Chapter 3. Oscillation

"Trigonometry is a sine of the times."

— Anonymous

In Chapters 1 and 2, we carefully worked out an object-oriented structure to make something move on the screen, using the concept of a vector to represent location, velocity, and acceleration driven by forces in the environment. We could move straight from here into topics such as particle systems, steering forces, group behaviors, etc. If we did that, however, we'd skip an important area of mathematics that we're going to need: ***trigonometry***, or the mathematics of triangles, specifically right triangles.

Trigonometry is going to give us a lot of tools. We'll get to think about angles and angular velocity and acceleration. Trig will teach us about the sine and cosine functions, which when used properly can yield an nice ease-in, ease-out wave pattern. It's going to allow us to calculate more complex forces in an environment that involves angles, such as a pendulum swinging or a box sliding down an incline.

So this chapter is a bit of a mishmash. We'll start with the basics of angles in Processing and cover many trigonometric topics, tying it all into forces at the end. And by taking this break now, we'll also pave the way for more advanced examples that require trig later in this book.

3.1 Angles

OK. Before we can do any of this stuff, we need to make sure we understand what it means to be an angle in Processing. If you have experience with Processing, you've undoubtedly encountered this issue while using the `rotate()` function to rotate and spin objects.

The first order of business is to cover ***radians*** and ***degrees***. You're probably familiar with the concept of an angle in ***degrees***. A full rotation goes from 0 to 360 degrees. 90 degrees (a right angle) is 1/4th of 360, shown below as two perpendicular lines.

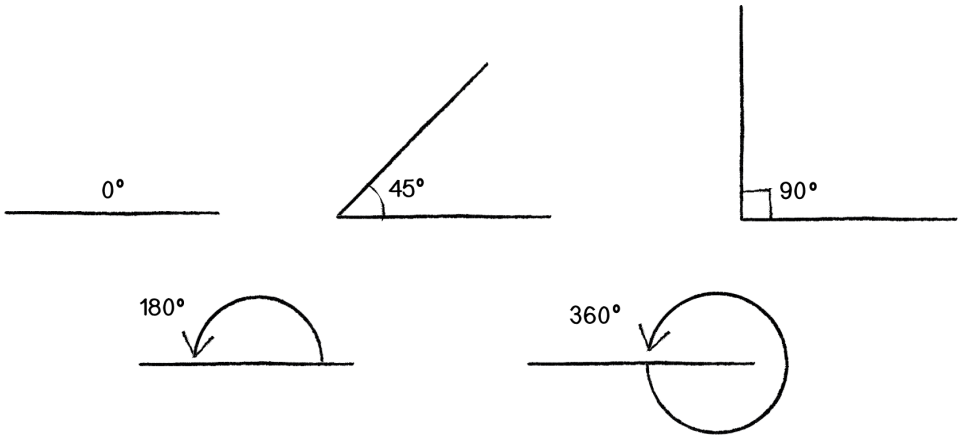


Figure 3.1

It's fairly intuitive for us to think of angles in terms of degrees. For example, the square in Figure 3.2 is rotated 45 degrees around its center.

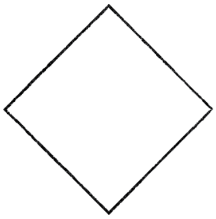


Figure 3.3

Processing, however, requires angles to be specified in ***radians***. A radian is a unit of measurement for angles defined by the ratio of the length of the arc of a circle to the radius of that circle. One radian is the angle at which that ratio equals one (see Figure 3.1). 180 degrees = PI radians, 360 degrees = 2*PI radians, 90 degrees = PI/2 radians, etc.

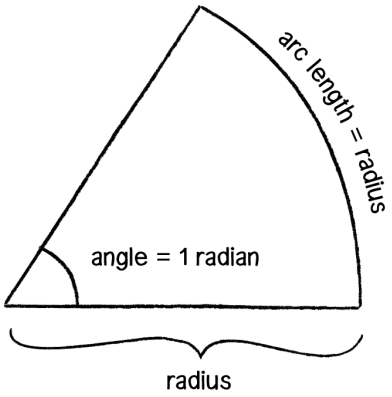


Figure 3.3

The formula to convert from degrees to radians is:

radians = 2 * PI * (degrees / 360)

Thankfully, if we prefer to think in degrees but code with radians, Processing makes this easy. The `radians()` function will automatically convert values from degrees to radians, and the constants `PI` and `TWO_PI` provide convenient access to these commonly used numbers (equivalent to 180 and 360 degrees, respectively). The following code, for example, will rotate shapes by 60 degrees.

```
float angle = radians(60);
rotate(angle);
```

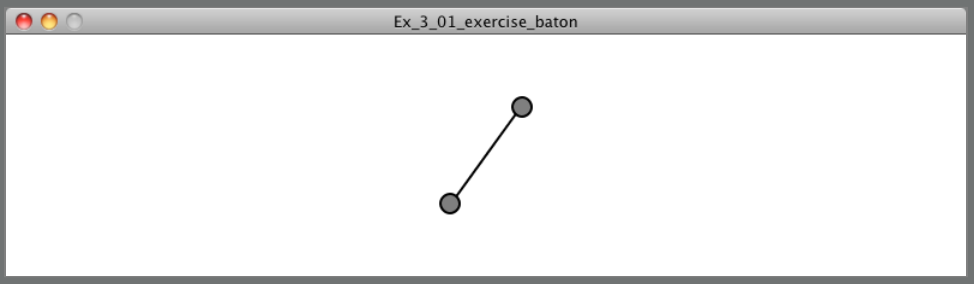
If you are not familiar with how rotation is implemented in Processing, I would suggest this tutorial: Processing - Transform 2D (http://www.processing.org/learning/transform2d/).

What is PI?

The mathematical constant pi (or π) is a real number defined as the ratio of a circle's circumference (the distance around the perimeter) to its diameter (a straight line that passes through the circle's center). It is equal to approximately 3.14159 and can be accessed in Processing with the built-in variable `PI`.

Exercise 3.1

Rotate a baton-like object (see below) around its center using `translate()` and `rotate()`.



3.2 Angular Motion

Remember all this stuff?

```
location = location + velocity
velocity = velocity + acceleration
```

The stuff we dedicated almost all of Chapters 1 and 2 to? Well, we can apply exactly the same logic to a rotating object.

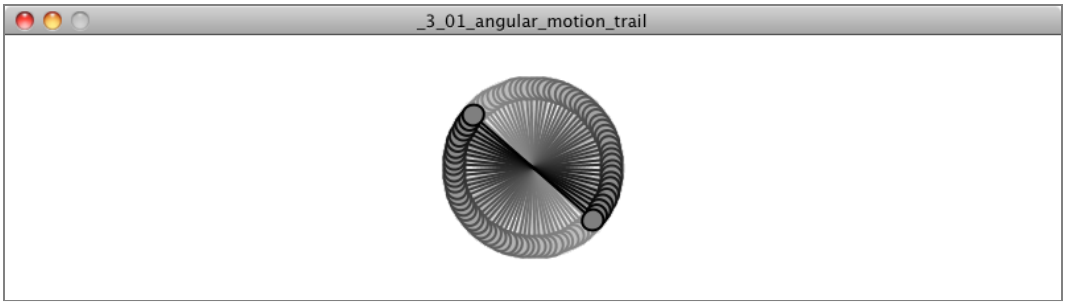
```
angle = angle + angular velocity
angular velocity = angular velocity + angular acceleration
```

In fact, the above is actually simpler than what we started with because an angle is a *scalar* quantity—a single number, not a vector!

Using the answer from Exercise 3.1 above, let's say we wanted to rotate a baton in Processing by some angle. We would have code like:

```
translate(width/2,height/2);
rotate(angle);
line(-50,0,50,0);
ellipse(50,0,8,8);
ellipse(-50,0,8,8);
```

Adding in our principles of motion brings us to the following example.



Example 3.1: Angular motion using rotate()

```
float angle = 0;                                  Location

float aVelocity = 0;                              Velocity

float aAcceleration = 0.001;                      Acceleration

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

void draw() {
  background(255);

  fill(175);
  stroke(0);
  rectMode(CENTER);
  translate(width/2,height/2);
  rotate(angle);
  line(-50,0,50,0);
  ellipse(50,0,8,8);
  ellipse(-50,0,8,8);

  aVelocity += aAcceleration;                     Angular equivalent of
                                                  velocity.add(acceleration);

  angle += aVelocity;                             Angular equivalent of location.add(velocity);
}
```

The baton starts onscreen with no rotation and then spins faster and faster as the angle of rotation accelerates.

This idea can be incorporated into our `Mover` object. For example, we can add the variables related to angular motion to our `Mover`.

```
class Mover {

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

  float angle = 0;
  float aVelocity = 0;
  float aAcceleration = 0;
```

And then in `update()`, we update both location and angle according to the same algorithm!

```
 void update() {

    velocity.add(acceleration);                    Regular old-fashioned motion
    location.add(velocity);


    aVelocity += aAcceleration;                    Newfangled angular motion
    angle += aVelocity;

    acceleration.mult(0);
  }
```

Of course, for any of this to matter, we also would need to rotate the object when displaying it.

```
  void display() {
    stroke(0);
    fill(175,200);
    rectMode(CENTER);
    pushMatrix();                                  pushMatrix() and popMatrix() are
                                                   necessary so that the rotation of this shape
                                                   doesn't affect the rest of our world.
    translate(location.x,location.y);              Set the origin at the shape's location.

    rotate(angle);                                 Rotate by the angle.

    rect(0,0,mass*16,mass*16);
    popMatrix();
  }
```

Now, if we were to actually go ahead and run the above code, we wouldn't see anything new. This is because the angular acceleration (`float aAcceleration = 0;`) is initialized to zero. For the object to rotate, we need to give it an acceleration! Certainly, we could hard-code in a different number.

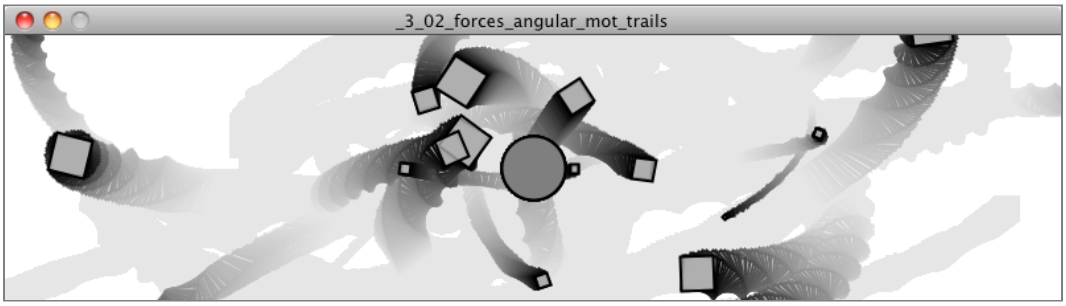
```
float aAcceleration = 0.01;
```

However, we can produce a more interesting result by dynamically assigning an angular acceleration according to forces in the environment. Now, we could head far down this road, trying to model the physics of angular acceleration using the concepts of torque (http://en.wikipedia.org/wiki/Torque) and moment of inertia (http://en.wikipedia.org/wiki/Moment_of_inertia). Nevertheless, this level of simulation is beyond the scope of this book. (We will see more about modeling angular acceleration with a pendulum later in this chapter, as well as look at how Box2D realistically models rotational motion in Chapter 5.)

For now, a quick and dirty solution will do. We can produce reasonable results by simply calculating angular acceleration as a function of the object's acceleration vector. Here's one such example:

```
aAcceleration = acceleration.x;
```

Yes, this is completely arbitrary. But it does do something. If the object is accelerating to the right, its angular rotation accelerates in a clockwise direction; acceleration to the left results in a counterclockwise rotation. Of course, it's important to think about scale in this case. The *x* component of the acceleration vector might be a quantity that's too large, causing the object to spin in a way that looks ridiculous or unrealistic. So dividing the *x* component by some value, or perhaps constraining the angular velocity to a reasonable range, could really help. Here's the entire `update()` function with these tweaks added.



Example 3.2: Forces with (arbitrary) angular motion

```
void update() {

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

  aAcceleration = acceleration.x / 10.0;
  aVelocity += aAcceleration;
```

Calculate angular acceleration according to acceleration's horizontal direction and magnitude.

```
    aVelocity = constrain(aVelocity,-0.1,0.1);
    angle += aVelocity;

    acceleration.mult(0);
  }
```

Use constrain() to ensure that angular velocity doesn't spin out of control.

Exercise 3.2

Step 1: Create a simulation where objects are shot out of a cannon. Each object should experience a sudden force when shot (just once) as well as gravity (always present).

Step 2: Add rotation to the object to model its spin as it is shot from the cannon. How realistic can you make it look?

3.3 Trigonometry

I think it may be time. We've looked at angles, we've spun an object. It's time for: *sohcahtoa*. Yes, *sohcahtoa*. This seemingly nonsensical word is actually the foundation for a lot of computer graphics work. A basic understanding of trigonometry is essential if you want to calculate an angle, figure out the distance between points, work with circles, arcs, or lines. And *sohcahtoa* is a mnemonic device (albeit a somewhat absurd one) for what the trigonometric functions sine, cosine, and tangent mean.

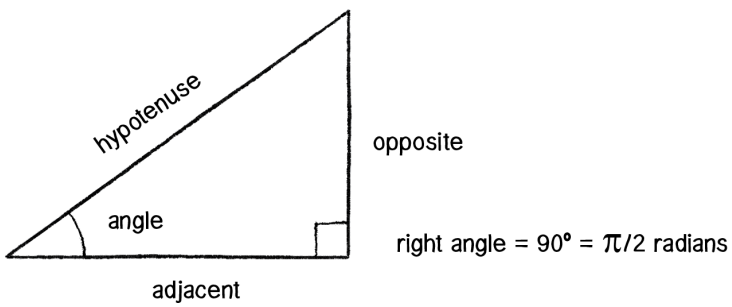


Figure 3.4

- ***soh***: sine = opposite / hypotenuse
- ***cah***: cosine = adjacent / hypotenuse
- ***toa***: tangent = opposite / adjacent

Take a look at Figure 3.4 again. There's no need to memorize it, but make sure you feel comfortable with it. Draw it again yourself. Now let's draw it a slightly different way (Figure 3.5).

See how we create a right triangle out of a vector? The vector arrow itself is the hypotenuse and the components of the vector (x and y) are the sides of the triangle. The angle is an additional means for specifying the vector's direction (or "heading").

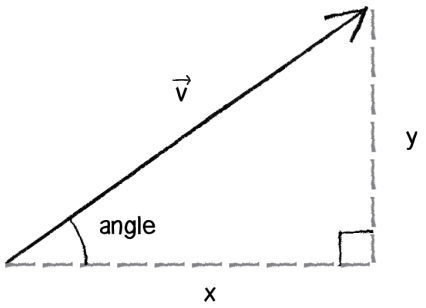
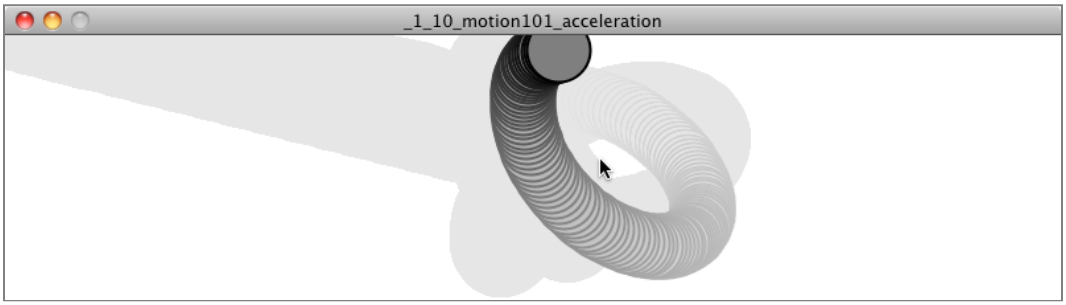


Figure 3.5

Because the trigonometric functions allow us to establish a relationship between the components of a vector and its direction + magnitude, they will prove very useful throughout this book. We'll begin by looking at an example that requires the tangent function.

3.4 Pointing in the Direction of Movement

Let's go all the way back to Example 1.10, which features a `Mover` object accelerating towards the mouse.



You might notice that almost all of the shapes we've been drawing so far are circles. This is convenient for a number of reasons, one of which is that we don't have to consider the question of rotation. Rotate a circle and, well, it looks exactly the same. However, there comes a time in all motion programmers' lives when they want to draw something on the screen that points in the direction of movement. Perhaps you are drawing an ant, or a car, or a spaceship. And when we say "point in the direction of movement," what we are really saying is "rotate according to the velocity vector." Velocity is a vector, with an x and a y component, but to rotate in Processing we need an angle, in radians. Let's draw our trigonometry diagram one more time, with an object's velocity vector (Figure 3.6).

OK. We know that the definition of tangent is:

$$tangent(angle) = \frac{velocity_y}{velocity_x}$$

The problem with the above is that we know velocity, but we don't know the angle. We have to solve for the angle. This is where a special function known as *inverse tangent* comes in, sometimes referred to as *arctangent* or tan^{-1}. (There is also an *inverse sine* and an *inverse cosine*.)

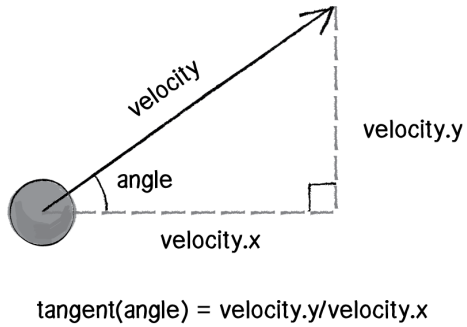


Figure 3.6

If the tangent of some value a equals some value b, then the inverse tangent of b equals a. For example:

if	*tangent(a) = b*
then	*a = arctangent(b)*

See how that is the inverse? The above now allows us to solve for the angle:

if	*tangent(angle) = $velocity_y$ / $velocity_x$*
then	*angle = arctangent($velocity_y$ / $velocity_x$)*

Now that we have the formula, let's see where it should go in our mover's `display()` function. Notice that in Processing, the function for arctangent is called `atan()`.

```
void display() {
  float angle = atan(velocity.y/velocity.x);        Solve for angle by using atan().

  stroke(0);
  fill(175);
  pushMatrix();
  rectMode(CENTER);
  translate(location.x,location.y);
  rotate(angle);                                    Rotate according to that angle.
  rect(0,0,30,10);
  popMatrix();
}
```

Now the above code is pretty darn close, and almost works. We still have a big problem, though. Let's consider the two velocity vectors depicted below.

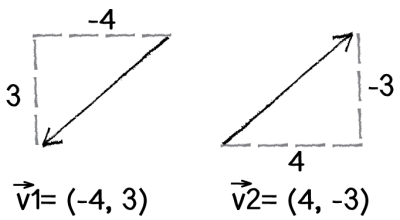
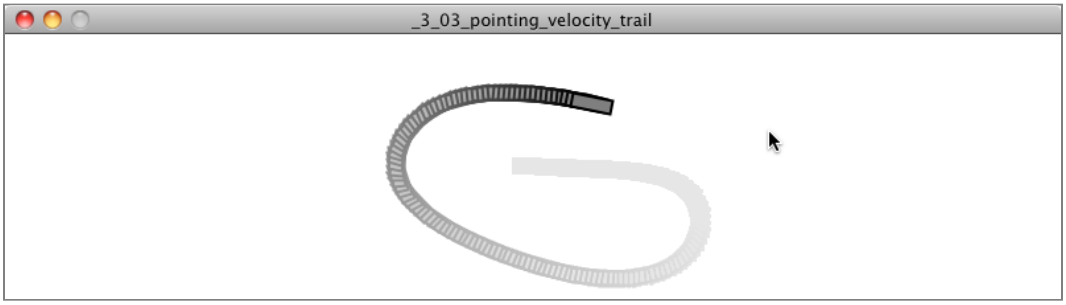


Figure 3.7

Though superficially similar, the two vectors point in quite different directions—opposite directions, in fact! However, if we were to apply our formula to solve for the angle to each vector...

```
V1 ⇒ angle = atan(-4/3) = atan(-1.25) = -0.9272952 radians = -53 degrees
V2 ⇒ angle = atan(4/-3) = atan(-1.25) = -0.9272952 radians = -53 degrees
```

...we get the same angle for each vector. This can't be right for both; the vectors point in opposite directions! The thing is, this is a pretty common problem in computer graphics. Rather than simply using `atan()` along with a bunch of conditional statements to account for positive/negative scenarios, Processing (along with pretty much all programming environments) has a nice function called `atan2()` that does it for you.



Example 3.3: Pointing in the direction of motion

```
void display() {
  float angle = atan2(velocity.y,velocity.x);

  stroke(0);
  fill(175);
  pushMatrix();
  rectMode(CENTER);
  translate(location.x,location.y);
```

Using atan2() to account for all possible directions

```
    rotate(angle);                    Rotate according to that angle.
    rect(0,0,30,10);
    popMatrix();
  }
```

To simplify this even further, the `PVector` class itself provides a function called `heading()`, which takes care of calling `atan2()` for you so you can get the 2D direction angle, in radians, for any Processing `PVector`.

```
    float angle = velocity.heading();    The easiest way to do this!
```

Exercise 3.3

Create a simulation of a vehicle that you can drive around the screen using the arrow keys: left arrow accelerates the car to the left, right to the right. The car should point in the direction in which it is currently moving.

3.5 Polar vs. Cartesian Coordinates

Any time we display a shape in Processing, we have to specify a pixel location, a set of x and y coordinates. These coordinates are known as ***Cartesian coordinates***, named for René Descartes, the French mathematician who developed the ideas behind Cartesian space.

Another useful coordinate system known as ***polar coordinates*** describes a point in space as an angle of rotation around the origin and a radius from the origin. Thinking about this in terms of a vector:

Cartesian coordinate—the *x*,*y* components of a vector
Polar coordinate—the magnitude (length) and direction (angle) of a vector

Processing's drawing functions, however, don't understand polar coordinates. Whenever we want to display something in Processing, we have to specify locations as *(x,y)* Cartesian coordinates. However, sometimes it is a great deal more convenient for us to think in polar coordinates when designing. Happily for us, with trigonometry we can convert back and forth between polar and Cartesian, which allows us to design with whatever coordinate system we have in mind but always draw with Cartesian coordinates.

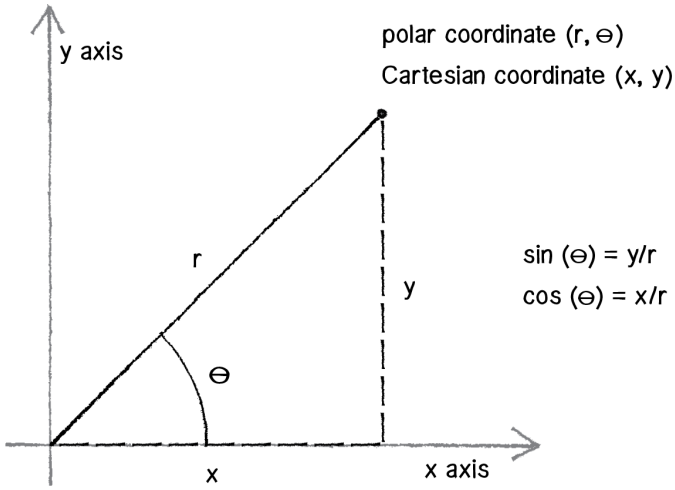


Figure 3.8: The Greek letter θ (theta) is often used to denote an angle. Since a polar coordinate is conventionally referred to as (r, θ), we'll use theta as a variable name when referring to an angle.

```
sine(theta)   = y/r   →   y = r * sine(theta)
cosine(theta) = x/r   →   x = r * cosine(theta)
```

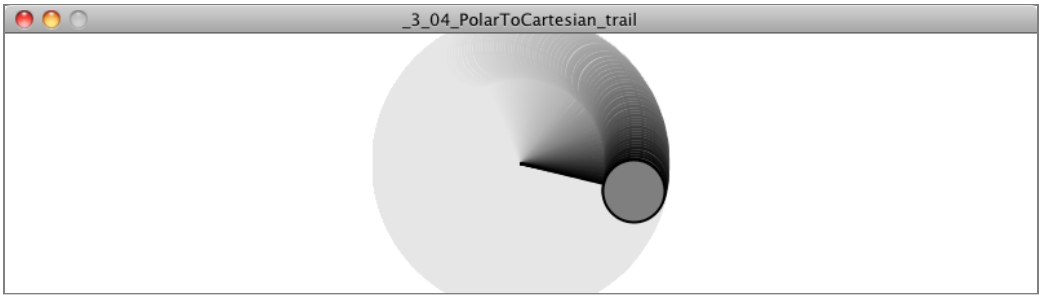
For example, if r is 75 and `theta` is 45 degrees (or PI/4 radians), we can calculate x and y as below. The functions for sine and cosine in Processing are `sin()` and `cos()`, respectively. They each take one argument, an angle measured in radians.

```
float r = 75;
float theta = PI / 4;

float x = r * cos(theta);
float y = r * sin(theta);
```

Converting from polar (r,theta) to Cartesian (x,y)

This type of conversion can be useful in certain applications. For example, to move a shape along a circular path using Cartesian coordinates is not so easy. With polar coordinates, on the other hand, it's simple: increment the angle!

Here's how it is done with global variables `r` and `theta`.



Example 3.4: Polar to Cartesian

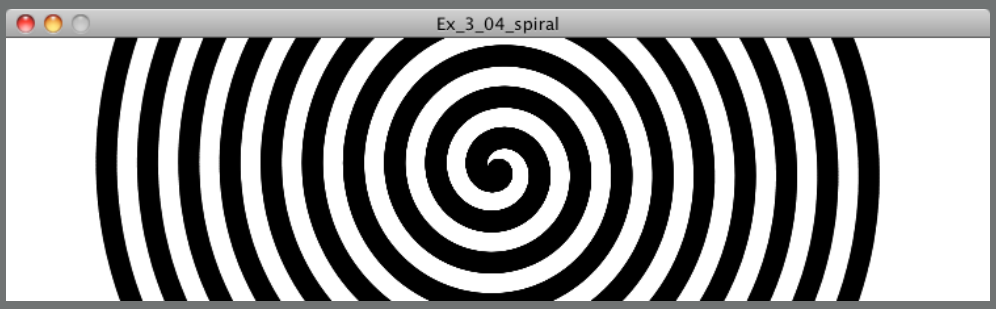
```
float r = 75;
float theta = 0;

void setup() {
  size(640,360);
  background(255);
}

void draw() {

  float x = r * cos(theta);
  float y = r * sin(theta);
```

Polar coordinates (r,theta) are converted to Cartesian (x,y) for use in the ellipse() function.

```
  noStroke();
  fill(0);
  ellipse(x+width/2, y+height/2, 16, 16);

  theta += 0.01;
}
```

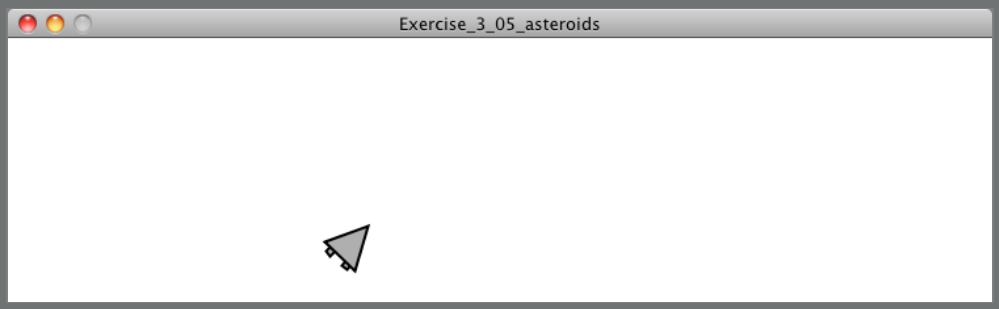
Exercise 3.4

Using Example 3.4 as a basis, draw a spiral path. Start in the center and move outwards. Note that this can be done by only changing one line of code and adding one line of code!



Exercise 3.5

Simulate the spaceship in the game Asteroids. In case you aren't familiar with Asteroids, here is a brief description: A spaceship (represented as a triangle) floats in two dimensional space. The left arrow key turns the spaceship counterclockwise, the right arrow key, clockwise. The *z* key applies a "thrust" force in the direction the spaceship is pointing.



3.6 Oscillation Amplitude and Period

Are you amazed yet? We've seen some pretty great uses of tangent (for finding the angle of a vector) and sine and cosine (for converting from polar to Cartesian coordinates). We could stop right here and be satisfied. But we're not going to. This is only the beginning. What sine and cosine can do for you goes beyond mathematical formulas and right triangles.

Let's take a look at a graph of the sine function, where *y = sine(x)*.

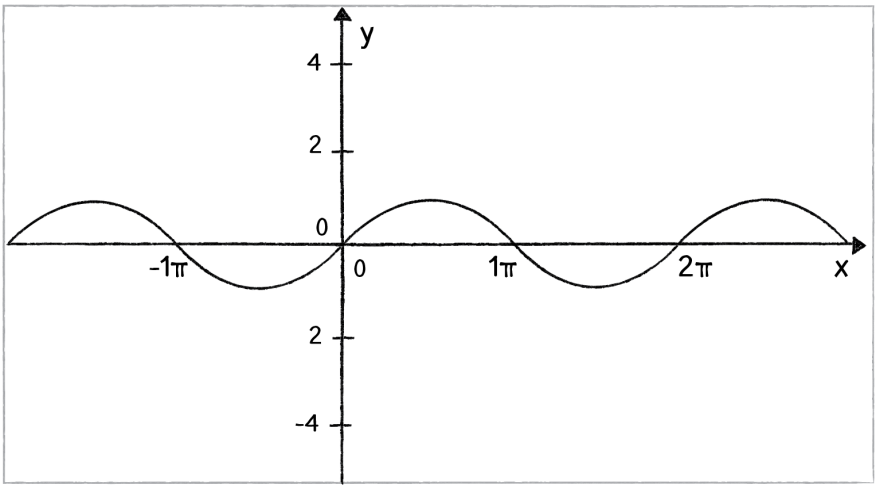
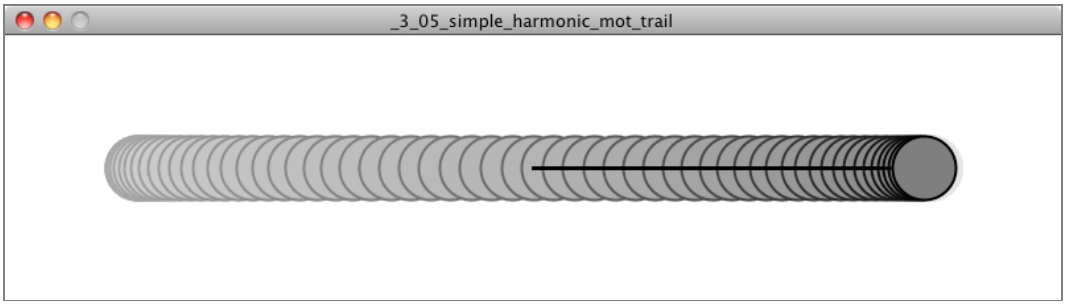


Figure 3.9: y = sine(x)

You'll notice that the output of the sine function is a smooth curve alternating between –1 and 1. This type of a behavior is known as ***oscillation***, a periodic movement between two points. Plucking a guitar string, swinging a pendulum, bouncing on a pogo stick—these are all examples of oscillating motion.

And so we happily discover that we can simulate oscillation in a Processing sketch by assigning the output of the sine function to an object's location. Note that this will follow the same methodology we applied to Perlin noise in the Introduction (see page 17).

Let's begin with a really basic scenario. We want a circle to oscillate from the left side to the right side of a Processing window.



This is what is known as ***simple harmonic motion*** (or, to be fancier, "the periodic sinusoidal oscillation of an object"). It's going to be a simple program to write, but before we get into the code, let's familiarize ourselves with some of the terminology of oscillation (and waves).

Simple harmonic motion can be expressed as any location (in our case, the x location) as a function of time, with the following two elements:

- ***Amplitude***: The distance from the center of motion to either extreme
- ***Period***: The amount of time it takes for one complete cycle of motion

Looking at the graph of sine (Figure 3.9), we can see that the amplitude is 1 and the period is `TWO_PI`; the output of sine never rises above 1 or below -1; and every `TWO_PI` radians (or 360 degrees) the wave pattern repeats.

Now, in the Processing world we live in, what is amplitude and what is period? Amplitude can be measured rather easily in pixels. In the case of a window 200 pixels wide, we would oscillate from the center 100 pixels to the right and 100 pixels to the left. Therefore:

```
float amplitude = 100;                    Our amplitude is measured in pixels.
```

Period is the amount of time it takes for one cycle, but what is time in our Processing world? I mean, certainly we could say we want the circle to oscillate every three seconds. And we could track the milliseconds—using `millis()` —in Processing and come up with an elaborate algorithm for oscillating an object according to real-world time. But for us, real-world time doesn't really matter. The real measure of time in Processing is in frames. The oscillating motion should repeat every 30 frames, or 50 frames, or 1000 frames, etc.

```
float period = 120;                       Our period is measured in frames (our unit
                                          of time for animation).
```

Once we have the amplitude and period, it's time to write a formula to calculate x as a function of time, which we now know is the current frame count.

```
float x = amplitude * cos(TWO_PI * frameCount / period);
```

Let's dissect the formula a bit more and try to understand each component. The first is probably the easiest. Whatever comes out of the cosine function we multiply by amplitude. We know that cosine will oscillate between -1 and 1. If we take that value and multiply it by amplitude then we'll get the desired result: a value oscillating between -amplitude and amplitude. (Note: this is also a place where we could use Processing's `map()` function to map the output of cosine to a custom range.)

Now, let's look at what is inside the cosine function:

```
TWO_PI * frameCount / period
```

What's going on here? Let's start with what we know. We know that cosine will repeat every 2*PI radians—i.e. it will start at 0 and repeat at 2*PI, 4*PI, 6*PI, etc. If the period is 120, then we want the oscillating motion to repeat when the `frameCount` is at 120 frames, 240 frames, 360 frames, etc. `frameCount` is really the only variable; it starts at 0 and counts upward. Let's take a look at what the formula yields with those values.

frameCount	frameCount / period	TWO_PI * frameCount / period
0	0	0
60	0.5	PI
120	1	TWO_PI
240	2	2 * TWO_PI (or 4* PI)
etc.		

`frameCount` divided by `period` tells us how many cycles we've completed—are we halfway through the first cycle? Have we completed two cycles? By multiplying that number by `TWO_PI`, we get the result we want, since `TWO_PI` is the number of radians required for one cosine (or sine) to complete one cycle.

Wrapping this all up, here's the Processing example that oscillates the x location of a circle with an amplitude of 100 pixels and a period of 120 frames.

Example 3.5: Simple Harmonic Motion

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

void draw() {
  background(255);

  float period = 120;
  float amplitude = 100;
  float x = amplitude * cos(TWO_PI * frameCount / period);

  stroke(0);
  fill(175);
  translate(width/2,height/2);
  line(0,0,x,0);
  ellipse(x,0,20,20);
}
```

Calculating horizontal location according to the formula for simple harmonic motion

It's also worth mentioning the term ***frequency***: the number of cycles per time unit. Frequency is equal to 1 divided by `period`. If the period is 120 frames, then only 1/120th of a cycle is completed in one frame, and so frequency = 1/120. In the above example, we simply chose to define the rate of oscillation in terms of period and therefore did not need a variable for frequency.

Exercise 3.6

Using the sine function, create a simulation of a weight (sometimes referred to as a "bob") that hangs from a spring from the top of the window. Use the `map()` function to calculate the vertical location of the bob. Later in this chapter, we'll see how to recreate this same simulation by modeling the forces of a spring according to Hooke's law.

3.7 Oscillation with Angular Velocity

An understanding of the concepts of oscillation, amplitude, and frequency/period is often required in the course of simulating real-world behaviors. However, there is a slightly easier way to rewrite the above example with the same result. Let's take one more look at our oscillation formula:

```
float x = amplitude * cos(TWO_PI * frameCount / period);
```

And let's rewrite it a slightly different way:

```
float x = amplitude * cos ( some value that increments slowly );
```

If we care about precisely defining the period of oscillation in terms of frames of animation, we might need the formula the way we first wrote it, but we can just as easily rewrite our example using the concept of angular velocity (and acceleration) from section 3.2 (see page 104). Assuming:

```
float angle = 0;
float aVelocity = 0.05;
```

in `draw()`, we can simply say:

```
angle += aVelocity;
float x = amplitude * cos(angle);
```

`angle` is our "some value that increments slowly."

Example 3.6: Simple Harmonic Motion II

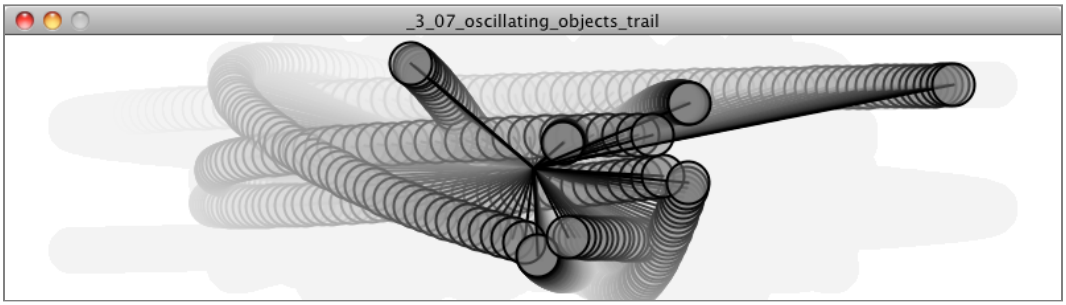
```
float angle = 0;
float aVelocity = 0.05;

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

void draw() {
  background(255);

  float amplitude = 100;
  float x = amplitude * cos(angle);
  angle += aVelocity;

  ellipseMode(CENTER);
  stroke(0);
  fill(175);
  translate(width/2,height/2);
  line(0,0,x,0);
  ellipse(x,0,20,20);
}
```

Using the concept of angular velocity to increment an angle variable

Just because we're not referencing it directly doesn't mean that we've eliminated the concept of `period`. After all, the greater the angular velocity, the faster the circle will oscillate (therefore lowering the period). In fact, the number of times it takes to add up the angular velocity to get to `TWO_PI` is the period or:

```
period = TWO_PI / angular velocity
```

Let's expand this example a bit more and create an `Oscillator` class. And let's assume we want the oscillation to happen along both the x-axis (as above) and the y-axis. To do this, we'll need two angles, two angular velocities, and two amplitudes (one for each axis). Another perfect opportunity for `PVector`!



Example 3.7: Oscillator objects

```
class Oscillator  {

  PVector angle;                                          Using a PVector to track two angles!
  PVector velocity;
  PVector amplitude;

  Oscillator()  {
    angle = new PVector();
    velocity = new PVector(random(-0.05,0.05),random(-0.05,0.05));
    amplitude = new PVector(random(width/2),random(height/2));
  }                                                       Random velocities and amplitudes

  void oscillate()  {
    angle.add(velocity);
  }

  void display()  {
    float x = sin(angle.x)*amplitude.x;                   Oscillating on the x-axis
    float y = sin(angle.y)*amplitude.y;                   Oscillating on the y-axis

    pushMatrix();
    translate(width/2,height/2);
    stroke(0);
    fill(175);
```

```
    line(0,0,x,y);
    ellipse(x,y,16,16);
    popMatrix();
  }
}
```

Drawing the Oscillator as a line connecting a circle

Exercise 3.7

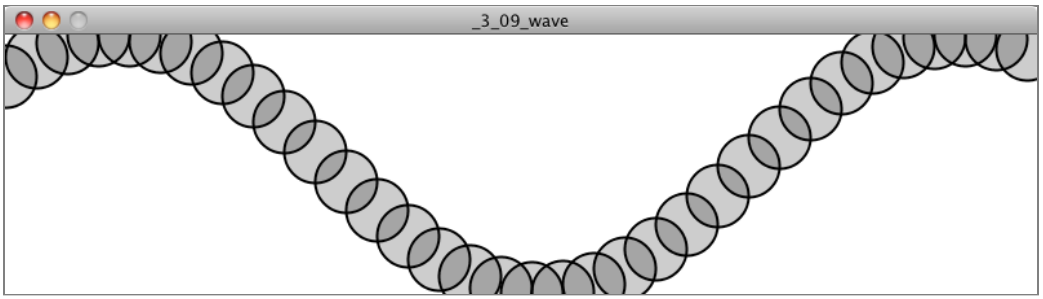
Try initializing each `Oscillator` object with velocities and amplitudes that are not random to create some sort of regular pattern. Can you make the oscillators appear to be the legs of a insect-like creature?

Exercise 3.8

Incorporate angular acceleration into the `Oscillator` object.

3.8 Waves

If you're saying to yourself, "Um, this is all great and everything, but what I really want is to draw a wave onscreen," well, then, the time has come. The thing is, we're about 90% there. When we oscillate a single circle up and down according to the sine function, what we are doing is looking at a single point along the x-axis of a wave pattern. With a little panache and a `for` loop, we can place a whole bunch of these oscillating circles next to each other.



This wavy pattern could be used in the design of the body or appendages of a creature, as well as to simulate a soft surface (such as water).

Here, we're going to encounter the same questions of amplitude (height of pattern) and period. Instead of period referring to time, however, since we're looking at the full wave, we

can talk about period as the width (in pixels) of a full wave cycle. And just as with simple oscillation, we have the option of computing the wave pattern according to a precise period or simply following the model of angular velocity.

Let's go with the simpler case, angular velocity. We know we need to start with an angle, an angular velocity, and an amplitude:

```
float angle = 0;
float angleVel = 0.2;
float amplitude = 100;
```

Then we're going to loop through all of the x values where we want to draw a point of the wave. Let's say every 24 pixels for now. In that loop, we're going to want to do three things:

1. Calculate the y location according to amplitude and sine of the angle.
2. Draw a circle at the *(x,y)* location.
3. Increment the angle according to angular velocity.

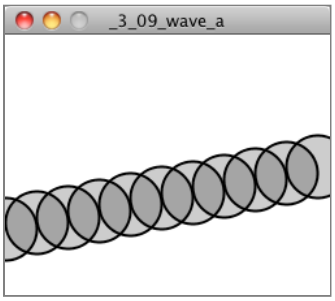
```
for (int x = 0; x <= width; x += 24) {

  float y = amplitude*sin(angle);

  ellipse(x,y+height/2,48,48);

  angle += angleVel;
}
```

1) Calculate the y location according to amplitude and sine of the angle.

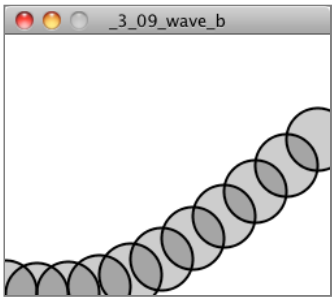
2) Draw a circle at the (x,y) location.

3) Increment the angle according to angular velocity.

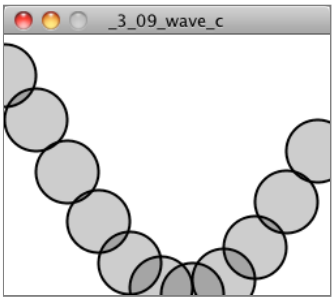
Let's look at the results with different values for `angleVel`:



angleVel = 0.05

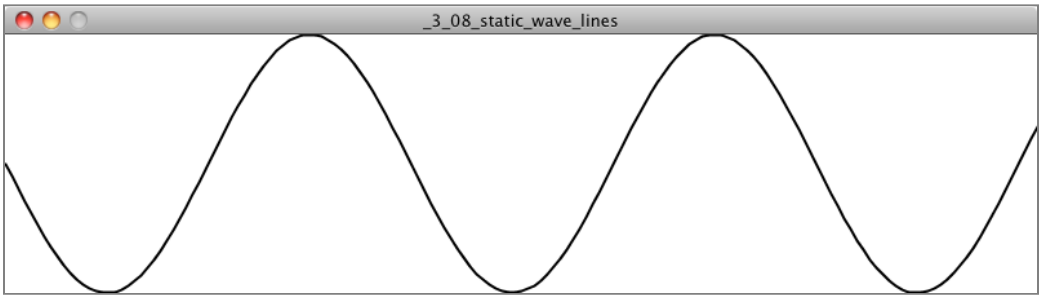


angleVel = 0.2



angleVel = 0.4

Notice how, although we're not precisely computing the period of the wave, the higher the angular velocity, the shorter the period. It's also worth noting that as the period becomes shorter, it becomes more and more difficult to make out the wave itself as the distance between the individual points increases. One option we have is to use `beginShape()` and `endShape()` to connect the points with a line.



Example 3.8: Static wave drawn as a continuous line

```
float angle = 0;
float angleVel = 0.2;
float amplitude = 100;

size(400,200);
background(255);

stroke(0);
strokeWeight(2);
noFill();

beginShape();
for (int x = 0; x <= width; x += 5) {
  float y = map(sin(angle),-1,1,0,height);
  vertex(x,y);
  angle +=angleVel;
}
endShape();
```

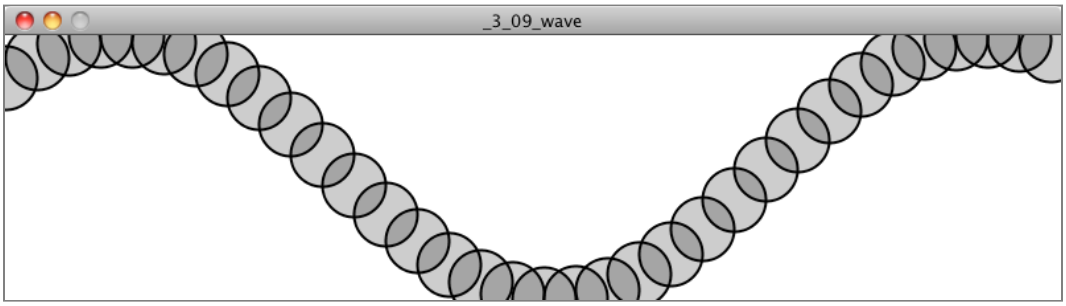
Here's an example of using the map() function instead.

With beginShape() and endShape(), you call vertex() to set all the vertices of your shape.

You may have noticed that the above example is static. The wave never changes, never undulates. This additional step is a bit tricky. Your first instinct might be to say: "Hey, no problem, we'll just let theta be a global variable and let it increment from one cycle through `draw()` to another."

While it's a nice thought, it doesn't work. If you look at the wave, the righthand edge doesn't match the lefthand; where it ends in one cycle of `draw()` can't be where it starts in the next. Instead, what we need to do is have a variable dedicated entirely to tracking what value of

angle the wave should start with. This angle (which we'll call `startAngle`) increments with its own angular velocity.



Example 3.9: The Wave

```
float startAngle = 0;
float angleVel = 0.1;

void setup() {
  size(400,200);
}

void draw() {
  background(255);

float angle = startAngle;
```

In order to move the wave, we start at a different theta value each frame. startAngle += 0.02;

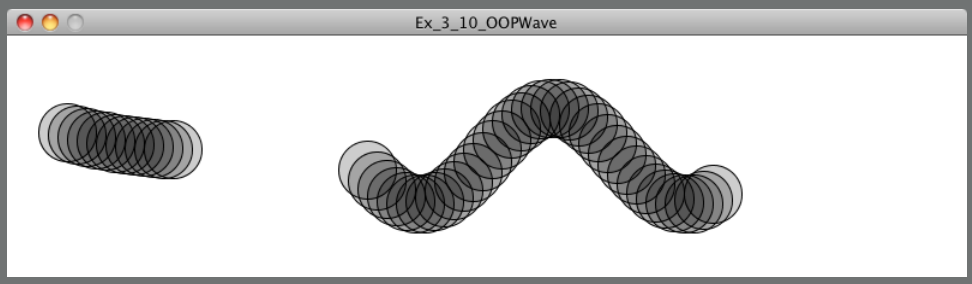
```
  for (int x = 0; x <= width; x += 24) {
    float y = map(sin(angle),-1,1,0,height);
    stroke(0);
    fill(0,50);
    ellipse(x,y,48,48);
    angle += angleVel;
  }
}
```

Exercise 3.9

Try using the Perlin noise function instead of sine or cosine with the above example.

Exercise 3.10

Encapsulate the above examples into a `Wave` class and create a sketch that displays two waves (with different amplitudes/periods) as in the screenshot below. Move beyond plain circles and lines and try visualizing the wave in a more creative way.



Exercise 3.11

More complex waves can be produced by the values of multiple waves together. Create a sketch that implements this, as in the screenshot below.