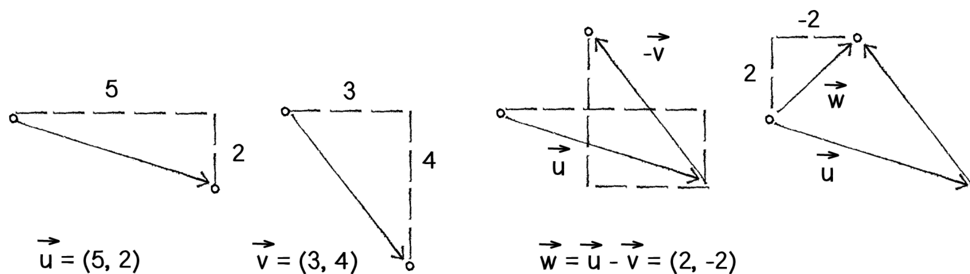
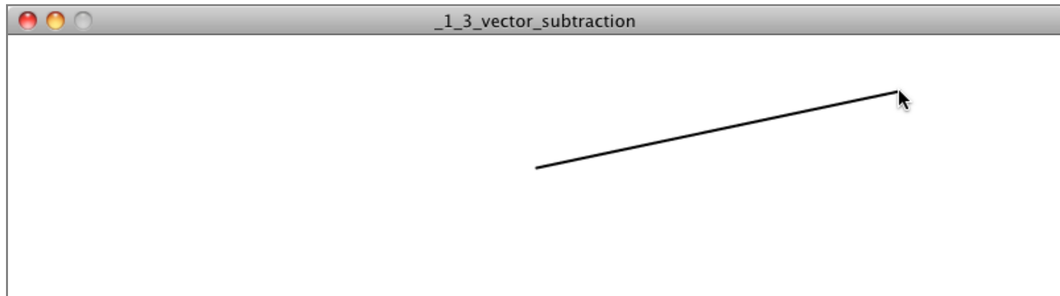


Figure 1.7: Vector Subtraction

and so the function inside `p5.Vector` looks like:

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

The following example demonstrates vector subtraction by taking the difference between two points—the mouse position and the center of the window. Note the use of translate to visualize the resulting vector as a line from the center `(width/2, height/2)` to the mouse.



Example 1.3: Vector subtraction

```
function setup() {
  createCanvas(640, 360);
}

function draw() {
  background(255);

  let mouse  = createVector(mouseX, mouseY);
  let center = createVector(width/2, height/2);
```

Two p5.Vector, one for the mouse location and one for the center of the window

```
  stroke(200);
  line(0, 0, mouse.x, mouse.y);
  line(0, 0, center.x, center.y);
```

Draw the original two vectors

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

Vector subtraction!

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

Draw a line to represent the result of subtraction. Notice how I move the origin with translate() to place the vector

```
}
```

Vector multiplication

Moving on to multiplication, you have to think a little bit differently. Multiplying a vector, typically refers to the process of ***scaling*** a vector. If I want to scale a vector to twice its size or one-third of its size (leaving its direction the same), I would say: “Multiply the vector by 2” or “Multiply the vector by 1/3.” Note that this is multiplying a vector by a scalar, a single number, not another vector.

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

$\vec{w} = \vec{u} * n$

can be written as:

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

Let's look at an example with vector notation.

$\vec{u} = (-3, 7)$
$n = 3$

$\vec{w} = \vec{u} * n$
$w_x = -3 * 3$
$w_y = 7 * 3$

$\vec{w} = (-9, 21)$

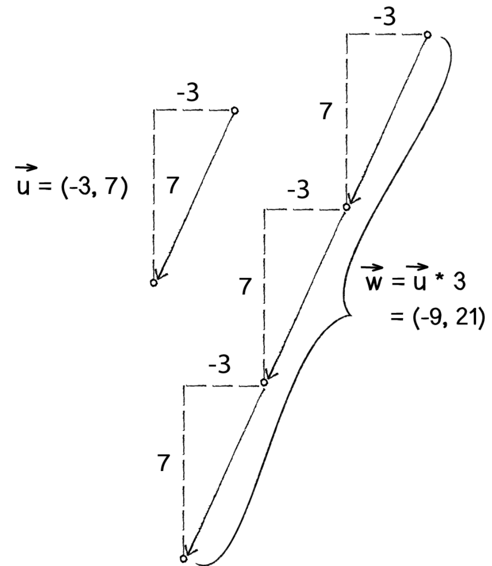


Figure 1.8: Scaling a vector

Therefore, the function inside the `p5.Vector` class is written as:

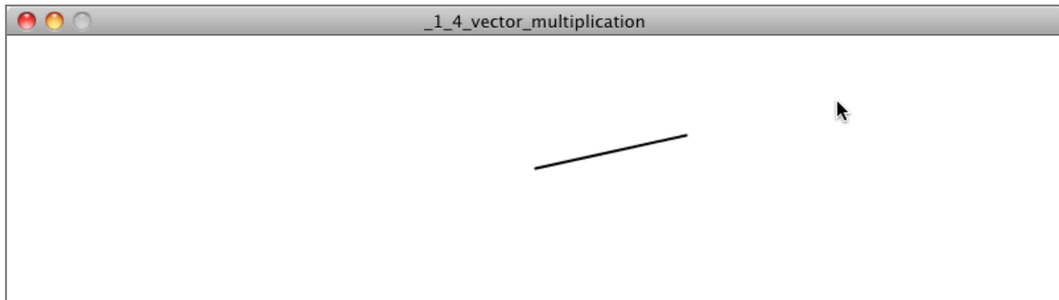
```
mult(n) {
  this.x = this.x * n;
  this.y = this.y * n;
}
```

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

And implementing multiplication in code is as simple as:

```
let u = vector(-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

```
function setup() {
  createCanvas(640, 360);
}

function draw() {
  background(255);

  let mouse = createVector(mouseX, mouseY);
  let center = createVector(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—simply replace the multiplication sign (asterisk) with the division sign (forward slash).

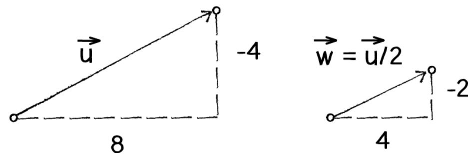


Figure 1.9

```
div(n) {
  this.x = this.x / n;
  this.y = this.y / n;
}

let u = createVector(8, -4);
u.div(2);
```

Dividing a vector! The vector is now half its original size (divided by 2).

More Number Properties with Vectors

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

The associative rule: $(n * m) * \vec{v} = n * (m * \vec{v})$

The distributive rule with 2 scalars, 1 vector: $(n + m) * \vec{v} = (n * \vec{v}) + (m * \vec{v})$

The distributive rule with 2 vectors, 1 scalar: $(\vec{u} + \vec{v}) * n = (\vec{u} * n) + (\vec{v} * n)$

1.5 Vector Magnitude

Multiplication and division, as I just described, change the length of a vector 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.

Notice in Figure 1.10 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 lucky to have this right triangle, because once upon a time, a Greek mathematician named Pythagoras discovered a lovely formula to describe the relationship between the sides and hypotenuse of a right triangle.

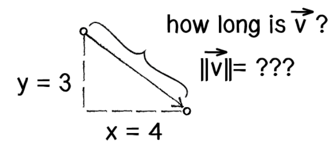


Figure 1.10: The length or "magnitude" of a vector $v \rightarrow$ is often written as: $\| v \rightarrow \|$

The Pythagorean theorem is *a* squared plus *b* squared equals *c* squared, for right triangles.

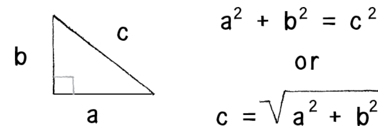


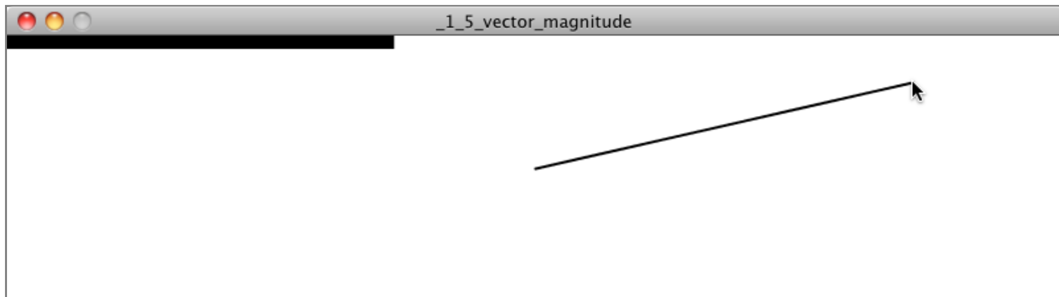
Figure 1.11: The Pythagorean Theorem

Armed with this formula, we can now compute the magnitude of $\vec{v}$ as follows:

$$||\vec{v}|| = \sqrt{v_x * v_x + v_y * v_y}$$

or in `p5.Vector`:

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



Example 1.5: Vector magnitude

```
function setup() {
  createCanvas(640, 360);
}

function draw() {
  background(255);

  let mouse = createVector(mouseX, mouseY);
  let center = createVector(width/2, height/2);
  mouse.sub(center);

  const m = mouse.mag();
  fill(0);
  rect(0, 0, m, 10);
```

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.

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

}
```

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 1. 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 1, turning it into what is called a ***unit vector***.

A unit vector describes a vector's direction without regard to its length. You'll see this come in especially handy once I start to work with forces in Chapter 2.

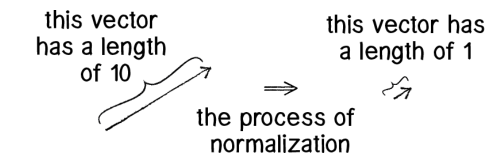


Figure 1.12

For any given vector $\vec{u}$, its unit vector (written as $\hat{u}$) is calculated as follows:

$$\hat{u} = \frac{\vec{u}}{||\vec{u}||}$$

In other words, to normalize a vector, 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 a right triangle, you then need to scale the hypotenuse down by dividing by 5. In that process the sides shrink, divided by 5 as well.

In the `p5.Vector` class, the normalization function is written as follows:

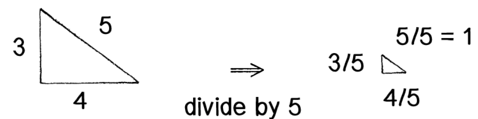
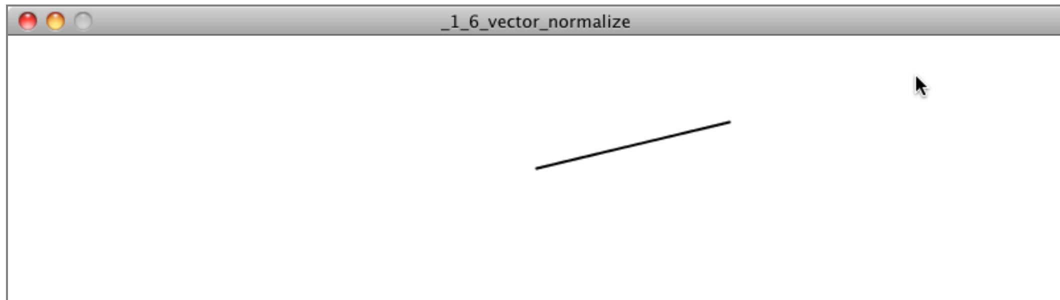


Figure 1.13

```
normalize() {
  let m = this.mag();
  this.div(m);
}
```

Of course, there's one small issue. What if the magnitude of the vector is 0? You can't divide by 0! Some quick error checking will fix that right up:

```
normalize() {
 let m = this.mag();
 if (m > 0) {
   this.div(m);
 }
}
```



Example 1.6: Normalizing a vector

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

  let mouse = createVector(mouseX, mouseY);
  let center = createVector(width/2, height/2);
  mouse.sub(center);

  mouse.normalize();
  mouse.mult(50);
```

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.

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

}
```

1.7 Vector Motion: Velocity

All this vector math stuff sounds like something you should know about, but why? How will it actually help you write code? Patience. It will take some time before the awesomeness of using the `p5.Vector` class fully comes to light. This is actually a common occurrence when first learning a new data structure. For example, when you first learn about an array, it might seem like more work to use an array than to 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 vectors. 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, focus on how it works. What does it mean to program motion using vectors? You've seen the beginning of this in Example 1.2 (see page 0): the bouncing ball. An object on screen has a position (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 position:

```
position.add(velocity);
```

And then the object is drawn at that position:

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

This is Motion 101.

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

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

In this case, I'm going to create a generic `Mover` class that will describe a thing moving around the screen. And so I must consider the following two questions:

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

The "Motion 101" algorithm has the answers to these questions. A `Mover` object has two pieces of data: `position` and `velocity`, which are both `p5.Vector` objects.

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

Its functionality is about as simple. The `Mover` needs to move and it needs to be seen. I'll implement these needs as functions named `update()` and `display()`. I'll put all of the motion logic code in `update()` and draw the object in `display()`.

```
update() {
  this.position.add(this.velocity);          The Mover moves.
}

display() {
  stroke(0);
  fill(175);
  ellipse(this.position.x, this.position.y, 16,     The Mover is displayed.
16);
}
}
```

I'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:

```
let m = new Mover();
```

In this case, let's arbitrarily decide to initialize the `Mover` object by giving it a random position and a random velocity. Note the use of `this` with all variables that are part of the `Mover` object.

```
constructor() {
  this.position = createVector(random(width), random(height));
  this.velocity = createVector(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, I spent the beginning of this chapter discussing the `p5.Vector` class. The `p5.Vector` class is the template for making the `position` object and the `velocity` object. 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 something that holds data (and functionality). That data can be numbers or other objects (arrays too)! You'll see this over and over again in this book. For example, in Chapter 4 (see page 145) I'll write a class to describe a system of particles. That `ParticleSystem` object will include a list of `Particle` objects…and each `Particle` object will have as its data several `p5.Vector` 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 canvas. For now let's do something simple, and have it wrap around the edges.

```
checkEdges() {
```

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

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

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

Now that the `Mover` class is finished, I can move onto `setup()>` and `draw()`. First, declare a `Mover` object:

```
let mover;
```

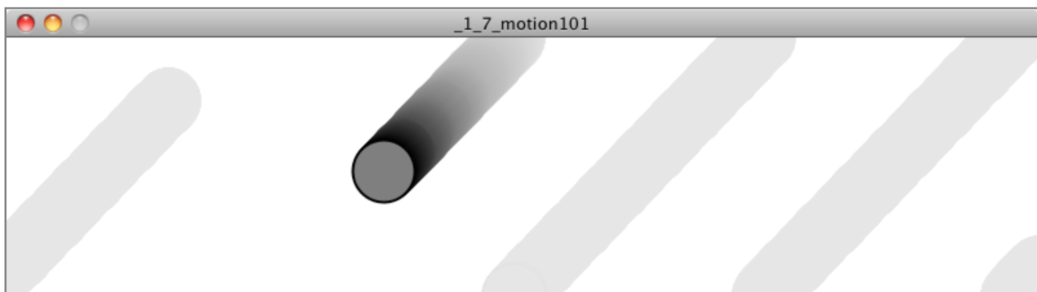
Then create and initialize the mover in `setup()`:

```
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)

```
let mover;

function setup() {
  createCanvas(640, 360);
```

Declare Mover object.

```
  mover = new Mover();                          Create Mover object.
}

function draw() {
  background(255);

  mover.update();                               Call functions on Mover object.
  mover.checkEdges();
  mover.display();
}


class Mover {

  constructor() {
    this.position = createVector(random(width),  The object has two vectors: position and velocity.
random(height));
    this.velocity = createVector(random(-2, 2),
random(-2, 2));
  }

  update() {
    this.position.add(this.velocity);           Motion 101: position changes by velocity.
  }

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

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

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

1.8 Vector Motion: Acceleration

OK. At this point, you hopefully feel comfortable with two things: (1) what a vector is and (2) how to use vectors inside of an object to keep track of its position and movement. This is an excellent first step and deserves a mild round of applause. Before standing ovations and screaming fans, however, you need to make one more, somewhat bigger 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 sophisticated motion, for motion that appears in the real world around us, I need to add one more vector to the class—`acceleration`.

The strict definition of ***acceleration*** I'm using here is: *the rate of change of velocity*. Think about that definition for a moment. Is this a new concept? Not really. Velocity is defined as *the rate of change of position*. In essence, I am developing a "trickle-down" effect. Acceleration affects velocity, which in turn affects position (for some brief foreshadowing, this point will become even more crucial in the next chapter, when I look at how forces affect acceleration, which affects velocity, which affects position). In code, this reads:

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

As an exercise, from this point forward, I'm going to make a rule for myself. I will write every example in the rest of this book without ever touching the value of velocity and position (except to initialize them). In other words, the goal for programming motion is: Come up with an algorithm for how to calculate acceleration and let the trickle-down effect work its magic. (In truth, I'll find reasons to break this rule, and will break it often but it's a useful constraint to begin with to illustrate the principles behind the motion algorithm.) And so I need to come up with some ways to calculate acceleration:

Acceleration Algorithms!

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

Algorithm #1, *a constant acceleration*, is not particularly interesting, but it is the simplest and gives me a starting point to incorporate acceleration into the code. The first thing I need to do is add another variable to the `Mover` class:

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

```
    this.acceleration = createVector();          A new vector for acceleration
  }
```

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

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

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

```
constructor() {
```

Let’s start the `Mover` object in the middle of the window…

```
this.position = createVector(width/2, height/2);
```

…with an initial velocity of zero.

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

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

```
this.acceleration = createVector(-0.001, 0.01);
  }
```

Maybe you’re thinking, “Gosh, those values seem awfully small!” That’s right, they are quite tiny. Acceleration values (measured in pixels) accumulate over time in the velocity, about thirty times per second depending on the sketch’s frame rate. And so to keep the magnitude of the velocity vector within a reasonable range, the acceleration values should remain quite small. I can also manage this by incorporating the `p5.Vector` function `limit()`.

```
this.velocity.limit(10);                         The limit() function constrains the magnitude of a
                                                 vector.
```

This translates to the following:

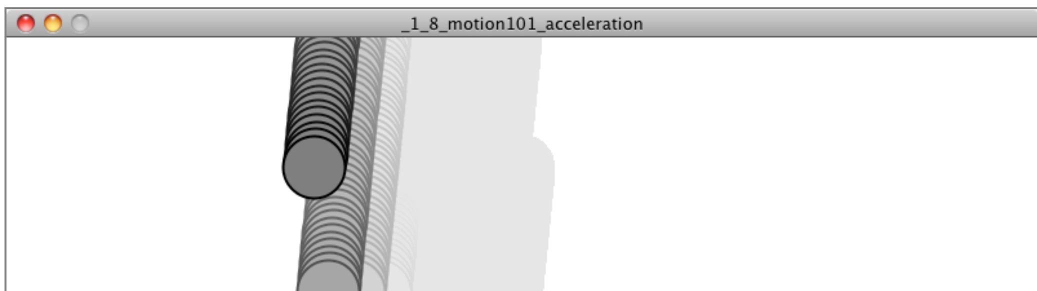
What is the magnitude of velocity? If it’s less than 10, no worries; just leave it as is. If it’s more than 10, however, reduce it to 10!

Exercise 1.4

Write the `limit()` function for the `p5.Vector` class.

```
limit(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 {

  constructor() {
    this.position = createVector(width/2, height/2);
    this.velocity = createVector(0, 0);
    this.acceleration = createVector(-0.001,
0.01);
    this.topspeed = 10;
  }

  update() {
    this.velocity.add(this.acceleration);
    this.velocity.limit(this.topspeed);
```

Acceleration is the key!

The variable topspeed will limit the magnitude of velocity.

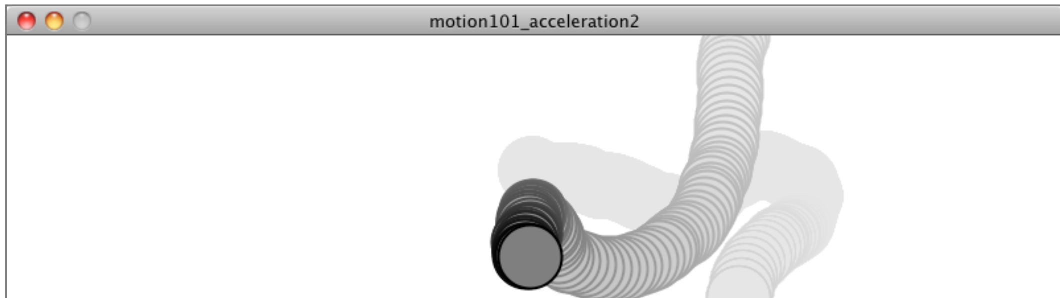
Velocity changes by acceleration and is limited by topspeed.

```
    this.position.add(this.velocity);
  }

  display() {}        // display() is the same.

  checkEdges() {}     // checkEdges() is the same.
}
```

Exercise 1.5

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 on to Algorithm #2, *a random acceleration*. In this case, instead of initializing acceleration in the object's constructor, I want to pick a new acceleration each cycle, i.e. each time `update()` is called.



Example 1.9: Motion 101 (velocity and random acceleration)

```
update() {

  this.acceleration = p5.Vector.random2D();   // The random2D() function will give us a unit vector pointing in a random direction.

  this.velocity.add(this.acceleration);
  this.velocity.limit(this.topspeed);
  this.position.add(this.velocity);
}
```

Because the random vector is a normalized one, I can try scaling it:

(a) scaling the acceleration to a constant value

```
this.acceleration = p5.Vector.random2D();
this.acceleration.mult(0.5);                    Constant
```

(b) scaling the acceleration to a random value

```
this.acceleration = p5.Vector.random2D();
this.acceleration.mult(random(2));              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—magnitude or direction. Acceleration is used to steer an object, and you’ll see this again and again in future chapters as I begin to program objects that make decisions about how to move about the canvas.

Exercise 1.6

Referring back to the Introduction (see page 18), implement acceleration according to Perlin noise.

1.9 Static vs. Non-Static Functions

You might have noticed something a bit odd and unfamiliar in the previous example. The `random2D()` function used to create a random unit vector was called on the `p5.Vector` class name itself! This is what is known as a “static” function and is in a key concept underlying the `p5.Vector` class. Algorithm #3 (*accelerate towards the mouse*), in fact, requires further exploration of this concept: the difference between using ***static*** methods and ***non-static*** methods.

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

```
let x = 0;
let 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. I could write the corresponding code pretty easily based on what we’ve learned about `p5.Vector`.

```
let v = createVector(0, 0);
let u = createVector(4, 5);
v.add(u);
```

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

Let's take a look at another example of some simple floating point math:

```
let x = 0;
let y = 5;

let z = x + y;
```

x has the value of 0, I 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 numbers. However, it's not so obvious with mathematical operations using `p5.Vector`. Let's try to rewrite the above code with vectors based on what we know so far.

```
let v = createVector(0,0);
let u = createVector(4,5);
const w = v.add(u);                    Don't be fooled; this is incorrect!!!
```

The above might seem like a good guess, but it's just not the way the `p5.Vector` class works. If you look at the definition of `add()` . . .

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

you'll see that this code does not accomplish my goal. First, it does not return a new `p5.Vector` object and second, it changes the value of the vector upon which it is called. In order to add two vector objects together and return the result as a new vector, I must use the static `add()` function.

Functions that are called 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 `p5.Vector` objects, v and u:

```
p5.Vector.add(v, u);                   Static: called from the class name.
```

```
v.add(u);                              Not static: called from an object instance.
```

When writing your own classes, static functions are very rarely needed, so it's likely you not have encountered them before. `p5.Vector`'s static functions generic mathematical operations to be performed on vectors without having to adjust the value of one of the input vectors. Let's look at how I might write the static version of `add()`:

```
static add(v1, v2) {
```

The static version of add allows two vectors to be added together and the result assigned to a new vector while leaving the original vectors (v and u above) intact.

```
  let v3 = createVector(v1.x + v2.x, v1.y + v2.y);
  return v3;
}
```

The key difference here is that the function creates a new vector (`v3`) and returns the sum of the components of `v1` and `v2` in `v3` without changing the values of either original vector.

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

```
let v = createVector(0, 0);
let u = createVector(4, 5);
const w = v.add(u);
let w = p5.Vector.add(v, u);
```

The `p5.Vector` class has static versions of `add()`, `sub()`, `mult()`, and `div()`.

Exercise 1.7

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

- The vector v equals (1,5).
- The vector u equals v multiplied by 2.
- The vector w equals v minus u.
- Divide the vector w by 3.

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

1.10 Interactivity with Acceleration

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

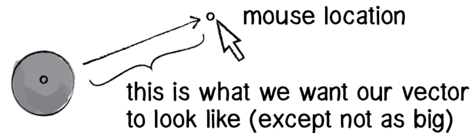


Figure 1.14

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

In Figure 1.15, you see that the vector (`dx`,`dy`) can be calculated by subtracting the object's position from the mouse's position.

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

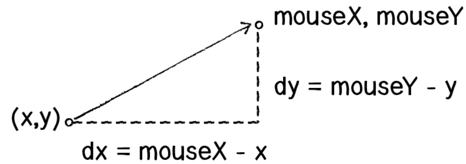


Figure 1.15

Let's rewrite the above using `p5.Vector` syntax. Assuming I'm writing this code inside `Mover` class and thus have access to the object's `position`, I then have:

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

Look! I'm using the static reference to sub() because I want a new p5.Vector pointing from one point to another.

I now have a vector `dir` that points from the mover's position all the way to the mouse. If the object were to actually accelerate using that vector, it would appear instantaneously at the mouse position. This does not make for a smooth animation, of course. The next step therefore is to decide how quickly that object should accelerate toward the mouse.

In order to set the magnitude (whatever it may be) of the acceleration vector, I must first _ that direction vector. That's right, you said it. *Normalize*. If I can shrink the vector down to its unit vector (of length one) then I can easily scale it to any value. One multiplied by anything equals anything.

```
let anything = _________;
dir.normalize();
dir.mult(anything);
```

Any number!

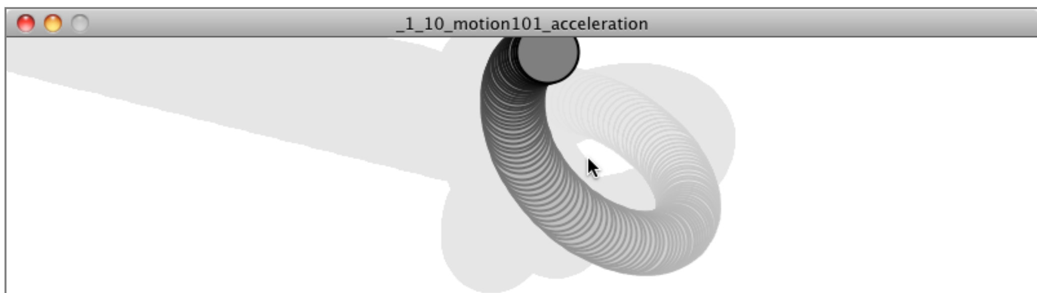
To summarize, take the following steps:

1. Calculate a vector that points from the object to the target position (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.

I have a confession to make. This is such a common operation (normalization then scaling) that `p5.Vector` includes a method to do just that—set the magnitude of a vector to a value. That function is `setMag()`.

```
let anything = ?????
dir.setMag(anything);
```

In this last example, to emphasize the math, I'm going to write the code with both steps separate, but this is likely the last time I‘ll do that and you‘ll see `setMag()` in examples going forward.



Example 1.10: Accelerating towards the mouse

```
update() {

  let mouse = createVector(mouseX, mouseY);
  let dir = p5.Vector.sub(mouse, position);        Step 1: Compute direction

  dir.normalize();                                 Step 2: Normalize

  dir.mult(0.5);                                   Step 3: Scale

  this.acceleration = dir;                         Step 4: Accelerate
```

```
    this.velocity.add(this.acceleration);
    this.velocity.limit(this.topspeed);
    this.position.add(this.velocity);

  }
```

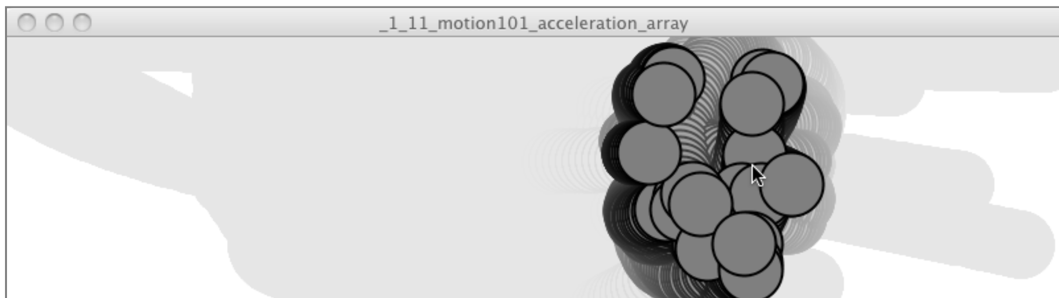
You may be wondering why the circle doesn't stop when it reaches the target. It's important to note that the object moving has no knowledge about trying to stop at a destination; it only knows where the destination is and tries to go accelerate there at a fixed rate regardless of how far away it is. This means it will inevitably overshoot the target and have to turn around, again accelerating towards the destination, overshooting it again, and so on and so forth. Stay tuned; in later chapters I'll show you how to program an object to ***arrive*** at a target (slow down on approach).

This example is remarkably close to the concept of gravitational attraction (in which the object is attracted to the mouse position). Gravitational attraction will be covered in more detail in the next chapter. However, one thing missing in this example is calculating the magnitude of acclerartion which in gravity's case is inversely proportional to distance. In other words, the closer the object is to the mouse, the faster it accelerates.

Exercise 1.8

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 movers (rather than just one).



Example 1.11: Array of movers accelerating towards the mouse

```
let movers = [];                                    An array of objects
```

```
function setup() {
  createCanvas(640, 360);
  background(255);
  for (let i = 0; i < 20; i++) {
    movers[i] = new Mover();
  }
}

function draw() {
  background(255);

  for (int i = 0; i < movers.length; i++) {
    movers[i].update();
    movers[i].display();
  }
}

class Mover {

  constructor() {
    this.position = createVector(random(width), random(height));
    this.velocity = createVector();
    this.acceleration = createVector();
    this.topspeed = 5;
  }

  update() {

    let mouse = createVector(mouseX, mouseY);
    let dir = p5.Vector.sub(mouse,
this.position);

    dir.normalize();
    dir.mult(0.5);
    this.acceleration = dir;

    this.velocity.add(this.acceleration);
    this.velocity.limit(this.topspeed);
    this.position.add(this.velocity);
    }

  display() {
    stroke(0);
    fill(175);
    ellipse(this.position.x, this.position.y,
16, 16);
  }
```

Initialize each object in the array. (`movers[i] = new Mover();`)

Calling functions on all the objects in the array (`movers[i].update(); movers[i].display();`)

Algorithm for calculating acceleration:

Find the vector pointing towards the mouse.

Normalize.

Scale.

Set to acceleration.

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

Display the Mover

```
  checkEdges() {                                    What to do at the edges
    if (this.position.x > width) {
      this.position.x = 0;
    } else if (this.position.x > 0) {
      this.position.x = width;
    }

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

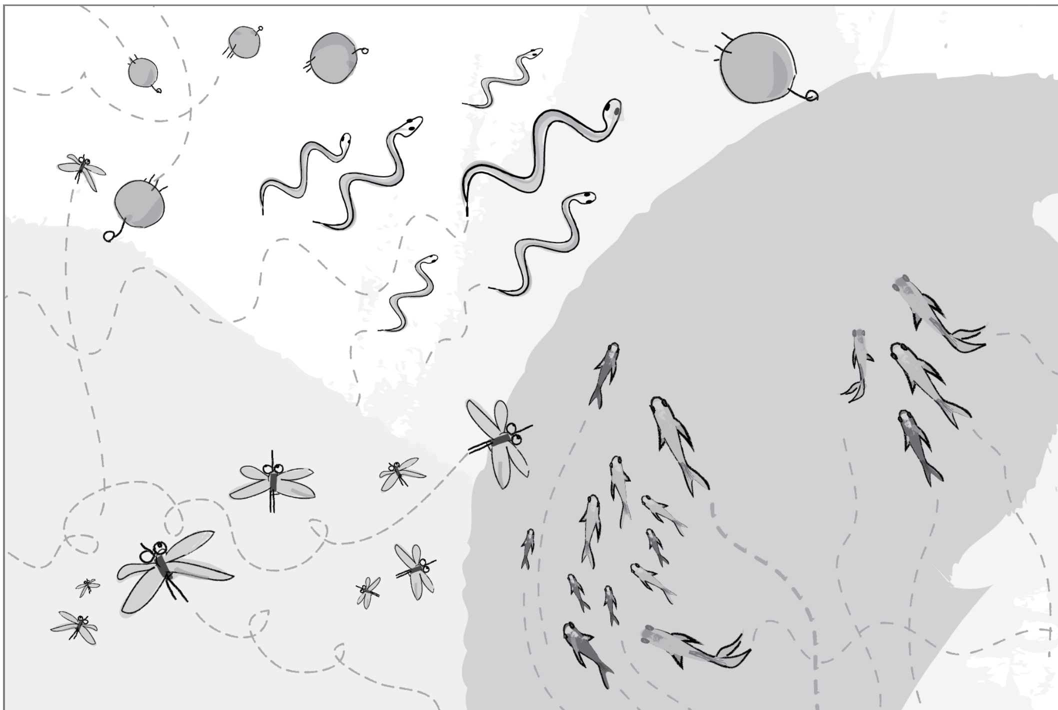


Figure 1.16: The Ecosystem Project

The 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 at a time. One idea for this is a simulation of an ecosystem. 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 vector? Try to give the creature a personality through its behavior (rather than through its visual design, although that is of course worth exploring as well).